

# Cardano.BM - benchmarking and logging

Alexander Diemand

Andreas Triantafyllos

November 2018

### **Abstract**

This is a framework that combines logging, benchmarking and monitoring. Complex evaluations of STM or monadic actions can be observed from outside while reading operating system counters before and after, and calculating their differences, thus relating resource usage to such actions. Through interactive configuration, the runtime behaviour of logging or the measurement of resource usage can be altered. Further reduction in logging can be achieved by redirecting log messages to an aggregation function which will output the running statistics with less frequency than the original message.

# Contents

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>Cardano BM</b>  | <b>3</b> |
| 1.1      | Overview   | 3        |
| 1.2      | Introduction   | 3        |
| 1.2.1    | Logging with Trace   | 3        |
| 1.2.2    | Setup procedure  | 3        |
| 1.2.3    | Measuring <i>Observables</i>   | 3        |
| 1.2.4    | Information reduction in <b>Aggregation</b>  | 3        |
| 1.2.5    | Output selection   | 3        |
| 1.2.6    | Monitoring   | 3        |
| 1.3      | Examples   | 3        |
| 1.3.1    | Observing evaluation of a STM action   | 3        |
| 1.3.2    | Observing evaluation of a monad action   | 3        |
| 1.3.3    | Simple example showing plain logging   | 3        |
| 1.3.4    | Complex example showing logging, aggregation of log items, and observing <i>IO</i> actions | 5        |
| 1.4      | Code listings  | 9        |
| 1.4.1    | Cardano.BM.Observer.STM  | 9        |
| 1.4.2    | Cardano.BM.Observer.Monad  | 11       |
| 1.4.3    | BaseTrace  | 14       |
| 1.4.4    | Cardano.BM.Trace   | 15       |
| 1.4.5    | Cardano.BM.Setup   | 20       |
| 1.4.6    | Cardano.BM.Counters  | 21       |
| 1.4.7    | Cardano.BM.Counters.Common   | 22       |
| 1.4.8    | Cardano.BM.Counters.Dummy  | 23       |
| 1.4.9    | Cardano.BM.Counters.Linux  | 24       |
| 1.4.10   | Cardano.BM.Data.Aggregated   | 30       |
| 1.4.11   | Cardano.BM.Data.Backend  | 35       |
| 1.4.12   | Cardano.BM.Data.Configuration  | 36       |
| 1.4.13   | Cardano.BM.Data.Counter  | 37       |
| 1.4.14   | Cardano.BM.Data.LogItem  | 38       |
| 1.4.15   | Cardano.BM.Data.Observable   | 40       |
| 1.4.16   | Cardano.BM.Data.Output   | 40       |
| 1.4.17   | Cardano.BM.Data.Severity   | 41       |
| 1.4.18   | Cardano.BM.Data.SubTrace   | 42       |
| 1.4.19   | Cardano.BM.Data.Trace  | 42       |
| 1.4.20   | Cardano.BM.Configuration   | 43       |
| 1.4.21   | Cardano.BM.Configuration.Model   | 43       |
| 1.4.22   | Cardano.BM.Output.Switchboard  | 51       |

|  |    |
|--|----|
| 1.4.23 Cardano.BM.Output.Log . . . . .         | 53 |
| 1.4.24 Cardano.BM.Output.EKGView . . . . .     | 60 |
| 1.4.25 Cardano.BM.Output.Aggregation . . . . . | 63 |

# Chapter 1

## Cardano BM

### 1.1 Overview

In figure 1.1 we display the relationships among modules in *Cardano.BM*. The arrows indicate import of a module. The arrows with a triangle at one end would signify "inheritance" in object-oriented programming, but we use it to show that one module replaces the other in the namespace, thus refines its interface.

### 1.2 Introduction

#### 1.2.1 Logging with **Trace**

#### 1.2.2 Setup procedure

#### Hierarchy of **Traces**

#### 1.2.3 Measuring *Observables*

#### 1.2.4 Information reduction in **Aggregation**

#### 1.2.5 Output selection

#### 1.2.6 Monitoring

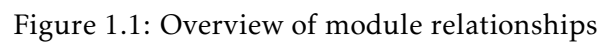
### 1.3 Examples

#### 1.3.1 Observing evaluation of a STM action

#### 1.3.2 Observing evaluation of a monad action

#### 1.3.3 Simple example showing plain logging

```
{-# LANGUAGE OverloadedStrings #-}
module Main
  (main)
  where
import Control.Concurrent (threadDelay)
import Cardano.BM.Configuration.Static (defaultConfigStdout)
```



```
import Cardano.BM.Setup (setupTrace)
import Cardano.BM.Trace (logDebug, logError, logInfo, logNotice,
                        logWarning)

main :: IO ()
main = do
    c ← defaultConfigStdout
    tr ← setupTrace (Right c) "simple"

    logDebug tr "this is a debug message"
    logInfo   tr "this is an information."
    logNotice tr "this is a notice!"
    logWarning tr "this is a warning!"
    logError  tr "this is an error!"

    threadDelay 80000
```



Figure 1.2: Setup procedure

```
return ()
```

### 1.3.4 Complex example showing logging, aggregation of log items, and observing IO actions

#### Module header and import directives

```

{-# LANGUAGE OverloadedStrings #-}
module Main
  (main)
  where
import Control.Concurrent (threadDelay)
import qualified Control.Concurrent.Async as Async
import Control.Monad (forM, forM_)
import GHC.Conc.Sync (STM, TVar, atomically, newTVar, readTVar, writeTVar)
import Data.Text (pack)
import System.Random

import qualified Cardano.BM.Configuration.Model as CM
import Cardano.BM.Data.Aggregated (Measurable (..))
import Cardano.BM.Data.AggregatedKind
import Cardano.BM.Data.BackendKind
import Cardano.BM.Data.LogItem
import Cardano.BM.Data.Observable
import Cardano.BM.Data.Output
import Cardano.BM.Data.Severity
import Cardano.BM.Data.SubTrace
import Cardano.BM.Observer.Monad (bracketObserveIO)
import qualified Cardano.BM.Observer.STM as STM
import Cardano.BM.Setup
import Cardano.BM.Trace

```

## Define configuration

The output can be viewed in EKG on <http://localhost:12789>.

```

config :: IO CM.Configuration
config = do
  c ← CM.empty
  CM.setMinSeverity c Debug
  CM.setSetupBackends c [KatipBK, AggregationBK, EKGViewBK]
  CM.setDefaultBackends c [KatipBK]
  CM.setSetupScribes c [ScribeDefinition {
    scName = "stdout"
    ,scKind = StdoutSK
    ,scRotation = Nothing
  }
    ,ScribeDefinition {
    scName = "out.odd.json"
    ,scKind = FileJsonSK
    ,scRotation = Nothing
  }
    ,ScribeDefinition {
    scName = "out.even.json"
    ,scKind = FileJsonSK
    ,scRotation = Nothing
  }
    ,ScribeDefinition {
    scName = "out.txt"
    ,scKind = FileTextSK
    ,scRotation = Nothing
  }
  ]
  CM.setDefaultScribes c ["StdoutSK::stdout"]
  CM.setScribes c "complex.random" (Just ["StdoutSK::stdout", "FileTextSK::out.txt"])
  CM.setScribes c "#aggregated.complex.random" (Just ["StdoutSK::stdout"])
  forM_ [(1 :: Int)..10] $ \x →
    if odd x
    then
      CM.setScribes c ("#aggregation.complex.observeSTM." <> (pack $ show x)) $ Just ["FileJsonSK::out"]
    else
      CM.setScribes c ("#aggregation.complex.observeSTM." <> (pack $ show x)) $ Just ["FileJsonSK::out"]
  CM.setSubTrace c "complex.random" (Just $ TeeTrace "ewma")
  CM.setSubTrace c "#ekgview"
  (Just $ FilterTrace [(Drop (StartsWith "#ekgview.#aggregation.complex.random"),
    Unhide [(EndsWith ".count"),
      (EndsWith ".avg"),
      (EndsWith ".mean")]),
    (Drop (StartsWith "#ekgview.#aggregation.complex.observeIO"),
      Unhide [(Contains "close.RTS"),

```



```

        (Contains "diff_OFF.RTS"))])
    ])
    CM.setSubTrace c "complex.observeIO" (Just $ ObservableTrace [GhcRtsStats,MemoryStats])
    forM_ [(1 :: Int)..10] $ \x →
        CM.setSubTrace
            c
            ("complex.observeSTM." <> (pack $ show x))
            (Just $ ObservableTrace [GhcRtsStats,MemoryStats])
    CM.setBackends c "complex.random" (Just [AggregationBK,KatipBK])
    CM.setBackends c "complex.random.ewma" (Just [AggregationBK])
    CM.setBackends c "complex.observeIO" (Just [AggregationBK])
    forM_ [(1 :: Int)..10] $ \x →
        CM.setBackends
            c
            ("complex.observeSTM." <> (pack $ show x))
            (Just [AggregationBK])
    CM.setAggregatedKind c "complex.random.rr" (Just StatsAK)
    CM.setAggregatedKind c "complex.random.ewma.rr" (Just (EwmaAK 0.42))
    CM.setBackends c "#aggregation.complex.observeIO" (Just [EKGVViewBK])
    CM.setBackends c "#aggregation.complex.random" (Just [EKGVViewBK])
    CM.setBackends c "#aggregation.complex.random.ewma" (Just [EKGVViewBK])
    CM.setEKGVport c 12789
    return c

```

### Thread that outputs a random number to a **Trace**

```

randomThr :: Trace IO → IO (Async.Async ())
randomThr trace = do
    logInfo trace "starting random generator"
    trace' ← subTrace "random" trace
    proc ← Async.async (loop trace')
    return proc
    where
        loop tr = do
            threadDelay 500000 -- 0.5 second
            num ← randomRIO (42 - 42, 42 + 42) :: IO Double
            lo ← LogObject <$> mkLOMeta <*> pure (LogValue "rr" (PureD num))
            traceNamedObject tr lo
            loop tr

```

### Thread that observes an IO action

```

observeIO :: Trace IO → IO (Async.Async ())
observeIO trace = do
    logInfo trace "starting observer"

```

```

proc ← Async.async (loop trace)
return proc
where
  loop tr = do
    threadDelay 5000000 -- 5 seconds
    _ ← bracketObserveIO tr "observeIO" $ do
      num ← randomRIO (100000, 200000) :: IO Int
      ls ← return $ reverse $ init $ reverse $ 42 : [1..num]
      pure $ const ls ()
    loop tr

```

**Thread that observes an IO action which downloads a txt in order to observe the I/O statistics**

disabled for now! on Mac OSX this function was blocking all IO.

```

observeDownload :: Trace IO → IO ()
observeDownload trace = loop trace
where
  loop tr = do
    threadDelay 10000000 -- 10 seconds
    tr' ← appendName "observeDownload" tr
    bracketObserveIO tr' "" $ do
      license ← openURI "http://www.gnu.org/licenses/gpl.txt"
      case license of
        Right bs → logNotice tr' $ pack $ take 100 $ BS8.unpack bs
        Left _ → return ()
      threadDelay 500000 -- .5 second
      pure ()
    loop tr

```

**Threads that observe STM actions on the same TVar**

```

observeSTM :: Trace IO → IO [Async.Async ()]
observeSTM trace = do
  logInfo trace "starting STM observer"
  tvar ← atomically $ newTVar ([1..1000] :: [Int])
  -- spawn 10 threads
  proc ← forM [(1 :: Int)..10] $ \x → Async.async (loop trace tvar (pack $ show x))
  return proc
where
  loop tr tvarlist name = do
    threadDelay 10000000 -- 10 seconds
    STM.bracketObserveIO tr ("observeSTM." <> name) (stmAction tvarlist)
    loop tr tvarlist name

stmAction :: TVar [Int] → STM ()
stmAction tvarlist = do

```

```

list ← readTVar tvarlist
writeTVar tvarlist $ reverse $ init $ reverse $ list
pure ()

```

### Main entry point

```

main :: IO ()
main = do
  -- create configuration
  c ← config
  -- create initial top-level Trace
  tr ← setupTrace (Right c) "complex"
  logNotice tr "starting program; hit CTRL-C to terminate"
  logInfo tr "watch its progress on http://localhost:12789"
  {-start thread sending unbounded sequence of random numbers to a trace which aggregates them into a
  procRandom ← randomThr tr
  -- start thread endlessly reversing lists of random length
  procObsvIO ← observeIO tr
  -- start threads endlessly observing STM actions operating on the same TVar
  procObsvSTMs ← observeSTM tr
  -- wait for observer thread to finish, ignoring any exception
  _ ← forM procObsvSTMs Async.waitCatch
  -- wait for observer thread to finish, ignoring any exception
  _ ← Async.waitCatch procObsvIO
  -- wait for random thread to finish, ignoring any exception
  _ ← Async.waitCatch procRandom
  return ()

```

## 1.4 Code listings

### 1.4.1 Cardano.BM.Observer.STM

```

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

```

#### Observe STM action in a named context

With given name, create a **SubTrace** according to **Configuration** and run the passed *STM* action on it.

```

bracketObserveIO :: Trace IO → Text → STM.STM t → IO t
bracketObserveIO logTrace0 name action = do
  logTrace ← subTrace name logTrace0

```

```

let subtrace = typeofTrace logTrace
bracketObserveIO' subtrace logTrace action
where
bracketObserveIO' :: SubTrace → Trace IO → STM.STM t → IO t
bracketObserveIO' NoTrace _ act =
  STM.atomically act
bracketObserveIO' subtrace logTrace act = do
  mCountersid ← observeOpen subtrace logTrace
  -- run action; if an exception is caught will be logged and rethrown.
  t ← (STM.atomically act) 'catch' (λ(e :: SomeException) → (logError logTrace (pack (show e)) >> throwM e))
  case mCountersid of
    Left openException →
      -- since observeOpen faced an exception there is no reason to call observeClose
      -- however the result of the action is returned
      logNotice logTrace ("ObserveOpen: " <> pack (show openException))
    Right countersid → do
      res ← observeClose subtrace logTrace countersid [ ]
      case res of
        Left ex → logNotice logTrace ("ObserveClose: " <> pack (show ex))
        _ → pure ()
  pure t

```

### Observe STM action in a named context and output captured log items

The STM action might output messages, which after "success" will be forwarded to the logging trace. Otherwise, this function behaves the same as **Observe STM action in a named context**.

```

bracketObserveLogIO :: Trace IO → Text → STM.STM (t, [LogObject]) → IO t
bracketObserveLogIO logTrace0 name action = do
  logTrace ← subTrace name logTrace0
  let subtrace = typeofTrace logTrace
  bracketObserveLogIO' subtrace logTrace action
where
bracketObserveLogIO' :: SubTrace → Trace IO → STM.STM (t, [LogObject]) → IO t
bracketObserveLogIO' NoTrace _ act = do
  (t, _) ← STM.atomically $ stmWithLog act
  pure t
bracketObserveLogIO' subtrace logTrace act = do
  mCountersid ← observeOpen subtrace logTrace
  -- run action, return result and log items; if an exception is
  -- caught will be logged and rethrown.
  (t, as) ← (STM.atomically $ stmWithLog act) 'catch'
    (λ(e :: SomeException) → (logError logTrace (pack (show e)) >> throwM e))
  case mCountersid of
    Left openException →
      -- since observeOpen faced an exception there is no reason to call observeClose
      -- however the result of the action is returned

```

```

    logNotice logTrace ("ObserveOpen: " <> pack (show openException))
Right countersid → do
  res ← observeClose subtrace logTrace countersid as
  case res of
    Left ex → logNotice logTrace ("ObserveClose: " <> pack (show ex))
    _ → pure ()
pure t

```

### 1.4.2 Cardano.BM.Observer.Monad

#### Monad.bracketObserverIO

Observes an *IO* action and adds a name to the logger name of the passed in **Trace**. An empty *Text* leaves the logger name untouched.

