

ForSyDe Model of an Equalizer System

Ingo Sander

This report is an extract from Sander's PhD Thesis [2] presenting an equalizer example, modelled using shallow-embedded signals.

NOTE: in the code listings, the path of the `Equalizer` module within the `forsyde-shallow-examples` project has been replaced with ...

1 Equalizer System

1.1 Overview

The main task of the equalizer system is to adjust the audio signal according to the `buttonControl`, that works as a user interface. In addition, the bass level must not exceed a predefined threshold to avoid damage to the speakers.

This specification can be naturally decomposed into four functions shown in Figure 1. The subsystems `buttonControl` and `distortionControl`, are control dominated (grey shaded), while the `audioFilter` and the `audioAnalyzer` are data flow dominated subsystems.

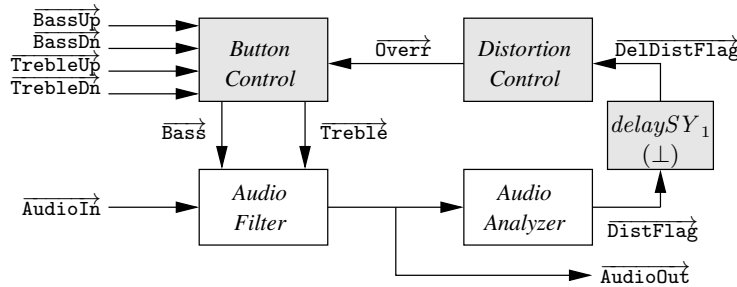


Figure 1: Subsystems of the `equalizer`

The `buttonControl` subsystem monitors the button inputs and the override signal from the subsystem `distortionControl` and adjusts the current bass and treble levels. This information is passed to the subsystem `audioFilter`, which receives the audio input, and filters and amplifies the audio signal according to the current bass and treble levels. This signal, the output signal of the equalizer, is analyzed by the `audioAnalyzer` subsystem, which determines, whether the bass exceeds a predefined threshold. The result of this analysis is passed to the subsystem `distortionControl`, which decides, if a minor or major violation is encountered and issues the necessary commands to the `buttonControl` subsystem.

The frequency characteristics of the `equalizer` is adjusted by the coefficients for the three FIR-filters in the `audioFilter`.

```

module ...Equalizer.Equalizer (
  equalizer
) where

import ForSyDe.Shallow

import ...Equalizer.ButtonControl
import ...Equalizer.DistortionControl
import ...Equalizer.AudioAnalyzer
import ...Equalizer.AudioFilter

```

The structure of the equalizer is expressed as a network of blocks:

```

equalizer lpCoeff bpCoeff hpCoeff dftPts
  bassUp bassDn trebleUp trebleDn input = (bass, treble)
  where
    (bass, treble) = buttonControl overrides bassUp bassDn
                    trebleUp trebleDn
    output         = audioFilter lpCoeff bpCoeff hpCoeff bass
                    treble input
    distFlag       = audioAnalyzer dftPts output
    overrides      = distortionControl delayedDistFlag
    delayedDistFlag = delaySY Abst distFlag

```

Since the equalizer contains a feedback loop, the signal **distFlag** is delayed one event cycle using the initial value \perp .

2 Internal Types

2.1 Overview

This module is a collection of data types that are used in the equalizer model.

```

module ...Equalizer.EqualizerTypes where

data AnalyzerMsg = Pass
                | Fail
                deriving (Show, Read, Eq)

data OverrideMsg = Lock
                | CutBass
                | Release
                deriving (Show, Read, Eq)

data Sensor = Active deriving (Show, Read, Eq)

data Button = BassDn
            | BassUp
            | TrebleDn
            | TrebleUp
            deriving (Show, Read, Eq)

```

3 Button Control Subsystem

3.1 Overview

The subsystem **buttonControl** works as a user interface in the equalizer system. It receives the four input signals $\overrightarrow{\text{BassDn}}$, $\overrightarrow{\text{BassUp}}$, $\overrightarrow{\text{TrebleDn}}$, $\overrightarrow{\text{TrebleUp}}$ and the override signal **override** from the **distortionControl** and calculates the new bass and treble values for the output signals $\overrightarrow{\text{Bass}}$ and $\overrightarrow{\text{Treble}}$. The subsystem contains the main processes **buttonInterface** and **levelControl**. The process **levelControl** outputs a new value, if

