

Université Paris VIII

Ecole Doctorale Cognition, Langage, Interaction

Laboratoire Paragraphe – CiTu

Sciences de l'Information et de la Communication

The Ardour DAW – Latency Compensation and Anywhere-to-Anywhere Signal Routing Systems

Robin Gareus

Thesis presented in fulfilment of the requirements for the degree of
Doctor of Philosophy

2017

Date: February 11, 2018

Revision: print-1-g09fb821

Keywords: Digital Audio, Latency, Latency Compensation, Process Graph, Synchronisation, Algorithms

Mots-clés: Algorithmes, Synchronisation, Son – Enregistrement et reproduction – Techniques numériques, Trasmission numériques, Routage (informationne)

Copyright: ©2012-2017 Robin Gareus. This is an open-access article distributed under the terms of the Creative Commons Attribution 3.0 Unported License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Université Paris VIII

Ecole Doctorale Cognition, Langage, Interaction

Laboratoire Paragraphe – CiTu

Sciences de l'Information et de la Communication

The Ardour DAW – Latency Compensation and Anywhere-to-Anywhere Signal Routing Systems

Robin **Gareus**

supervised by

Khaldoun **Zreik**, Prof. Dr.

Digital Humanities Department

Université Paris 8

defense: 8/Dec/2017

jury members

Victor **Lazzarini**

Laurent **Pottier**

Maynooth University Université Jean Monnet

Winfried **Ritsch**

Isis **Truck**

IEM Graz

Université Paris 8

Contents

Abstract	IX
Abstract (French)	X
Acknowledgements	XII
Résumé substantiel en français	1
I On Latency and Digital Audio	53
1 Introduction	53
1.1 Digital Audio	53
1.2 Latency and Latency Compensation	54
1.3 Signal Routing	60
1.4 Latency compensation in anywhere-to-anywhere routing systems . . .	61
1.5 State of the art	64
2 Research and Development	65
2.1 Background, motivation and a short history of Ardour	65
2.2 Prototyping	67
2.3 Free Software & Collaborations	69
II The Digital Audio Ecosystem	73
3 Sources of Latency	73
3.1 Sound propagation through air	73
3.2 Digital-to-Analogue and Analogue-to-Digital conversion	73

3.3	Computer Architecture	74
3.4	Operating System	75
3.5	Digital Signal Processing	75
3.6	Processing Period	76
4	Input/Output Latency Measurement	77
5	Plug-ins	79
5.1	Examples of Latent Plugins	80
5.2	Plugin Standards	82
III	Architecture and Building Blocks	85
6	Overview	85
7	Architecture	86
8	Internal Route Architecture	90
9	Automation Event Handling	91
10	Ports	93
11	Portengine	96
12	Aux Sends and Busses	98
13	IO-Processors and Deliveries	99
IV	Latency Compensation	103
14	Synchronization	103

15 Time Alignment	104
16 Execution and Latency Graphs	109
16.1 Routing and Feeds	112
16.2 Upstream/Capture Latency	113
16.3 Downstream/Playback Latency	113
16.4 Calculating Latencies	114
16.5 Send, Return and Insert Latency	117
16.6 Ambiguous Latency	122
16.7 Additional Graph Considerations	124
17 Processing	125
17.1 The Process Cycle	125
17.2 The Process Callback	128
17.3 Parallel Execution	131
18 Route Processing	134
18.1 Setting Processor Latencies	138
18.2 Processing Route Buffers	139
18.3 Playback & Capture	142
18.4 Disk-writer - Write Behind	143
18.5 Disk-reader - Read Ahead	145
18.6 Capturing Processor	147
18.7 Side-chains and Modulation	148
19 Vari-speed	148

V Conclusion	151
VI Appendix	153
A Delaylines in Ardour	153
A.1 ARDOUR::DelayLine	153
A.2 ARDOUR::FixedDelay	153
B Verification Tools	154
B.1 Bounce recording via physical cable	154
B.2 Introspection	156
B.3 No Delay Line	156
B.4 First Non Zero Sample	157
B.5 Audio/MIDI Alignment Testing	157
C Audio Signal Measurement and Visualization	159
C.1 On Measurements	160
C.2 Introduction to Audio Signal Meters	160
C.3 Meter Types and Standards	161
C.4 Standardisation	173
C.5 Software Implementation	174
Bibliography	179
List of Tables	185
List of Figures	187

Abstract

In inherently latent digital systems it is not trivial to compensate for latency, particularly in situations of complex signal routing graphs as is the case in a Digital Audio Workstation.

While the general problem is of mathematical nature, design complexities arise in real-time audio systems due to constraints by hardware, system-architecture and engineering.

To construct a system providing for full-graph latency compensation with anywhere-to-anywhere routing capabilities, it is insufficient to merely describe mechanisms. The complete system has to be designed as one and prototyped to take real-world limitations into account.

This research was carried out using Ardour, a digital audio workstation, which is freely available under the GPL free-software licence. This thesis is as much a design-report as it is research documentation.

A complete breakdown of building-blocks and interaction is presented, most of which has also been implemented beyond a proof-of-concept with the goal to bridge the gap between professional audio production systems and freely accessible documentation for research and development.

While ostensibly focusing on Ardour, this thesis describes generic concepts of Audio Workstations like Ports, Tracks, Busses, and DSP Processors, as well as operational interaction between them.

Basic concepts common to all digital I/O processes and sources of latency are explained, and process- and latency graphs are illustrated to provide a complete picture. General issues related to time-alignment, both local, and global, as well as more DAW specific cases like parameter-automation and parallel-execution are discussed. Algorithms are modelled with pseudocode where appropriate and application programming interfaces are presented as examples to concepts throughout the text.

Résumé

Dans des systèmes numériques essentiellement latents, compenser la latence n'est pas trivial, en particulier lorsque les graphes de routage du signal sont complexes comme c'est souvent le cas dans une station audionumérique (DAW).

Tandis que le problème général est de nature mathématique, des complications apparaissent dans la conception de systèmes audio en temps réel à cause des contraintes du matériel, de l'architecture du système, ou de l'ingénierie.

Pour construire un système fournissant une compensation de latence sur l'intégralité du graphe avec possibilité de connecter n'importe quelle source à n'importe quelle destination, uniquement décrire les mécanismes est insuffisant. Le système complet doit être conçu d'un bloc à l'aide de prototypes pour prendre en compte les limitations du monde réel.

Cette recherche a été menée en utilisant Ardour, une station audionumérique librement disponible sous licence libre GPL. Cette thèse est autant un rapport de conception qu'une documentation de recherche.

Une analyse complète des éléments de base et de leurs interactions est présentée. La plupart ont été implémentés au delà de la démonstration de faisabilité, dans le but de combler l'écart entre les systèmes professionnels de production audio et la documentation librement accessible pour la recherche et le développement.

Même si elle s'attache ostensiblement à Ardour, cette thèse décrit les concept génériques des stations audio tels que les Ports, les pistes (Tracks), les bus (Busses) et les processeurs de traitement numériques du signal (Processors) ainsi que les interactions opérationnelles entre eux.

Les concepts de base communs à toutes les entrées/sorties numériques sont expliqués ainsi que les sources de latence. Les graphes de traitement et de latence sont illustrés pour présenter une vue d'ensemble.

Les problèmes généraux rencontrés lors de l'alignement temporel, tant local que

global, seront étudiés, de même que des cas plus spécifiques aux stations audionumériques comme l'automatisation des paramètres et le parallélisme d'exécution. Les algorithmes sont modélisés en pseudocode lorsque c'est approprié, et des interfaces de programmation sont proposées en exemple pour les concepts rencontrés au long du texte.

Acknowledgements

*One of the secrets of life is that
all that is really worth the doing is what we do for others.
(Lewis Carroll)*

First and foremost, I would like to thank my supervisor and advisor, Prof. Khaldoun Zreik, for having an open mind on exploring the unconventional, and supporting technical projects in the context of media-art.

Every interaction with Khaldoun has been an absolute pleasure and without his dedication to support digital art and infrastructure to create it, I would not be here writing this thesis. I am very grateful of the leeway that he granted, particularly by providing support remotely and internationally. Khaldoun has supported and encouraged me throughout my graduate years and his resourcefulness in international collaborations was great inspiration.

I would like to thank Maurice Benayoun for giving me the opportunity to join CiTu and become the technical-backbone of his work on Art after Technology. He pushed me to engineer what appeared to be unreasonable and it was also Maurice who motivated me writing a PhD thesis in the first place. Sharing his international exhibition opportunities and collaborations has been an amazing experience that is to stay with me for time to come.

Kudos go to Paul Davis. This thesis would not have been possible without him laying the groundwork on what is today Ardour. His expertise on software architecture has been critical during periods of prototyping and endless discussions on IRC proved to become outstanding advise. Over the years Paul has played a major role in shaping me in my personal and professional life.

I would like to thank Chris Goddard countless hours of beta-testing, measuring and lending his lending his expertise on hands-on pro-audio, as well as Julien Rivaud for

proof-reading, providing feedback, as well as translating the thesis to French on short notice.

Ben Loftis at harrisonconsoles made it possible to realize this work as product on the market and financial support from harrisonconsoles.com was vital during the last years.

I would like to especially thank Prof. Victor Lazzarini, who not only volunteered to serve as on the review committee, but whose performance and presentation at the Linux Audio Conference 2006 laid the cornerstone to my interest in Linux-Audio. The opportunity to co-host that conference with him 5 years later has been a great honour.