Microbenchmarking steps:

1. Create a *trace* which will have been configured to observe things besides logging.

```

import qualified Cardano.BM.Configuration.Model as CM
○○○
c ← config
trace@(ctx, _) ← setupTrace (Right c) "demo-playground"
where
  config :: IO CM.Configuration
  config = do
    c ← CM.empty
    CM.setMinSeverity c Debug
    CM.setSetupBackends c [KatipBK, AggregationBK]
    CM.setDefaultBackends c [KatipBK, AggregationBK]
    CM.setSetupScribes c [ScribeDefinition {
      scName = "stdout"
    , scKind = StdoutSK
    , scRotation = Nothing
    }
    ]
    CM.setDefaultScribes c ["StdoutSK::stdout"]
  return c

```

2. *c* is the **Configuration** of *trace*. In order to enable the collection and processing of measurements (min, max, mean, std-dev) *AggregationBK* is needed.

```
CM.setDefaultBackends c [KatipBK, AggregationBK]
```

in a configuration file (YAML) means

```

defaultBackends:
  - KatipBK
  - AggregationBK

```

3. Set the measurements that you want to take by changing the configuration of the *trace* using *setSubTrace*, in order to declare the namespace where we want to enable the particular measurements and the list with the kind of measurements.

```
CM.setSubTrace
  (configuration ctx)
  "demo-playground.submit-tx"
  (Just $ ObservableTrace observablesSet)
where
  observablesSet = [ MonotonicClock, MemoryStats ]
```

4. Find an action to measure. e.g.:

```
runProtocolWithPipe x hdl proto 'catch' (λProtocolStopped → return ())
```

and use *bracketObserveIO*. e.g.:

```
bracketObserveIO trace "submit-tx" $
  runProtocolWithPipe x hdl proto 'catch' (λProtocolStopped → return ())
```

---

```
bracketObserveIO :: Trace IO → Text → IO t → IO t
bracketObserveIO logTrace0 name action = do
  logTrace ← subTrace name logTrace0
  bracketObserveIO' (typeofTrace logTrace) logTrace action
where
  bracketObserveIO' :: SubTrace → Trace IO → IO t → IO t
  bracketObserveIO' NoTrace _ act = act
  bracketObserveIO' subtrace logTrace act = do
    mCountersid ← observeOpen subtrace logTrace
    -- run action; if an exception is caught will be logged and rethrown.
    t ← act 'catch' (λ(e :: SomeException) → (logError logTrace (pack (show e)) >> throwM e))
    case mCountersid of
      Left openException →
        -- since observeOpen faced an exception there is no reason to call observeClose
        -- however the result of the action is returned
        logNotice logTrace ("ObserveOpen: " <> pack (show openException))
      Right countersid → do
        res ← observeClose subtrace logTrace countersid [ ]
        case res of
          Left ex → logNotice logTrace ("ObserveClose: " <> pack (show ex))
          _ → pure ()
    pure t
```

### Monadic.bracketObserverM

Observes a *MonadIO m*  $\Rightarrow m$  action and adds a name to the logger name of the passed in *Trace*. An empty *Text* leaves the logger name untouched.

```

bracketObserveM :: (MonadCatch m, MonadIO m) => Trace IO -> Text -> m t -> m t
bracketObserveM logTrace0 name action = do
  logTrace <- liftIO $ subTrace name logTrace0
  bracketObserveM' (typeofTrace logTrace) logTrace action
where
  bracketObserveM' :: (MonadCatch m, MonadIO m) => SubTrace -> Trace IO -> m t -> m t
  bracketObserveM' NoTrace _ act = act
  bracketObserveM' subtrace logTrace act = do
    mCountersid <- liftIO $ observeOpen subtrace logTrace
    -- run action; if an exception is caught will be logged and rethrown.
    t <- act 'catch'
    (\(e :: SomeException) -> (liftIO (logError logTrace (pack (show e))) >> throwM e)))
  case mCountersid of
    Left openException ->
      -- since observeOpen faced an exception there is no reason to call observeClose
      -- however the result of the action is returned
      liftIO $ logNotice logTrace ("ObserveOpen: " <> pack (show openException))
    Right countersid -> do
      res <- liftIO $ observeClose subtrace logTrace countersid [ ]
      case res of
        Left ex -> liftIO (logNotice logTrace ("ObserveClose: " <> pack (show ex)))
        _ -> pure ()
  pure t

```

### observerOpen

```

observeOpen :: SubTrace -> Trace IO -> IO (Either SomeException CounterState)
observeOpen subtrace logTrace = (do
  identifier <- newUnique
  -- take measurement
  counters <- readCounters subtrace
  let state = CounterState identifier counters
  if counters == [ ]
  then return ()
  else do
    -- send opening message to Trace
    traceNamedObject logTrace <<=
      LogObject < $ > mkLOMeta < * > pure (ObserveOpen state)
    return (Right state)) 'catch' (return o Left)

```

### observeClose

```

observeClose :: SubTrace -> Trace IO -> CounterState -> [LogObject] -> IO (Either SomeException ())
observeClose subtrace logTrace initState logObjects = (do
  let identifier = csIdentifier initState

```

```

    initialCounters = csCounters initState
-- take measurement
counters ← readCounters subtrace
if counters ≡ []
then return ()
else do
    -- send closing message to Trace
    traceNamedObject logTrace ≪
        LogObject < $ > mkLOMeta < * > pure (ObserveClose (CounterState identifier counters))
    -- send diff message to Trace
    traceNamedObject logTrace ≪
        LogObject < $ > mkLOMeta < * > pure (ObserveDiff (CounterState identifier (diffCounters initialCounters)))
-- trace the messages gathered from inside the action
forM_ logObjects $ traceNamedObject logTrace
return (Right ())) 'catch' (return ◦ Left)

```

### 1.4.3 BaseTrace

#### Contravariant

A covariant is a functor:  $F A \rightarrow F B$

A contravariant is a functor:  $F B \rightarrow F A$

$Op\ a\ b$  implements the inverse to 'arrow' " $getOp :: b \rightarrow a$ ", which when applied to a **BaseTrace** of type " $Op\ (m\ ())\ s$ ", yields " $s \rightarrow m\ ()$ ". 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 " $fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$ ". In case of a contravariant functor, it is the dual function " $contramap :: (a \rightarrow b) \rightarrow f\ b \rightarrow f\ a$ " which is defined.

In the following instance,  $runTrace$  extracts type " $Op\ (m\ ())\ s$ " to which  $contramap$  applies  $f$ , thus " $f\ s \rightarrow m\ ()$ ". 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 " $getOp :: b \rightarrow a$ " which translates to " $s \rightarrow m\ ()$ ", 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 o m x → n x) → BaseTrace m s → BaseTrace n s
natTrace nat (BaseTrace (Op tr)) = BaseTrace $ Op $ nat o 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****Utilities**

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

```
natTrace :: (forall x o m x → n x) → Trace m → Trace n
natTrace nat (ctx, trace) = (ctx, BaseTrace.natTrace nat trace)
```

Access type of **Trace**.

```
typeofTrace :: Trace m → SubTrace
typeofTrace (ctx, _) = tracetype ctx
```

Update type of **Trace**.

```
updateTracetype :: SubTrace → Trace m → Trace m
updateTracetype subtr (ctx, tr) = (ctx {tracetype = subtr}, tr)
```

**Enter new named context**

The context name is created and checked that its size is below a limit (currently 80 chars). The minimum severity that a log message must be labelled with is looked up in the configuration and recalculated.

```
appendName :: MonadIO m ⇒ LoggerName → Trace m → m (Trace m)
appendName name (ctx, trace) = do
  let prevLoggerName = loggerName ctx
      prevMinSeverity = minSeverity ctx
      newLoggerName = appendWithDot prevLoggerName name
      globMinSeverity ← liftIO $ Config.minSeverity (configuration ctx)
      namedSeverity ← liftIO $ Config.inspectSeverity (configuration ctx) newLoggerName
  case namedSeverity of
    Nothing → return (ctx {loggerName = newLoggerName}, trace)
    Just sev → return (ctx {loggerName = newLoggerName
```

```

    ,minSeverity = max (max sev prevMinSeverity) globMinSeverity}
    ,trace)
appendWithDot :: LoggerName → LoggerName → LoggerName
appendWithDot "" newName = T.take 80 newName
appendWithDot xs "" = xs
appendWithDot xs newName = T.take 80 $ xs <> "." <> newName

```

### Contramap a trace and produce the naming context

```

named :: BaseTrace.BaseTrace m (LogNamed i) → LoggerName → BaseTrace.BaseTrace m i
named trace name = contramap (LogNamed name) trace

```

### Trace a **LogObject** through

```

traceNamedObject
  :: MonadIO m
  ⇒ Trace m
  → LogObject
  → m ()
traceNamedObject trace@(ctx,logTrace) lo@(LogObject _ lc) = do
  let lname = loggerName ctx
  doOutput ← case (typeofTrace trace) of
    FilterTrace filters →
      case lc of
        LogValue lname _ →
          return $ evalFilters filters (lname <> "." <> lname)
        _ →
          return $ evalFilters filters lname
    TeeTrace secName → do
      -- create a newly named copy of the LogObject
      BaseTrace.traceWith (named logTrace (lname <> "." <> secName)) lo
      return True
    _ → return True
  if doOutput
  then BaseTrace.traceWith (named logTrace lname) lo
  else return ()

```

### Evaluation of **FilterTrace**

A filter consists of a *DropName* and a list of *UnhideNames*. If the context name matches the *DropName* filter, then at least one of the *UnhideNames* must match the name to have the evaluation of the filters return *True*.

```

evalFilters :: [(DropName, UnhideNames)] → LoggerName → Bool
evalFilters fs nm =

```

```

    all (λ(no, yes) → if (dropFilter nm no) then (unhideFilter nm yes) else True) fs
  where
    dropFilter :: LoggerName → DropName → Bool
    dropFilter name (Drop sel) = {-not -} (matchName name sel)
    unhideFilter :: LoggerName → UnhideNames → Bool
    unhideFilter _ (Unhide [ ]) = False
    unhideFilter name (Unhide us) = any (λsel → matchName name sel) us
    matchName :: LoggerName → NameSelector → Bool
    matchName name (Exact name') = name ≡ name'
    matchName name (StartsWith prefix) = T.isPrefixOf prefix name
    matchName name (EndsWith postfix) = T.isSuffixOf postfix name
    matchName name (Contains name') = T.isInfixOf name' name

```

### 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.

TODO remove *locallock*

```

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

stdoutTrace :: TraceNamed IO
stdoutTrace = BaseTrace.BaseTrace $ Op $ λ(LogNamed logname (LogObject _ lc)) →
  withMVar locallock $ \_ →
    case lc of
      (LogMessage logItem) →
        output logname $ liPayload logItem
      obj →
        output logname $ toStrict (encodeToLazyText obj)
  where
    output nm msg = TIO.putStrLn $ nm <> " :: " <> msg

```

### Concrete Trace into a TVar

```

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

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

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

```

Check a log item's severity against the **Trace**'s minimum severity

do we need three different **minSeverity** defined?

We do a lookup of the global **minSeverity** in the configuration. And, a lookup of the **minSeverity** for the current named context. These values might have changed in the meanwhile.