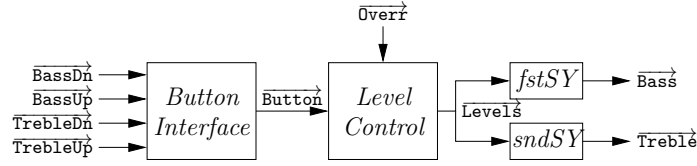


Figure 2: The Subsystem `buttonControl`

either the signal $\overrightarrow{\text{Button}}$ or the signal $\overrightarrow{\text{Override}}$ is present, otherwise the output value is absent. The process `holdLevel` is modeled by means of `holdSY(0.0,0.0)` that outputs the last present value, if the input value is absent. The process `unzipSY` transforms a signal of tuples (the current bass and treble level) into a tuple of signals (a bass and a treble signal).

```

module ... Equalizer.ButtonControl (
  buttonControl
) where

import ForSyDe.Shallow
import ... Equalizer.EqualizerTypes

data State    = Operating
              | Locked deriving(Eq, Show)

type Level    = Double
type Bass     = Level
type Treble   = Level

buttonControl :: Signal (AbstExt OverrideMsg)
              → Signal (AbstExt Sensor)
              → Signal (AbstExt Sensor)
              → Signal (AbstExt Sensor)
              → Signal (AbstExt Sensor)
              → (Signal Bass, Signal Treble)

buttonControl overrides bassDn bassUp trebleDn trebleUp
  = (bass, treble)
  where (bass, treble) = unzipSY levels
        levels = holdSY (0.0, 0.0) $ levelControl button overrides
        button = buttonInterface bassDn bassUp trebleDn trebleUp

```

3.2 The Process `buttonInterface`

The `buttonInterface` monitors the four input buttons $\overrightarrow{\text{BassDn}}$, $\overrightarrow{\text{BassUp}}$, $\overrightarrow{\text{TrebleDn}}$, $\overrightarrow{\text{TrebleUp}}$ and indicates if a button is pressed. If two or more buttons are pressed the conflict is resolved by the priority order of the buttons.

```

buttonInterface :: Signal (AbstExt Sensor)
               → Signal (AbstExt Sensor)
               → Signal (AbstExt Sensor)
               → Signal (AbstExt Sensor)
               → Signal (AbstExt Button)

buttonInterface bassUp bassDn trebleUp trebleDn
  = zipWith4SY f bassUp bassDn trebleUp trebleDn
  where f (Prst Active) _ _ _ = Prst BassUp
        f _ (Prst Active) _ _ = Prst BassDn
        f _ _ (Prst Active) _ = Prst TrebleUp
        f _ _ _ (Prst Active) = Prst TrebleDn
        f _ _ _ _             = Abst

