ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ РФ
МГТУ им. Н.Э.Баумана

23 0102

Согласовано

_____ *Галкин В. А*
“____” _____ _____

Утверждаю

_____ *Галкин В. А*
“____” _____ _____

Курсовая работа по дисциплине «Сетевые технологии»

«Локальная безадаптерная сеть»
Листинг PowerCom

Студент 4 курса группы ИУ5-72
_____ *Гуща А. В*
“____” _____ _____

Студент 4 курса группы ИУ5-72
_____ *Нардид А. Н*
“____” _____ _____

Студент 4 курса группы ИУ5-72
_____ *Оганян Л. П*
“____” _____ _____

Москва
2013 г.

Содержание

1 MAIN.HS

```haskell
module Main (main) where

import Paths_PowerCom
import Application.Layer

import Control.Monad (forever)
import Control.Distributed.Process
import Control.Distributed.Process.Node
import Network.Transport.Chan
import System.Exit
import System.Environment

exitMsg :: (ProcessId, String) -> Process ()
exitMsg (_, msg) = case msg of
  "exit" -> liftIO exitSuccess
  _      -> return ()

main :: IO ()
main = do
  args <- getArgs

  t <- createTransport
  node <- newLocalNode t initRemoteTable
  gladeFile <- getDataFileName "views/gui.glade"

  runProcess node $ do
    rootId <- getSelfPid
    initApplicationLayer gladeFile (convertArgs args) rootId
    forever $ receiveWait [match exitMsg]

  where
    convertArgs args = case length args of
        2 -> Just (head args, args !! 1)
        _ -> Nothing
```

2 EVENT.HS

```haskell
module Event (
      Event
    , initEvent
    , tag
    , getTag
    , riseEvent
    , checkEvent
    ) where

import Data.IORef
```

```haskell
import Control.Distributed.Process

data Event a = Event (IORef Bool) (IORef a)

initEvent :: a -> IO (Event a)
initEvent val = do
    flagRef <- newIORef False
    valRef <- newIORef val
    return $ Event flagRef valRef

getTag :: Event a -> IO a
getTag (Event _ val) = readIORef val

tag :: Event a -> a -> IO (Event a)
tag (Event flag valRef) val = do
    writeIORef valRef val
    return $! Event flag valRef

riseEvent :: Event a -> IO(Event a)
riseEvent (Event flagRef val) = do
    writeIORef flagRef True
    return $! Event flagRef val

checkEvent :: Event a -> (a -> Process b) -> b -> Process b
checkEvent (Event flagRef valRef) f failVal = do
    flag <- liftIO $ readIORef flagRef
    if flag then do
        val <- liftIO $ do
            writeIORef flagRef False
            readIORef valRef
        f val
    else return failVal
```

3 UTILITY.HS

```haskell
module Utility (
      while
    , exitMsg
    ) where

import Control.Distributed.Process
import Control.Monad

exitMsg :: (ProcessId, String) -> Process Bool
exitMsg (_, msg) = case msg of
  "exit" -> return False
  _      -> return True

while :: Process Bool -> Process ()
while f = do
    val <- f
```

```
17      when val $ while f
```

4 APPLICATION

4.1 ChatView.hs

```
1  module Application.ChatView (
2        initChatTextView
3      , putUserMessage
4      , putInfoMessage
5      , putErrorMessage
6      , textViewGetAllText
7      , textViewSetText
8      ) where
9
10 import Graphics.UI.Gtk
11 import Data.Time
12 import Data.Functor
13 import System.Locale
14
15 putUserMessage :: TextView -> String -> String -> IO ()
16 putUserMessage textView username msg = do
17     timeStr <- formatTime defaultTimeLocale "%T" <$> getCurrentTime
18     buffer <- textViewGetBuffer textView
19     bufferAddStringWithTag buffer ("[" ++ timeStr ++ ": " ++ username
    ++ "]: ") "UsernameColor"
20     bufferAddStringWithTag buffer (msg++"\n") "MessageColor"
21
22     textViewScrollToEnd textView
23
24 putInfoMessage :: TextView -> String -> IO ()
25 putInfoMessage textView msg = do
26     buffer <- textViewGetBuffer textView
27     bufferAddStringWithTag buffer (msg++"\n") "InfoColor"
28
29     textViewScrollToEnd textView
30
31 putErrorMessage :: TextView -> String -> IO ()
32 putErrorMessage textView msg = do
33     buffer <- textViewGetBuffer textView
34     bufferAddStringWithTag buffer (msg++"\n") "ErrorColor"
35
36     textViewScrollToEnd textView
37
38 textViewScrollToEnd :: TextView -> IO ()
39 textViewScrollToEnd textView = do
40     buffer <- textViewGetBuffer textView
41     endIter <- textBufferGetEndIter buffer
42     textViewScrollToIter textView endIter 0.0 Nothing
43     return ()
44
```

```
45 bufferAddStringWithTag :: TextBuffer -> String -> String -> IO ()
46 bufferAddStringWithTag buffer string tagName =  do
47     oldEnd <- textBufferGetEndIter buffer
48     line <- textIterGetLine oldEnd
49     offset <- textIterGetLineOffset oldEnd
50 
51     textBufferInsert buffer oldEnd string
52 
53     newEnd <- textBufferGetEndIter buffer
54     newBegin <- textBufferGetIterAtLineOffset buffer line offset
55     textBufferApplyTagByName buffer tagName newBegin newEnd
56 
57     return ()
58 
59 textViewGetAllText :: TextView -> IO String
60 textViewGetAllText textView = do
61     buffer <- textViewGetBuffer textView
62     beginIter <- textBufferGetStartIter buffer
63     endIter <- textBufferGetEndIter buffer
64     textBufferGetText buffer beginIter endIter True
65 
66 bufferDeleteAllText :: TextBuffer -> IO ()
67 bufferDeleteAllText buffer = do
68     beginIter <- textBufferGetStartIter buffer
69     endIter <- textBufferGetEndIter buffer
70     textBufferDelete buffer beginIter endIter
71 
72 textViewSetText :: TextView -> String -> IO ()
73 textViewSetText textView text = do
74     buffer <- textViewGetBuffer textView
75     bufferDeleteAllText buffer
76     bufferAddStringWithTag buffer text "HistoryColor"
77 
78 initChatTextView :: Builder -> IO TextView
79 initChatTextView builder = do
80     textView <- builderGetObject builder castToTextView "MessageArea"
81     buffer <- textViewGetBuffer textView
82     tagTable <- textBufferGetTagTable buffer
83 
84     usernameColorTag <- textTagNew $ Just "UsernameColor"
85     usernameColorTag `set`
86         [ textTagBackground := "White"
87         , textTagForeground := "Dark Green"
88         ]
89     textTagTableAdd tagTable usernameColorTag
90 
91     messageColorTag <- textTagNew $ Just "MessageColor"
92     messageColorTag `set`
93         [ textTagBackground := "White"
94         , textTagForeground := "Dark Blue"
95         ]
96     textTagTableAdd tagTable messageColorTag
97 
98     errorColorTag <- textTagNew $ Just "ErrorColor"
```

```
99      errorColorTag `set`
100         [ textTagBackground := "White"
101         , textTagForeground := "Crimson"
102         ]
103     textTagTableAdd tagTable errorColorTag
104
105     infoColorTag <- textTagNew $ Just "InfoColor"
106     infoColorTag `set`
107         [ textTagBackground := "White"
108         , textTagForeground := "Cadet Blue"
109         ]
110     textTagTableAdd tagTable infoColorTag
111
112     historyColorTag <- textTagNew $ Just "HistoryColor"
113     historyColorTag `set`
114         [ textTagBackground := "White"
115         , textTagForeground := "Chocolate"
116         ]
117     textTagTableAdd tagTable historyColorTag
118
119     return textView
```