A third filter is the **minSeverity** defined in the current context.

```

traceConditionally
  :: MonadIO m
  => Trace m
  → LogObject
  → m ()

traceConditionally logTrace@(ctx, _) msg@(LogObject _ (LogMessage item)) = do
  globminsev ← liftIO $ Config.minSeverity (configuration ctx)
  globnamesev ← liftIO $ Config.inspectSeverity (configuration ctx) (loggerName ctx)
  let minsev = max (minSeverity ctx) $ max globminsev (fromMaybe Debug globnamesev)
      flag = (liSeverity item) ≥ minsev
  when flag $ traceNamedObject logTrace msg
traceConditionally logTrace logObject =
  traceNamedObject 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
  → T.Text
  → m ()

traceNamedItem trace p s m =
  traceConditionally trace ≪
    LogObject < $ > liftIO mkLOMeta
      < * > pure (LogMessage LogItem {liSelection = p
        , liSeverity = s
        , liPayload = m
        })

```

Logging functions

```

logDebug, logInfo, logNotice, logWarning, logError, logCritical, logAlert, logEmergency
  :: MonadIO m => Trace m → T.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
logCritical   logTrace = traceNamedItem logTrace Both Critical
logAlert      logTrace = traceNamedItem logTrace Both Alert
logEmergency  logTrace = traceNamedItem logTrace Both Emergency
logDebugS, logInfoS, logNoticeS, logWarningS, logErrorS, logCriticalS, logAlertS, logEmergencyS
  :: MonadIO m => Trace m → T.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
logCriticalS  logTrace = traceNamedItem logTrace Private Critical
logAlertS     logTrace = traceNamedItem logTrace Private Alert
logEmergencyS logTrace = traceNamedItem logTrace Private Emergency
logDebugP, logInfoP, logNoticeP, logWarningP, logErrorP, logCriticalP, logAlertP, logEmergencyP
  :: MonadIO m => Trace m → T.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
logCriticalP  logTrace = traceNamedItem logTrace Public Critical
logAlertP     logTrace = traceNamedItem logTrace Public Alert
logEmergencyP logTrace = traceNamedItem logTrace Public Emergency
logDebugUnsafeP, logInfoUnsafeP, logNoticeUnsafeP, logWarningUnsafeP, logErrorUnsafeP,
  logCriticalUnsafeP, logAlertUnsafeP, logEmergencyUnsafeP
  :: MonadIO m => Trace m → T.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
logCriticalUnsafeP logTrace = traceNamedItem logTrace PublicUnsafe Critical
logAlertUnsafeP  logTrace = traceNamedItem logTrace PublicUnsafe Alert
logEmergencyUnsafeP logTrace = traceNamedItem logTrace PublicUnsafe Emergency

```

### subTrace

Transforms the input **Trace** according to the **Configuration** using the logger name of the current **Trace** appended with the new name. If the empty *Text* is passed, then the logger name remains untouched.

```

subTrace :: MonadIO m => T.Text → Trace m → m (Trace m)
subTrace name tr@(ctx, _) = do

```

```

let newName = appendWithDot (loggerName ctx) name
subtrace0 ← liftIO $ Config.findSubTrace (configuration ctx) newName
let subtrace = case subtrace0 of Nothing → Neutral; Just str → str
case subtrace of
  Neutral      → do
    tr' ← appendName name tr
    return $ updateTracetype subtrace tr'
  UntimedTrace → do
    tr' ← appendName name tr
    return $ updateTracetype subtrace tr'
  TeeTrace _   → do
    tr' ← appendName name tr
    return $ updateTracetype subtrace tr'
  FilterTrace _ → do
    tr' ← appendName name tr
    return $ updateTracetype subtrace tr'
  NoTrace      → return $ updateTracetype subtrace (ctx, BaseTrace.BaseTrace $ Op $ \_ → pure ())
  DropOpening  → return $ updateTracetype subtrace (ctx, BaseTrace.BaseTrace $ Op $
    λ(LogNamed _ lo@(LogObject _ lc)) → do
      case lc of
        ObserveOpen _ → return ()
        _             → traceNamedObject tr lo)
  ObservableTrace _ → do
    tr' ← appendName name tr
    return $ updateTracetype subtrace tr'

```

### 1.4.5 Cardano.BM.Setup

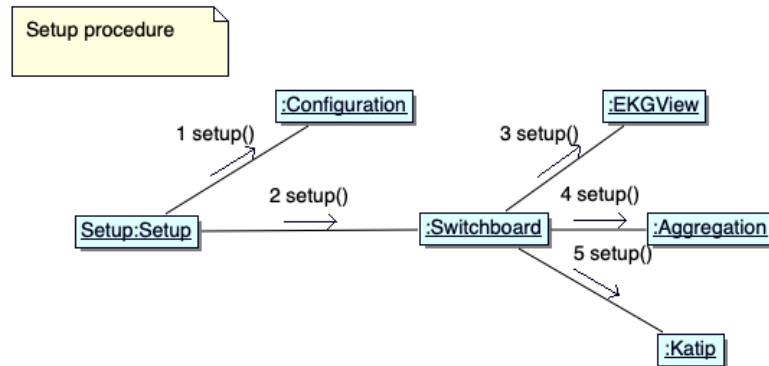


Figure 1.3: Setup procedure

**setupTrace**

Setup a new **Trace** (**Trace**) with either a given **Configuration** (**Configuration.Model**) or a *FilePath* to a configuration file.

```

setupTrace :: MonadIO m => Either FilePath Config.Configuration -> Text -> m (Trace m)
setupTrace (Left cfgFile) name = do
  c ← liftIO $ Config.setup cfgFile
  setupTrace_ c name
setupTrace (Right c) name = setupTrace_ c name
setupTrace_ :: MonadIO m => Config.Configuration -> Text -> m (Trace m)
setupTrace_ c name = do
  sb ← liftIO $ Switchboard.realize c
  sev ← liftIO $ Config.minSeverity c
  ctx ← liftIO $ newContext "" c sev sb
  subTrace name $ natTrace liftIO (ctx, Switchboard.mainTrace sb)

```

**withTrace**

```

withTrace :: MonadIO m => Config.Configuration -> Text -> (Trace m -> m t) -> m t
withTrace cfg name action = do
  logTrace ← setupTrace (Right cfg) name
  action logTrace

```

**newContext**

```

newContext :: LoggerName
  -> Config.Configuration
  -> Severity
  -> Switchboard.Switchboard
  -> IO TraceContext
newContext name cfg sev sb = do
  return $ TraceContext {
    loggerName = name
    , configuration = cfg
    , minSeverity = sev
    , tracetype = Neutral
    , shutdown = unrealize sb
  }

```

**1.4.6 Cardano.BM.Counters**

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

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

```
{-# LANGUAGE CPP #-}
```

```

# if defined (linux_HOST_OS)
# define LINUX
# endif
module Cardano.BM.Counters
(
    Platform.readCounters
    ,diffTimeObserved
    ,getMonoClock
) where
# ifdef LINUX
import qualified Cardano.BM.Counters.Linux as Platform
# else
import qualified Cardano.BM.Counters.Dummy as Platform
# endif

import Cardano.BM.Counters.Common (getMonoClock)
import Cardano.BM.Data.Aggregated (Measurable (..))
import Cardano.BM.Data.Counter

```

#### Calculate difference between clocks

```

diffTimeObserved :: CounterState → CounterState → Measurable
diffTimeObserved (CounterState id0 startCounters) (CounterState id1 endCounters) =
    let
        startTime = getMonotonicTime startCounters
        endTime = getMonotonicTime endCounters
    in
    if (id0 == id1)
    then endTime - startTime
    else error "these clocks are not from the same experiment"
where
    getMonotonicTime counters = case (filter isMonotonicClockCounter counters) of
        [(Counter MonotonicClockTime _ mus)] → mus
        _ → error "A time measurement is missing!"
    isMonotonicClockCounter :: Counter → Bool
    isMonotonicClockCounter = (MonotonicClockTime ==) ∘ cType

```

#### 1.4.7 Cardano.BM.Counters.Common

Common functions that serve *readCounters* on all platforms.

```

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

```



**Read monotonic clock**

```

getMonoClock :: IO [Counter]
getMonoClock = do
  t ← getMonotonicTimeNSec
  return [Counter MonotonicClockTime "monoclock" $ Microseconds (t `div` 1000)]

```

**Read GHC RTS statistics**

Read counters from GHC's *RTS* (runtime system). The values returned are as per the last GC (garbage collection) run.

```

readRTSStats :: IO [Counter]
readRTSStats = do
  iscollected ← GhcStats.getRTSStatsEnabled
  if iscollected
    then ghcstats
    else return []
  where
    ghcstats :: IO [Counter]
    ghcstats = do
      -- need to run GC?
      rts ← GhcStats.getRTSStats
      let getrts = ghcval rts
      return [getrts (Bytes ∘ fromIntegral ∘ GhcStats.allocated_bytes, "bytesAllocated")
            ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_live_bytes, "liveBytes")
            ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_large_objects_bytes, "largeBytes")
            ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_compact_bytes, "compactBytes")
            ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_slop_bytes, "slopBytes")
            ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_mem_in_use_bytes, "usedMemBytes")
            ,getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.gc_cpu_ns, "gcCpuNs")
            ,getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.gc_elapsed_ns, "gcElapsedNs")
            ,getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.cpu_ns, "cpuNs")
            ,getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.elapsed_ns, "elapsedNs")
            ,getrts (PureI ∘ toInteger ∘ GhcStats.gcs, "gcNum")
            ,getrts (PureI ∘ toInteger ∘ GhcStats.major_gcs, "gcMajorNum")
            ]
    ghcval :: GhcStats.RTSStats → ((GhcStats.RTSStats → Measurable), Text) → Counter
    ghcval s (f, n) = Counter RTSStats n $ (f s)

```

**1.4.8 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.

The only supported measurements are monotonic clock time and RTS statistics for now.

```

readCounters :: SubTrace → IO [Counter]
readCounters NoTrace = return []

```

```

readCounters Neutral          = return [ ]
readCounters (TeeTrace _)     = return [ ]
readCounters (FilterTrace _) = return [ ]
readCounters UntimedTrace    = return [ ]
readCounters DropOpening     = return [ ]
readCounters (ObservableTrace tts) = foldrM ( $\lambda(sel, fun) a \rightarrow$ 
    if any ( $\equiv sel$ ) tts
    then ( $fun \gg \lambda xs \rightarrow return \$ a ++ xs$ )
    else return a) [ ] selectors
where
    selectors = [ (MonotonicClock, getMonoClock)
        -- , (MemoryStats, readProcStatM)
        -- , (ProcessStats, readProcStats)
        -- , (IOStats, readProcIO)
        , (GhcRtsStats, readRTSStats)
        ]

```

#### 1.4.9 Cardano.BM.Counters.Linux

we have to expand the *readMemStats* function  
to read full data from *proc*

```

readCounters :: SubTrace  $\rightarrow$  IO [Counter]
readCounters NoTrace          = return [ ]
readCounters Neutral          = return [ ]
readCounters (TeeTrace _)     = return [ ]
readCounters (FilterTrace _) = return [ ]
readCounters UntimedTrace    = return [ ]
readCounters DropOpening     = return [ ]
readCounters (ObservableTrace tts) = foldrM ( $\lambda(sel, fun) a \rightarrow$ 
    if any ( $\equiv sel$ ) tts
    then ( $fun \gg \lambda xs \rightarrow return \$ a ++ xs$ )
    else return a) [ ] selectors
where
    selectors = [ (MonotonicClock, getMonoClock)
        , (MemoryStats, readProcStatM)
        , (ProcessStats, readProcStats)
        , (IOStats, readProcIO)
        ]

```

```

pathProc :: FilePath
pathProc = "/proc/"
pathProcStat :: ProcessID  $\rightarrow$  FilePath
pathProcStat pid = pathProc </> (show pid) </> "stat"
pathProcStatM :: ProcessID  $\rightarrow$  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)
  let ps = zip colnames ps0
  psUseful = filter (("unused" ≠) ∘ fst) ps
  return $ map (λ(n,i) → Counter MemoryCounter n (PureI i)) psUseful
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 in the kernel source file
fs/proc/array.c.
```

The fields, in order, with their proper scanf(3) format specifiers, are listed below. Whether or not certain of these fields display valid information is governed by a ptrace access mode PTRACE\_MODE\_READ\_FSCREDS | PTRACE\_MODE\_NOAUDIT check (refer to ptrace(2)). If the check denies access, then the field value is displayed as 0. The affected fields are indicated with the marking [PT].

- (1) pid %d  
The process ID.
- (2) comm %s  
The filename of the executable, in parentheses. This is visible whether or not the executable 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 contained in the combination of bits 31 to 20 and 7 to 0; the major device number is in bits 15 to 8.)
- (8) tpgid %d  
The ID of the foreground process group of the controlling terminal of the process.
- (9) flags %u  
The kernel flags word of the process. For bit meanings, see the PF\_\* defines in the Linux kernel source file include/linux/sched.h. Details depend on the kernel version.
- 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 required loading a memory page from disk.
- (11) cminflt %lu  
The number of minor faults that the process's waited-for children have made.
- (12) majflt %lu  
The number of major faults the process has made which have required loading a memory page from disk.
- (13) cmajflt %lu  
The number of major faults that the process's waited-for children have made.
- (14) utime %lu  
Amount of time that this process has been scheduled in user mode, measured in clock ticks (divide by sysconf(\_SC\_CLK\_TCK)). This includes guest time, guest\_time (time spent running a virtual CPU, see below), so that applications that are not aware of the guest time field do

- not lose that time from their calculations.
- (15) stime %lu  
Amount of time that this process has been scheduled in kernel mode, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`).
- (16) cutime %ld  
Amount of time that this process's waited-for children have been scheduled in user mode, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`). (See also `times(2)`.) This includes 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 scheduled in kernel mode, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`).
- (18) priority %ld  
(Explanation for Linux 2.6) For processes running a real-time scheduling policy (policy below; see `sched_setscheduler(2)`), this is the negated scheduling priority, minus one; that is, a number in the range -2 to -100, corresponding to real-time priorities 1 to 99. For processes running under a non-real-time scheduling policy, this is the raw nice value (`setpriority(2)`) as represented in the kernel. The kernel stores nice values as numbers in the range 0 (high) to 39 (low), corresponding to the user-visible nice range of -20 to 19.
- (19) nice %ld  
The nice value (see `setpriority(2)`), a value in the range 19 (low priority) to -20 (high priority).
- (20) num\_threads %ld  
Number of threads in this process (since Linux 2.6). Before kernel 2.6, this field was hard 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 due to an interval timer. Since kernel 2.6.17, this field is no longer maintained, and is hard coded as 0.
- (22) starttime %llu  
The time the process started after system boot. In kernels before Linux 2.6, this value was expressed in jiffies. Since Linux 2.6, the value is expressed in clock ticks (divide by `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. This is just the pages which count toward text, data, or stack space. This does not include pages which have not 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 description of `RLIMIT_RSS` in `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 stack page for the process.
- (30) kstkeip %lu [PT]  
The current EIP (instruction pointer).