```

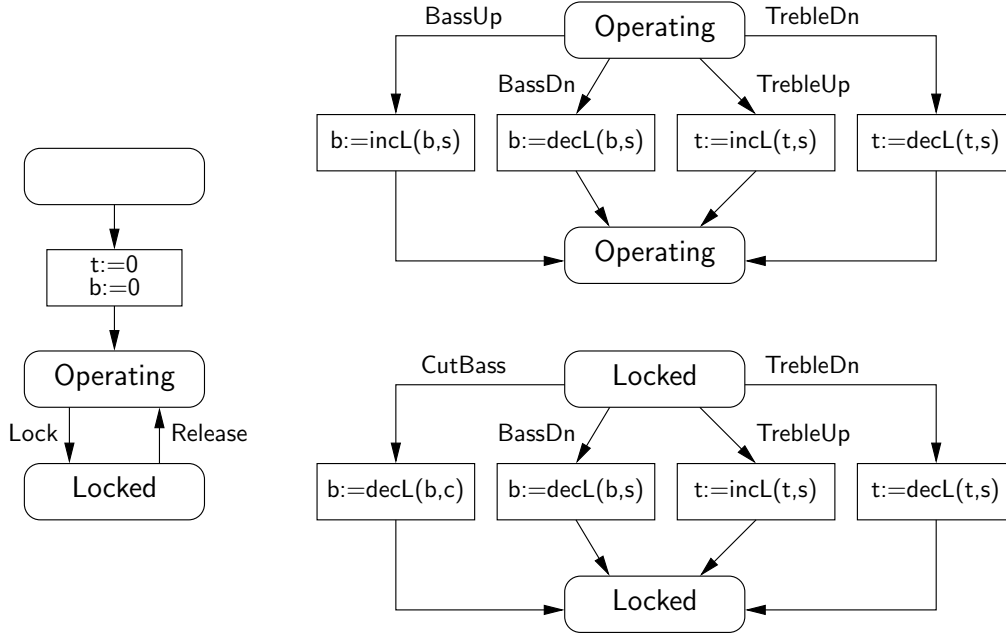


Figure 3: The State Diagram of the Process `levelControl`

3.3 The Process `levelControl`

The process has a local state that consists of a mode and the current values for the bass and treble levels (Figure 3). The `levelControl` has two modes, in the mode `Operating` the bass and treble values are stepwise changed in 0.2 steps. However, there exists maximum and minimum values which are -5.0 and +5.0. The process enters the mode `Locked` when the `Override` input has the value `Lock`. In this mode an additional increase of the bass level is prohibited and even decreased by 1.0 in case the `Override` signal has the value `CutBass`. The subsystem returns to the `Operating` mode on the override value `Release`. The output of the process is an absent extended signal of tuples with the current bass and treble levels.

```

levelControl :: Signal (AbstExt Button)
              → Signal (AbstExt OverrideMsg)
              → Signal (AbstExt (Bass,Treble))
levelControl button overrides
  = mealy2SY nextState output (initState, initLevel) button overrides

nextState :: (State,(Double,Double)) → AbstExt Button
          → AbstExt OverrideMsg → (State,(Double,Double))
nextState (state, (bass, treble)) button override
  = (newState, (newBass, newTreble))
  where
    newState = if state == Operating
               then if override == Prst Lock
                    then Locked
                    else Operating
               else if override == Prst Release
                    then Operating
                    else Locked

    newBass = if state == Locked
              then if override == Prst CutBass
                   then decreaseLevel bass cutStep

```

```

        else if button == Prst BassDn
            then decreaseLevel bass step
            else bass
    else if button == Prst BassDn
        then decreaseLevel bass step
        else if button == Prst BassUp
            then increaseLevel bass step
            else bass

    newTreble = if button == Prst TrebleDn
                then decreaseLevel treble step
                else if button == Prst TrebleUp
                    then increaseLevel treble step
                    else treble

output :: (a, (Bass, Treble))
      → AbstExt Button
      → AbstExt OverrideMsg
      → AbstExt (Bass,Treble)
output _      Abst Abst = Abst
output (_, levels) _      = Prst levels

```

The process uses the following initial values.

```

initState = Operating
initLevel = (0.0, 0.0)
maxLevel  = 5.0
minLevel  = -5.0
step      = 0.2
cutStep   = 1.0

```

The process uses the following auxiliary functions.

```

decreaseLevel :: Level → Level → Level
decreaseLevel level step = if reducedLevel >= minLevel
                           then reducedLevel
                           else minLevel

    where reducedLevel = level - step

increaseLevel :: Level → Level → Level
increaseLevel level step = if increasedLevel <= maxLevel
                           then increasedLevel
                           else maxLevel

    where increasedLevel = level + step

```

4 Audio Filter Subsystem

4.1 Overview

Figure 4 shows the structure of the **audioFilter**. The task of this subsystem is to amplify different frequencies of the audio signal independently according to the assigned levels. The audio signal is splitted into three identical signals, one for each frequency region. The signals are filtered and then amplified according to the assigned amplification level. As the equalizer in this design only has a bass and treble control, the middle frequencies are not amplified. The output signal from the **audioFilter** is the addition of the three filtered and amplified signals.

We model this structure as a network of blocks directly from Figure 4. It consists of three filters, two amplifiers and an adder. These blocks are modeled in the process layer. The **audioFilter** has the filter coefficients for the low pass, band pass and high pass filter as parameters.