4.2 Gui.hs

```
1  module Application.Gui (
2        initGui
3      , runGui
4      ) where
5
6  import Graphics.UI.Gtk
7  import Application.OptionDialog
8  import Application.ChatView
9  import Application.UserList
10 import Application.Types
11 import Channel.Options
12
13 import Control.Monad.IO.Class (liftIO)
14 import Control.Concurrent
15 import Data.Functor
16 import Data.IORef
17 import Data.Foldable
18
19 createAboutDialog :: IO ()
20 createAboutDialog = do
21     dialog <- aboutDialogNew
22     set dialog
23         [ aboutDialogName      := "About application"
24         , aboutDialogVersion   := "1.1"
25         , aboutDialogCopyright := "Copyright 2013 Gushcha Anton,
    Nardid Anatoliy, Oganyan Levon"
26         , aboutDialogComments  := "Application for messaging within
    serial port."
```

```
27          , aboutDialogLicense    := Just license]
28      dialog `on` response $ const $ widgetHideAll dialog
29      widgetShowAll dialog
30
31  license :: String
32  license = "PowerCom is free software: you can redistribute it and/or
        modify\n\
33  \it under the terms of the GNU General Public License as published
        by\n\
34  \the Free Software Foundation, either version 3 of the License, or\n\
35  \(at your option) any later version.\n\
36  \\n\
37  \PowerCom is distributed in the hope that it will be useful,\n\
38  \but WITHOUT ANY WARRANTY; without even the implied warranty of\n\
39  \MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the\n\
40  \GNU General Public License for more details.\n\
41  \\n\
42  \You should have received a copy of the GNU General Public License\n\
43  \along with PowerCom.  If not, see <http://www.gnu.org/licenses/>."
44
45  saveAction :: IORef (Maybe String) -> GuiApi -> IO ()
46  saveAction lastSaveRef api = do
47      lastSave <- readIORef lastSaveRef
48      case lastSave of
49          Nothing -> saveAsAction lastSaveRef api
50          Just fileName -> saveChatToFile fileName api
51
52  saveAsAction :: IORef (Maybe String) -> GuiApi -> IO ()
53  saveAsAction lastSaveRef api = do
54      dialog <- newSaveDialog
55      withFileChooserDo dialog $ \s -> do
56          writeIORef lastSaveRef $ Just s
57          saveChatToFile s api
58      widgetDestroy dialog
59
60  openAction :: IORef (Maybe String) -> TextView -> IO ()
61  openAction lastSaveRef chatView = do
62      dialog <- newOpenDialog
63      withFileChooserDo dialog $ \s -> do
64          writeIORef lastSaveRef $ Just s
65          loadChatFromFile chatView s
66      widgetDestroy dialog
67
68  withFileChooserDo :: FileChooserDialog -> (String -> IO ()) -> IO ()
69  withFileChooserDo dialog action = do
70      dialResponse <- dialogRun dialog
71      case dialResponse of
72          ResponseOk -> do
73              newFileNameOpt <- fileChooserGetFilename dialog
74              forM_ newFileNameOpt action
75          _ -> return ()
76
77  newSaveDialog :: IO FileChooserDialog
78  newSaveDialog = fileChooserDialogNew Nothing Nothing
```

```
    FileChooserActionSave [("Save", ResponseOk), ("Cancel",
    ResponseCancel)]
79
80 newOpenDialog :: IO FileChooserDialog
81 newOpenDialog = fileChooserDialogNew Nothing Nothing
    FileChooserActionOpen [("Open", ResponseOk), ("Cancel",
    ResponseCancel)]
82
83 saveChatToFile :: FilePath -> GuiApi -> IO ()
84 saveChatToFile filename api = writeFile filename =<< getChatText api
85
86 loadChatFromFile :: TextView -> FilePath -> IO ()
87 loadChatFromFile textView fileName = textViewSetText textView =<<
    readFile fileName
88
89 initGui :: FilePath -> Maybe (String, String) -> GuiCallbacks -> IO
    (Window, ChannelOptions, GuiApi)
90 initGui gladeFile initArgs callbacks = do
91     initGUI
92     builder <- builderNew
93     builderAddFromFile builder gladeFile
94
95     -- Binding main window
96     mainWindow <- builderGetObject builder castToWindow "MainWindow"
97     onDestroy mainWindow mainQuit
98
99     -- Exit item
100    exitItem <- builderGetObject builder castToMenuItem "ExitItem"
101    exitItem `on` menuItemActivate $ mainQuit
102
103    -- Show about dialog
104    aboutItem <- builderGetObject builder castToMenuItem "AboutItem"
105    aboutItem `on` menuItemActivate $ createAboutDialog
106
107    -- OptionDialog
108    (optionsRef, setupOptions') <- setupOptionDialog builder
    callbacks initArgs
109    options <- readIORef optionsRef
110
111    -- TextView for messages
112    chatTextView <- initChatTextView builder
113
114    -- Send buffer
115    sendEntry <- builderGetObject builder castToEntry "SendEntry"
116
117    let sendBtnAction = do
118            msg <- entryGetText sendEntry
119            username <- userName <$> readIORef optionsRef
120            putUserMessage chatTextView username msg
121            sendMessageCallback callbacks msg
122            entrySetText sendEntry ""
123
124    sendEntry `on` keyPressEvent $ tryEvent $ do
125        "Return" <- eventKeyName
```

```
126         liftIO sendBtnAction
127
128     -- Send Button
129     sendButton <- builderGetObject builder castToButton "SendButton"
130     sendButton `on` buttonActivated $ sendBtnAction
131
132     -- Connect button
133     connectButton <- builderGetObject builder castToToolButton
    "ConnectButton"
134     onToolButtonClicked connectButton $ connectCallback callbacks
135
136     -- Disconnect button
137     disconnectButton <- builderGetObject builder castToToolButton
    "DisconnectButton"
138     onToolButtonClicked disconnectButton $ disconnectCallback
    callbacks
139
140     -- User list
141     (_, addUser', removeUser') <- initUserList builder (userName
    options)
142
143     let api = GuiApi {
144                 printMessage = putUserMessage     chatTextView
145               , printInfo    = putInfoMessage     chatTextView
146               , printError   = putErrorMessage    chatTextView
147               , setupOptions = setupOptions'
148               , getChatText  = textViewGetAllText chatTextView
149               , addUser      = addUser'
150               , removeUser   = removeUser'
151               }
152
153     -- save dialog
154     fileNameRef <- newIORef (Nothing :: Maybe String)
155     saveItem <- builderGetObject builder castToMenuItem "SaveItem"
156     saveItem `on` menuItemActivate $ saveAction fileNameRef api
157
158     -- save as dialog
159     saveAsItem <- builderGetObject builder castToMenuItem "SaveAsItem"
160     saveAsItem `on` menuItemActivate $ saveAsAction fileNameRef api
161
162     -- open dialog
163     openItem <- builderGetObject builder castToMenuItem "OpenItem"
164     openItem `on` menuItemActivate $ openAction fileNameRef
    chatTextView
165
166     return (mainWindow, options, api)
167
168 runGui :: Window -> IO ()
169 runGui mainWindow = do
170     -- Yielding GTK thread
171     timeoutAddFull (yield >> return True) priorityDefaultIdle 1
172     widgetShowAll mainWindow
```

```
173     mainGUI
```

4.3 Layer.hs

```
1  module Application.Layer (
2      initApplicationLayer
3      ) where
4
5  import Application.Gui
6  import Application.Types
7  import Channel.Layer
8  import Channel.Options
9  import Utility (while, exitMsg)
10 import Event
11
12 import Control.Distributed.Process
13 import Control.Monad (forever)
14 import Control.Concurrent (yield)
15
16 data AppEvents =
17     AppEvents
18     {
19       sendEvent           :: Event String
20     , connectEvent        :: Event ()
21     , disconnectEvent     :: Event ()
22     , optionChangedEvent :: Event (ChannelOptions, ChannelOptions)
23     }
24
25 initAppEvents :: IO AppEvents
26 initAppEvents = do
27     sendEvent'             <- initEvent ""
28     connectEvent'          <- initEvent ()
29     disconnectEvent'       <- initEvent ()
30     optionChangedEvent'    <- initEvent (defaultOptions,
   defaultOptions)
31
32     return
33         AppEvents
34         {
35           sendEvent           = sendEvent'
36         , connectEvent        = connectEvent'
37         , disconnectEvent     = disconnectEvent'
38         , optionChangedEvent = optionChangedEvent'
39         }
40
41 callbacks :: AppEvents -> GuiCallbacks
42 callbacks events =
43     GuiCallbacks {
44                    sendMessageCallback    = \msg -> do
45                     newEvent <- tag (sendEvent events) msg
46                     riseEvent newEvent
47                     return ()
```

```
48
49                  , connectCallback       = do
50                     riseEvent $ connectEvent events
51                     return ()
52
53                  , disconnectCallback    = do
54                     riseEvent $ disconnectEvent events
55                     return ()
56
57                  , optionChangedCallback = \opt oldopt -> do
58                     newEvent <- tag (optionChangedEvent events) (opt,
    oldopt)
59                     riseEvent newEvent
60                     return ()
61
62                  }
63
64 printUserMessage :: GuiApi -> (ProcessId, String, String, String) ->
    Process Bool
65 printUserMessage api (_, _, user, msg) = do
66     liftIO $ printMessage api user msg
67     return True
68
69 printInfoMessage :: GuiApi -> (ProcessId, String, String) -> Process
    Bool
70 printInfoMessage api (_, _, msg) = do
71     liftIO $ printInfo api msg
72     return True
73
74 printErrorMessage :: GuiApi -> (ProcessId, String, String) -> Process
    Bool
75 printErrorMessage api (_, _, msg) = do
76     liftIO $ printError api msg
77     return True
78
79 setupOptionsHandler :: GuiApi -> (ProcessId, String, ChannelOptions)
    -> Process Bool
80 setupOptionsHandler api (_, _, options) = do
81     liftIO $ setupOptions api options
82     return True
83
84 userConnectHandler :: GuiApi -> (ProcessId, String, String) ->
    Process Bool
85 userConnectHandler api (_, _, name) = do
86     liftIO $ addUser api name
87     return True
88
89 userDisconnectHandler :: GuiApi -> (ProcessId, String, String) ->
    Process Bool
90 userDisconnectHandler api (_, _, name) = do
91     liftIO $ removeUser api name
92     return True
93
94 initApplicationLayer :: FilePath -> Maybe (String, String) ->
```

```
    ProcessId -> Process ()
initApplicationLayer gladeFile args rootId = do
    spawnLocal $ do

      events <- liftIO initAppEvents
      (mainWindow, options, api) <- liftIO $ initGui gladeFile args $
    callbacks events

      thisId <- getSelfPid
      channelId <- initChannelLayer thisId options

      spawnLocal $ do
          liftIO $ runGui mainWindow
          mapM_ (`send` (thisId, "exit")) [thisId, channelId, rootId]

      spawnLocal $ forever $ do
        checkEvent (sendEvent events) (\s -> send channelId (thisId,
    "send", s)) ()
        checkEvent (connectEvent events) (\() -> send channelId
    (thisId, "connect")) ()
        checkEvent (disconnectEvent events) (\() -> send channelId
    (thisId, "disconnect")) ()
        checkEvent (optionChangedEvent events) (\(opt, oldopt) -> do
          liftIO $ removeUser api $ userName oldopt
          liftIO $ addUser api $ userName opt
          send channelId (thisId, "options", opt, oldopt)) ()
        liftIO yield

      while $ receiveWait [
                matchIf (\(_, com)       -> com == "exit")
    exitMsg
              , matchIf (\(_, com, _, _) -> com == "message")    $
    printUserMessage       api
              , matchIf (\(_, com, _)    -> com == "info")       $
    printInfoMessage       api
              , matchIf (\(_, com, _)    -> com == "error")      $
    printErrorMessage      api
              , matchIf (\(_, com, _)    -> com == "options")    $
    setupOptionsHandler    api
              , matchIf (\(_, com, _)    -> com == "connect")    $
    userConnectHandler     api
              , matchIf (\(_, com, _)    -> com == "disconnect") $
    userDisconnectHandler  api]

    return ()
```