I would like to show my appreciation to other members of the review committee, in particular Prof. Laurent Pottier and ao. Prof. Winfried Ritsch who support the use of free audio-software in academia, as well as Prof. Julius O. Smith and Fernando Lopez-Lezcano to inviting me to Stanford over the course of this PhD.

A final special mention is due to Fons Adriaensen whose company is always enjoyable despite or even because of his knack for tirelessly pointing out technical flaws and implementation mistakes. His knowledge of DSP and the willingness to share in-depth information over a bottle of wine compares to no other expert.

A general shout goes out to the LAD community, friends close and far and all the wonderful people, yes you, who joined me during various periods along the road in the last 5 years. It's been quite a ride!

I'm grateful to my parents. Mum, who ensured I knew that I could do whatever I wanted in life and my dad who introduced me to computers and music early in my life, an inspiration that still lingers even after all those years and may explain a lot in the pages to come.

To Carolina,

You have given me more love than a person can ever hope to receive.

Première partie

Latence et Audio numérique

1 Introduction

1.1 Audio numérique

Tel qu'il est perçu par les humains, le son est un phénomène purement analogique : une onde de pression longitudinale se propageant dans un milieu. Il a été décrit dans plusieurs domaines de la physique et a été source de recherches interdisciplinaires sur l'acoustique, étudiant les propriétés des ondes mécaniques ou encore les sensations auditives qu'elles évoquent aux humains.

Une des propriétés clef de l'audio analogique est que c'est un signal continu. C'est vrai d'une onde de pression tangible dans l'air comme de la transcription électrique de ce signal : à tout moment il existe une valeur représentant le signal à cet instant exact : pression, tension, ou intensité.

Les représentations numériques de l'audio sont discrètes. L'information est échantillonnée à intervalles donnés, et convertie en une suite de nombres. Contrairement à un traitement du signal purement analogique, l'audio numérique est représenté par des échantillons isolés dans le temps.

En numérique, la granularité minimum d'un signal audio est un échantillon, ce qui correspond à 20μ secondes pour une fréquence d'échantillonnage de 48000 échantillons

par seconde.

Même s'il peut sembler qu'un tel signal discontinu, lorsqu'il est de nouveau converti vers le monde analogique, résulterait en un signal en escalier, il n'en est rien. D'après le théorème d'échantillonnage de Nyquist-Shannon[1], il y a une unique correspondance entre le signal analogique et sa représentation numérique en suite d'échantillons audio discrets si la fréquence maximum est au plus la moitié de la fréquence d'échantillonnage.

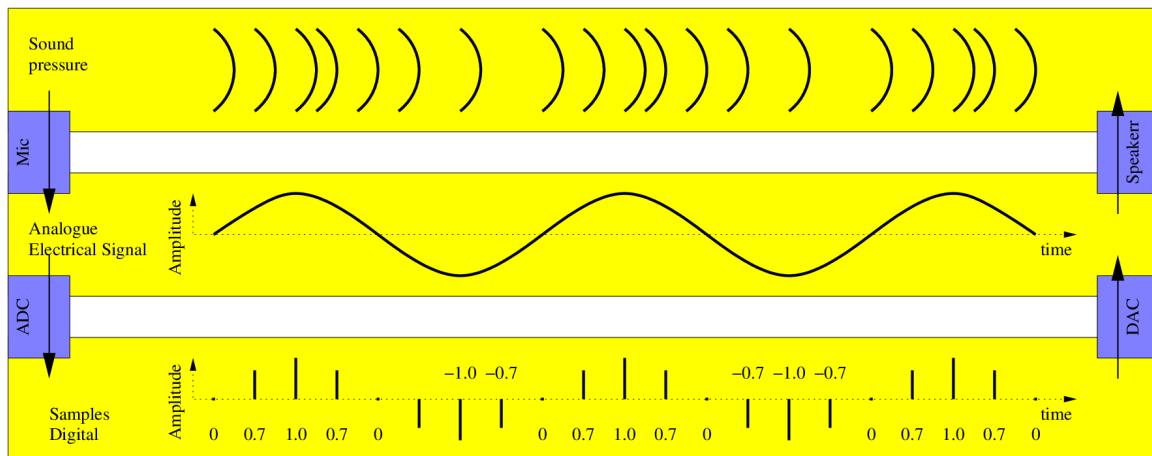


Figure fr-1 – Un signal audio, de l'onde de pression sonore à la représentation numérique discrète en passant par le signal électrique continu

Bien qu'il y ait des motivations techniques à choisir de hautes fréquences d'échantillonnage¹, une fréquence d'échantillonnage d'au moins deux fois la plus haute fréquence audible par l'oreille humaine suffit à couvrir les usages génériques de la production audio. Cette limite de l'audition humaine a été établie à 20kHz, ce qui est appuyé par près d'un siècle de données expérimentales. Ceci amène à une fréquence

¹par exemple des filtres passe-bande pour des convertisseurs analogique vers numérique ou réciproquement

d'échantillonnage juste au dessus de 40000 échantillons par seconde : en général 44,1kHz pour les CD ou 48kHz pour les DVD².

1.2 Latence et compensation de la latence

Un des effets secondaires de la conversion analogique vers numérique ou numérique vers analogique est qu'elle prend du temps.

Un convertisseur analogique-numérique échantillonne le signal à intervalles réguliers (par exemple les $20\mu s$ mentionnés précédemment). La valeur est seulement disponible après la conversion. Au moment où la représentation numérique est disponible, le signal correspondant est déjà du passé. De la même façon, le convertir en retour avec un convertisseur numérique-analogique engendre un délai supplémentaire.

En comparaison avec un véritable by-pass analogique du signal, tout signal échantillonné numériquement puis reconstruit sera en retard.

En superposant un signal avec une copie retardée de lui-même, plusieurs effets peuvent arriver. Si le signal retardé a un décalage de phase de 180° , les deux signaux s'annulent l'un l'autre et l'on obtient un silence complet comme présenté à la figure *fr-2*.

Dans le cas général, ce type d'interférences est appelé filtre en peigne : plusieurs fréquences du spectre s'annulent, tandis que d'autres sont amplifiées.

Si le décalage entre les deux signaux est grand (ou qu'il n'est pas constant), cela produit une modulation d'amplitude ou un *battement* audible. Un délai encore plus

²Une fréquence d'échantillonnage trop élevée est considérée délétère : «les ultra-sons peuvent produire de la distorsion dans les amplificateurs analogiques et causer des artefacts de crénélage dans le spectre audible»[2].

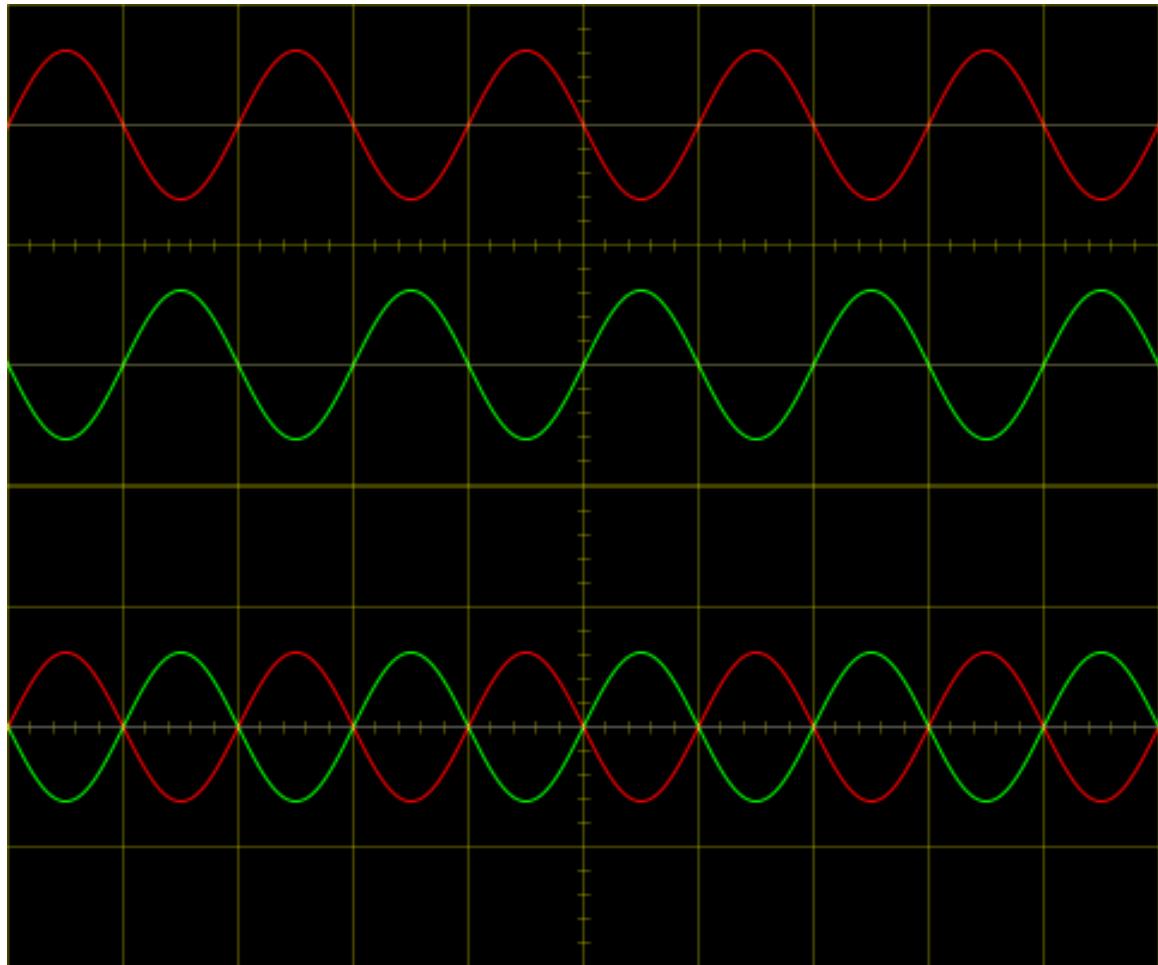


Figure fr-2 – Deux sinusoïdes décalées de 180 degrés dont le cumul est un silence complet.