- (31) signal %lu  
The bitmap of pending signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use /proc/[pid]/status instead.
- (32) blocked %lu  
The bitmap of blocked signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use /proc/[pid]/status instead.
- (33) sigignore %lu  
The bitmap of ignored signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use /proc/[pid]/status instead.
- (34) sigcatch %lu  
The bitmap of caught signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use /proc/[pid]/status instead.
- (35) wchan %lu [PT]  
This is the "channel" in which the process is waiting. It is the address of a location in the kernel where the process is sleeping. The corresponding symbolic name can be found in /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 processes scheduled under a real-time policy, or 0, for non-real-time processes (see sched\_setscheduler(2)).
- (41) policy %u (since Linux 2.5.19)  
Scheduling policy (see sched\_setscheduler(2)). Decode using the SCHED\_\* constants in 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 guest operating system), 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 clock ticks (divide by sysconf(\_SC\_CLK\_TCK)).
- (45) start\_data %lu (since Linux 3.3) [PT]  
Address above which program initialized and uninitialized (BSS) data are placed.
- (46) end\_data %lu (since Linux 3.3) [PT]  
Address below which program initialized and uninitialized (BSS) data are placed.
- (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)*

*let ps = zip colnames ps0*

*psUseful = filter ((*"unused"*  $\neq$ )  $\circ$  fst) ps*

*return \$ map ( $\lambda(n,i) \rightarrow$  Counter StatInfo *n* (PureI *i*)) psUseful*

*where*

*colnames :: [Text]*

*colnames = [ "pid", "unused", "unused", "ppid", "pgroup", "session", "ttynr", "tpgid", "flags", "minflt", "cminflt", "majflt", "cmajflt", "utime", "stime", "cutime", "cstime", "priority", "nice", "num", "itrealvalue", "starttime", "vsize", "rss", "rsslim", "startcode", "endcode", "startstack", "signal", "blocked", "sigignore", "sigcatch", "wchan", "nswap", "cnsnap", "exitcode", "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. This is simply the sum of bytes which this process passed to read(2) and similar system calls. It includes things such as terminal I/O and is unaffected by whether or not actual physical disk I/O was required (the read might have been satisfied from pagecache).

**wchar:** characters written

The number of bytes which this task has caused, or shall cause to be written to disk. Similar caveats apply here as with rchar.

**syscr:** read syscalls

Attempt to count the number of read I/O operations—that is, system calls such as read(2) and pread(2).

```

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

read_bytes: bytes read
    Attempt to count the number of bytes which this process really did cause to be fetched from the
    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 sent to the storage layer.

cancelled_write_bytes:
    The big inaccuracy here is truncate. If a process writes 1MB to a file and then deletes the
    file, it will in fact perform no writeout. But it will have been accounted as having caused 1MB
    of write. In other words: this field represents the number of bytes which this process caused
    to not happen, by truncating pagecache. A task can cause "negative" I/O too. If this task
    truncates some dirty pagecache, some I/O which another task has been accounted for (in its
    write_bytes) will not be happening.

```

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

Permission to access this file is governed by a ptrace access mode `PTRACE_MODE_READ_FSCREDS` check; see `ptrace(2)`.

```

readProcIO :: IO [Counter]
readProcIO = do
    pid ← getProcessID
    ps0 ← readProcList (pathProcIO pid)
    let ps = zip3 colnames ps0 units
    return $ map (\(n,i,u) → Counter IOCounter n (u i)) ps
where
    colnames :: [Text]
    colnames = [ "rchar", "wchar", "syscr", "syscw", "rbytes", "wbytes", "cxwbytes" ]
    units = [ Bytes ∘ fromInteger, Bytes ∘ fromInteger, PureI, PureI, Bytes ∘ fromInteger, Bytes ∘ fromInteger, Bytes ∘ fromInteger ]

```

#### 1.4.10 Cardano.BM.Data.Aggregated

##### Measurable

A **Measurable** may consist of different types of values. Time measurements are strict, so are *Bytes* which are externally measured. The real or integral numeric values are lazily linked, so we can decide later to drop them.

```

data Measurable = Microseconds {-# UNPACK #-} !Word64
    | Nanoseconds {-# UNPACK #-} !Word64
    | Seconds      {-# UNPACK #-} !Word64
    | Bytes        {-# UNPACK #-} !Word64
    | PureD        Double
    | PureI        Integer
    deriving (Eq, Ord, Generic, ToJSON)

```

**Measurable** can be transformed to an integral value.



```

getInteger :: Measurable → Integer
getInteger (Microseconds a) = toInteger a
getInteger (Nanoseconds a) = toInteger a
getInteger (Seconds a)      = toInteger a
getInteger (Bytes a)        = toInteger a
getInteger (PureI a)        = a
getInteger (PureD a)        = round a

```

**Measurable** can be transformed to a rational value.

```

getDouble :: Measurable → Double
getDouble (Microseconds a) = fromIntegral a
getDouble (Nanoseconds a)  = fromIntegral a
getDouble (Seconds a)      = fromIntegral a
getDouble (Bytes a)        = fromIntegral a
getDouble (PureI a)        = fromIntegral a
getDouble (PureD a)        = a

```

It is a numerical value, thus supports functions to operate on numbers.

```

instance Num Measurable where
  (+) (Microseconds a) (Microseconds b) = Microseconds (a + b)
  (+) (Nanoseconds a) (Nanoseconds b) = Nanoseconds (a + b)
  (+) (Seconds a)      (Seconds b)      = Seconds      (a + b)
  (+) (Bytes a)        (Bytes b)        = Bytes        (a + b)
  (+) (PureI a)        (PureI b)        = PureI        (a + b)
  (+) (PureD a)        (PureD b)        = PureD        (a + b)
  (+) _ _              = error "Trying to add values with different units"

  (*) (Microseconds a) (Microseconds b) = Microseconds (a * b)
  (*) (Nanoseconds a) (Nanoseconds b) = Nanoseconds (a * b)
  (*) (Seconds a)      (Seconds b)      = Seconds      (a * b)
  (*) (Bytes a)        (Bytes b)        = Bytes        (a * b)
  (*) (PureI a)        (PureI b)        = PureI        (a * b)
  (*) (PureD a)        (PureD b)        = PureD        (a * b)
  (*) _ _              = error "Trying to multiply values with different units"

  abs (Microseconds a) = Microseconds (abs a)
  abs (Nanoseconds a) = Nanoseconds (abs a)
  abs (Seconds a)      = Seconds      (abs a)
  abs (Bytes a)        = Bytes        (abs a)
  abs (PureI a)        = PureI        (abs a)
  abs (PureD a)        = PureD        (abs a)

  signum (Microseconds a) = Microseconds (signum a)
  signum (Nanoseconds a) = Nanoseconds (signum a)
  signum (Seconds a)      = Seconds      (signum a)
  signum (Bytes a)        = Bytes        (signum a)
  signum (PureI a)        = PureI        (signum a)
  signum (PureD a)        = PureD        (signum a)

  negate (Microseconds a) = Microseconds (negate a)

```

```

negate (Nanoseconds a) = Nanoseconds (negate a)
negate (Seconds a)     = Seconds      (negate a)
negate (Bytes a)       = Bytes        (negate a)
negate (PureI a)       = PureI        (negate a)
negate (PureD a)       = PureD        (negate a)
fromInteger = PureI

```

Pretty printing of **Measurable**.

**instance** Show **Measurable** **where**

```

show v@(Microseconds a) = show a -- ++ showUnits v
show v@(Nanoseconds a)  = show a -- ++ showUnits v
show v@(Seconds a)      = show a -- ++ showUnits v
show v@(Bytes a)        = show a -- ++ showUnits v
show v@(PureI a)        = show a -- ++ showUnits v
show v@(PureD a)        = show a -- ++ showUnits v

showUnits :: Measurable → String
showUnits (Microseconds _) = " s"
showUnits (Nanoseconds _)  = " ns"
showUnits (Seconds _)      = " s"
showUnits (Bytes _)        = " B"
showUnits (PureI _)        = ""
showUnits (PureD _)        = ""

-- show in S.I. units
showSI :: Measurable → String
showSI (Microseconds a) = show (fromFloatDigits ((fromIntegral a) / (1000000 :: Float))) ++
                             showUnits (Seconds a)
showSI (Nanoseconds a) = show (fromFloatDigits ((fromIntegral a) / (1000000000 :: Float))) ++
                             showUnits (Seconds a)
showSI v@(Seconds a)   = show a ++ showUnits v
showSI v@(Bytes a)     = show a ++ showUnits v
showSI v@(PureI a)     = show a ++ showUnits v
showSI v@(PureD a)     = show a ++ showUnits v

```

## Stats

A **Stats** statistics is strictly computed.

```

data BaseStats = BaseStats {
  fmin :: !Measurable,
  fmax :: !Measurable,
  fcount :: !Word64,
  fsum_A :: !Double,
  fsum_B :: !Double
} deriving (Generic, ToJSON, Show)

instance Eq BaseStats where
  (BaseStats mina maxa counta sumAa sumBa) == (BaseStats minb maxb countb sumAb sumBb) =

```

```

    mina ≡ minb ∧ maxa ≡ maxb ∧ counta ≡ countb ∧
    abs (sumAa - sumAb) < 1.0e-4 ∧
    abs (sumBa - sumBb) < 1.0e-4

```

```

data Stats = Stats {
  flast :: !Measurable,
  -- ftime :: !Measurable,
  fbasic :: !BaseStats,
  fdelta :: !BaseStats
} deriving (Eq, Generic, ToJSON, Show)

```

```

meanOfStats :: BaseStats → Double
meanOfStats = fsum_A

```

```

stdevOfStats :: BaseStats → Double
stdevOfStats s =
  if fcount s < 2
  then 0
  else sqrt $ (fsum_B s) / (fromInteger $ fromIntegral (fcount s) - 1)

```

**instance Semigroup Stats** disabled for the moment, because not needed.

We use a parallel algorithm to update the estimation of mean and variance from two sample statistics. (see [https://en.wikipedia.org/wiki/Algorithms\\_for\\_calculating\\_variance#Parallel\\_algorithm](https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Parallel_algorithm))

```

instance Semigroup Stats where
  (<>) a b = let counta = fcount a
              countb = fcount b
              newcount = counta + countb
              delta = fsum_A b - fsum_A a
              in
  Stats { flast = flast b -- right associative
        , fmin   = min (fmin a) (fmin b)
        , fmax   = max (fmax a) (fmax b)
        , fcount = newcount
        , fsum_A = fsum_A a + (delta / fromInteger newcount)
        , fsum_B = fsum_B a + fsum_B b + (delta * delta) * (fromInteger (counta * countb) / fromInteger newcount)
        }

```

```

stats2Text :: Stats → Text
stats2Text (Stats slast sbasic sdelta) =
  pack $
    "{ last=" ++ show slast ++
    ", basic-stats=" ++ showStats' (sbasic) ++
    ", delta-stats=" ++ showStats' (sdelta) ++
    " }"

```

```

where
  showStats' :: BaseStats → String

```

```
showStats' s =
  ", { min=" + show (fmin s) ++
  ", max=" + show (fmax s) ++
  ", mean=" + show (meanOfStats s) ++ showUnits (fmin s) ++
  ", std-dev=" + show (stdevOfStats s) ++
  ", count=" + show (fcount s) ++
  " }"
```

### Exponentially Weighted Moving Average (EWMA)

Following [https://en.wikipedia.org/wiki/Moving\\_average#Exponential\\_moving\\_average](https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average) we calculate the exponential moving average for a series of values  $Y_t$  according to:

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

```
data EWMA = EmptyEWMA {alpha :: Double}
  | EWMA {alpha :: Double
  , avg :: Measurable
  } deriving (Show, Eq, Generic, ToJSON)
```

### Aggregated

```
data Aggregated = AggregatedStats Stats
  | AggregatedEWMA EWMA
  deriving (Eq, Generic, ToJSON)
```

**instance Semigroup Aggregated** disabled for the moment, because not needed.

```
instance Semigroup Aggregated where
  (<>) (AggregatedStats a) (AggregatedStats b) =
    AggregatedStats (a <> b)
  (<>) _ _ = error "Cannot combine different objects"
```

```
singletonStats :: Measurable → Aggregated
```

```
singletonStats a =
```

```
  let stats = Stats {flast = a
    ,fbasic = BaseStats
      {fmin = a
      ,fmax = a
      ,fcount = 1
      ,fsum_A = getDouble a
      ,fsum_B = 0}
    ,fdelta = BaseStats
      {fmin = 0
```

```

    ,fmax = 0
    ,fcount = 0
    ,fsum_A = 0
    ,fsum_B = 0}
  }
in
  AggregatedStats stats

```

```

instance Show Aggregated where
  show (AggregatedStats astats) =
    "{ stats = " ++ show astats ++ " }"
  show (AggregatedEWMA a) = show a

```

#### 1.4.11 Cardano.BM.Data.Backend

##### Accepts a NamedLogItem

Instances of this type class accept a **NamedLogItem** and deal with it.

```

class IsEffectuator t where
  effectuate :: t → NamedLogItem → IO ()
  effectuatefrom :: forall s ◦ (IsEffectuator s) ⇒ t → NamedLogItem → s → IO ()
  default effectuatefrom :: forall s ◦ (IsEffectuator s) ⇒ t → NamedLogItem → s → IO ()
  effectuatefrom t nli _ = effectuate t nli

```

##### Declaration of a Backend

A backend is life-cycle managed, thus can be *realized* and *unrealized*.

```

class (IsEffectuator t) ⇒ IsBackend t where
  typeof    :: t → BackendKind
  realize    :: Configuration → IO t
  realizefrom :: forall s ◦ (IsEffectuator s) ⇒ Trace IO → s → IO t
  default realizefrom :: forall s ◦ (IsEffectuator s) ⇒ Trace IO → s → IO t
  realizefrom (ctx, _) _ = realize (configuration ctx)
  unrealize :: t → IO ()

```

##### Backend

This data structure for a backend defines its behaviour as an **IsEffectuator** when processing an incoming message, and as an **IsBackend** for unrealizing the backend.

```

data Backend = MkBackend
  { bEffectuate :: NamedLogItem → IO ()
  , bUnrealize :: IO ()
  }

```

### 1.4.12 Cardano.BM.Data.Configuration

Data structure to help parsing configuration files.