4.4 OptionDialog.hs

```
module Application.OptionDialog (
      setupOptionDialog
    , defaultOptions
    ) where
```

```
5
6  import Graphics.UI.Gtk
7  import Application.Types
8  import System.Hardware.Serialport hiding (send)
9
10 import Control.Monad.IO.Class (liftIO)
11 import Control.Applicative
12 import Control.Monad
13
14 import Data.Word
15 import Data.List
16 import Data.IORef
17 import Data.Maybe
18
19
20 import Channel.Options
21 import Physical.Detector
22
23 -- |Fills combo with list of showable values and return function to
24 -- matching that values with combo elements
25 createEnumCombo :: (Eq a) => ComboBox          -- ^ Combo box to fill
26     -> (a -> String)                           -- ^ Function to map
   elem into string, show for instance
27     -> [a]                                     -- ^ List of values the
   combo be filled
28     -> IO (a -> Maybe Int)                     -- ^ Matching function
   to search values in the combo
29 createEnumCombo combo f descr = do
30     comboBoxSetModelText combo
31     mapM_ (comboBoxAppendText combo . f) descr
32     return $ \val -> elemIndex val descr
33
34 -- | Data type used to mapping options to option dialog combos indexes
35 data OptionMappings = OptionMappings
36     {
37       speedMapping     :: CommSpeed -> Maybe Int
38     , stopBitMapping   :: StopBits  -> Maybe Int
39     , parityMapping    :: Parity    -> Maybe Int
40     , portWordMapping  :: Word8     -> Maybe Int
41     }
42
43 defaultOptionsWithArgs :: Maybe (String, String) -> ChannelOptions
44 defaultOptionsWithArgs args = case args of
45     Nothing -> defaultOptions
46     Just (portname, username) -> defaultOptions { portName =
   portname, userName = username}
47
48 getOptionElements :: Builder -> IO (ComboBox, Entry, ComboBox,
   ComboBox, ComboBox, ComboBox)
49 getOptionElements builder = (,,,,,) <$>
50     getComboBox "PortNameCombo" <*>
51     getEntry "UserNameEntry" <*>
52     getComboBox "SpeedCombo" <*>
53     getComboBox "StopBitCombo" <*>
```

```
54      getComboBox "ParityBitCombo" <*>
55      getComboBox "WordBitCombo"
56      where
57          getEntry     = builderGetObject builder castToEntry
58          getComboBox  = builderGetObject builder castToComboBox
59
60
61  setupGuiOptions :: Builder -> OptionMappings -> ChannelOptions -> IO
        ChannelOptions
62  setupGuiOptions builder mappings options = do
63
        (_,userNameEntry,speedCombo,stopBitCombo,parityBitCombo,wordBitCombo)
        <- getOptionElements builder
64
65      --entrySetText portNameEntry $ portName options
66      entrySetText userNameEntry $ userName options
67      comboBoxSetActive speedCombo     $ fromMaybe 0 $ speedMapping
        mappings $ portSpeed options
68      comboBoxSetActive stopBitCombo   $ fromMaybe 0 $ stopBitMapping
        mappings $ portStopBits options
69      comboBoxSetActive parityBitCombo $ fromMaybe 0 $ parityMapping
        mappings $ portParityBits options
70      comboBoxSetActive wordBitCombo   $ fromMaybe 0 $ portWordMapping
        mappings $ portWordBits options
71
72      return options
73
74
75  collectOptions :: Builder -> IO ChannelOptions
76  collectOptions builder = do
77
        (portNameCombo,userNameEntry,speedCombo,stopBitCombo,parityBitCombo,wor
78          <- getOptionElements builder
79
80      portNameVal    <- getFromCombo portNameCombo
81      userNameVal    <- entryGetText userNameEntry
82      portSpeedVal   <- string2PortSpeed <$> getFromCombo speedCombo
83      stopBitVal     <- string2StopBit   <$> getFromCombo stopBitCombo
84      parityBitVal   <- string2ParityBit <$> getFromCombo parityBitCombo
85      wordBitVal     <- getWordBit       <$> getFromCombo wordBitCombo
86
87      return ChannelOptions
88          {
89            portName       = portNameVal
90          , userName       = userNameVal
91          , portSpeed      = portSpeedVal
92          , portStopBits   = stopBitVal
93          , portParityBits = parityBitVal
94          , portWordBits   = wordBitVal
95          }
96      where
97          getFromCombo :: ComboBox -> IO String
98          getFromCombo combo = do
99              maybeText <- comboBoxGetActiveText combo
```

```
100             case maybeText of
101                 Just str -> return str
102                 Nothing  -> return ""
103         getWordBit s = case s of
104             "" -> 7
105             _  -> read s :: Word8
106 
107 setupOptionDialog :: Builder -> GuiCallbacks -> Maybe (String, String) -> IO (IORef ChannelOptions, ChannelOptions -> IO ())
108 setupOptionDialog builder callbacks initArgs = do
109     optionDialog <- builderGetObject builder castToDialog "OptionDialog"
110     optionDialog `set` [windowDeletable := False]
111 
112     -- Combos
113     portNameCombo <- builderGetObject builder castToComboBox "PortNameCombo"
114     createEnumCombo portNameCombo id =<< getSerialPorts
115 
116     speedCombo <- builderGetObject builder castToComboBox "SpeedCombo"
117     speedMatch <- createEnumCombo speedCombo portSpeed2String
118         [CS110
119         ,CS300
120         ,CS600
121         ,CS1200
122         ,CS2400
123         ,CS4800
124         ,CS9600
125         ,CS19200
126         ,CS38400
127         ,CS57600
128         ,CS115200]
129 
130     stopBitCombo <- builderGetObject builder castToComboBox "StopBitCombo"
131     stopBitMatch <- createEnumCombo stopBitCombo stopBit2String [One,Two]
132 
133     parityBitCombo <- builderGetObject builder castToComboBox "ParityBitCombo"
134     parityBitMatch <- createEnumCombo parityBitCombo parityBit2String [Even, Odd, NoParity]
135 
136     wordBitCombo <- builderGetObject builder castToComboBox "WordBitCombo"
137     wordBitMatch <- createEnumCombo wordBitCombo show [7,8]
138 
139     let mappings = OptionMappings
140             {
141               speedMapping    = speedMatch
142             , stopBitMapping  = stopBitMatch
143             , parityMapping   = parityBitMatch
144             , portWordMapping = wordBitMatch
145             }
```

```
146     -- Setup options
147     initOptions <- setupGuiOptions builder mappings $
        defaultOptionsWithArgs initArgs
148     options <- newIORef initOptions
149
150     -- OptionDialog item
151     optionItem <- builderGetObject builder castToMenuItem "OptionItem"
152     optionItem `on` menuItemActivate $ widgetShowAll optionDialog
153
154     -- OptionDialog tool button
155     optionButton <- builderGetObject builder castToToolButton
        "OptionButton"
156     onToolButtonClicked optionButton $ widgetShowAll optionDialog
157
158     optionDialog `on` keyPressEvent $ tryEvent $ do
159         "Return" <- eventKeyName
160         liftIO $ dialogResponse optionDialog $ ResponseUser 1
161
162     optionDialog `on` keyPressEvent $ tryEvent $ do
163         "Escape" <- eventKeyName
164         liftIO $ dialogResponse optionDialog $ ResponseUser 2
165
166     optionDialog `on` response $ \respId -> do
167           case respId of
168             ResponseUser 1 -> do
169                 newOptions <- collectOptions builder
170                 oldOptions <- readIORef options
171                 writeIORef options newOptions
172                 optionChangedCallback callbacks newOptions oldOptions
173             ResponseUser 2 -> return ()
174             _ -> return ()
175           widgetHideAll optionDialog
176
177     -- OptionDialog
178     return (options, void . setupGuiOptions builder mappings)
```

4.5 Types.hs

```
1  module Application.Types (
2        GuiCallbacks(..)
3      , GuiApi(..)
4      ) where
5
6  import Channel.Options
7
8  data GuiCallbacks = GuiCallbacks {
9        sendMessageCallback    :: String -> IO ()
10     , connectCallback        :: IO ()
11     , disconnectCallback     :: IO ()
12     , optionChangedCallback :: ChannelOptions -> ChannelOptions -> IO
       ()
```

```
}

data GuiApi = GuiApi {
      printMessage :: String -> String -> IO ()
    , printInfo    :: String -> IO ()
    , printError   :: String -> IO ()
    , setupOptions :: ChannelOptions -> IO ()
    , getChatText  :: IO String
    , addUser      :: String -> IO ()
    , removeUser   :: String -> IO ()
}
```

4.6 UserList.hs

```
module Application.UserList (
      initUserList
    ) where

import Graphics.UI.Gtk
import Data.List (elemIndex)
import Control.Monad

defaultUserIcon :: String
defaultUserIcon =  "comotron-user"

addUserToList :: ListStore (String, String) -> String -> IO ()
addUserToList store name = do
    list <- listStoreToList store
    case elemIndex name $ map snd list of
        Just _  -> return ()
        Nothing -> void $ listStoreAppend store (defaultUserIcon,
   name)

removeUserFromList :: ListStore (String, String) -> String -> IO ()
removeUserFromList store name = do
    list <- listStoreToList store
    case elemIndex name $ map snd list of
        Just i  -> listStoreRemove store i
        Nothing -> return ()


initUserList :: Builder -> String -> IO (TreeView, String -> IO (),
   String -> IO ())
initUserList builder username = do
    treeView <- builderGetObject builder castToTreeView "UserListView"
    store <- listStoreNew [(defaultUserIcon, username)]

    treeModelSetColumn store (makeColumnIdString 0) fst
    treeModelSetColumn store (makeColumnIdString 1) snd
    treeViewSetModel treeView store