< 10ms	Perçu comme simultané
10 - 20 ms	Seuil de coïncidence
> 20 ms	Perception de deux évènements distincts

Table fr-1 – Règle empirique pour la latence audio.

grand est perçu comme un écho. On peut prendre la règle empirique suivante : lorsque le décalage est supérieur à 10ms, les humains distinguent deux évènements séparés, même si cela dépend aussi du type de signal et de sa fréquence fondamentale.

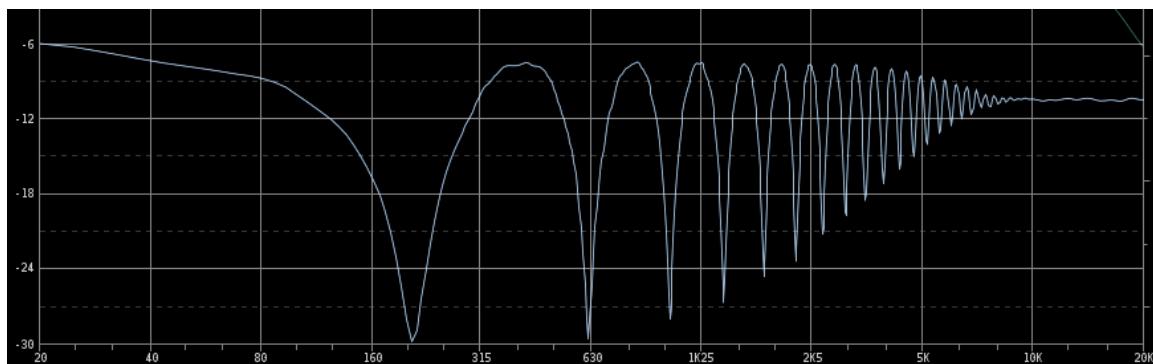


Figure fr-3 – Spectre d'un bruit rose échantillonné à 48kHz combiné à une copie de lui-même retardée de 116 échantillons. Sans le retard on verrait un spectre constant vers environ –6dBFS. La forme du spectre avec des creux périodiques explique le nom de filtre en peigne.

Superposer des signaux avec des retards différents produit des effets qui sont en général indésirables lors de la production de musique.

Si le retard est nettement plus petit que la fréquence limite de l'audition humaine, et que le signal est échantillonné avec une fréquence suffisante, l'effet est négligeable. De nombreuses solutions purement matérielles fonctionnent avec une latence sous la milliseconde et leur court temps de réponse est la raison pour laquelle beaucoup de systèmes professionnels haut de gamme sont implémentés en utilisant des processeurs de signal numériques dédiés sous forme de circuits électroniques, et ne sont pas construits à l'aide d'ordinateurs génériques. Cependant, un avantage majeur d'un ordinateur tout-usage est sa versatilité. Alors qu'un matériel dédié se limite en général à un unique usage, un PC offre de la flexibilité.

Le premier système audio numérique date de la fin des années 1950 ; la production

musicale généraliste a entrepris une transition majeure de l'analogique vers le domaine numérique depuis le début des années 80, mais ce n'est pas avant la fin des années 1990 que les processus de production et post-production dominants ont emboité le pas.

À l'avènement du 21^e siècle, des ordinateurs suffisamment puissants pour traiter l'audio en temps réel sont devenus largement accessibles. Les stations audio-numériques sont sorties de la niche professionnelle pour permettre au grand public d'accéder à des outils d'écriture, d'enregistrement et de production.

La conception classique d'un PC consiste en une chaîne de divers composants, de l'interface audio, à la puce *southbridge* du processeur en passant par les bus PCI ou USB. L'architecture d'un ordinateur moderne est optimisée pour la bande passante et non le débit. Cette contrainte favorise le traitement de larges blocs de donnée plutôt qu'un échantillon à la fois³. Bien que traiter les données en bloc réduit l'utilisation globale des ressources système et optimise la performance générale, cela introduit un temps de réponse minimal.

Le délai causé est petit, mais non négligeable. Par exemple, pour 64 échantillons de période à 48k échantillons par seconde, une période de délai en entrée et une en sortie :

$$2 \cdot 64[\text{échantillons}] / 48000[\text{échantillons/seconde}] = 2.66[\text{ms}]$$

Pour mettre ce résultat en perspective, c'est le temps que met le son à franchir 90cm

³SIMD : single instruction, multiple data (une seule instruction, plusieurs données, comme les instructions SSE), arbitrage de bus PCI, transferts de données par rafale

dans l'air soit la distance moyenne entre un piano et l'oreille du pianiste.

Un PC générique non optimisé aura le plus souvent des retards plus importants : par exemple l'ordonnanceur des systèmes Microsoft Windows utilise des tranches de temps de 16ms pour sa granularité par défaut.

Le terme technique pour ce retard est *Latence*.

La latence est le temps de réponse d'un système à un stimulus donné.

Dans le contexte audio elle est généralement divisée en latence de capture et latence de reproduction.

- *Latence de capture* : c'est le temps nécessaire pour que l'audio numérisé soit disponible pour le traitement numérique. En général c'est une période audio.
- *Latence de reproduction* : c'est le temps nécessaire pour que l'audio numérique soit traité et envoyé à la sortie de la chaîne de traitement. Au mieux c'est une période audio.

Mais cette division est un détail d'implémentation et n'a que peu d'intérêt dans un premier temps⁴. Ce qui importe plus est la combinaison des deux. La latence d'aller-retour est le temps nécessaire pour qu'un événement audio donné soit capturé, traité, et reproduit.

Il est important de noter que sur les systèmes PC, la latence de traitement est l'objet d'un choix : elle peut être baissée en respectant les limites imposées seulement

⁴Nous verrons plus tard que la distinction est importante lors d'un enregistrement avec lecture, en alignant l'audio.

par le matériel (périphérique audio, processeur, et vitesse du bus) ou le pilote audio. Des latences plus basses augmentent la charge du système puisqu'il doit traiter l'audio en morceaux plus petits qui arrivent bien plus fréquemment. Plus basse est la latence, plus haute est la probabilité que le système ne réussisse pas à respecter l'échéance du traitement, ce qui donne un *x-run* (une mémoire tampon pleine ou au contraire avec trop peu de données) avec son joyeux train de clics, pops et craquements dans le flux audio.

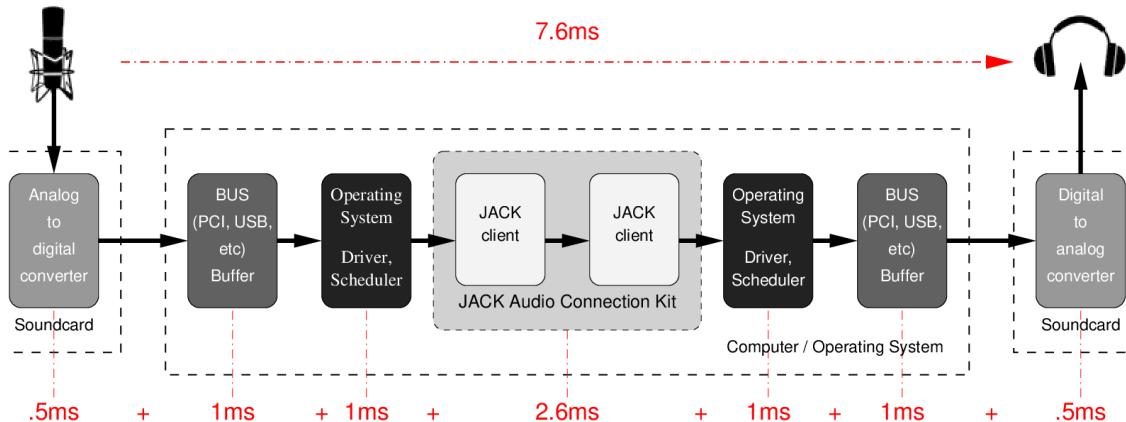


Figure fr-4 – Chaîne de latence. Les valeurs données en exemple correspondent à un PC typique. Avec du matériel professionnel et un système optimisé, la latence totale d'aller-retour est souvent plus basse. Le point important est que la latence, toujours additive, est la somme de nombreux facteurs indépendants.

Une faible latence n'est pas toujours désirable. Elle a quelques inconvénients : le plus visible est la consommation accrue d'énergie ; le microprocesseur devant traiter plein de petits blocs de données audio, il est constamment actif et ne peut pas entrer en mode d'économie d'énergie (on peut penser aux bruits de ventilateur). De plus, si plus

d'une application est impliquée dans le traitement du son, chacune doit fonctionner pour un temps court et strict à l'intérieur de chaque cycle audio, ce qui donne une charge système bien plus élevée et augmente le risque de x-runs.