#### Representation

```

type Port = Int
data Representation = Representation
  { minSeverity    :: Severity
  , rotation      :: RotationParameters
  , setupScribes  :: [ScribeDefinition]
  , defaultScribes :: [(ScribeKind, Text)]
  , setupBackends :: [BackendKind]
  , defaultBackends :: [BackendKind]
  , hasEKG       :: Maybe Port
  , hasGUI       :: Maybe Port
  , options      :: HM.HashMap Text Object
  }
deriving (Generic, Show, ToJSON, FromJSON)

```

#### parseRepresentation

```

parseRepresentation :: FilePath → IO Representation
parseRepresentation fp = do
  repr :: Representation ← decodeFileThrow fp
  return $ implicit_fill_representation repr

```

after parsing the configuration representation we implicitly correct it.

```

implicit_fill_representation :: Representation → Representation
implicit_fill_representation =
  remove_ekgview_if_not_defined ∘
  filter_duplicates_from_backends ∘
  filter_duplicates_from_scribes ∘
  union_setup_and_usage_backends ∘
  add_ekgview_if_port_defined ∘
  add_katip_if_any_scribes
where
  filter_duplicates_from_backends r =
    r { setupBackends = mkUniq $ setupBackends r }
  filter_duplicates_from_scribes r =
    r { setupScribes = mkUniq $ setupScribes r }
  union_setup_and_usage_backends r =
    r { setupBackends = setupBackends r <> defaultBackends r }
  remove_ekgview_if_not_defined r =
    case hasEKG r of
      Nothing → r { defaultBackends = filter (λbk → bk ≠ EKGViewBK) (defaultBackends r)

```

```

    ,setupBackends = filter ( $\lambda bk \rightarrow bk \neq \text{EKGVViewBK}$ ) (setupBackends r)
  }
  Just _  $\rightarrow$  r
add_ekgview_if_port_defined r =
  case hasEKG r of
    Nothing  $\rightarrow$  r
    Just _  $\rightarrow$  r {setupBackends = setupBackends r <> [EKGVViewBK]}
add_katip_if_any_scribes r =
  if (any  $\neg$  [null $ setupScribes r, null $ defaultScribes r])
  then r {setupBackends = setupBackends r <> [KatipBK]}
  else r
mkUniq :: Ord a  $\Rightarrow$  [a]  $\rightarrow$  [a]
mkUniq = Set.toList  $\circ$  Set.fromList

```

### 1.4.13 Cardano.BM.Data.Counter

#### Counter

```

data Counter = Counter
  { cType :: CounterType
  , cName :: Text
  , cValue :: Measurable
  }
  deriving (Eq, Show, Generic, ToJSON)

data CounterType = MonotonicClockTime
  | MemoryCounter
  | StatInfo
  | IOCounter
  | CpuCounter
  | RTSSStats
  deriving (Eq, Show, Generic, ToJSON)

instance ToJSON Microsecond where
  toJSON = toJSON  $\circ$  toMicroseconds
  toEncoding = toEncoding  $\circ$  toMicroseconds

```

#### Names of counters

```

nameCounter :: Counter  $\rightarrow$  Text
nameCounter (Counter MonotonicClockTime _ _) = "Time-interval"
nameCounter (Counter MemoryCounter _ _) = "Mem"
nameCounter (Counter StatInfo _ _) = "Stat"
nameCounter (Counter IOCounter _ _) = "IO"
nameCounter (Counter CpuCounter _ _) = "Cpu"
nameCounter (Counter RTSSStats _ _) = "RTS"

```

## CounterState

```

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)

```

## Difference between counters

```

diffCounters :: [Counter] → [Counter] → [Counter]
diffCounters openings closings =
  getCountersDiff openings closings
where
  getCountersDiff :: [Counter]
    → [Counter]
    → [Counter]
  getCountersDiff as bs =
    let
      getName counter = nameCounter counter <> cName counter
      asNames = map getName as
      aPairs = zip asNames as
      bsNames = map getName bs
      bs' = zip bsNames bs
      bPairs = HM.fromList bs'
    in
      catMaybes $ (flip map) aPairs $ λ(name, Counter _ _ startValue) →
        case HM.lookup name bPairs of
          Nothing → Nothing
          Just counter → let endValue = cValue counter
                        in Just counter {cValue = endValue − startValue}

```

### 1.4.14 Cardano.BM.Data.LogItem

#### LoggerName

A **LoggerName** has currently type *Text*.

```

type LoggerName = Text

```



**NamedLogItem**

```
type NamedLogItem = LogNamed LogObject
```

**LogNamed**

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

```
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)
```

**Logging of outcomes with LogObject**

```
data LogObject = LogObject LOMeta LOContent
  deriving (Generic, Show, ToJSON)
```

Meta data for a **LogObject**:

```
data LOMeta = LOMeta {
  _timestamp :: {-# UNPACK #-} !UTCTime
  , _tid :: {-# UNPACK #-} !ThreadId
  }
  deriving (Show)
instance ToJSON LOMeta where
  toJSON (LOMeta _timestamp _tid) =
    object [ "timestamp" . = _timestamp, "tid" . = show _tid ]
mkLOMeta :: IO LOMeta
mkLOMeta =
  LOMeta < $ > getCurrentTime
    < * > myThreadId
```

Payload of a **LogObject**:

```
data LOContent = LogMessage LogItem
  | LogValue Text Measurable
  | ObserveOpen CounterState
  | ObserveDiff CounterState
  | ObserveClose CounterState
  | AggregatedMessage [(Text, Aggregated)]
  | KillPill
  deriving (Generic, Show, ToJSON)
```

## LogItem

TODO `liPayload :: ToObject`

```
data LogItem = LogItem
  { liSelection :: LogSelection
  , liSeverity  :: Severity
  , liPayload  :: Text -- TODO should become ToObject
  } deriving (Show, Generic, ToJSON)

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, FromJSON)
```

### 1.4.15 Cardano.BM.Data.Observable

#### ObservableInstance

```
data ObservableInstance = MonotonicClock
  | MemoryStats
  | ProcessStats
  | IOStats
  | GhcRtsStats
  deriving (Generic, Eq, Ord, Show, FromJSON, ToJSON, Read)
```

### 1.4.16 Cardano.BM.Data.Output

#### OutputKind

```
data OutputKind = TVarList (STM.TVar [LogObject])
  | TVarListNamed (STM.TVar [LogNamed LogObject])
  deriving (Eq)
```

#### ScribeKind

This identifies katip's scribes by type.

```
data ScribeKind = FileTextSK
  | FileJsonSK
  | StdoutSK
  | StderrSK
  deriving (Generic, Eq, Ord, Show, FromJSON, ToJSON)
```

### ScribeId

A scribe is identified by **ScribeKind** x *Filename*

```
type ScribeId = Text -- (ScribeKind :: Filename)
```

### ScribeDefinition

This identifies katip's scribes by type.

```
data ScribeDefinition = ScribeDefinition
  { scKind :: ScribeKind
  , scName :: Text
  , scRotation :: Maybe RotationParameters
  }
  deriving (Generic, Eq, Ord, Show, FromJSON, ToJSON)
```

#### 1.4.17 Cardano.BM.Data.Severity

##### Severity

The intended meaning of severity codes:

Debug *detailed information about values and decision flow* **Info** general information of events; progressing properly Notice *needs attention; something  $\rightarrow$  progressing properly* **Warning** may continue into an error condition if continued Error *unexpected set of event or condition occurred* **Critical** error condition causing degrade of operation Alert *a subsystem is no longer operating correctly, likely requires manual* at this point, the system can never progress without additional intervention

We were informed by the Syslog taxonomy: [https://en.wikipedia.org/wiki/Syslog#Severity\\_level](https://en.wikipedia.org/wiki/Syslog#Severity_level)

```
data Severity = Debug
  | Info
  | Notice
  | Warning
  | Error
  | Critical
  | Alert
  | Emergency
  deriving (Show, Eq, Ord, Generic, ToJSON, Read)

instance FromJSON Severity where
  parseJSON = withText "severity" $ \case
    "Debug"    → pure Debug
    "Info"     → pure Info
    "Notice"   → pure Notice
    "Warning"  → pure Warning
    "Error"    → pure Error
    "Critical" → pure Critical
    "Alert"    → pure Alert
    "Emergency" → pure Emergency
    _          → pure Info -- catch all
```

### 1.4.18 Cardano.BM.Data.SubTrace

#### SubTrace

```

data NameSelector = Exact Text | StartsWith Text | EndsWith Text | Contains Text
                  deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)
data DropName     = Drop NameSelector
                  deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)
data UnhideNames = Unhide [NameSelector]
                  deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)
data SubTrace = Neutral
              | UntimedTrace
              | NoTrace
              | TeeTrace LoggerName
              | FilterTrace [(DropName, UnhideNames)]
              | DropOpening
              | ObservableTrace [ObservableInstance]
              deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)

```

### 1.4.19 Cardano.BM.Data.Trace

#### Trace

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

```

type Trace m = (TraceContext, TraceNamed m)

```

#### TraceNamed

A **TraceNamed** is a specialized **Contravariant** of type **NamedLogItem**, a **LogNamed** with payload **LogObject**.

```

type TraceNamed m = BaseTrace m (NamedLogItem)

```

#### TraceContext

We keep the context's name and a reference to the **Configuration** in the **TraceContext**.

```

data TraceContext = TraceContext
  { loggerName :: LoggerName
  , configuration :: Configuration
  , tracetype   :: SubTrace
  , minSeverity :: Severity
  , shutdown    :: IO ()
  }

```

### 1.4.20 Cardano.BM.Configuration

see `Cardano.BM.Configuration.Model` for the implementation.

```
getOptionOrDefault :: CM.Configuration → Text → Text → IO (Text)
getOptionOrDefault cg name def = do
  opt ← CM.getOption cg name
  case opt of
    Nothing → return def
    Just o → return o
```

### 1.4.21 Cardano.BM.Configuration.Model

`Configuration.Model`

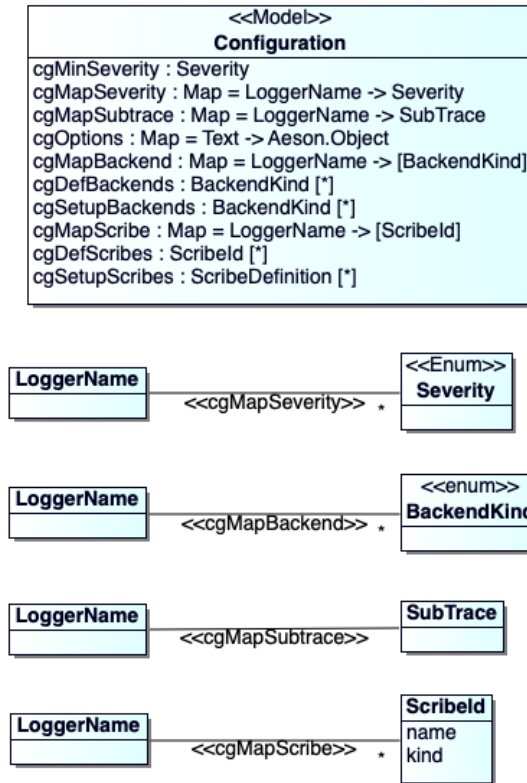


Figure 1.4: Configuration model

```
type ConfigurationMVar = MVar ConfigurationInternal
newtype Configuration = Configuration
  {getCG :: ConfigurationMVar}
-- Our internal state; see -"Configuration model"-
data ConfigurationInternal = ConfigurationInternal
```

```

{cgMinSeverity    ::Severity
-- minimum severity level of every object that will be output
,cgMapSeverity    ::HM.HashMap LoggerName Severity
-- severity filter per loggernaem
,cgMapSubtrace    ::HM.HashMap LoggerName SubTrace
-- type of trace per loggernaem
,cgOptions        ::HM.HashMap Text Object
-- options needed for tracing, logging and monitoring
,cgMapBackend     ::HM.HashMap LoggerName [BackendKind]
-- backends that will be used for the specific loggernaem
,cgDefBackendKs   ::[BackendKind]
-- backends that will be used if a set of backends for the
-- specific loggernaem is not set
,cgSetupBackends  ::[BackendKind]
-- backends to setup; every backend to be used must have
-- been declared here
,cgMapScribe      ::HM.HashMap LoggerName [ScribeId]
-- katip scribes that will be used for the specific loggernaem
,cgMapScribeCache ::HM.HashMap LoggerName [ScribeId]
-- map to cache info of the cgMapScribe
,cgDefScribes     ::[ScribeId]
-- katip scribes that will be used if a set of scribes for the
-- specific loggernaem is not set
,cgSetupScribes   ::[ScribeDefinition]
-- katip scribes to setup; every scribe to be used must have
-- been declared here
,cgMapAggregatedKind :: HM.HashMap LoggerName AggregatedKind
-- kind of Aggregated that will be used for the specific loggernaem
,cgDefAggregatedKind :: AggregatedKind
-- kind of Aggregated that will be used if a set of scribes for the
-- specific loggernaem is not set
,cgPortEKG        ::Int
-- port for EKG server
,cgPortGUI        ::Int
-- port for changes at runtime (NOT IMPLEMENTED YET)
} deriving (Show,Eq)

```

### Backends configured in the Switchboard

For a given context name return the list of backends configured, or, in case no such configuration exists, return the default backends.

```

getBackends :: Configuration → LoggerName → IO [BackendKind]
getBackends configuration name =
  withMVar (getCG configuration) $ \cg → do
    let outs = HM.lookup name (cgMapBackend cg)
    case outs of
      Nothing → do

```

```

        return (cgDefBackendKs cg)
    Just os → return os

getDefaultBackends :: Configuration → IO [BackendKind]
getDefaultBackends configuration =
    withMVar (getCG configuration) $ \cg → do
        return (cgDefBackendKs cg)

setDefaultBackends :: Configuration → [BackendKind] → IO ()
setDefaultBackends configuration bes = do
    cg ← takeMVar (getCG configuration)
    putMVar (getCG configuration) $ cg {cgDefBackendKs = bes}

setBackends :: Configuration → LoggerName → Maybe [BackendKind] → IO ()
setBackends configuration name be = do
    cg ← takeMVar (getCG configuration)
    putMVar (getCG configuration) $ cg {cgMapBackend = HM.alter (\_ → be) name (cgMapBackend cg)}

```

### Backends to be setup by the Switchboard

Defines the list of **Backends** that need to be setup by the **Switchboard**.

```

setSetupBackends :: Configuration → [BackendKind] → IO ()
setSetupBackends configuration bes = do
    cg ← takeMVar (getCG configuration)
    putMVar (getCG configuration) $ cg {cgSetupBackends = bes}

getSetupBackends :: Configuration → IO [BackendKind]
getSetupBackends configuration =
    withMVar (getCG configuration) $ \cg →
        return $ cgSetupBackends cg

```

### Scribes configured in the Log backend

For a given context name return the list of scribes to output to, or, in case no such configuration exists, return the default scribes to use.

```

getScribes :: Configuration → LoggerName → IO [ScribeId]
getScribes configuration name = do
    (updateCache, scribes) ← withMVar (getCG configuration) $ \cg → do
        let defs = cgDefScribes cg
        let mapScribe = cgMapScribe cg
        let find_s lname = case HM.lookup lname mapScribe of
            Nothing →
                case dropToDot lname of
                    Nothing → defs
                    Just lname' → find_s lname'
        Just os → os
    let outs = HM.lookup name (cgMapScribeCache cg)
    -- look if scribes are already cached