```

```
36      return (treeView, addUserToList store, removeUserFromList store)
```

5 CHANNEL

5.1 Buffer.hs

```
1  module Channel.Buffer (
2        MessageBuffer
3      , initMessageBuffer
4      , addMessagePart
5      , collectMessage
6      , clearBuffer
7      , isMessageReady
8      ) where
9
10 import Data.IORef
11 import Control.Distributed.Process
12 import Control.Monad
13
14 type MessageBuffer = IORef (String, [String], Int)
15
16 initMessageBuffer :: Process MessageBuffer
17 initMessageBuffer = liftIO $ newIORef ("", [], 0)
18
19 addMessagePart :: MessageBuffer -> String -> Process ()
20 addMessagePart buff s = liftIO $ do
21     (name, raw, n) <- readIORef buff
22     when (length raw < n) $ writeIORef buff (name, raw ++ [s], n)
23
24 isMessageReady :: MessageBuffer -> Process Bool
25 isMessageReady buff = liftIO $ do
26     (_, raw, n) <- readIORef buff
27     return $ length raw == n
28
29 collectMessage :: MessageBuffer -> Process (String, String)
30 collectMessage buff = liftIO $ do
31     (name, raw, _) <- readIORef buff
32     return (name, concat raw)
33
34 clearBuffer :: MessageBuffer -> String -> Int -> Process ()
35 clearBuffer buff name n = liftIO $ writeIORef buff (name, [], n)
```

5.2 Connection.hs

```
1  module Channel.Connection (
2        Connection
3      , initConnection
4      , closeConnection
```

```
5       , openConnection
6       , setRemoteUsername
7       , remoteUserName
8       , ifConnectedWithError
9       , ifConnected
10      , ifNotConnected
11      , connectHandler
12      , disconnectHandler
13      , sendFrameWithDisconnect
14      ) where
15
16  import Channel.Options
17  import Channel.Sending
18  import Channel.Miscs
19  import Channel.Frame
20
21  import Data.IORef
22  import Data.Functor
23  import Control.Monad
24  import Control.Applicative
25  import Control.Distributed.Process
26
27  -- | Connection is bool value with remote user name
28  type Connection = IORef (Bool, String)
29
30  initConnection :: Process Connection
31  initConnection = liftIO $ newIORef (False, "")
32
33  closeConnection :: Connection -> Process ()
34  closeConnection conn = liftIO $ do
35      (_, uname) <- readIORef conn
36      writeIORef conn (False, uname)
37
38  -- User name is sended with link frame
39  openConnection :: Connection -> Process ()
40  openConnection conn = liftIO $ do
41      (_, uname) <- readIORef conn
42      writeIORef conn (True, uname)
43
44  isConnected :: Connection -> Process Bool
45  isConnected conn = liftIO $ fst <$> readIORef conn
46
47  setRemoteUsername :: Connection -> String -> Process ()
48  setRemoteUsername conn uname = do
49      bool <- isConnected conn
50      liftIO $ writeIORef conn (bool, uname)
51
52  remoteUserName :: Connection -> Process String
53  remoteUserName conn = liftIO $ snd <$> readIORef conn
54
55  ifConnectedWithError :: Connection -> ProcessId -> Process () ->
        Process ()
56  ifConnectedWithError connRef errorTransitId action = do
57      connection <- isConnected connRef
```

```
58      thisId <- getSelfPid
59      if connection then action
60      else send errorTransitId (thisId, "error", "Connection is not
    established!")
61
62  ifConnected :: Connection -> Process () -> Process ()
63  ifConnected = withConnectionDo True
64
65  ifNotConnected :: Connection -> Process () -> Process ()
66  ifNotConnected = withConnectionDo False
67
68  withConnectionDo :: Bool -> Connection -> Process () -> Process ()
69  withConnectionDo state connRef action = do
70      connection <- isConnected connRef
71      when (connection == state) action
72
73  getIdConOpt :: InnerChannelOptions
74                  -> Connection -> Process (ProcessId, ChannelOptions,
    Bool)
75  getIdConOpt optionsRef conn = (,,) <$> getSelfPid <*> getOptions
    optionsRef <*> isConnected conn
76
77  connectHandler :: ProcessId -> Connection -> InnerChannelOptions ->
    (ProcessId, String) -> Process Bool
78  connectHandler physLayerId conn optionsRef (senderId, _) = do
79      (thisId, options, connection) <- getIdConOpt optionsRef conn
80      if connection then return True else do
81          informSender senderId "Connecting..."
82          send physLayerId (thisId, "reopen", options)
83          connResult <- expect :: Process Bool
84          if not connResult then return True else do
85              sendRes <- sendFrameWithAck physLayerId $ LinkFrame $
    userName options
86              if sendRes then openConnection conn >> return True
87              else do
88                  informSenderError senderId "Remote host is not
    answering!"
89                  return True
90
91
92  disconnectHandler :: ProcessId -> Connection -> InnerChannelOptions
    -> (ProcessId, String) -> Process Bool
93  disconnectHandler physLayerId conn optionsRef (senderId, _) = do
94      (_, options, connection) <- getIdConOpt optionsRef conn
95      when connection $ do
96          informSender senderId "Disconnecting..."
97          sendFrameWithDisconnect conn senderId physLayerId $
    UnlinkFrame $ userName options
98          closeConnection conn
99      return True
100
101 sendFrameWithDisconnect :: Connection -> ProcessId -> ProcessId ->
    Frame -> Process ()
102 sendFrameWithDisconnect conn transitId targetId frame =
```

```
103     disconnectOnFail transitId conn $ sendFrameWithAck targetId frame
104
105 disconnectOnFail :: ProcessId -> Connection -> Process Bool ->
    Process ()
106 disconnectOnFail transitId conn action = do
107     res <- action
108     unless res $ ifConnected conn $ do
109         uname <- remoteUserName conn
110         sendDisconnectUser transitId uname
111         informSenderError transitId "Remote host is not answering!
    Connection closed."
112         closeConnection conn
```

5.3 ConnectionChecker.hs

```
1 module Channel.ConnectionChecker (
2       spawnConnectionChecker
3     ) where
4
5 import Channel.Connection
6 import Channel.Frame
7
8 import Control.Distributed.Process
9 import Control.Monad (forever)
10 import Control.Concurrent (threadDelay)
11
12 checkDelay :: Int
13 checkDelay = 5000000
14
15 spawnConnectionChecker :: ProcessId -> ProcessId -> Connection ->
    Process ProcessId
16 spawnConnectionChecker chanLayerId appLayerId conn = spawnLocal $
    forever $ do
17     ifConnected conn $ send chanLayerId (appLayerId, "transit-frame",
    toByteString Upcheck)
18     liftIO $ threadDelay checkDelay
```