Cependant il y a quelques situations où une faible latence est réellement importante, parce qu'elles requièrent une réponse très rapide de l'ordinateur.

- *Jouer des instruments virtuels* : un long retard entre l'appui sur les touches et le son produit pas l'instrument dérègle la précision rythmique de la plupart des instrumentistes.
- *Écoute de contrôle logicielle* : si une chanteuse entend sa propre voix passant par deux itinéraires différents, son crâne et des casques, une latence importante peut être perturbante.
- *Effets en temps réel* : Ce cas est similaire au jeu d'instruments virtuels ; au lieu d'instruments virtuels ou de synthétiseurs on considère des effets numériques appliqués à des instruments réels. Une latence faible est importante lorsqu'on utilise l'ordinateur comme groupe d'effets (par exemple une pédale d'effets pour guitare) — en outre une synchronisation précise peut être importante lorsqu'on déclenche manuellement des effets sonores comme un «delay».
- *Mixage «live»* (en direct) : certains ingénieurs du son utilisent un ordinateur pour mixer des concerts en direct. Essentiellement c'est une combinaison de ce qui précède : écoute de contrôle sur scène, traitement d'effets et équalisation. C'est

en réalité plus épineux car l'on ne veut pas seulement une faible latence (l'audio ne doit pas être trop en retard sur l'interprétation) mais aussi une latence stable (qui minimise la gigue) pour les lignes à retard entre les enceintes frontales et du fond de la salle.

Dans beaucoup d'autres cas — comme la lecture, l'enregistrement, le re-recording, le mixage, le mastering, etc. — la latence n'est pas cruciale. Elle peut être relativement grande et facilement être *compensée*. Expliquons cette affirmation : lors du mixage ou du mastering, il importe peu que 10ms ou 100ms s'écoulent entre le moment où la touche déclenchant la lecture est enfoncee, et la sortie du son par l'enceinte. C'est la même chose en enregistrant avec un décompte utilisant un métronome.

Cependant, lors d'un enregistrement il est important que le son enregistré soit aligné en interne avec le son existant reproduit au même instant.

C'est là que la compensation de latence entre en jeu. Il y a deux possibilités pour compenser la latence dans une station audio-numérique (DAW) :

- *read-ahead* : Le DAW commence à lire en avance (par rapport à la position de lecture) de sorte à ce que lorsque le son arrive aux enceintes un court moment plus tard, il est exactement aligné avec le matériau qui est enregistré.
- *write-behind* : Dans le même but, l'audio entrant peut être retardé pour l'aligner de nouveau avec l'audio joué à ce moment.

1.3 Routage du signal

Pendant la production musicale, le son passe par diverses étapes. Elles peuvent correspondre à des connexions explicites, par exemple un microphone branché dans un pré-amplificateur branché sur une réverbération par l'ingénieur du son ; ou des connexions explicites à l'intérieur d'une boîte d'effets, qui sont opaques pour l'ingénieur du son. Les connexions explicites sont généralement faites en utilisant une baie de brassage (patchbay) qui permet de router arbitrairement une source vers différentes destinations. Certaines baies permettent de combiner les signaux, mais la manière courante de mélanger les signaux est d'utiliser une table de mixage audio qui permet de contrôler les niveaux individuels des signaux. Un aspect important est l'existence de sous-groupes, c'est-à-dire la combinaison de plusieurs sources apparentées dans un même *bus* pour partager les contrôleurs et les réglages d'effets ; par exemple un bus «batterie» qui regroupe les différents microphones d'une batterie en un mixage stéréo final.

La correspondance des signaux vers les bus n'est en aucun cas constituée de connexions exclusives. Par exemple, tous les microphones de la batterie peuvent être combinés à différents niveaux dans un bus pour une gestion commune du volume tandis que certains de ces microphones (par exemple les cymbales) sont aussi connectées à un second bus pour la réverbération et l'écho. Le routage effectif peut ainsi être un réseau complexe de connexions.

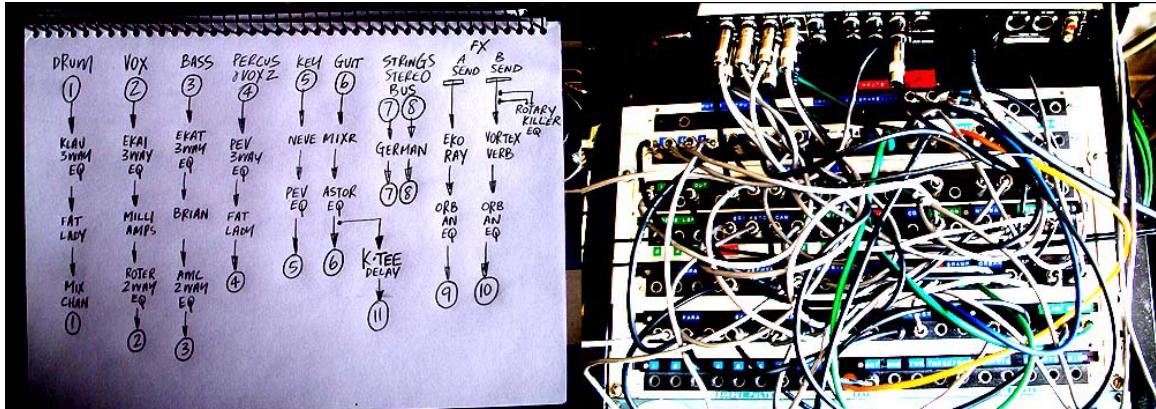


Figure fr-5 – Une baie de brassage audio analogique typique avec son diagramme de routage (source ekadek.com)

1.4 Compensation de latence dans les systèmes à routage arbitraire

Quand les signaux suivent différent chemins avant d'être combinés, l'on doit s'assurer qu'ils sont alignés temporellement. Des écarts aussi faibles qu'un échantillon peuvent produire des effets indésirables comme l'annulation de phase ou le filtrage en peigne. Comme l'on le voit à la figure *fr-3*, des différences plus grandes sont encore plus perceptibles.

Combiner un signal avec une copie de lui même retardée de $50\mu\text{s}$ (soit environ 10 échantillons à 192kHz) produit une interférence destructrice vers 10kHz (note : ce retard serait équivalent à placer un second haut-parleur 1,7cm derrière le haut-parleur qui joue le même son)[3].

Dans un système purement analogique ce n'est en général pas une préoccupation. Les signaux électriques se déplacent à la vitesse de la lumière (très proche de cette

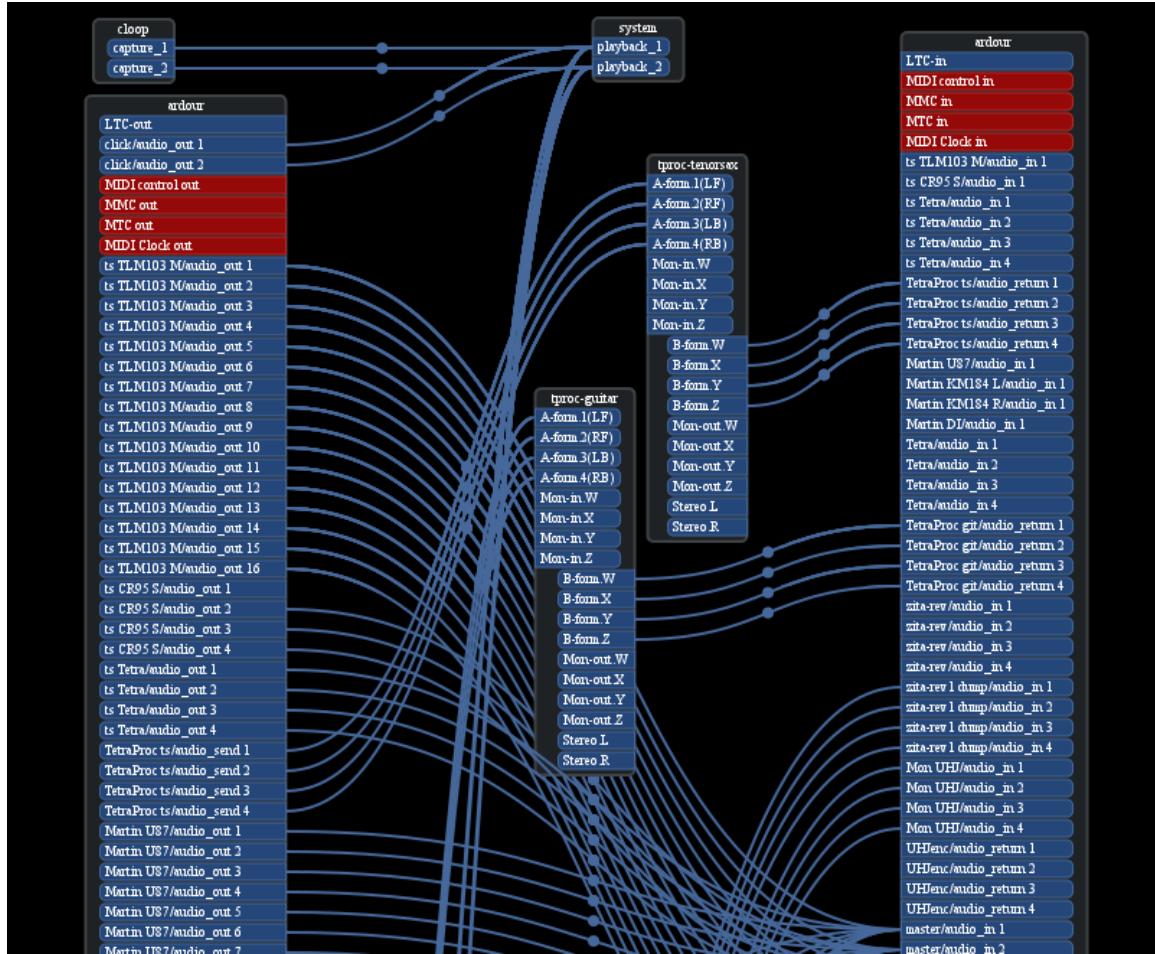


Figure fr-6 – Routage numérique — ici les connections des ports JACK sont visibles dans le logiciel ‘patchage’

vitesse en réalité), ce qui est plusieurs ordres de grandeur au dessus du spectre auditif.