```

```

    return $ case outs of
      -- if no cached scribes found; search the appropriate scribes that
      -- they must inherit and update the cached map
      Nothing → (True, find_s name)
      Just os → (False, os)

    when updateCache $ setCachedScribes configuration name $ Just scribes
    return scribes
  where
    dropToDot :: Text → Maybe Text
    dropToDot ts = dropToDot' (breakOnEnd "." ts)
    dropToDot' (_, "") = Nothing
    dropToDot' (name', _) = Just $ dropWhileEnd (≡ ' . ') name'

  getCachedScribes :: Configuration → LoggerName → IO (Maybe [ScribeId])
  getCachedScribes configuration name =
    withMVar (getCG configuration) $ λcg → do
      return $ HM.lookup name $ cgMapScribeCache cg

  setScribes :: Configuration → LoggerName → Maybe [ScribeId] → IO ()
  setScribes configuration name scribes = do
    cg ← takeMVar (getCG configuration)
    putMVar (getCG configuration) $
      cg {cgMapScribe = HM.alter (\_ → scribes) name (cgMapScribe cg)}

  setCachedScribes :: Configuration → LoggerName → Maybe [ScribeId] → IO ()
  setCachedScribes configuration name scribes = do
    cg ← takeMVar (getCG configuration)
    putMVar (getCG configuration) $
      cg {cgMapScribeCache = HM.alter (\_ → scribes) name (cgMapScribeCache cg)}

  setDefaultScribes :: Configuration → [ScribeId] → IO ()
  setDefaultScribes configuration scs = do
    cg ← takeMVar (getCG configuration)
    putMVar (getCG configuration) $ cg {cgDefScribes = scs}

```

### Scribes to be setup in the **Log** backend

Defines the list of *Scribes* that need to be setup in the **Log** backend.

```

  setSetupScribes :: Configuration → [ScribeDefinition] → IO ()
  setSetupScribes configuration sds = do
    cg ← takeMVar (getCG configuration)
    putMVar (getCG configuration) $ cg {cgSetupScribes = sds}

  getSetupScribes :: Configuration → IO [ScribeDefinition]
  getSetupScribes configuration =
    withMVar (getCG configuration) $ λcg → do
      return $ cgSetupScribes cg

```



**AggregatedKind to define the type of measurement**

For a given context name return its *AggregatedKind* or in case no such configuration exists, return the default *AggregatedKind* to use.

```

getAggregatedKind :: Configuration → LoggerName → IO AggregatedKind
getAggregatedKind configuration name =
  withMVar (getCG configuration) $ \cg → do
    let outs = HM.lookup name (cgMapAggregatedKind cg)
    case outs of
      Nothing → do
        return (cgDefAggregatedKind cg)
      Just os → return $ os

setDefaultAggregatedKind :: Configuration → AggregatedKind → IO ()
setDefaultAggregatedKind configuration defAK = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgDefAggregatedKind = defAK}

setAggregatedKind :: Configuration → LoggerName → Maybe AggregatedKind → IO ()
setAggregatedKind configuration name ak = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgMapAggregatedKind = HM.alter (\_ → ak) name (cgMapAggregatedKi

```

**Access port numbers of EKG, GUI**

```

getEKGport :: Configuration → IO Int
getEKGport configuration =
  withMVar (getCG configuration) $ \cg → do
    return $ cgPortEKG cg

setEKGport :: Configuration → Int → IO ()
setEKGport configuration port = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgPortEKG = port}

getGUIport :: Configuration → IO Int
getGUIport configuration =
  withMVar (getCG configuration) $ \cg → do
    return $ cgPortGUI cg

setGUIport :: Configuration → Int → IO ()
setGUIport configuration port = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgPortGUI = port}

```

**Options**

```

getOption :: Configuration → Text → IO (Maybe Text)
getOption configuration name = do

```

```

withMVar (getCG configuration) $ \cg →
  case HM.lookup name (cgOptions cg) of
    Nothing → return Nothing
    Just o → return $ Just $ pack $ show o

```

### Global setting of minimum severity

```

minSeverity :: Configuration → IO Severity
minSeverity configuration = withMVar (getCG configuration) $ \cg →
  return $ cgMinSeverity cg
setMinSeverity :: Configuration → Severity → IO ()
setMinSeverity configuration sev = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgMinSeverity = sev}

```

### Relation of context name to minimum severity

```

inspectSeverity :: Configuration → Text → IO (Maybe Severity)
inspectSeverity configuration name = do
  withMVar (getCG configuration) $ \cg →
    return $ HM.lookup name (cgMapSeverity cg)
setSeverity :: Configuration → Text → Maybe Severity → IO ()
setSeverity configuration name sev = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgMapSeverity = HM.alter (\_ → sev) name (cgMapSeverity cg)}

```

### Relation of context name to SubTrace

A new context may contain a different type of **Trace**. The function **appendName** (Enter new named context) will look up the **SubTrace** for the context's name.

```

findSubTrace :: Configuration → Text → IO (Maybe SubTrace)
findSubTrace configuration name = do
  withMVar (getCG configuration) $ \cg →
    return $ HM.lookup name (cgMapSubtrace cg)
setSubTrace :: Configuration → Text → Maybe SubTrace → IO ()
setSubTrace configuration name trafo = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgMapSubtrace = HM.alter (\_ → trafo) name (cgMapSubtrace cg)}

```

### Parse configuration from file

Parse the configuration into an internal representation first. Then, fill in **Configuration** after refinement.

```

setup :: FilePath → IO Configuration
setup fp = do
  r ← R.parseRepresentation fp
  setupFromRepresentation r

setupFromRepresentation :: R.Representation → IO Configuration
setupFromRepresentation r = do
  cgreg ← newEmptyMVar
  let mapseverity = HM.lookup "mapSeverity" (R.options r)
      mapbackends = HM.lookup "mapBackends" (R.options r)
      mapsubtrace = HM.lookup "mapSubtrace" (R.options r)
      mapscribes = HM.lookup "mapScribes" (R.options r)
      mapAggregatedKinds = HM.lookup "mapAggregatedkinds" (R.options r)
      mapScribe = parseScribeMap mapscribes
  putMVar cgreg $ ConfigurationInternal
    { cgMinSeverity = R.minSeverity r
    , cgMapSeverity = parseSeverityMap mapseverity
    , cgMapSubtrace = parseSubtraceMap mapsubtrace
    , cgOptions = R.options r
    , cgMapBackend = parseBackendMap mapbackends
    , cgDefBackendKs = R.defaultBackends r
    , cgSetupBackends = R.setupBackends r
    , cgMapScribe = mapScribe
    , cgMapScribeCache = mapScribe
    , cgDefScribes = r_defaultScribes r
    , cgSetupScribes = R.setupScribes r
    , cgMapAggregatedKind = parseAggregatedKindMap mapAggregatedKinds
    , cgDefAggregatedKind = StatsAK
    , cgPortEKG = r_hasEKG r
    , cgPortGUI = r_hasGUI r
    }
  return $ Configuration cgreg

where
  parseSeverityMap :: Maybe (HM.HashMap Text Value) → HM.HashMap Text Severity
  parseSeverityMap Nothing = HM.empty
  parseSeverityMap (Just hmv) = HM.mapMaybe mkSeverity hmv
  mkSeverity (String s) = Just (read (unpack s) :: Severity)
  mkSeverity _ = Nothing

  parseBackendMap Nothing = HM.empty
  parseBackendMap (Just hmv) = HM.map mkBackends hmv
  mkBackends (Array bes) = catMaybes $ map mkBackend $ Vector.toList bes
  mkBackends _ = []
  mkBackend (String s) = Just (read (unpack s) :: BackendKind)
  mkBackend _ = Nothing

```

```

parseScribeMap Nothing = HM.empty
parseScribeMap (Just hmv) = HM.map mkScribes hmv
mkScribes (Array scs) = catMaybes $ map mkScribe $ Vector.toList scs
mkScribes (String s) = [(s :: ScribeId)]
mkScribes _ = []
mkScribe (String s) = Just (s :: ScribeId)
mkScribe _ = Nothing

parseSubtraceMap :: Maybe (HM.HashMap Text Value) → HM.HashMap Text SubTrace
parseSubtraceMap Nothing = HM.empty
parseSubtraceMap (Just hmv) = HM.mapMaybe mkSubtrace hmv
mkSubtrace (String s) = Just (read (unpack s) :: SubTrace)
mkSubtrace (Object hm) = mkSubtrace' (HM.lookup "tag" hm) (HM.lookup "contents" hm)
mkSubtrace _ = Nothing
mkSubtrace' Nothing _ = Nothing
mkSubtrace' _ Nothing = Nothing
mkSubtrace' (Just (String tag)) (Just (Array cs)) =
  if tag == "ObservableTrace"
  then Just $ ObservableTrace $ map (λ(String s) → (read (unpack s) :: ObservableInstance)) $ Vector.toList cs
  else Nothing
mkSubtrace' _ _ = Nothing

r_hasEKG repr = case (R.hasEKG repr) of
  Nothing → 0
  Just p → p
r_hasGUI repr = case (R.hasGUI repr) of
  Nothing → 0
  Just p → p
r_defaultScribes repr = map (λ(k,n) → pack (show k) <> " :: " <> n) (R.defaultScribes repr)

parseAggregatedKindMap Nothing = HM.empty
parseAggregatedKindMap (Just hmv) =
  let
    listv = HM.toList hmv
    mapAggregatedKind = HM.fromList $ catMaybes $ map mkAggregatedKind listv
  in
    mapAggregatedKind
mkAggregatedKind (name, String s) = Just (name, read (unpack s) :: AggregatedKind)
mkAggregatedKind _ = Nothing

```

### Setup empty configuration

```

empty :: IO Configuration
empty = do
  cgreg ← newEmptyMVar
  putMVar cgreg $ ConfigurationInternal Debug HM.empty HM.empty HM.empty HM.empty [] [] HM.empty H
  return $ Configuration cgreg

```

### 1.4.22 Cardano.BM.Output.Switchboard

#### Switchboard

```

type SwitchboardMVar = MVar SwitchboardInternal
newtype Switchboard = Switchboard
    {getSB :: SwitchboardMVar}
data SwitchboardInternal = SwitchboardInternal
    {sbQueue :: TBQ.TBQueue NamedLogItem
    ,sbDispatch :: Async.Async ()
    }

```

#### Trace that forwards to the Switchboard

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

```

mainTrace :: Switchboard → TraceNamed IO
mainTrace sb = BaseTrace.BaseTrace $ Op $ λlognamed → do
    effectuate sb lognamed

```

#### Process incoming messages

Incoming messages are put into the queue, and then processed by the dispatcher. The queue is initialized and the message dispatcher launched.

```

instance IsEffectuator Switchboard where
    effectuate switchboard item = do
        let writequeue :: TBQ.TBQueue NamedLogItem → NamedLogItem → IO ()
        writequeue q i = do
            nocapacity ← atomically $ TBQ.isFullTBQueue q
            if nocapacity
            then return ()
            else atomically $ TBQ.writeTBQueue q i
        withMVar (getSB switchboard) $ λsb →
            writequeue (sbQueue sb) item

```

#### Switchboard implements Backend functions

Switchboard is an **Declaration of a Backend**

```

instance IsBackend Switchboard where
    typeof _ = SwitchboardBK
    realize cfg =
        let spawnDispatcher
            :: Configuration
            → [(BackendKind, Backend)]
            → TBQ.TBQueue NamedLogItem

```

```

    → IO (Async.Async ())
spawnDispatcher config backends queue =
  let sendMessage nli befilter = do
    selectedBackends ← getBackends config (lnName nli)
    let selBEs = befilter selectedBackends
    forM_ backends $ λ(bek, be) →
      when (bek ∈ selBEs) (bEffectuate be $ nli)
  qProc = do
    nli ← atomically $ TBQ.readTBQueue queue
    case lnItem nli of
      LogObject _ KillPill →
        forM_ backends (λ(., be) → bUnrealize be)
      LogObject _ (AggregatedMessage _) → do
        sendMessage nli (filter (≠ AggregationBK))
        qProc
      _ → sendMessage nli id >> qProc
  in
    Async.async qProc
in do
  q ← atomically $ TBQ.newTBQueue 2048
  sbref ← newEmptyMVar
  putMVar sbref $ SwitchboardInternal q $ error "uninitialized dispatcher"
  let sb :: Switchboard = Switchboard sbref
  backends ← getSetupBackends cfg
  bs ← setupBackends backends cfg sb []
  dispatcher ← spawnDispatcher cfg bs q
  -- link the given Async to the current thread, such that if the Async
  -- raises an exception, that exception will be re-thrown in the current
  -- thread, wrapped in ExceptionInLinkedThread.
  Async.link dispatcher
  modifyMVar_ sbref $ λsbInternal → return $ sbInternal {sbDispatch = dispatcher}
  return sb
unrealize switchboard = do
  let clearMVar :: MVar a → IO ()
    clearMVar = void ∘ tryTakeMVar
  (dispatcher, queue) ← withMVar (getSB switchboard) (λsb → return (sbDispatch sb, sbQueue sb))
  -- send terminating item to the queue
  lo ← LogObject <$> mkLOMeta <*> pure KillPill
  atomically $ TBQ.writeTBQueue queue $ LogNamed "kill.switchboard" lo
  -- wait for the dispatcher to exit
  res ← Async.waitCatch dispatcher
  either throwM return res
  (clearMVar ∘ getSB) switchboard

```

### Realizing the backends according to configuration

```

setupBackends :: [BackendKind]
    → Configuration
    → Switchboard
    → [(BackendKind, Backend)]
    → IO [(BackendKind, Backend)]
setupBackends [ ] _ _ acc = return acc
setupBackends (bk : bes) c sb acc = do
    be' ← setupBackend' bk c sb
    setupBackends bes c sb ((bk, be') : acc)
setupBackend' :: BackendKind → Configuration → Switchboard → IO Backend
setupBackend' SwitchboardBK _ _ = error "cannot instantiate a further Switchboard"
setupBackend' EKGViewBK c _ = do
    be :: Cardano.BM.Output → EKGView.EKGView ← Cardano.BM.Output → EKGView.realize c
    return MkBackend
    { bEffectuate = Cardano.BM.Output → EKGView.effectuate be
    , bUnrealize = Cardano.BM.Output → EKGView.unrealize be
    }
setupBackend' AggregationBK c sb = do
    let trace = mainTrace sb
    ctx = TraceContext { loggerName = ""
    , configuration = c
    , minSeverity = Debug
    , tracetype = Neutral
    , shutdown = pure ()
    }
    be :: Cardano.BM.Output → Aggregation.Aggregation ← Cardano.BM.Output → Aggregation.realizefrom (ctx,
    return MkBackend
    { bEffectuate = Cardano.BM.Output → Aggregation.effectuate be
    , bUnrealize = Cardano.BM.Output → Aggregation.unrealize be
    }
setupBackend' KatipBK c _ = do
    be :: Cardano.BM.Output → Log.Log ← Cardano.BM.Output → Log.realize c
    return MkBackend
    { bEffectuate = Cardano.BM.Output → Log.effectuate be
    , bUnrealize = Cardano.BM.Output → Log.unrealize be
    }