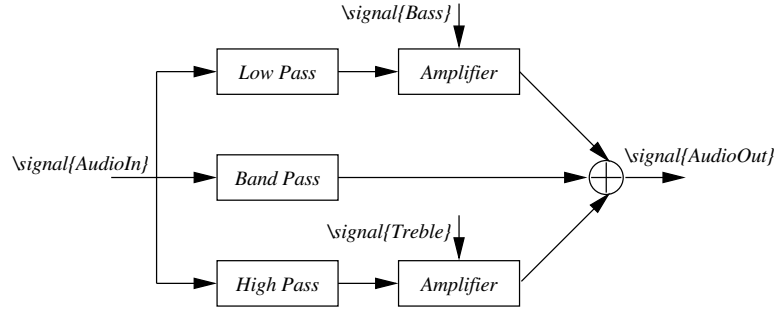


Figure 4: Subsystems of the *Audio Filter*

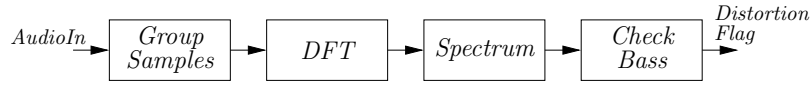


Figure 5: The *Audio Analyzer* subsystem

```

module ... Equalizer.AudioFilter where

import ForSyDe.Shallow
import ForSyDe.Shallow.Utility.FIR

audioFilter :: Floating a => Vector a → Vector a → Vector a
              → Signal a → Signal a → Signal a → Signal a
audioFilter lpCoeff bpCoeff hpCoeff bass treble audioIn = audioOut
  where audioOut      = zipWith3SY add3 bassPath middlePath treblePath
        bassPath      = ((amplify bass) . lowPass) audioIn
        middlePath     = bandPass audioIn
        treblePath     = ((amplify treble) . highPass) audioIn
        lowPass        = firSY lpCoeff
        bandPass       = firSY bpCoeff
        highPass       = firSY hpCoeff
        amplify        = zipWithSY scale
        add3 x y z      = x + y + z
        scale x y       = y * (base ** x)
        base           = 1.1

```

5 Audio Analyzer Subsystem

5.1 Overview

The `audioAnalyzer` analyzes the current bass level and raises a flag when the bass level exceeds a limit.

As illustrated in Figure 5 the `audioAnalyzer` is divided into four blocks. The input signal is first grouped into samples of size N in the process `groupSamples` and then processed with a `dft` in order to get the frequency spectrum of the signal. Then the power spectrum is calculated in `spectrum`. In `checkBass` the lowest frequencies are compared with a threshold value. If they exceed this value, the output `distortionFlag` will have the value Fail.

Since `groupSamples` needs N cycles for the grouping, it produces $N - 1$ absent values \perp for each grouped sample. Thus the following processes `dft`, `spectrum` and `checkBass` are all Ψ -extended in order to be able to process the absent value \perp .

```

| module ... Equalizer.AudioAnalyzer (

```

```

    audioAnalyzer
  ) where

import Data.Complex
import ForSyDe.Shallow
import ...Equalizer.EqualizerTypes

input = 0.1 :- 0.2 :- input

limit :: Double
limit = 1.0

nLow :: Int
nLow = 3

audioAnalyzer :: Int → Signal Double → Signal (AbstExt AnalyzerMsg)
audioAnalyzer pts = mapSY (psi checkBass) — Check Bass
                  . mapSY (psi spectrum) — Spectrum
                  . mapSY (psi (dft pts)) — DFT
                  . groupSY pts — Group Samples
                  . mapSY toComplex

spectrum :: (RealFloat a) => Vector (Complex a) → Vector a
spectrum = mapV log10 . selectLow nLow . mapV power . selectHalf .
  dropV 1
  where
    log10 x = log x / log 10
    power x = (magnitude x) ^ 2
    selectLow n xs = takeV n xs
    selectHalf xs = takeV half xs
    where half = floor (fromIntegral (lengthV xs) / 2)

checkBass :: Vector Double → AnalyzerMsg
checkBass = checkLimit limit . sumV
  where
    checkLimit limit x
      | x > limit = Fail
      | otherwise = Pass
    sumV vs = foldlV (+) 0.0 vs

toComplex x = x :+ 0

```

6 Distorsion Control Subsystem

The block `distortionControl` is directly developed from the SDL-specification, that has been used for the MASCOT-model [1]. The specification is shown in Figure 6.