Dans le monde numérique cependant, la plus petite granularité est un échantillon (par exemple $20\mu\text{s}$ à 48kHz). Les effets numériques introduisent souvent un retard. Par exemple un équaliseur requiert en général un ou deux échantillons pour réagir ; ou encore un limiteur utilise des échantillons qui suivent pour déterminer le niveau de signal.

La plupart des effets numériques ont une latence fixée qu'ils annoncent à la station

audio-numérique hôte du module de l'effet.

Un exemple classique est l'enregistrement d'une guitare et d'une voix avec plusieurs microphones, et un traitement du son enregistré par plusieurs effets qui induisent des retards de signal différents, avant de mixer le résultat pour la reproduction audio.

Des signaux suivant des routes différentes de la source à la sortie doivent être alignées pour éviter des bizarreries auditives. Pour contourner ce problème de nombreuses solutions matérielles limitent les possibilités de routage ou ne permettent d'ajouter des effets qu'à certains bus dans la chaîne globale du signal audio. Dans le cas de pédaux audio matériels la chaîne d'effets est généralement figée sans aucune possibilité de personnaliser le routage. Pour réduire la complexité beaucoup de stations audio-numériques proposent un nombre limité de départs/retours auxiliaires vers des canaux d'effets ; la plupart n'autorisent pas la sortie directe de pistes ou de bus voire fournissent des points d'entrée et de sortie uniques.

Dans un système professionnel de post-production, l'alignement de l'audio sur un code temporel externe (par exemple vidéo) ajoute un niveau supplémentaire de complexité. En première approche, il y a trois points possibles de synchronisation :

- *Aligner les pistes* : Les pistes individuelles, voire les pistes d'un même type (MIDI, audio) voient leur sortie alignée. Cela permet un routage du signal flexible mais augmente possiblement la latence totale.
- *Aligner les sous-groupes ou les bus* : Uniquement aligner les bus après l'obtention de leur signal par mixage.

- *Sortie principale* : Tous les signaux sont alignés sur une référence unique — la sortie principale.

Il existe une quatrième option qui est d'utiliser toutes les propositions ci-dessus ou seulement une partie si c'est pertinent à mesure que les connexions sont établies. Cette variante est le sujet de cette thèse : permettre un routage flexible entre n'importe quelles sources et n'importe quelles destinations, en incluant les sorties individuelles des pistes, tout en minimisant la latence globale.

1.5 État de l'art

Pour le moment la plupart des outils professionnels résolvent ce problème soit avec du matériel dédié fourni par le vendeur, ou en limitant arbitrairement les options de routage à la disposition de l'utilisateur. Ces approches et compromis sont spécifiques à chaque vendeur voire chaque implémentation.

Dans la plupart des cas, la compensation de latence est aussi réduite à la compensation du seul retard introduit par les effets (Plugin Delay Compensation), appliquée uniquement aux pistes lors de la lecture. Par exemple «l'algorithme de compensation décale chaque piste de la quantité requise»[4], les départs/retours sont ignorés, « les pistes de retour ne sont pas compensées »[4]. C'est similaire à l'approche d'Ardour 3.

Un autre produit impose que les routes soient linéaires, et a même une contrainte supplémentaire : « la compensation PDC automatique ne fonctionne pas pour les effets ayant plusieurs sorties »[5].

Au moins un vendeur n'offre la compensation PDC que pour ses produits haut de gamme ou professionnels, et nécessite des actions explicites de l'utilisateur pour l'activer[6].

Pour aligner l'audio lors de la capture, une suggestion courante est simplement d'enregistrer avec des petites tailles de tampon pour atténuer les conséquences, ou bien utiliser du matériel dédié fourni par le vendeur[6].

De nombreuses stations audio-numériques limitent aussi les possibilités de placement des insertions d'effets (uniquement après l'atténuateur par exemple), alors qu'il manque le concept de bus à d'autres. Les informations fournies dans les guides de l'utilisateur sont souvent éparses, et l'on discute du sujet dans les forums internet en proposant diverses solutions alternatives pour ajouter manuellement les lignes de retard à la bonne place.

Une part importante des applications non-professionnelles (ou *semi-pro*) ne comprennent pas du tout la latence.

2 Recherche et Développement

2.1 Contexte, motivation, et court historique d'Ardour

Le but de ces recherches est d'implémenter une compensation de latence professionnelle pour la station audio-numérique Ardour, et les fonctionnalités apparentées pour rendre Ardour encore plus adaptée aux production et post-production professionnelles.

Un des concepts essentiels d'Ardour est de faciliter un routage de n'importe quelle

source vers n’importe quelle destination ; cette fonctionnalité s’appuie sur le JACK Audio Connection Kit pour le routage inter-applications et n’est pas courante dans les autres stations audio-numériques.

Ardour a été créé en 1999 par Paul DAVIS ; c’était un enregistreur numérique sur disque dur sous licence du logiciel libre. La disponibilité du code source et la liberté assurée par la licence GPL en font un candidat idéal pour la recherche.

Ardour a un support rudimentaire pour l’alignement des pistes audio, lors de l’enregistrement et de la lecture, mais les options de routage diverses — en particulier les départs auxiliaires — ou encore les entrées live et la latence des bus ne sont pas prises en compte. Tout cela a provoqué des rapports de bogue récurrents remontant à l’année 2005 (Ardour 0.99).

Ardour 2 se concentrat sur un flux de travail purement audio ; cette limitation explique les nombreuses demandes d’amélioration pour rendre Ardour utilisable en dehors du champ de l’ingénierie audio classique. En particulier, l’auteur de cette thèse s’intéresse aux bandes son de film et aux procédures de travail apparentées.

Ardour 3 a vu son développement débuter en 2009 et a gagné diverses nouvelles fonctionnalités jusqu’à sa sortie en 2013. La plus marquante est la présence de pistes MIDI, mais l’évolution entre la version 3 et les précédentes est substantielle. De grosses parties — comme l’exécution en parallèle des modules de traitement du signal — ne sont pas directement visibles par l’utilisateur. D’autres ouvrent la voie de l’adoption d’Ardour par les studios de post-production professionnelle, comme : la

grille de routage, un système d'exportation restructuré, une section dédiée à l'écoute de contrôle, et la possibilité de synchroniser Ardour avec un code temporel (timecode) externe.

Pendant ce développement, l'auteur a contribué grandement aux algorithmes de poursuite et synchronisation temporelles, a implémenté une frise chronologique vidéo (video-timeline) ainsi que les infrastructures nécessaires, et a réécrit la mesure des niveaux pour la conformer aux standards existants. Ardour 3.5 remplit plusieurs critères pour la production de bandes son cinématographiques, et a déjà été utilisé depuis sa sortie pour enregistrer, éditer et mixer des longs métrages et divers courts-métrages.

Malgré le développement rapide de nombreuses fonctionnalités qui a mené à Ardour 3, une compensation de latence digne de ce nom est toujours une fonctionnalité majeure qui manque à l'appel et doit être abordée.

Bien qu'une bonne partie du développement d'Ardour 3 fut loin d'être triviale et que de nombreux mécanismes complexes furent implantés à l'aide de discussions et de planification de bas niveau, il apparut clairement qu'une compensation de latence décente ne peut pas être construite de la même façon. Le style traditionnel de développement est de faire progresser des modules dédiés et parties ciblées.

La compensation de latence concerne pour sa part plusieurs zones majeures réparties dans le logiciel entier :

- Le *backend* du moteur audio (ports d'entrée/sortie).
- La synchronisation de lecture (position de lecture de la session, timecode externe

et variation de vitesse).

- Les pistes audio et MIDI (lecture en avance, écriture en retard, tampons et caches).
- L'interface de gestion de la latence des modules d'effet.
- Le routage du signal, les chaines parallèles et les modifications du système de routage.
- Le graphe d'exécution du traitement.

Chacune de ces parties est elle-même un module compliqué, et leurs interactions sont plutôt complexes. En réalisant les prototypes, il est devenu clair qu'une compensation de latence correcte requiert une planification d'envergure ; c'est la raison d'être de cette thèse.

2.2 Réalisation des prototypes

En 2013-2014, à la fin du cycle de développement d'Ardour 3.x un prototype a été développé qui intégrait les bus ainsi que la latence des effets et synthétiseurs MIDI dans le graphe de traitement de la station audio-numérique. C'était pour une grande part un projet de recherche et d'expérimentation pour explorer la faisabilité et identifier les briques nécessaires ou les zones dont l'architecture devait être revue.