```

#### 1.4.23 Cardano.BM.Output.Log

##### Internal representation

```

type LogMVar = MVar LogInternal
newtype Log = Log
    { getK :: LogMVar }

```

```
data LogInternal = LogInternal
  { kLogEnv :: K.LogEnv
  , configuration :: Config.Configuration }
```

**Log** implements *effectuate*

```
instance IsEffectuator Log where
  effectuate katip item = do
    c ← withMVar (getK katip) $ \k → return (configuration k)
    selscribes ← getScribes c (lnName item)
    forM_ selscribes $ \sc → passN sc katip item
```

**Log** implements backend functions

```
instance IsBackend Log where
  typeOf _ = KatipBK
  realize config = do
    let updateEnv :: K.LogEnv → IO UTCTime → K.LogEnv
        updateEnv le timer =
          le { K._logEnvTimer = timer, K._logEnvHost = "hostname" }
        register :: [ScribeDefinition] → K.LogEnv → IO K.LogEnv
        register [ ] le = return le
        register (defsc : dscs) le = do
          let kind = scKind defsc
              name = scName defsc
              name' = pack (show kind) <> " : " <> name
              scr ← createScribe kind name
              register dscs ≡ K.registerScribe name' scr scribeSettings le
          mockVersion :: Version
          mockVersion = Version [0,1,0,0] [ ]
          scribeSettings :: KC.ScribeSettings
          scribeSettings =
            let bufferSize = 5000 -- size of the queue (in log items)
            in
              KC.ScribeSettings bufferSize
            createScribe FileTextSK name = mkTextFileScribe (FileDescription $ unpack name) False
            createScribe FileJsonSK name = mkJsonFileScribe (FileDescription $ unpack name) False
            createScribe StdoutSK _ = mkStdoutScribe
            createScribe StderrSK _ = mkStderrScribe
          cfoKey ← Config.getOptionOrDefault config (pack "cfokey") (pack "<unknown>")
          le0 ← K.initLogEnv
            (K.Namespace [ "iohk" ])
            (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
    scribes ← getSetupScribes config
    le ← register scribes le1
    kref ← newEmptyMVar
    putMVar kref $ LogInternal le config
    return $ Log kref
unrealize katip = do
    le ← withMVar (getK katip) $ \k → return (kLogEnv k)
    void $ K.closeScribes le

example :: IO ()
example = do
    config ← Config.setup "from_some_path.yaml"
    k ← setup config
    passN (pack (show StdoutSK)) k $ LogNamed
        { lnName = "test"
        , lnItem = LogMessage $ LogItem
            { liSelection = Both
            , liSeverity = Info
            , liPayload = "Hello!"
            }
        }
    passN (pack (show StdoutSK)) k $ LogNamed
        { lnName = "test"
        , lnItem = LogValue "cpu-no" 1
        }

```

Needed instances for *katip*:

```

deriving instance K.ToObject LogObject
deriving instance K.ToObject LogItem
deriving instance K.ToObject (Maybe LOContent)
instance KC.LogItem LogObject where
    payloadKeys _ _ = KC.AllKeys
instance KC.LogItem LogItem where
    payloadKeys _ _ = KC.AllKeys
instance KC.LogItem (Maybe LOContent) where
    payloadKeys _ _ = KC.AllKeys

```

### Log.passN

The following function copies the **NamedLogItem** to the queues of all scribes that match on their name. Compare start of name of scribe to (*show backend* <> " : : "). This function is non-blocking.

```

passN :: Text → Log → NamedLogItem → IO ()
passN backend katip namedLogItem = do
    env ← withMVar (getK katip) $ \k → return (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 (LogObject lometa loitem) = lnItem namedLogItem
      let (sev, msg, payload) = case loitem of
        (LogMessage logItem) →
          (liSeverity logItem, liPayload logItem, Nothing)
        (ObserveDiff _) →
          let text = toStrict (encodeToLazyText loitem)
          in
            (Info, text, Just loitem)
        (ObserveOpen _) →
          let text = toStrict (encodeToLazyText loitem)
          in
            (Info, text, Just loitem)
        (ObserveClose _) →
          let text = toStrict (encodeToLazyText loitem)
          in
            (Info, text, Just loitem)
        (AggregatedMessage aggregated) →
          let text = T.concat $ (flip map) aggregated $ λ(name, agg) →
              "\n" <> name <> ": " <> pack (show agg)
          in
            (Info, text, Nothing)
        (LogValue name value) →
          (Debug, name <> " = " <> pack (show value), Nothing)
      KillPill →
        (Info, "Kill pill received!", Nothing)
    if (msg ≡ "") ∧ (isNothing payload)
    then return ()
    else do
      let threadIdText = KC.mkThreadIdText (tid lometa)
      let ns = lnName namedLogItem
      let itemTime = tstamp lometa
      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
      }

```

```

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

```

## Scribes

```

mkStdoutScribe :: IO K.Scribe
mkStdoutScribe = mkTextFileScribeH stdout True
mkStderrScribe :: IO K.Scribe
mkStderrScribe = mkTextFileScribeH stderr True
mkTextFileScribeH :: Handle → Bool → IO K.Scribe
mkTextFileScribeH handler color = do
    mkFileScribeH handler formatter color
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 ())
    → Bool
    → IO K.Scribe
mkFileScribeH h formatter colorize = do
    hSetBuffering h LineBuffering
    locklocal ← newMVar ()
    let logger :: forall a ◦ K.LogItem a ⇒ K.Item a → IO ()
        logger item = withMVar locklocal $ \_ →
            formatter h colorize K.V0 item
    pure $ K.Scribe logger (hClose h)
mkTextFileScribe :: FileDescription → Bool → IO K.Scribe
mkTextFileScribe fdesc colorize = do
    mkFileScribe fdesc formatter colorize
where
    formatter :: Handle → Bool → K.Verbosity → K.Item a → IO ()
    formatter hdl colorize' v' item =
        case KC._itemMessage item of
            K.LogStr "" →
                -- if message is empty do not output it
                return ()
            _ → do
                let tmsg = toLazyText $ formatItem colorize' v' item
                TIO.hPutStrLn hdl tmsg
mkJsonFileScribe :: FileDescription → Bool → IO K.Scribe
mkJsonFileScribe fdesc colorize = do
    mkFileScribe fdesc formatter colorize
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
    -- if a message is contained in item then only the
    -- message is printed and not the data
    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)

mkFileScribe
  :: FileDescription
  → (forall a ◦ K.LogItem a ⇒ Handle → Bool → K.Verbosity → K.Item a → IO ())
  → Bool
  → IO K.Scribe
mkFileScribe fdesc formatter colorize = do
  let prefixDir = prefixPath fdesc
  (createDirectoryIfMissing True prefixDir)
  'catchIO' (prtoutException ("cannot log prefix directory: " ++ prefixDir))
  let fpath = filePath fdesc
  h ← catchIO (openFile fpath WriteMode) $
    λe → do
      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 K.V0 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 Log.Severity
sev2klog :: Severity → K.Severity
sev2klog = λcase
    Debug    → K.DebugS
    Info     → K.InfoS
    Notice   → K.NoticeS
    Warning  → K.WarningS
    Error    → K.ErrorS
    Critical → K.CriticalS
    Alert    → K.AlertS
    Emergency → K.EmergencyS

data FileDescription = FileDescription {
    filePath :: !FilePath
    deriving (Show)
prefixPath :: FileDescription → FilePath
prefixPath = takeDirectory ∘ filePath
-- display message and stack trace of exception on stdout
prtoutException :: Exception e ⇒ String → e → IO ()
prtoutException msg e = do
    putStrLn msg
    putStrLn ("exception: " ++ displayException e)

```

### 1.4.24 Cardano.BM.Output.EKGView

#### Structure of EKGView

```

type EKGViewMVar = MVar EKGViewInternal
newtype EKGView = EKGView
    {getEV :: EKGViewMVar}
data EKGViewInternal = EKGViewInternal
    {evQueue :: TBQ.TBQueue (Maybe NamedLogItem)
    ,evLabels :: EKGViewMap
    ,evServer :: Server
    }

```

#### Relation from variable name to label handler

We keep the label handlers for later update in a *HashMap*.

```

type EKGViewMap = HM.HashMap Text Label.Label

```

#### Internal Trace

This is an internal **Trace**, named "#ekgview", which can be used to control the messages that are being displayed by EKG.

```

ekgTrace :: EKGView → Configuration → IO (Trace IO)
ekgTrace ekg c = do
    let trace = ekgTrace' ekg
    ctx = TraceContext {loggerName = ""
    ,configuration = c
    ,minSeverity = Debug
    ,tracetype = Neutral
    ,shutdown = pure ()
    }
    Trace.subTrace "#ekgview" (ctx, trace)
where
    ekgTrace' :: EKGView → TraceNamed IO
    ekgTrace' ekgview = BaseTrace.BaseTrace $ Op $ λ(LogNamed lognamed lo) → do
        let setlabel :: Text → Text → EKGViewInternal → IO (Maybe EKGViewInternal)
        setlabel name label ekg_i@(EKGViewInternal _ labels server) =
            case HM.lookup name labels of
                Nothing → do
                    ekghdl ← getLabel name server
                    Label.set ekghdl label
                    return $ Just $ ekg_i {evLabels = HM.insert name ekghdl labels}
                Just ekghdl → do
                    Label.set ekghdl label
                    return Nothing

```

```

update :: LogObject → LoggerName → EKGViewInternal → IO (Maybe EKGViewInternal)
update (LogObject _ (LogMessage logitem)) logname ekg_i =
    setlabel logname (liPayload logitem) ekg_i
update (LogObject _ (LogValue iname value)) logname ekg_i =
    let logname' = logname <> "." <> iname
    in
        setlabel logname' (pack $ show value) ekg_i
update _ _ _ = return Nothing
ekgup ← takeMVar (getEV ekgview)
let -- strip off some prefixes not necessary for display
    lognam1 = case stripPrefix "#ekgview.#aggregation." lognamed of
        Nothing → lognamed
        Just ln' → ln'
    logname = case stripPrefix "#ekgview." lognam1 of
        Nothing → lognam1
        Just ln' → ln'
upd ← update lo logname ekgup
case upd of
    Nothing → putMVar (getEV ekgview) ekgup
    Just ekgup' → putMVar (getEV ekgview) ekgup'

```

### EKG view is an effectuator

Function *effectuate* is called to pass in a **NamedLogItem** for display in EKG. If the log item is an *AggregatedStats* message, then all its constituents are put into the queue.

**instance IsEffectuator EKGView where**

```

effectuate ekgview item = do
    ekg ← readMVar (getEV ekgview)
    let queue a = do
        nocapacity ← atomically $ TBQ.isFullTBQueue (evQueue ekg)
        if nocapacity
        then return ()
        else atomically $ TBQ.writeTBQueue (evQueue ekg) (Just a)
    case (lnItem item) of
        (LogObject lometa (AggregatedMessage ags)) → liftIO $ do
            let logname = lnName item
            traceAgg :: [(Text, Aggregated)] → IO ()
            traceAgg [] = return ()
            traceAgg ((n, AggregatedEWMA ewma) : r) = do
                queue $ LogNamed (logname <> "." <> n) $ LogObject lometa (LogValue "avg" $ avg ewma)
            traceAgg r
            traceAgg ((n, AggregatedStats stats) : r) = do
                let statsname = logname <> "." <> n
                qbasestats s' nm = do
                    queue $ LogNamed nm $ LogObject lometa (LogValue "mean" (PureD $ meanOfStats s'))
                    queue $ LogNamed nm $ LogObject lometa (LogValue "min" $ fmin s')

```

```

        queue $ LogNamed nm $ LogObject lometa (LogValue "max" $ fmax s')
        queue $ LogNamed nm $ LogObject lometa (LogValue "count" $ PureI $ fromIntegral $ fcount s')
        queue $ LogNamed nm $ LogObject lometa (LogValue "stdev" (PureD $ stdevOfStats s'))
        queue $ LogNamed statsname $ LogObject lometa (LogValue "last" $ flast stats)
        qbasestats (fbasic stats) $ statsname <> ".basic"
        qbasestats (fdelta stats) $ statsname <> ".delta"
        traceAgg r
    traceAgg ags
    (LogObject _ (LogMessage _)) → queue item
    (LogObject _ (LogValue _ _)) → queue item
    _ → return ()

```

## **EKGView** implements **Backend** functions

**EKGView** is an **IsBackend**

**instance IsBackend EKGView where**

```

    typeOf _ = EKGViewBK
    realize config = do
        evref ← newEmptyMVar
        let ekgview = EKGView evref
        evport ← getEKGport config
        ehdl ← forkServer "127.0.0.1" evport
        ekghdl ← getLabel "iohk-monitoring version" ehdl
        Label.set ekghdl $ pack (showVersion version)
        ekgtrace ← ekgTrace ekgview config
        queue ← atomically $ TBQ.newTBQueue 512
        dispatcher ← spawnDispatcher queue ekgtrace
        -- link the given Async to the current thread, such that if the Async
        -- raises an exception, that exception will be re-thrown in the current
        -- thread, wrapped in ExceptionInLinkedThread.
        Async.link dispatcher
        putMVar evref $ EKGViewInternal
            { evLabels = HM.empty
            , evServer = ehdl
            , evQueue = queue
            }
        return ekgview
    unrealize ekgview = do
        ekg ← takeMVar $ getEV ekgview
        killThread $ serverThreadId $ evServer ekg

```

## Asynchronously reading log items from the queue and their processing

```

spawnDispatcher :: TBQ.TBQueue (Maybe NamedLogItem)
    → Trace.Trace IO

```



```

        → IO (Async.Async ())
spawnDispatcher evqueue trace =
  Async.async $ qProc
where
  qProc = do
    maybeItem ← atomically $ TBQ.readTBQueue evqueue
    case maybeItem of
      Just (LogNamed logname logvalue) → do
        trace' ← Trace.appendName logname trace
        Trace.traceNamedObject trace' logvalue
        qProc
      Nothing → return () -- stop here

```

### Interactive testing **EKGView**

```

test :: IO ()
test = do
  c ← Cardano.BM.Setup.setupTrace (Left "test/config.yaml") "ekg"
  ev ← Cardano.BM.Output ◦ EKGView.realize c
  effectuate ev $ LogNamed "test.questions" (LogValue "answer" 42)
  effectuate ev $ LogNamed "test.monitor023" (LogMessage (LogItem Public Warning "!!!! ALARM !!!!"))