5.4 CyclicCode.hs

```
1 module Channel.CyclicCode (
2       codeCyclic
3     , decodeCyclic
4     , prop_codeDecodeEq
5     , prop_polyConverting
6     , prop_Word8BitCount
7     , prop_quotRemPoly
8     , prop_simpleCoding
9     , prop_fullCodingDecoding
10    , prop_falseWord4Coding
```

```
11      , prop_falseWord8Coding
12      ) where
13
14  import qualified Data.ByteString as BS
15  import qualified Data.ByteString.Char8 as CH
16  import Math.Polynomial
17  import Data.Bits
18  import Data.Sequence (foldrWithIndex, fromList)
19  import Data.Word
20  import Control.Monad
21  import Test.QuickCheck hiding ( (.&.) )
22
23  type Word4 = Word8 -- only for semantic concise
24  type Word7 = Word8
25
26  data Bit = Bit Bool
27      deriving Eq
28
29  instance Show Bit where
30      show (Bit val) = if val then "1" else "0"
31
32  instance Num Bit where
33      (+) (Bit a) (Bit b) = case (a, b) of
34          (True, True) -> Bit False
35          (_, True)    -> Bit True
36          (True, _)    -> Bit True
37          _            -> Bit False
38
39      (-) = (+)
40      (*) (Bit a) (Bit b) = case (a, b) of
41          (False, _)   -> Bit False
42          (_, False)   -> Bit False
43          _            -> Bit True
44
45      abs ba = ba
46      signum ba = ba
47      fromInteger int = Bit $ int > 0
48
49  instance Fractional Bit where
50      (/) ba _ = ba
51      fromRational = undefined
52
53  word8ToPoly :: Word8 -> Poly Bit
54  word8ToPoly wd = poly LE $ map
55      (Bit . testBit wd) [0 .. bitSize wd - 1]
56
57  polyToWord8 :: Poly Bit -> Word8
58  polyToWord8 = foldrWithIndex coeff2Bit 0 . fromList . polyCoeffs LE
59      where
60          coeff2Bit :: Int -> Bit -> Word8 -> Word8
61          coeff2Bit i (Bit b) acc = if b then acc `setBit` i else acc
62
63  codeCyclic :: BS.ByteString -> BS.ByteString
64  codeCyclic = BS.pack . concatMap (\(a,b) -> [a, b]) . map codeWord8 .
```

```
    BS.unpack
65
66 codeWord8 :: Word8 -> (Word7, Word7)
67 codeWord8 wd = (codeWord4 highWord, codeWord4 lowWord)
68     where highWord = (wd .&. 0xF0) `shiftR` 4
69           lowWord  = wd .&. 0x0F
70
71 codeWord4 :: Word4 -> Word7 -- n = 7 k = 4
72 codeWord4 wd = polyToWord8 finalPoly
73     where
74         polyGen     = poly BE [1,0,1,1]
75         wordPoly    = word8ToPoly wd
76         shiftedPoly = wordPoly `multPoly` poly BE [1, 0, 0, 0] -- (n
    - k) = 3
77         reminder    = shiftedPoly `remPoly` polyGen
78         finalPoly   = shiftedPoly `addPoly` reminder
79
80 decodeCyclic :: BS.ByteString -> Maybe BS.ByteString
81 decodeCyclic = mPack . mapM decodeWord8 . makePairs . BS.unpack
82     where
83         mPack = liftM BS.pack
84
85 makePairs :: [a] -> [(a, a)]
86 makePairs [] = []
87 makePairs (_:[]) = []
88 makePairs (x1:x2:xs) = (x1, x2) : makePairs xs
89
90 decodeWord8 :: (Word7, Word7) -> Maybe Word8
91 decodeWord8 (a, b) = mShiftL4 (decodeWord4 a) `mOr` decodeWord4 b
92     where
93         mShiftL4 = liftM $ flip shiftL 4
94         mOr = liftM2 (.|.)
95
96 decodeWord4 :: Word7 -> Maybe Word4
97 decodeWord4 wd = if syndrome == zero then Just finalWord else Nothing
98     where
99         polyGen     = poly BE [1,0,1,1]
100        wordPoly    = word8ToPoly wd
101        syndrome    = wordPoly `remPoly` polyGen
102        finalWord   = (wd `shiftR` 3) .&. 0x0F
103
104 -- Testing
105 prop_codeDecodeEq :: Word8 -> Bool
106 prop_codeDecodeEq wd = case decodeWord8 $ codeWord8 wd of
107     Nothing  -> False
108     Just val -> wd == val
109
110 prop_polyConverting :: Word8 -> Bool
111 prop_polyConverting wd = wd == polyToWord8 (word8ToPoly wd)
112
113 prop_Word8BitCount :: Word8 -> Bool
114 prop_Word8BitCount wd = bitSize wd == 8
115
```

```haskell
prop_quotRemPoly :: Word8 -> Word8 -> Bool
prop_quotRemPoly a b = (b == 0) || (newa == pa)
    where newa   = addPoly (multPoly q pb) r
          (q, r) = quotRemPoly pa pb
          pa = word8ToPoly a
          pb = word8ToPoly b

prop_simpleCoding :: Word8 -> Bool
prop_simpleCoding wd = case decodeWord4 $ codeWord4 cutedWd of
    Nothing -> False
    Just val -> val == cutedWd
    where cutedWd = wd .&. 0x0F

prop_fullCodingDecoding :: String -> Bool
prop_fullCodingDecoding s = case decodeCyclic $ codeCyclic bs of
    Nothing -> False
    Just val -> val == bs
    where bs = CH.pack s

newtype BitError = BitError Int
    deriving (Eq, Show)

instance Arbitrary BitError where
    arbitrary = oneof $ map (return . BitError) [0 .. 7]
    shrink _ = []

prop_falseWord4Coding :: Word8 -> BitError -> Bool
prop_falseWord4Coding wd (BitError i) = case decodeWord4 $
   complementBit (codeWord4 cutedWd) i of
    Nothing -> True
    Just _ -> False
    where cutedWd = wd .&. 0x0F

prop_falseWord8Coding :: Word8 -> BitError -> BitError -> Bool
prop_falseWord8Coding wd (BitError i1) (BitError i2) =
    case decodeWord8 (cwd1 `complementBit` i1, cwd2 `complementBit`
   i2) of
        Nothing -> True
        Just _ -> False
    where
        (cwd1, cwd2) = codeWord8 wd
```

5.5 Frame.hs

```haskell
module Channel.Frame (
      Frame(..)
    , FrameClass(..)
    , prop_toByteString
    ) where

import qualified Data.ByteString as BS
import qualified Data.ByteString.Lazy as BL
```

```haskell
import qualified Data.ByteString.UTF8 as UTF

import Data.Functor
import Data.Word

import Data.Binary.Strict.Get
import Data.Binary.Put

import Control.Monad
import Control.Applicative

import Test.QuickCheck

class (Eq a) => FrameClass a where
    toByteString :: a -> BS.ByteString
    fromByteString :: BS.ByteString -> (Either String a,
   BS.ByteString)

frameType :: Frame -> Word8
frameType frame = case frame of
                    InformationFrame _ _ -> 0x00
                    DataPartFrame _      -> 0x01
                    LinkFrame        _   -> 0x02
                    UnlinkFrame      _   -> 0x03
                    AckFrame             -> 0x04
                    RetFrame             -> 0x05
                    OptionFrame      _   -> 0x06
                    Upcheck              -> 0x07

data Frame = InformationFrame String Word32
           | DataPartFrame String
           | OptionFrame [(String, String)]
           | LinkFrame   String
           | UnlinkFrame String
           | AckFrame
           | RetFrame
           | Upcheck
           deriving (Show, Eq)

instance Arbitrary Frame where
    arbitrary = oneof [ InformationFrame <$> (arbitrary :: Gen
   String) <*> (arbitrary :: Gen Word32)
                      , DataPartFrame <$> (arbitrary :: Gen String)
                      , LinkFrame   <$> (arbitrary :: Gen String)
                      , UnlinkFrame <$> (arbitrary :: Gen String)
                      , return AckFrame
                      , return RetFrame
                      , OptionFrame <$> (arbitrary :: Gen [(String,
   String)])
                      , return Upcheck]

    shrink (OptionFrame os) = [OptionFrame nos | nos <- shrink os]
    shrink _ = []

```

```
60 -- TODO: Move to binary class instead of custom
61 {-instance Binary Frame where
62     put = put . toByteString
63     get = do
64         (res, _) <- liftM fromByteString
65         case res of
66             Right frame -> return frame
67             Left err -> error err-}
68
69 int2word :: Int -> Word32
70 int2word = fromInteger . toInteger
71
72 word2int :: Word32 -> Int
73 word2int = fromInteger . toInteger
74
75 instance FrameClass Frame where
76     toByteString frame = BS.concat . BL.toChunks $ runPut $ case
    frame of
77                              InformationFrame u n -> putBounded $
    putMarkedString u >> putWord32be n
78                              DataPartFrame s      -> putBounded $
    putMarkedString s
79                              LinkFrame   u        -> putBounded $
    putMarkedString u
80                              UnlinkFrame u        -> putBounded $
    putMarkedString u
81                              AckFrame             -> putShort
82                              RetFrame             -> putShort
83                              OptionFrame      os  -> putBounded $
    putListLength os >> putOptions os
84                              Upcheck              -> putShort
85                            where
86                              putBegin          = putWord8 (frameType
    frame)
87                              putShort          = putBegin
88                              putListLength     = putWord32be .
    int2word . length
89                              putBSLength       = putWord32be .
    int2word . BS.length
90                              putMarkedString s = let bs =
    UTF.fromString s in putBSLength bs >> putByteString bs
91                              putBounded      m = putBegin >> m
92                              putOptions        = mapM_ (\(key,value)
    -> putMarkedString key >> putMarkedString value)
93
94     fromByteString = runGet parseFrame
95                              where
96                                  parseFrame :: Get Frame
97                                  parseFrame = do
98                                      frameTypeId <- getWord8
99                                      case frameTypeId of
100                                         0x00 -> return
    InformationFrame `ap` parseMarkedString `ap` getWord32be
101                                         0x01 -> return DataPartFrame
```

```
    `ap` parseMarkedString
                                        0x02 -> return LinkFrame
    `ap` parseMarkedString
                                        0x03 -> return UnlinkFrame
    `ap` parseMarkedString
                                        0x04 -> return AckFrame
                                        0x05 -> return RetFrame
                                        0x06 -> return OptionFrame
    `ap` parseKeyValue
                                        0x07 -> return Upcheck
                                        _    -> fail "Unknown frame
    type!"
                                parseMarkedString = do
                                    len <- getWord32be
                                    body <- getByteString $ word2int
    len
                                    return $ UTF.toString body

parseKeyValue :: Get [(String, String)]
parseKeyValue =  do
    pairsCount <- getWord32be
    mapM parsePair [1..pairsCount]
    where
        parsePair :: a -> Get (String, String)
        parsePair _ = do
            keyCount <- getWord32be
            key <- getByteString $ word2int keyCount
            valueCount <- getWord32be
            value <- getByteString $ word2int valueCount
            return (UTF.toString key, UTF.toString value)

-- Testing
prop_toByteString :: Frame -> Bool
prop_toByteString f = case fst $ fromByteString $ toByteString f of
                        Left  _ -> False
                        Right v -> v == f
```