Le mécanisme nécessite d'analyser le graphe de routage pour trouver une solution qui induit une latence additionnelle minimale. Tous les signaux s'alignent sur une

référence temporelle commune, le contrôle global de lecture, ou tête de lecture. Des chemins de signal n'ayant aucune dépendance l'un envers l'autre peuvent être traités en parallèle et gérés en bloc. Pourtant, même les chemins indépendants dans le graphe du signal doivent être alignés dans le temps ; c'est pourquoi le graphe d'ordonnancement du traitement n'est qu'en partie utile pour calculer les latences.

Un mécanisme d'alignement de toutes les pistes l'une par rapport à l'autre, ainsi qu'une contrainte d'alignement d'au moins une piste en termes absolus sont nécessaires pour assurer un synchronisme.

L'alignement de pistes indépendantes s'obtient en retardant le signal du chemin le plus court. Cependant, si l'un des signaux n'est pas une entrée live mais est lue depuis le disque, le pointeur de lecture peut être décalé pour appliquer un mécanisme de lecture en avance. Au lieu de retarder plus l'audio entrant pour l'aligner avec des pistes déjà enregistrées, la lecture en avance permet d'obtenir une latence globale plus basse dans le système.

Pour calculer le graphe de latence, l'on doit pouvoir répondre aux deux questions suivantes pour chaque nœud (port audio) du graphe :

- Combien de temps les données lues du port ont-elles mis pour arriver d'un bord du graphe (capture) ?
- Combien de temps le données écrites dans le port vont-elles mettre pour atteindre le bord du graphe (reproduction) ?

En fonction du routage, il peut ne pas y avoir initialement de réponse seule et

unique à ces questions, mais un intervalle minimum/maximum de latences dans chaque direction. Comme présenté à la figure *fr-4*, la latence globale est une combinaison des latences systémique et interne, et un alignement absolu requiert des outils pour quantifier les latences externes.

Le routage est représenté par un graphe orienté acyclique, qui est ordonné topologiquement et parcouru ensuite dans les deux directions pour calculer les latences. Finalement, des lignes à retard sont ajoutées pour équilibrer les latences de chaque noeud en fonction des points de synchronisation choisis.

Un tel prototype a été construit pour la station Ardour 3, comme une démonstration de faisabilité comprenant la compensation tant des latences du matériel que celles des bus et pistes audio, en utilisant le bus master comme point de synchronisation commun (voir la figure *fr-7*).

De nombreuses briques logiques créées lors du prototypage — par exemple des lignes à retard audio et MIDI — peuvent être utilisées pour l'implémentation finale ; cependant le but principal du prototype était d'explorer et d'identifier les zones de difficultés.

L'implémentation initiale de la démonstration de faisabilité a mené au développement de nombreux pré-requis et changements d'architecture en préparation d'une implémentation complète. Ardour 4 s'est détaché de JACK comme *backend* obligatoire et a mis en place une abstraction de toutes les entrées/sorties audio ainsi que les calculs de latence matérielle. Ardour 4.x inclut aussi un outil intégré pour mesurer et calibrer les latences systémiques audio et MIDI, et c'est durant le cycle de développement de

la version 4.x que de nombreuses briques internes de l'implémentation ont été mises à jour pour ouvrir la voie de l'alignement temporel.

Divers outils et utilitaires externes ont été créés lors de l'étape de prototypage ; certains sont décrit dans les annexes. En particulier, des plugins pour simuler une latence, de même que des outils de mesure pour permettre la validation. Un autre ajout notable est l'intégration d'un moteur de scripts qui permet l'introspection et l'accès direct à l'intérieur du système. Cela s'est avéré inestimable pour le débogage et la rectification des cas particuliers.

Une étape cruciale pour permettre la compensation de latence fut la séparation des entrées/sorties du disque d'une piste vers des processeurs dédiés, ce qui a été entrepris à la fin du processus de développement d'Ardour 5.

Ce n'est pas avant la fin du cycle Ardour 5.x en 2017 qu'a démarré une implémentation effective de la compensation de latence sur l'ensemble du graphe.

Un système complet incluant les pistes MIDI aussi bien que les données génériques (automatisation) et divers points de synchronisation est décrite à la partie IV et la branche «master» du dépôt Git du futur Ardour 6 contient déjà cette fonctionnalité.

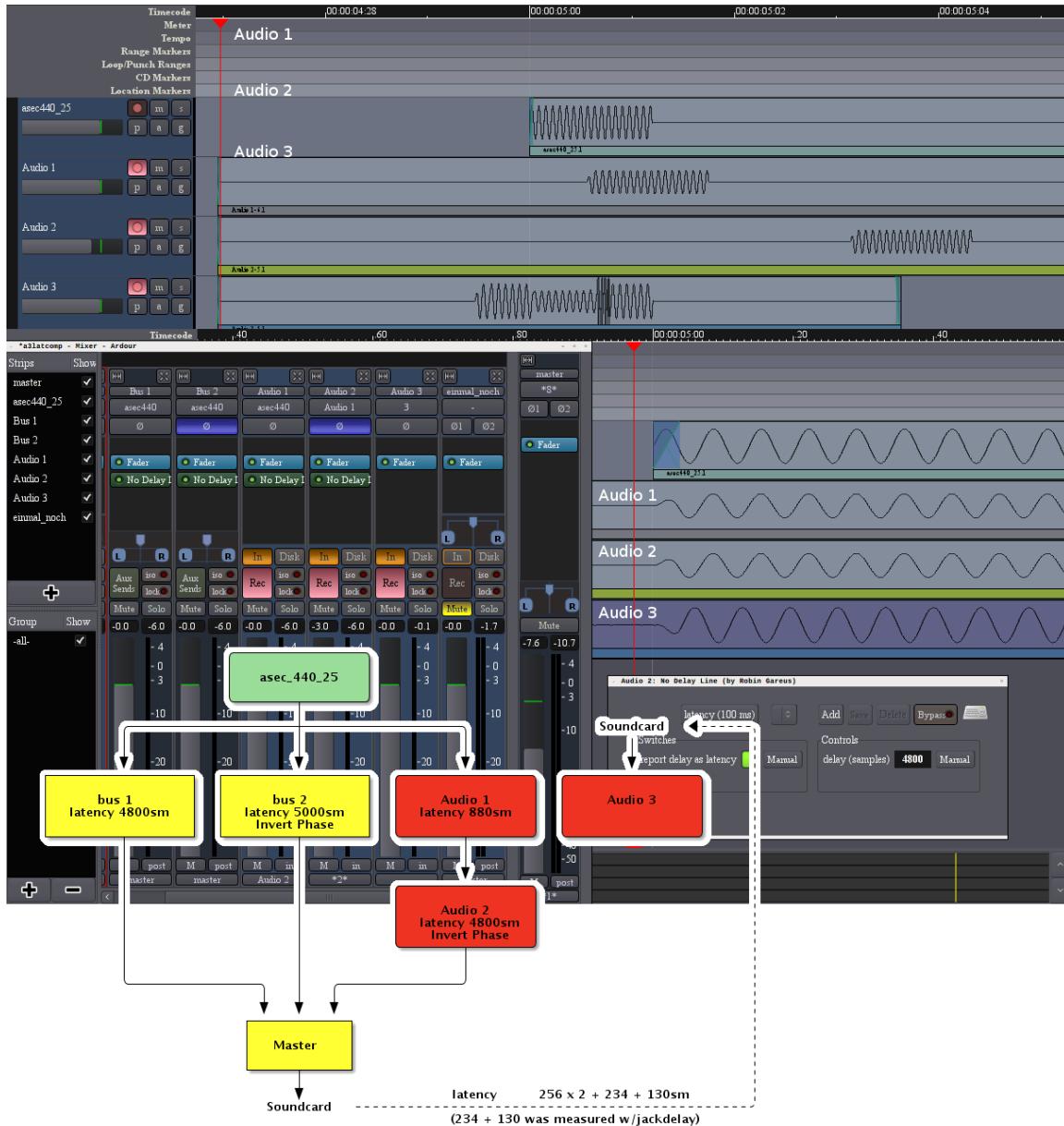


Figure fr-7 – En haut : état précédent d’Ardour 3, avant le travail sur la compensation de latence. En bas : prototype démontrant la faisabilité d’une compensation de latence correcte. L’encart montre le diagramme de routage (vert : lecture ; rouge : pistes d’enregistrement ; jaune : bus)

Deuxième partie

L'écosystème audio numérique

Pour pouvoir compenser la latence, il faut connaitre quelle quantité il y a. Ce chapitre détaille les diverses sources de latence, et les méthodes de consultation ou de mesure de cette latence.

3 Les sources de latence

3.1 La propagation du son dans l'air

Puisque le son est une perturbation mécanique dans un fluide, il se propage à une vitesse relativement faible d'environ 340m/s. Ce n'est pas directement pertinent pour les problèmes de compensation de latence, mais une configuration décente des microphones et des hauts-parleurs est un pré-requis.

Il est en revanche notable que de nombreux musiciens apprennent à accepter une latence fixe donnée. En particulier dans les instruments mécaniques — comme un orgue d'église — la mécanique elle-même impose une latence. En comparaison, le son d'une guitare acoustique ou d'un piano (distance : $\approx 50\text{cm}$) a une latence d'environ une à deux millisecondes à cause du temps de propagation du son entre l'instrument et l'oreille, alors qu'un orgue peut mettre jusqu'à une seconde à réagir.

La vitesse du son aussi significative dans les grandes salles de concert où de nombreux jeux d'enceintes sont présents. Des lignes de retard appropriées doivent être ajoutées

pour aligner le signal des différents hauts-parleurs avec le son venant de la scène.