```

#### 1.4.25 Cardano.BM.Output.Aggregation

##### Internal representation

```

type AggregationMVar = MVar AggregationInternal
newtype Aggregation = Aggregation
  { getAg :: AggregationMVar }
data AggregationInternal = AggregationInternal
  { agQueue :: TBQ.TBQueue (Maybe NamedLogItem)
  , agDispatch :: Async.Async ()
  }

```

##### Relation from context name to aggregated statistics

We keep the aggregated values (**Aggregated**) for a named context in a *HashMap*.

```

type AggregationMap = HM.HashMap Text AggregatedExpanded

```

##### Info for Aggregated operations

Apart from the **Aggregated** we keep some valuable info regarding to them; such as when was the last time it was sent.

```

type Timestamp = Word64
data AggregatedExpanded = AggregatedExpanded
  { aeAggregated :: !Aggregated
  , aeResetAfter :: !(Maybe Word64)
  , aeLastSent :: {-# UNPACK #-} !Timestamp
  }

```

### Aggregation implements *effectuate*

Aggregation is an **Accepts a NamedLogItem** Enter the log item into the **Aggregation** queue.

```

instance IsEffectuator Aggregation where
  effectuate agg item = do
    ag ← readMVar (getAg agg)
    nocapacity ← atomically $ TBQ.isFullTBQueue (agQueue ag)
    if nocapacity
    then return ()
    else atomically $! TBQ.writeTBQueue (agQueue ag) $ Just item

```

### Aggregation implements **Backend** functions

Aggregation is an **Declaration of a Backend**

```

instance IsBackend Aggregation where
  typeOf _ = AggregationBK
  realize _ = error "Aggregation cannot be instantiated by 'realize'"
  realizefrom trace0@(ctx, _) _ = do
    trace ← Trace.subTrace "#aggregation" trace0
    aggref ← newEmptyMVar
    aggregationQueue ← atomically $ TBQ.newTBQueue 2048
    dispatcher ← spawnDispatcher (configuration ctx) HM.empty aggregationQueue trace
    -- link the given Async to the current thread, such that if the Async
    -- raises an exception, that exception will be re-thrown in the current
    -- thread, wrapped in ExceptionInLinkedThread.
    Async.link dispatcher
    putMVar aggref $ AggregationInternal aggregationQueue dispatcher
    return $ Aggregation aggref
  unrealize aggregation = do
    let clearMVar :: MVar a → IO ()
        clearMVar = void ∘ tryTakeMVar
    (dispatcher, queue) ← withMVar (getAg aggregation) (λag →
      return (agDispatch ag, agQueue ag))
    -- send terminating item to the queue
    atomically $ TBQ.writeTBQueue queue Nothing
    -- wait for the dispatcher to exit
    res ← Async.waitCatch dispatcher

```

```
either throwM return res
(clearMVar ◦ getAg) aggregation
```

### Asynchronously reading log items from the queue and their processing

```
spawnDispatcher :: Configuration
                → AggregationMap
                → TBQ.TBQueue (Maybe NamedLogItem)
                → Trace.Trace IO
                → IO (Async.Async ())

spawnDispatcher conf aggMap aggregationQueue trace = Async.async $ qProc aggMap
where
  qProc aggregatedMap = do
    maybeItem ← atomically $ TBQ.readTBQueue aggregationQueue
    case maybeItem of
      Just (LogNamed logname lo@(LogObject lm _)) → do
        (updatedMap, aggregations) ← update lo logname aggregatedMap
        unless (null aggregations) $
          sendAggregated (LogObject lm (AggregatedMessage aggregations)) logname
        qProc updatedMap
      Nothing → return ()

  update :: LogObject
        → LoggerName
        → AggregationMap
        → IO (AggregationMap, [(Text, Aggregated)])

  update (LogObject _ (LogValue iname value)) logname agmap = do
    let fullname = logname <> " ." <> iname
    aggregated ←
      case HM.lookup fullname agmap of
        Nothing → do
          -- if Aggregated does not exist; initialize it.
          aggregatedKind ← getAggregatedKind conf fullname
          case aggregatedKind of
            StatsAK → return $ singletonStats value
            EwmaAK aEWMA → do
              let initEWMA = EmptyEWMA aEWMA
              return $ AggregatedEWMA $ ewma initEWMA value
          Just a → return $ updateAggregation value (aeAggregated a) (aeResetAfter a)
    now ← getMonotonicTimeNSec
    let aggregatedX = AggregatedExpanded {
      aeAggregated = aggregated
    , aeResetAfter = Nothing
    , aeLastSent = now
    }
    namedAggregated = [(iname, aeAggregated aggregatedX)]
    updatedMap = HM.alter (const $ Just $ aggregatedX) fullname agmap
```

```

-- use of HM.alter so that in future we can clear the Aggregated
-- by using as alter's arg a function which returns Nothing.
return (updatedMap,namedAggregated)
update (LogObject _ (ObserveDiff counterState)) logname agmap =
  updateCounters (csCounters counterState) (logname, "diff") agmap [ ]
update (LogObject _ (ObserveOpen counterState)) logname agmap =
  updateCounters (csCounters counterState) (logname, "open") agmap [ ]
update (LogObject _ (ObserveClose counterState)) logname agmap =
  updateCounters (csCounters counterState) (logname, "close") agmap [ ]
-- TODO for text messages aggregate on delta of timestamps
update _ _ agmap = return (agmap,[ ])
updateCounters :: [Counter]
  → (LoggerName,LoggerName)
  → AggregationMap
  → [(Text,Aggregated)]
  → IO (AggregationMap,[ (Text,Aggregated) ])
updateCounters [ ] _ aggrMap aggs = return $ (aggrMap,aggs)
updateCounters (counter : cs) (logname,msgname) aggrMap aggs = do
  let name = cName counter
      subname = msgname <> "." <> (nameCounter counter) <> "." <> name
      fullname = logname <> "." <> subname
      value = cValue counter
  aggregated ←
    case HM.lookup fullname aggrMap of
      -- if Aggregated does not exist; initialize it.
      Nothing → do
        aggregatedKind ← getAggregatedKind conf fullname
        case aggregatedKind of
          StatsAK → return $ singletonStats value
          EwmaAK aEWMA → do
            let initEWMA = EmptyEWMA aEWMA
            return $ AggregatedEWMA $ ewma initEWMA value
          Just a → return $ updateAggregation value (aeAggregated a) (aeResetAfter a)
  now ← getMonotonicTimeNSec
  let aggregatedX = AggregatedExpanded {
    aeAggregated = aggregated
    , aeResetAfter = Nothing
    , aeLastSent = now
  }
  namedAggregated = (subname,aggregated)
  updatedMap = HM.alter (const $ Just $ aggregatedX) fullname aggrMap
  updateCounters cs (logname,msgname) updatedMap (namedAggregated : aggs)
sendAggregated :: LogObject → Text → IO ()
sendAggregated aggregatedMsg@(LogObject _ (AggregatedMessage _)) logname = do
  -- enter the aggregated message into the Trace
  trace' ← Trace.appendName logname trace
  liftIO $ Trace.traceNamedObject trace' aggregatedMsg

```

```
-- ignore every other message that is not of type AggregatedMessage
sendAggregated _ _ = return ()
```

### Update aggregation

We distinguish an uninitialized from an already initialized aggregation. The latter is properly initialized.

We use Welford's online algorithm to update the estimation of mean and variance of the sample statistics. (see [https://en.wikipedia.org/wiki/Algorithms\\_for\\_calculating\\_variance#Welford's\\_Online](https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's_Online))

```
updateAggregation :: Measurable → Aggregated → Maybe Word64 → Aggregated
updateAggregation v (AggregatedStats s) resetAfter =
  let count = fcount (fbasic s)
      reset = maybe False (count ≥) resetAfter
  in
  if reset
  then
    singletonStats v
  else
    AggregatedStats $! Stats {flast = v
                             ,fbasic = updateBaseStats (count ≥ 1) v (fbasic s)
                             ,fdelta = updateBaseStats (count ≥ 2) (v - flast s) (fdelta s)
                             }
updateAggregation v (AggregatedEWMA e) _ = AggregatedEWMA $! ewma e v
updateBaseStats :: Bool → Measurable → BaseStats → BaseStats
updateBaseStats False _ s = s {fcount = fcount s + 1}
updateBaseStats True v s =
  let newcount = fcount s + 1
      newvalue = getDouble v
      delta = newvalue - fsum_A s
      dincr = (delta / fromIntegral newcount)
      delta2 = newvalue - fsum_A s - dincr
  in
  BaseStats {fmin = min (fmin s) v
            ,fmax = max (fmax s) v
            ,fcount = newcount
            ,fsum_A = fsum_A s + dincr
            ,fsum_B = fsum_B s + (delta * delta2)
            }
```

### Calculation of EWMA

Following [https://en.wikipedia.org/wiki/Moving\\_average#Exponential\\_moving\\_average](https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average) we calculate the exponential moving average for a series of values  $Y_t$  according to:

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

The pattern matching below ensures that the **EWMA** will start with the first value passed in, and will not change type, once determined.

```

ewma :: EWMA → Measurable → EWMA
ewma (EmptyEWMA a) v = EWMA a v
ewma (EWMA a s@(Microseconds _)) y@(Microseconds _) =
  EWMA a $ Microseconds $ round $ a * (getDouble y) + (1 - a) * (getDouble s)
ewma (EWMA a s@(Seconds _)) y@(Seconds _) =
  EWMA a $ Seconds $ round $ a * (getDouble y) + (1 - a) * (getDouble s)
ewma (EWMA a s@(Bytes _)) y@(Bytes _) =
  EWMA a $ Bytes $ round $ a * (getDouble y) + (1 - a) * (getDouble s)
ewma (EWMA a (PureI s)) (PureI y) =
  EWMA a $ PureI $ round $ a * (fromInteger y) + (1 - a) * (fromInteger s)
ewma (EWMA a (PureD s)) (PureD y) =
  EWMA a $ PureD $ a * y + (1 - a) * s
ewma _ _ = error "Cannot average on values of different type"

```

# Index

- Aggregated, 34
  - instance of Semigroup, 34
  - instance of Show, 35
- AggregatedExpanded, 63
- Aggregation, 63
- appendName, 15
- Backend, 35
- BaseTrace, 14
  - instance of Contravariant, 14
- Counter, 37
- Counters
  - Dummy
    - readCounters, 23
  - Linux
    - readCounters, 24
- CounterState, 38
- CounterType, 37
- diffCounters, 38
- diffTimeObserved, 22
- evalFilters, 16
- EWMA, 34
- ewma, 67
- getMonoClock, 23
- getOptionOrDefault, 43
- IsBackend, 35
- IsEffectuator, 35
- logAlert, 18
- logAlertP, 18
- logAlertS, 18
- logCritical, 18
- logCriticalP, 18
- logCriticalS, 18
- logDebug, 18
- logDebugP, 18
- logDebugS, 18
- logEmergency, 18
- logEmergencyP, 18
- logEmergencyS, 18
- logError, 18
- logErrorP, 18
- logErrorS, 18
- LoggerName, 38
- logInfo, 18
- logInfoP, 18
- logInfoS, 18
- LogItem, 40
  - liPayload, 40
  - liSelection, 40
  - liSeverity, 40
- LogNamed, 39
- logNotice, 18
- logNoticeP, 18
- logNoticeS, 18
- LogObject, 39
- LogSelection, 40
  - Both, 40
  - Private, 40
  - Public, 40
  - PublicUnsafe, 40
- logWarning, 18
- logWarningP, 18
- logWarningS, 18
- mainTrace, 51
- Measurable, 30
  - instance of Num, 31
  - instance of Show, 32
- nameCounter, 37
- NamedLogItem, 39
- natTrace, 15
- newContext, 21
- nominalTimeToMicroseconds, 22
- noTrace, 15
- ObservableInstance, 40

- GhcRtsStats, 40
- IOStats, 40
- MemoryStats, 40
- MonotonicClock, 40
- ProcessStats, 40
- OutputKind, 40
  - TVarList, 40
  - TVarListNamed, 40
- parseRepresentation, 36
- Port, 36
- readRTSStats, 23
- Representation, 36
- ScribeDefinition, 41
  - scKind, 41
  - scName, 41
  - scRotation, 41
- ScribeId, 41
- ScribeKind
  - FileJsonSK, 40
  - FileTextSK, 40
  - StderrSK, 40
  - StdoutSK, 40
- setupTrace, 21
- Severity, 41
  - Alert, 41
  - Critical, 41
  - Debug, 41
  - Emergency, 41
  - Error, 41
  - Info, 41
  - instance of FromJSON, 41
  - Notice, 41
  - Warning, 41
- singletonStats, 34
- Stats, 32
  - instance of Semigroup, 33
- stats2Text, 33
- stdoutTrace, 17
- SubTrace, 42
  - DropOpening, 42
  - FilterTrace, 42
    - NameOperator, 42
    - NameSelector, 42
  - Neutral, 42
  - NoTrace, 42
  - ObservableTrace, 42
  - TeeTrace, 42
  - UntimedTrace, 42
- subTrace, 19
- Switchboard, 51
  - instance of IsBackend, 51
  - instance of IsEffectuator, 51
  - setupBackends, 53
- Trace, 42
- traceConditionally, 18
- TraceContext, 42
  - configuration, 42
  - loggerName, 42
  - minSeverity, 42
  - shutdown, 42
  - tracetype, 42
- traceInTVar, 17
- traceInTVarIO, 17
- TraceNamed, 42
- traceNamedInTVarIO, 17
- traceNamedItem, 18
- traceNamedObject, 16
- traceWith, 14
- typeofTrace, 15
- updateAggregation, 67
- updateTracetype, 15
- withTrace, 21