5.6 Layer.hs

```
module Channel.Layer (
    initChannelLayer
    ) where

import Channel.Options
import Channel.Frame
import Channel.Buffer
import Channel.Connection
import Channel.Miscs
import Channel.Processing
import Channel.ConnectionChecker

import Physical.Layer
```

```haskell
import Utility (while, exitMsg)

import Control.Distributed.Process

import qualified Data.ByteString as BS

sendMessageHandler :: ProcessId -> Connection -> InnerChannelOptions -> (ProcessId, String, String) -> Process Bool
sendMessageHandler physLayerId conn optionsRef (senderId, _, msg) = do
    options <- getOptions optionsRef
    ifConnectedWithError conn senderId $
        mapM_ (sendFrameWithDisconnect conn senderId physLayerId) $ frameBuffers $ userName options
    return True
    where
        frameBuffers :: String -> [Frame]
        frameBuffers uname = startFrame : dataFrames msg
            where
                lengthInFrame :: Int
                lengthInFrame = 200

                startFrame :: Frame
                startFrame = InformationFrame uname $ fromIntegral dataFramesCount

                dataFramesCount :: Int
                dataFramesCount = (length msg `quot` lengthInFrame) + 1

                dataFrames :: String -> [Frame]
                dataFrames [] = []
                dataFrames s =  (DataPartFrame $ take lengthInFrame s) : dataFrames (drop lengthInFrame s)

-- TODO: Move to sending Frame instead of bytestring
transitFrameHandler :: ProcessId -> Connection -> (ProcessId, String, BS.ByteString) -> Process Bool
transitFrameHandler physLayerId conn (senderId, _, framebs) = do
    let (res, _) = fromByteString framebs
    case res of
        Right frame -> ifConnected conn $ sendFrameWithDisconnect conn senderId physLayerId frame
        Left _ -> return ()
    return True

changeOptionsHandler :: ProcessId -> Connection -> InnerChannelOptions -> (ProcessId, String, ChannelOptions, ChannelOptions) -> Process Bool
changeOptionsHandler physLayerId conn optionsRef (senderId, _, options, oldOptions) = do
    thisId <- getSelfPid
    setOptions optionsRef options
    send physLayerId (thisId, "reopen", options)
    res <- expect :: Process Bool
```

```
58      if not res then do
59          informSenderError senderId "Failed to reopen port!"
60          closeConnection conn
61      else ifConnected conn $ do
62          informSender senderId "Changing options..."
63          sendFrameWithDisconnect conn senderId physLayerId $
64              if userName oldOptions == userName options then frame
65              else frameWithRemoteNames
66      return True
67      where
68          frame :: Frame
69          frame = OptionFrame optionPairs
70
71          frameWithRemoteNames :: Frame
72          frameWithRemoteNames = OptionFrame $ optionPairs ++
73              [("remoteNameNew", userName options)
74              ,("remoteNameOld", userName oldOptions)]
75
76          optionPairs :: [(String, String)]
77          optionPairs = getOptionPairs options
78              [ "portSpeed"
79              , "portStopBits"
80              , "portParityBits"
81              , "portWordBits"]
82
83  transitError :: ProcessId -> (ProcessId, String, String) -> Process
        Bool
84  transitError transitId (_, _, msg) = informSenderError transitId msg
        >> return True
85
86  transitInfo :: ProcessId -> (ProcessId, String, String) -> Process
        Bool
87  transitInfo transitId (_, _, msg) = informSender transitId msg >>
        return True
88
89  initChannelLayer :: ProcessId -> ChannelOptions -> Process ProcessId
90  initChannelLayer appLayer options = spawnLocal $ do
91      thisId <- getSelfPid
92      optionsRef    <- initInnerOptions options
93      connection    <- initConnection
94      messageBuffer <- initMessageBuffer
95      physLayerId   <- initPhysicalLayer options thisId
96      spawnConnectionChecker thisId appLayer connection
97      while $ receiveWait [
98          -- From connection checker
99            matchIf (\(_, com, _)    -> com == "transit-frame") $
        transitFrameHandler   physLayerId connection
100         -- From application layer
101         , matchIf (\(_, com)       -> com == "exit")         exitMsg
102         , matchIf (\(_, com, _)    -> com == "send")       $
        sendMessageHandler   physLayerId connection optionsRef
103         , matchIf (\(_, com)       -> com == "connect")    $
        connectHandler       physLayerId connection optionsRef
104         , matchIf (\(_, com)       -> com == "disconnect") $
```

```
    disconnectHandler    physLayerId connection optionsRef
105       , matchIf (\(_, com, _, _) -> com == "options")    $
    changeOptionsHandler physLayerId connection optionsRef
106       -- From physical layer
107       , matchIf (\(_, com, _)    -> com == "error")      $
    transitError         appLayer
108       , matchIf (\(_, com, _)    -> com == "info")       $
    transitInfo          appLayer
109       , matchIf (\(_, com, _)    -> com == "frame" || com ==
    "frame-acked")
110           $ receiveFrameHandler physLayerId appLayer messageBuffer
    connection optionsRef]
111
112   send physLayerId (thisId, "exit")
```

5.7 Miscs.hs

```
1  module Channel.Miscs (
2        informSender
3      , informSenderError
4      , sendMessage
5      , sendConnectUser
6      , sendDisconnectUser
7      , sendReopenPort
8      , sendUpdateOptions
9      ) where
10
11 import Channel.Options
12 import Control.Distributed.Process
13 import Data.Typeable
14
15 informSender :: ProcessId -> String -> Process ()
16 informSender = sendTyped1 "info"
17
18 informSenderError :: ProcessId -> String -> Process ()
19 informSenderError = sendTyped1 "error"
20
21 sendMessage :: ProcessId -> String -> String -> Process ()
22 sendMessage = sendTyped2 "message"
23
24 sendConnectUser :: ProcessId -> String -> Process ()
25 sendConnectUser = sendTyped1 "connect"
26
27 sendDisconnectUser :: ProcessId -> String -> Process ()
28 sendDisconnectUser = sendTyped1 "disconnect"
29
30 sendReopenPort :: ProcessId -> ChannelOptions -> Process ()
31 sendReopenPort = sendTyped1 "reopen"
32
33 sendUpdateOptions :: ProcessId -> ChannelOptions -> Process ()
34 sendUpdateOptions = sendTyped1 "options"
35
```

```
sendTyped1 :: (Binary a, Typeable a) => String -> ProcessId -> a ->
    Process ()
sendTyped1 msgType targetId msg = do
    thisId <- getSelfPid
    send targetId (thisId, msgType, msg)

sendTyped2 :: (Binary a, Typeable a, Binary b, Typeable b) =>
    String -> ProcessId -> a -> b -> Process ()
sendTyped2 msgType targetId msg1 msg2 = do
    thisId <- getSelfPid
    send targetId (thisId, msgType, msg1, msg2)
```

5.8 Options.hs

```
{-# Language StandaloneDeriving, DeriveDataTypeable #-}
module Channel.Options (
      InnerChannelOptions
    , initInnerOptions
    , getOptions
    , setOptions
    , ChannelOptions(..)
    , defaultOptions
    , Binary(..)
    , portSpeed2String
    , string2PortSpeed
    , stopBit2String
    , string2StopBit
    , parityBit2String
    , string2ParityBit
    , updateOptionsFromPairs
    , getOptionPairs
    ) where

import System.Hardware.Serialport

import Data.Binary (Binary(..))
import Data.Binary.Get

import Data.Word
import Data.Typeable
import Data.IORef
import Control.Distributed.Process

deriving instance Typeable CommSpeed
deriving instance Typeable StopBits
deriving instance Typeable Parity
deriving instance Typeable FlowControl
deriving instance Typeable SerialPortSettings

deriving instance Show Parity
deriving instance Show StopBits