3.2 Les conversions analogique-numérique et numérique-analogique

Les signaux électriques se déplacent rapidement (avec une célérité de l'ordre de celle de la lumière) donc leur temps de propagation est négligeable dans ce contexte, mais les conversions entre les domaines analogique et numérique prennent un temps comparativement long à s'effectuer. Le retard dû à la conversion est en général en dessous d'une milliseconde pour un équipement professionnel.

3.3 L'architecture des ordinateurs

La source principale de latence est le cycle de traitement. Bien que le nombre d'échantillons traités par cycle est en général un choix de l'utilisateur, diverses contraintes sont imposées par le matériel autant que par l'architecture et le pilote audio.

Tout d'abord et surtout, le nombre d'échantillons par période est en général une puissance de deux. Ce n'est pas une contrainte stricte, mais une conséquence directe de l'interface entre le CPU et le Bus, où l'espace d'adressage est géré en binaire. Il n'y a souvent aucun intérêt à choisir un nombre entre deux puissances de deux car la quantité réellement transférée est arrondie à la puissance de deux supérieure. La charge microprocesseur pour transférer, disons 86 échantillons est égale à celle pour 128 échantillons alors que l'échéance arrive plus tôt.

Les cartes-son moyenne gamme classiques (par exemple les Intel HDA) imposent une limite basse de 32 échantillons par cycle, mais en fait en pratique elles ne peuvent pas

lire et écrire de manière fiable un flux de morceaux durant moins d'une milliseconde.

Des périodes de 256 échantillons par cycle sont courants dans le matériel de gamme moyenne. Des équipements professionnels existent ayant des tailles de fragment audio plus basses, mais pour plein de raisons pratiques il n'est pas réaliste d'avoir moins de 8 échantillons par période.

Dans un ordinateur à usage générique, divers autres facteurs peuvent interférer avec les faibles latences. Du côté matériel ils sont principalement liés à l'économie d'énergie : par exemple l'échelonnement des fréquences du microprocesseur ou du bus peuvent avoir un impact sur le temps de réponse minimum. De plus, des interfaces vidéos ou WiFi, des périphériques USB ou leurs pilotes peuvent accaparer les ressources de l'ordinateur pendant une longue durée s'ils sont mal conçus, et empêcher l'interface audio de suivre la cadence du flux de données.

La plupart des machines modernes proposent aussi une gestion du système et des vérifications de l'état du matériel (SMI : system management interrupts), qu'on ne peut pas désactiver sans risque. Ces interruptions peuvent prendre un certain temps à traiter.⁵

3.4 Le système d'exploitation

L'ordinateur doit recevoir et transmettre des données audio à intervalles réguliers, ce qui concerne de nombreux sous-systèmes (voir la figure *fr-4*).

⁵Des cartes-mères qui n'envoient jamais de SMI sont préférables pour un système audio temps-réel fiable ; c'est aussi une exigence des systèmes de transaction d'action en temps-réel qui ont le même genre de soucis de latence.

L’interface audio signale la fin d’un cycle en déclenchant une «interruption» (IRQ). Le système d’exploitation doit compléter sa tâche actuelle jusqu’à un point où il peut prendre en charge l’interruption. Les systèmes d’exploitation préemptifs — comme real-time Linux — ont un temps de réponse plus court car ils permettent de préempter le processus en cours pour gérer les tâches de haute priorité.

Basculer d’une tâche à l’autre dans un système d’exploitation implique l’échange du contexte de processus ce qui ajoute un délai supplémentaire. Changer de contexte a un cout fixe — le temps nécessaire pour sauvegarder et restaurer les registres, inversement proportionnel à la vitesse du CPU — et un cout variable qui dépend de la mémoire impactée — vidage du tampon de traduction d’adresses (TLB), maintien de la cohérence du cache. L’incidence est d’environ $10\mu\text{s}$ à $100\mu\text{s}$ [7].

3.5 Le traitement numérique du signal

De nombreux algorithmes traitant l’audio nécessitent un accès au contexte (les échantillons passés ou futurs) pour réaliser leurs calculs. L’audio entrant est mis en tampon et le traitement ne commence que lorsqu’assez de données ont été collectées. Au final, cela retarde le signal entrant.

3.6 Période de traitement

Traiter le signal, appliquer divers effets ou synthétiser des sons déclenchés par un évènement entrant impliquent d’effectuer des opérations mathématiques sur le signal, ce qui peut être très exigeant même pour les machines puissantes.

La plupart des CPU conçus depuis le milieu des années 90 proposent des instructions SIMD (single instruction, multiple data) qui permettent de traiter par bloc de larges fragments de données de manière efficace. Les plus connues sont les instructions SSE (Streaming SIMD Extensions), et leurs successeurs divers — SSE2, SSE3, SSE4 et AVX — sont depuis monnaie courante. Les instructions SIMD améliorent nettement les performances lorsqu'on effectue la même opération sur de multiples données. Traiter 32 échantillons d'un coup sera bien plus rapide que traiter 32 fois un seul échantillon. Plus la taille du bloc est grande, moins de temps CPU est nécessaire pour traiter la même quantité globale de données⁶.

La taille optimale de ce bloc dépend de l'algorithme de traitement et de considérations sur la performance et le cout.

C'est en général la cause principale de latence lorsqu'on utilise un ordinateur, mais elle est prévisible et peut être optimisée.

4 Mesurer les latences d'entrée/sortie

À cause de l'interaction complexe entre les parties mentionnées à la section 3 la seule procédure fiable pour établir la latence d'un système est de la mesurer. Une exception notable à ce fait est la situation d'un vendeur seul responsable de tous les composants du système entier, en incluant le matériel, les micro-logiciels (firmware) et la partie

⁶Bien que la relation entre la taille du bloc et le temps de traitement est monotone, elle n'est pas linéaire pour les petites tailles de blocs ; elle s'approche d'une relation linéaire au dessus de mille à huit mille échantillons.

logicielle. Certaines machines Apple Macintosh® font par exemple partie de cette catégorie.

C'est plutôt courant que la latence systémique additionnelle soit une fonction de la taille du tampon. La figure *fr-8* montre cet effet pour un périphérique USB. La mesure est effectuée grâce à une boucle fermée audio en émettant un signal test et en le capturant de nouveau après un aller-retour à travers la chaîne complète.

La boucle peut être fermée de différentes manières :

- Placer un haut-parleur près d'un microphone. Cette méthode est rarement utilisée, car la latence due à la propagation dans l'air est bien connue et n'a pas besoin d'être mesurée.
- Connecter la sortie de l'interface audio à son entrée en utilisant un câble. Cela peut être une boucle numérique ou analogique, en fonction de la nature des entrées/sorties utilisées. Une boucle numérique ne prendra pas en compte la latence de conversion analogique/numérique.

Utiliser un signal continu de partiels inharmoniques permet de mesurer la latence effective avec une précision inférieure à l'échantillon en inspectant la phase relative de chacune des tonalités de test. Un outil appelé *jack delay*[8] pouvant atteindre une précision d'un millième d'échantillon a été utilisé pour la mesure d'une PreSonus® Audiobox 1818VSL (voir la figure *fr-8*). Dans Ardour, le mécanisme de calibration des périphériques audio est basé sur le code de *jack delay*.

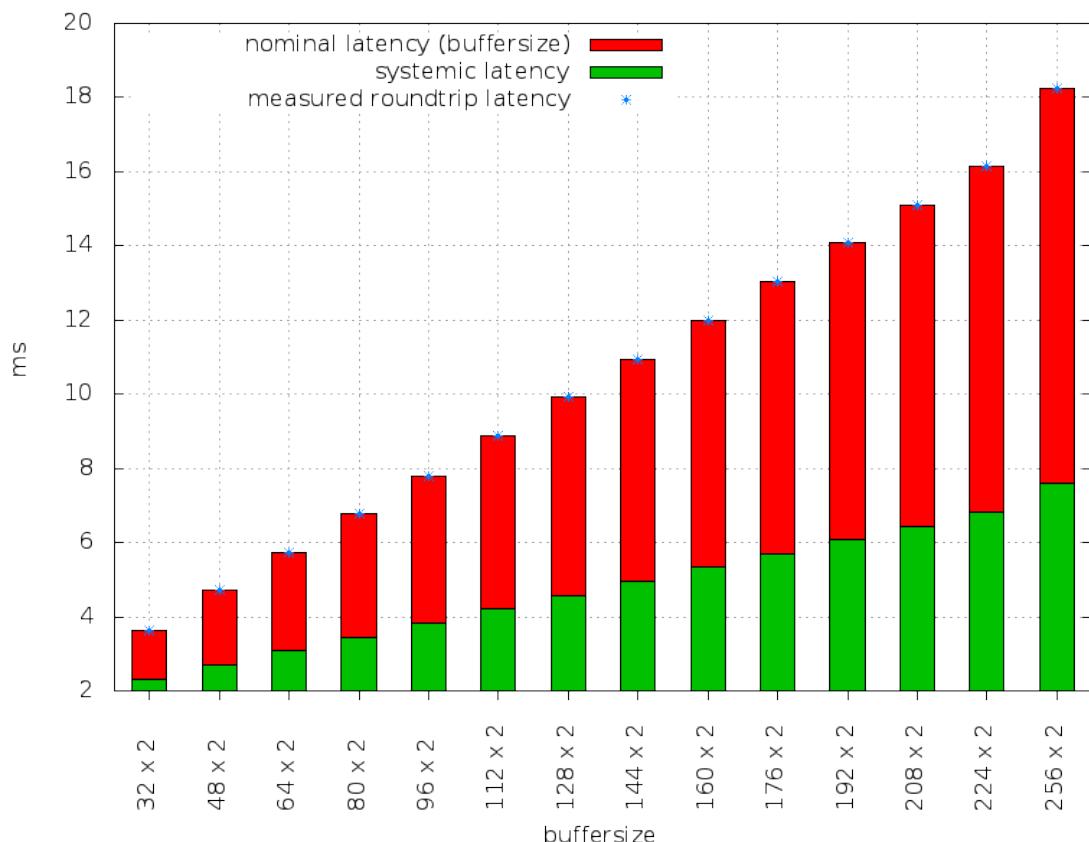


Figure fr-8 – Latence d'aller-retour mesurée sur une Presonus Audiobox VSL1818 à une fréquence d'échantillonnage de 48kHz.