The `distortionControl` is a single FSM, which is modeled by means of the skeleton `mealySY`. The global state is not only expressed by the explicit states - Passed, Failed and Locked -, but also by means of the variable `cnt`. The state machine has two possible input values, Pass and Fail, and three output values, Lock, Release and CutBass.

The `mealySY` creates a process that can be interpreted as a Mealy-machine. It takes two functions, `nxtSt` to calculate the next state and `out` to calculate the output. The state is represented by a pair of the explicit state and the variable `cnt`. The initial state is the same as in the SDL-model, given by the tuple (Passed, 0). The `nxtSt` function uses pattern matching. Whenever an input value matches a pattern of the `nxtSt` function the corresponding right hand side is evaluated, giving the next state. An event with an absent value leaves the state unchanged. The output function is modeled in a similar way. The output is absent, when no output message is indicated in the SDL-model.

```

| module ...Equalizer.DistortionControl (

```

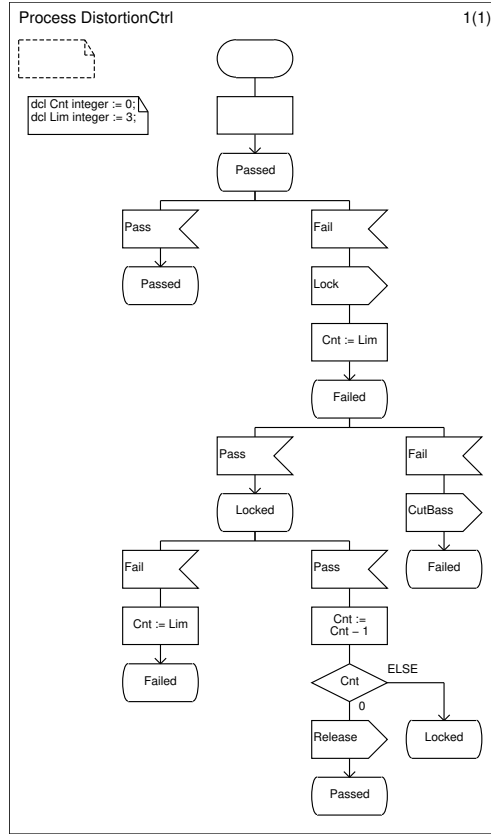


Figure 6: SDL-description of *Distortion Control*

```

distortionControl
) where

import ForSyDe.Shallow
import ...Equalizer.EqualizerTypes

data State = Passed
           | Failed
           | Locked

distortionControl :: Signal (AbstExt AnalyzerMsg)
                  → Signal (AbstExt OverrideMsg)

distortionControl distortion
  = mealySY nxtSt out (Passed, 0) distortion

lim = 3

--      State      Input      NextState
nxtSt (state, cnt) (Abst)      = (state, cnt)
nxtSt (Passed, cnt) (Prst Pass) = (Passed, cnt)
nxtSt (Passed, _ ) (Prst Fail)  = (Failed, lim)
nxtSt (Failed, cnt) (Prst Pass) = (Locked, cnt)
nxtSt (Failed, cnt) (Prst Fail) = (Failed, cnt)
nxtSt (Locked, _ ) (Prst Fail)  = (Failed, lim)
nxtSt (Locked, cnt) (Prst Pass) = (newSt, newCnt)
  where newSt = if (newCnt == 0)
                 then Passed

```



```

        else Locked
        newCnt = cnt - 1

—   State      Input      Output
out (Passed,_) (Prst Pass) = Abst
out (Passed,_) (Prst Fail) = Prst Lock
out (Failed,_) (Prst Pass) = Abst
out (Failed,_) (Prst Fail) = Prst CutBass
out (Locked,_) (Prst Fail) = Abst
out (Locked,cnt) (Prst Pass) = if (cnt == 1)
                                then Prst Release
                                else Abst
out _           Abst         = Abst

```

References

- [1] Per Bjuréus and Axel Jantsch. “MASCOT: A specification and cosimulation method integrating data and control flow”. In: *Proceedings of the Design and Test Europe Conference (DATE)*. Paris, France, Mar. 2000, pp. 161–168.
- [2] Ingo Sander. “System Modeling and Design Refinement in ForSyDe”. NR 20140805. PhD thesis. KTH, Microelectronics and Information Technology, IMIT, 2003, pp. xvi, 228. ISBN: 91-7283-501-X.