```

```
39 deriving instance Eq CommSpeed
40 deriving instance Eq StopBits
41 deriving instance Eq Parity
42
43 type InnerChannelOptions = IORef ChannelOptions
44
45 initInnerOptions :: ChannelOptions -> Process InnerChannelOptions
46 initInnerOptions opt = liftIO $ newIORef opt
47
48 getOptions :: InnerChannelOptions -> Process ChannelOptions
49 getOptions inner = liftIO $ readIORef inner
50
51 setOptions :: InnerChannelOptions -> ChannelOptions -> Process ()
52 setOptions inner opt = liftIO $ writeIORef inner opt
53
54 data ChannelOptions =
55     ChannelOptions
56     {
57       portName       :: String
58     , userName       :: String
59     , portSpeed      :: CommSpeed
60     , portStopBits   :: StopBits
61     , portParityBits :: Parity
62     , portWordBits   :: Word8
63     }
64     deriving (Typeable, Show)
65
66 defaultOptions :: ChannelOptions
67 defaultOptions = ChannelOptions
68         {
69           portName       = "COM1"
70         , userName       = "Username"
71         , portSpeed      = CS2400
72         , portStopBits   = Two
73         , portParityBits = NoParity
74         , portWordBits   = 8
75         }
76
77 portSpeed2String :: CommSpeed -> String
78 portSpeed2String spd = case spd of
79     CS110    -> "110"
80     CS300    -> "300"
81     CS600    -> "600"
82     CS1200   -> "1200"
83     CS2400   -> "2400"
84     CS4800   -> "4800"
85     CS9600   -> "9600"
86     CS19200  -> "19200"
87     CS38400  -> "38400"
88     CS57600  -> "57600"
89     CS115200 -> "115200"
90
91 string2PortSpeed :: String -> CommSpeed
92 string2PortSpeed str = case str of
```

```
93      "110"    -> CS110
94      "300"    -> CS300
95      "600"    -> CS600
96      "1200"   -> CS1200
97      "2400"   -> CS2400
98      "4800"   -> CS4800
99      "9600"   -> CS9600
100     "19200"  -> CS19200
101     "38400"  -> CS38400
102     "57600"  -> CS57600
103     "115200" -> CS115200
104     _        -> CS2400
105
106 stopBit2String :: StopBits -> String
107 stopBit2String sp =  case sp of
108     One -> "One"
109     Two -> "Two"
110
111 string2StopBit :: String -> StopBits
112 string2StopBit s = case s of
113     "One" -> One
114     "Two" -> Two
115     _     -> One
116
117 parityBit2String :: Parity -> String
118 parityBit2String p = case p of
119     Even     -> "Even"
120     Odd      -> "Odd"
121     NoParity -> "No parity"
122
123 string2ParityBit :: String -> Parity
124 string2ParityBit s = case s of
125     "Even"      -> Even
126     "Odd"       -> Odd
127     "No parity" -> NoParity
128     _           -> NoParity
129
130 getOptionPairs :: ChannelOptions -> [String] -> [(String, String)]
131 getOptionPairs options = foldl getProperty []
132     where
133         getProperty :: [(String, String)] -> String -> [(String,
    String)]
134         getProperty acc prop = case prop of
135             "portName"       -> (prop, portName options) : acc
136             "userName"       -> (prop, userName options) : acc
137             "portSpeed"      -> (prop, portSpeed2String $ portSpeed
      options) : acc
138             "portStopBits"   -> (prop, stopBit2String   $
    portStopBits   options) : acc
139             "portParityBits" -> (prop, parityBit2String $
    portParityBits options) : acc
140             "portWordBits"   -> (prop, show $ portWordBits options) :
    acc
141             _                -> acc
```

```
142
143 updateOptionsFromPairs :: [(String, String)] -> ChannelOptions ->
    ChannelOptions
144 updateOptionsFromPairs pairs options = foldl setProperty options pairs
145     where
146         setProperty :: ChannelOptions -> (String, String) ->
    ChannelOptions
147         setProperty opt (name, val) = case name of
148             "portName"       -> opt { portName = val }
149             "userName"       -> opt { userName = val }
150             "portSpeed"      -> opt { portSpeed = string2PortSpeed
    val }
151             "portStopBits"   -> opt { portStopBits = string2StopBit
    val }
152             "portParityBits" -> opt { portParityBits =
    string2ParityBit val }
153             "portWordBits"   -> opt { portWordBits = read val }
154             _                -> opt
155
156
157 instance Binary CommSpeed where
158     put sp = case sp of
159         CS110    -> put (0 :: Word8)
160         CS300    -> put (1 :: Word8)
161         CS600    -> put (2 :: Word8)
162         CS1200   -> put (3 :: Word8)
163         CS2400   -> put (4 :: Word8)
164         CS4800   -> put (5 :: Word8)
165         CS9600   -> put (6 :: Word8)
166         CS19200  -> put (7 :: Word8)
167         CS38400  -> put (8 :: Word8)
168         CS57600  -> put (9 :: Word8)
169         CS115200 -> put (10 :: Word8)
170
171     get = do
172         sp <- getWord8
173         case sp of
174             0  -> return CS110
175             1  -> return CS300
176             2  -> return CS600
177             3  -> return CS1200
178             4  -> return CS2400
179             5  -> return CS4800
180             6  -> return CS9600
181             7  -> return CS19200
182             8  -> return CS38400
183             9  -> return CS57600
184             10 -> return CS115200
185             _  -> return CS2400
186
187 instance Binary StopBits where
188     put sb = case sb of
189         One -> put (0 :: Word8)
190         Two -> put (1 :: Word8)
```

```
191
192     get = do
193         sb <- getWord8
194         case sb of
195             0 -> return One
196             1 -> return Two
197             _ -> return One
198
199 instance Binary Parity where
200     put pr = case pr of
201         Even     -> put (0 :: Word8)
202         Odd      -> put (1 :: Word8)
203         NoParity -> put (2 :: Word8)
204
205     get = do
206         pr <- getWord8
207         case pr of
208             0 -> return Even
209             1 -> return Odd
210             2 -> return NoParity
211             _ -> return NoParity
212
213 instance Binary FlowControl where
214     put fc = case fc of
215         Software      -> put (0 :: Word8)
216         NoFlowControl -> put (1 :: Word8)
217
218     get = do
219         fc <- getWord8
220         case fc of
221             0 -> return Software
222             1 -> return NoFlowControl
223             _ -> return NoFlowControl
224
225 instance Binary SerialPortSettings where
226     put (SerialPortSettings speed wordBits stopbits parityBits
    flowCnt t) = do
227         put speed
228         put wordBits
229         put stopbits
230         put parityBits
231         put flowCnt
232         put t
233
234     get = do
235         speed       <- get :: Get CommSpeed
236         wordBits    <- get :: Get Word8
237         stopbits    <- get :: Get StopBits
238         parityBits  <- get :: Get Parity
239         flowCnt     <- get :: Get FlowControl
240         t           <- get :: Get Int
241         return $ SerialPortSettings speed wordBits stopbits
    parityBits flowCnt t
242
```

```
243 instance Binary ChannelOptions where
244     put o = do
245         put (portName o)
246         put (userName o)
247         put (portSpeed o)
248         put (portStopBits o)
249         put (portParityBits o)
250         put (portWordBits o)
251
252     get = do
253         uname <- get :: Get String
254         pname <- get :: Get String
255         speed <- get :: Get CommSpeed
256         stbit <- get :: Get StopBits
257         party <- get :: Get Parity
258         wordb <- get :: Get Word8
259         return ChannelOptions
260                {
261                  portName = uname
262                , userName = pname
263                , portSpeed = speed
264                , portStopBits = stbit
265                , portParityBits = party
266                , portWordBits = wordb
267                }
```

5.9 Processing.hs

```
1  module Channel.Processing (
2        receiveFrameHandler
3     ) where
4
5  import Channel.Buffer
6  import Channel.Connection
7  import Channel.Frame
8  import Channel.Options
9  import Channel.Miscs
10 import Channel.Sending
11
12 import Control.Distributed.Process
13 import Control.Applicative
14 import Control.Monad
15
16 import qualified Data.ByteString as BS
17 import Data.List
18 import Data.Word
19
20 receiveFrameHandler :: ProcessId -> ProcessId -> MessageBuffer ->
       Connection -> InnerChannelOptions
21      -> (ProcessId, String, BS.ByteString) -> Process Bool
22 receiveFrameHandler physLayerId transitId messageBuffer conn
       optionsRef (_, com, byteFrame) = do
```

```
23      options <- getOptions optionsRef
24      case decodeFrame byteFrame of
25          Just frame ->
26              case frame of
27                  AckFrame -> return ()
28                  RetFrame -> return ()
29                  _ -> do
30                      -- prevent double sending for not fully processed
    frames
31                      when (com /= "frame-acked") $ sendFrame
    physLayerId AckFrame
32                      processFrame frame options
33          _ -> do
34              informSenderError transitId "Failed to recieve frame!"
35              when (com /= "frame-acked") $ sendFrame physLayerId
    RetFrame
36      return True
37      where
38          getRemoteNames :: [(String, String)] -> Maybe (String, String)
39          getRemoteNames props = (,) <$> getValue props "remoteNameNew"
    <*> getValue props "remoteNameOld"
40
41          getValue :: [(String, String)] -> String -> Maybe String
42          getValue props key = snd <$> find (\(k, _) -> k == key) props
43
44          word2int :: Word32 -> Int
45          word2int = fromInteger . toInteger
46
47          processFrame (InformationFrame name n) _ = clearBuffer
    messageBuffer name (word2int n)
48
49          processFrame (DataPartFrame s) _ = do
50              addMessagePart messageBuffer s
51              filled <- isMessageReady messageBuffer
52              when filled $ do
53                  (name, msg) <- collectMessage messageBuffer
54                  sendMessage transitId name msg
55
56          processFrame (OptionFrame props) options = do
57              case getRemoteNames props of
58                  Just (newName, oldName) -> do
59                      informSender transitId $ "Remote name changing, "
    ++ oldName ++ " to " ++ newName
60                      sendDisconnectUser transitId oldName
61                      sendConnectUser transitId newName
62                  Nothing -> return ()
63              let newOptions = updateOptionsFromPairs props options
64              setOptions optionsRef newOptions
65              informSender transitId "Recieved new options from other
    side, changing..."
66              sendReopenPort physLayerId newOptions
67              sendUpdateOptions transitId newOptions
68
69
```

```
70          processFrame (LinkFrame name) options = do
71              sendConnectUser transitId name
72              setRemoteUsername conn name
73              ifNotConnected conn $ do
74                  informSender transitId "Remote host connected!"
75                  sendFrameWithAck physLayerId $ LinkFrame $ userName
    options
76                  openConnection conn
77
78          processFrame (UnlinkFrame name) options = do
79              sendDisconnectUser transitId name
80              ifConnected conn $ do
81                  informSender transitId "Remote host disconnected!"
82                  sendFrameWithAck physLayerId $ UnlinkFrame $ userName
    options
83                  closeConnection conn
84
85          processFrame (RetFrame) _ = informSender transitId "Main
    handler got ret frame, it is bad!"
86          processFrame (AckFrame) _ = informSender transitId "Main
    handler got ack frame, it is bad!"
87          processFrame (Upcheck) _ = return () --informSender transitId
    "Upcheck got"
```

5.10 Sending.hs

```
1  module Channel.Sending (
2        sendFrameWithAck
3      , sendFrame
4      , codeFrame
5      , decodeFrame
6      ) where
7
8  import Channel.Frame
9  import Channel.CyclicCode
10
11 import Control.Distributed.Process
12 import Control.Monad
13
14 import qualified Data.ByteString as BS
15 import Data.Maybe
16
17 codeFrame :: Frame -> BS.ByteString
18 codeFrame = codeCyclic . toByteString
19
20 decodeFrame :: BS.ByteString -> Maybe Frame
21 decodeFrame bs = case decode bs of
22     Just (Right frame, _) -> Just frame
23     _ -> Nothing
24     where decode = liftM fromByteString . decodeCyclic
25
26 sendFrame :: ProcessId -> Frame -> Process ()
```

```haskell
sendFrame targetId frame = do
    thisId <- getSelfPid
    send targetId (thisId, "send", codeFrame frame)

sendFrameWithAck :: ProcessId -> Frame -> Process Bool
sendFrameWithAck targetId frame = do
    sendFrame targetId frame
    expectAck sendTries
    where
        sendTries = 3
        timeout = 1000000 -- 1 s

        expectAck :: Int -> Process Bool
        expectAck 0 = return False
        expectAck nTries = do
            res <- receiveTimeout timeout [
                matchIf innerAckRetMatcher innerAckRetHandler
              , matchIf (\(_, com, _) -> com == "frame") otherFrameHandler]
            case res of
                Just False -> expectAck nTries
                Just True -> return True
                Nothing -> return False
            where
                innerAckRetMatcher :: (ProcessId, String, BS.ByteString) -> Bool
                innerAckRetMatcher (_, com, bs) = com == "frame" && case decodeFrame bs of
                    Just AckFrame -> True
                    Just RetFrame -> True
                    _ -> False

                innerAckRetHandler :: (ProcessId, String, BS.ByteString) -> Process Bool
                innerAckRetHandler (_, _, bs) = case fromJust $ decodeFrame bs of
                        AckFrame -> return True
                        RetFrame -> do
                           sendFrame targetId frame
                           expectAck $ nTries-1
                        _ -> error "Recieved wrong frame! Impossible state!"

                otherFrameHandler :: (ProcessId, String, BS.ByteString) -> Process Bool
                otherFrameHandler (_, _, bs) = do
                    thisId <- getSelfPid
                    case decodeFrame bs of
                        Just decodedFrame -> do
                            sendFrame targetId AckFrame
                            send thisId (thisId, "frame-acked", codeFrame decodedFrame)
                            return False
                        _ -> do
```

```
73                          sendFrame targetId RetFrame
74                          return False
```

6 PHYSICAL

6.1 Detector.hs

```
1  module Physical.Detector (
2      getSerialPorts
3      ) where
4
5  import qualified System.Hardware.Serialport      as Serial
6  import System.Info (os)
7  import System.Directory (getDirectoryContents)
8  import Control.Exception (SomeException, try)
9  import Control.Monad (filterM)
10
11 import Physical.Options
12 import Channel.Options
13
14 -- TODO: serial port detection for other systems
15 getSerialPorts :: IO [FilePath]
16 getSerialPorts = case os of
17     "linux" -> do
18         fileList <- getDirectoryContents "/dev"
19         filterM isSerialPort $ map (\s -> "/dev/" ++ s) fileList
20     _ -> return []
21
22 isSerialPort :: FilePath -> IO Bool
23 isSerialPort fileName = do
24     res <- try (Serial.openSerial fileName $ channel2physicalOptions
       defaultOptions)
25         :: IO (Either SomeException Serial.SerialPort)
26     case res of
27         Right _ -> return True
28         Left _  -> return False
```

6.2 Layer.hs

```
1  module Physical.Layer (
2      initPhysicalLayer
3      ) where
4
5  import Control.Distributed.Process
6  import qualified Data.ByteString                 as BS
7  import Control.Exception (SomeException)
8  import Control.Monad (forever)
9
```

```
10 import Physical.Options
11 import Physical.Port
12 import Utility (while, exitMsg)
13
14 receiveFrameCycle :: ProcessId -> PortState -> Process ()
15 receiveFrameCycle channelId portState = do
16     liftIO $ putStrLn "Recieving thread started..."
17     forever $ do
18         frameResult <- receiveFrame portState
19         thisId <- getSelfPid
20         case frameResult of
21             Right bs -> send channelId (thisId, "frame", bs)
22             Left err -> send channelId (thisId, "error", "Error while
    receiving frame: " ++ err ++ "!")
23
24 sendFrameHandler :: PortState -> (ProcessId, String, BS.ByteString)
    -> Process Bool
25 sendFrameHandler portState (senderId, _, msg) = do
26     thisId <- getSelfPid
27     result <- sendFrame portState msg
28     case result of
29         Nothing -> return True
30         Just err -> do
31             send senderId (thisId, "error", err)
32             return True
33
34 reopenPortHandler :: PortState -> (ProcessId, String, ChannelOptions)
    -> Process Bool
35 reopenPortHandler portState (senderId, _, options) = do
36     thisId <- getSelfPid
37     res <- reopenPort portState options
38     case res of
39         Just err -> send senderId False >> send senderId (thisId,
    "error", err) >> return True
40         Nothing  -> send senderId True  >> return True
41
42 closePortHandler :: PortState -> (ProcessId, String) -> Process Bool
43 closePortHandler portState (_, _) = do
44     closePort portState
45     return True
46
47 physicalLayerCycle :: ChannelOptions -> ProcessId -> Process ()
48 physicalLayerCycle options channelId = do
49     thisId <- getSelfPid
50     send channelId (thisId, "info", "Physical layer initialized...")
51
52     initResult <- try (initPort options) :: Process (Either
    SomeException PortState)
53     case initResult of
54         Right port -> do
55             send channelId (thisId, "info", "Serial port opened...")
56
57             spawnLocal $ receiveFrameCycle channelId port
58
```

```
59              while $ receiveWait [
60                    matchIf (\(_, com)     -> com == "exit")
    exitMsg
61                  , matchIf (\(_, com, _) -> com == "send")
    (sendFrameHandler port)
62                  , matchIf (\(_, com, _) -> com == "reopen")
    (reopenPortHandler port)
63                  , matchIf (\(_, com) -> com == "close")
    (closePortHandler port)]
64
65              closePort port
66          Left ex -> do
67              send channelId (thisId, "error", "Exception while initing
    physical layer: " ++ show ex)
68              (_, _, newOptions) <- expect :: Process (ProcessId,
    String, ChannelOptions)
69              send channelId (thisId, "info", "Got new options, trying
    to init physical layer...")
70              physicalLayerCycle newOptions channelId
71
72 initPhysicalLayer :: ChannelOptions -> ProcessId -> Process ProcessId
73 initPhysicalLayer options channelId = spawnLocal $ physicalLayerCycle
    options channelId
```

6.3 Option.hs

```
1  module Physical.Options (
2        ChannelOptions(..)
3      , channel2physicalOptions
4      ) where
5
6  import Channel.Options
7  import System.Hardware.Serialport
8
9  channel2physicalOptions :: ChannelOptions -> SerialPortSettings
10 channel2physicalOptions channel = SerialPortSettings
11     {
12       commSpeed     = portSpeed      channel
13     , bitsPerWord   = portWordBits   channel
14     , stopb         = portStopBits   channel
15     , parity        = portParityBits channel
16     , flowControl   = NoFlowControl
17     , timeout       = 0 -- other values will throw you in the abyss of
    pain! Please, don't touch ;)
18     }
```

6.4 Port.hs

```
1  module Physical.Port (
```

```
2       PortState
3     , initPort
4     , closePort
5     , reopenPort
6     , receiveFrame
7     , serialSendSafe
8     , sendFrame
9     ) where
10
11 import qualified System.Hardware.Serialport     as Serial
12 import qualified Data.ByteString                as BS
13 import qualified Data.ByteString.Lazy           as BL
14 import Data.Binary.Strict.Get
15 import Data.Binary.Put
16 import Data.IORef
17 import Control.Exception (SomeException)
18 import Control.Concurrent (yield)
19 import Control.Distributed.Process
20 import Control.Monad
21
22 import Physical.Options
23
24 type PortState = IORef (Serial.SerialPort, Bool)
25
26 toStrict :: BL.ByteString -> BS.ByteString
27 toStrict = BS.concat . BL.toChunks
28
29 initPort :: ChannelOptions -> Process PortState
30 initPort channel = do
31     port <- liftIO $ Serial.openSerial (portName channel)
    (channel2physicalOptions channel)
32     liftIO $ newIORef (port, True)
33
34 closePort :: PortState -> Process ()
35 closePort portState = do
36     (port, opened) <- liftIO $ readIORef portState
37     when opened $ liftIO $ do
38         Serial.closeSerial port
39         writeIORef portState (port, False)
40
41 reopenPort :: PortState -> ChannelOptions -> Process (Maybe String)
42 reopenPort portState options = do
43     (_, opened) <- liftIO $ readIORef portState
44     when opened $ closePort portState
45
46     res <- try (liftIO $
47         Serial.openSerial (portName options) (channel2physicalOptions
    options))
48             :: Process (Either SomeException Serial.SerialPort)
49     case res of
50         Left ex -> return $ Just $ show ex
51         Right newPort -> liftIO $ writeIORef portState (newPort,
    True) >> return Nothing
52
```

```
53
54 receiveFrame :: PortState -> Process (Either String BS.ByteString)
55 receiveFrame portState = do
56     bsLengthRes <- receiveNonEmpty portState 4
57     case bsLengthRes of
58         Left ex        -> return $ Left ex
59         Right bsLength ->
60             case fst $ runGet getWord32be bsLength of
61                 Left _             -> return $ Left ("Parsing failed!
    " ++ show bsLength)
62                 Right frameLength -> receiveNonEmpty portState $
    fromIntegral frameLength
63     where
64         receiveNonEmpty :: PortState -> Int -> Process (Either String
    BS.ByteString)
65         receiveNonEmpty pstate msgLength = do
66             (port, opened) <- liftIO $ readIORef pstate
67             if not opened then return $ Left "Port closed!"
68             else do
69                 liftIO yield
70                 res <- try (liftIO $ Serial.recv port msgLength) ::
    Process (Either SomeException BS.ByteString)
71                 liftIO yield
72                 case res of
73                     Left ex          -> return $ Left (show ex)
74                     Right msg | BS.length msg == 0 -> receiveNonEmpty
    pstate msgLength
75                               | BS.length msg < msgLength -> do
76                                 resRec <- receiveNonEmpty pstate $
    msgLength - BS.length msg
77                                 case resRec of
78                                     Left ex -> return $ Left ex
79                                     Right rec -> return $ Right $
    BS.concat [msg, rec]
80                               | otherwise -> return $ Right msg
81
82 serialSendSafe :: PortState -> BS.ByteString -> Process (Maybe Int)
83 serialSendSafe portState msg = do
84     (port, _) <- liftIO $ readIORef portState
85     res <- try (liftIO $ Serial.send port msg) :: Process (Either
    SomeException Int)
86     case res of
87         Left _ -> return Nothing
88         Right l -> return $ Just l
89
90 sendFrame :: PortState -> BS.ByteString -> Process (Maybe String)
91 sendFrame portState msg = do
92     (_, opened) <- liftIO $ readIORef portState
93     if not opened then return Nothing
94     else do
95         sendLengthRes <- serialSendSafe portState bsLength
96         case sendLengthRes of
97             Just 4 -> do
98                 sendedMsgRes <- serialSendSafe portState msg
```

```
99                 case sendedMsgRes of
100                     Just _ -> return Nothing
101                     _      -> return $ Just "Failed to send frame
    body!"
102             _ -> return $ Just "Failed to send frame length!"
103         where
104             bsLength :: BS.ByteString
105             bsLength = toStrict $ runPut $ putWord32be $ fromIntegral
    frameLength
106             frameLength = BS.length msg
```