Connaitre précisément la latence d'aller/retour est essentiel pour un travail d'enregistrement en studio.

La boucle fermée décrite ci-dessus correspond à la situation d'une station audio lisant du contenu existant ou générant un battement de métronome et d'un musicien jouant par dessus ce contenu existant pour enregistrer son instrument. L'audio nouvellement capturé sera en retard d'un cycle d'aller-retour comparé au contenu existant reproduit.

Le problème se complique davantage : la latence systémique peut ne pas être

reproductible, et varier à chaque fois que le sous-système audio d'un PC est réinitialisé. C'est principalement le cas des périphériques USB : le bus USB impose des contraintes temporelles supplémentaires (les commandes peuvent être mises en attente à des intervalles de 1ms au plus, la taille des paquets de données USB est fixée). La solution adoptée par le module snd-usb du noyau Linux est d'ajouter un tampon circulaire pour séparer les tampons de transfert et d'exécution, ce qui ajoute un délai fixe, qui change à chaque fois que le périphérique audio est ouvert. Le code source d'autres pilotes n'est pas librement accessible, mais les mesures suggèrent que le même mécanisme est souvent utilisé. Ce n'est pas un problème tant que le périphérique reste ouvert pour utilisation par le pilote après la mesure de latence.

À l'heure d'écriture de ces lignes, la seule interface de programmation (API) d'un système d'exploitation qui permette de demander directement la latence matérielle (sans la mesurer explicitement) est CoreAudio sur OS X. Cependant, la précision des valeurs fournies dépend de l'interface audio ainsi que de son pilote et peut être imprécise pour les matériels tierce-partie.

Si une interface a plusieurs flux d'entrée/sortie, par exemple les cartes-son avec MIDI, les flux sont généralement non alignés et des latences systémiques différentes s'appliquent pour les flux individuels. Une exception notable est la norme IEEE1394 (Firewire), qui permet un multiplexage de paquets isochrones pour combiner plusieurs flux synchronisés dans la même tranche de données.

De même, lorsqu'on combine plusieurs interfaces dont la fréquence d'horloge est

synchronisée, la latence systémique peut quand même être différente pour chacun des périphériques.

La morale est que la latence systémique est une variable qui doit être mesurée et calibrée au moment de l'utilisation.

5 Plugins

Dans la plupart des stations audio-numériques le traitement du signal n'est pas effectué par la station elle-même, mais le système a une architecture modulaire. Le concept date du début des années 60 où dans les synthétiseurs modulaires, les modules n'étaient pas branchés de manière fixe mais connectés ensemble à l'aide de câbles ou une baie de brassage matricielle pour créer un son complexe. Les équivalents numériques de tels modules sont les plugins : des logiciels tierce-partie additionnels chargés dans la station (l'hôte), prévus pour étendre les capacités de l'environnement.

Les plugins sont typiquement considérés comme des périphériques virtuels par analogie avec les périphériques matériels comme les réverbérations, les compresseurs, les délais, synthétiseurs, etc. qui peuvent être branchés ensemble de manière modulaire. Cela permet aux éditeurs de stations hôtes de se concentrer sur la convivialité et l'efficacité de leurs produits en laissant les éditeurs spécialisés focaliser leur attention sur le traitement numérique du signal[9]. « Les plugins audio fournissent en général leur propre interface utilisateur, qui contient souvent des objets graphiques utilisés pour contrôler et visualiser les paramètres audio du plugin »[10].

L'algorithme de traitement utilisé par le plugin peut induire de la latence. Les raisons sont multiples et peuvent correspondre à la nature intrinsèque de l'effet comme à son implémentation particulière.

Comme le traitement du signal par les plugins est optionnel et distribué de manière non uniforme dans le flux du signal, c'est la source principale de la complexité de la compensation de latence.

5.1 Quelques exemples de plugins latents

Les plugins latents sont de deux sortes : les uns produisent des retards très courts, imperceptibles en tant que tels, mais qui produisent les interférences de phase ; les autres provoquent des retards plus importants.

Dans la première catégorie l'on retrouve les équaliseurs, les distorsions, les guides d'ondes et les effets de façonnage du son similaires. Bien qu'il soit possible d'implémenter de nombreuses variantes de ceux-ci en utilisant un traitement du signal sans latence, plusieurs modèles nécessitent quelques échantillons de latence pour réaligner la phase du signal, ou à cause d'implémentations à rétro-contrôle z^{-1} . Les équaliseurs à phase linéaire sont un exemple où le signal est retardé pour aligner la phase.

La deuxième catégorie se compose d'effets qui effectuent un traitement couteux et transigent sur l'usage d'un tampon pour diminuer la charge processeur, ainsi que d'effets qui doivent filtrer ou analyser l'audio avant de le traiter. Le retard est de l'ordre de 20Hz (2400 échantillons à la fréquence d'échantillonnage de 48kHz), pour inclure les basses fréquences du contenu.

Un exemple de la deuxième catégorie est le limiteur à pré-lecture (lookahead). Cet effet limite les pics d'un signal audio avec en général des courbes douces d'attaque et de retour. Si l'effet répond trop vite (temps d'attaque court) et atténue le signal rapidement, il y aura des artéfacts audibles. Un limiteur à pré-lecture remplit une mémoire tampon sur une certain intervalle glissant de temps, qu'il utilise pour détecter le niveau global en avance sur le traitement de l'audio, ce qui permet d'utiliser une réponse retardée et des changements lisses du gain.

De même, tout effet qui requiert du contexte sera latent. Entrent dans cette catégorie les réducteurs de bruit ou les pitch shifters (qui changent la hauteur du signal). Ils opèrent sur des fragments de contenu ; la taille de ces fragments est en général fixée et indépendante de la taille du tampon de traitement.

De nombreux effets qui réalisent des multiplications de matrices comme les simulations de baffles ou d'enceintes par convolution utilisent un noyau de convolution fixe dans l'implémentation. La nature de ces implémentations fixe une taille de bloc et une mémoire tampon est requise pour aligner ces blocs avec la taille du tampon de traitement de l'hôte. Ce sont à nouveau des compromis d'implémentation et la plupart des effets de convolution produisent de la latence.

La latence n'est pas forcément une valeur fixe mais peut dépendre de la configuration et des paramètres. Les plugins avec un rétro-contrôle court ont en général un nombre fixe d'échantillons de latence, alors que les processeurs qui doivent analyser le signal audio ont souvent un retard qui est une fraction de la période d'échantillonnage. Dans

de rares cas la latence dépend de la taille du tampon de l'hôte.

Une pratique courante qui requiert une compensation de latence est la *compression parallèle*. C'est une technique où l'on mixe le signal neutre avec un version hautement compressée de lui-même. Contrairement à la compression normale, cela préserve les attaques et les transitoires dans le canal clair, mais le retard induit par le compresseur — qui est généralement dans un bus et peut être branché à plusieurs sources — doit être compensé.

5.2 Les standards de plugins

Pour des raisons historiques et commerciales, une variété de standards existent pour les plugins. D'un point de vue abstrait, ils sont très similaires et ont la même fonctionnalité. Cependant, en regardant plus en détail on observe de nombreuses différences techniques.

Les standards de plugins les plus courants sont VST (Virtual Studio Technology, Steinberg), Audio Unit (Apple, OSX only) et LV2 (sous licence ISC). Une étude générale de ces standards dépasse le cadre de cette thèse.

Tous les standards fournissent des moyens aux plugins pour indiquer leur latence à l'hôte, mais des différences subtiles existent :

- Les plugins *Audio Unit* déclarent leur latence en secondes (en virgule flottante double précision). Ils sont interrogés à ce sujet de manière asynchrone une fois instanciés, et peuvent notifier l'hôte d'un changement de leur latence en utilisant une fonction de rappel (callback)⁷

⁷Lors du prototypage il a été découvert que lorsqu'on leur demande leur latence certains plugins

- La spécification *VST* permet aux plugins de fournir une valeur fixée, initialisée à l’instanciation. La valeur, appelée `initialDelay`, est donnée en échantillons.
- Dans le cas du standard *LV2*, la latence est un port de contrôle en sortie tout à fait classique, et n’est pas nécessairement constante. Le standard spécifie explicitement qu’un appel de la fonction de traitement avec une taille de tampon nulle enjoint le plugin de recalculer sa latence : « Un cas particulier : lorsque `sample_count` vaut 0, le plugin doit mettre à jour tous les ports de sortie qui représentent un instant donné du temps (c'est-à-dire les ports de contrôle, mais pas les ports audio). C'est particulièrement utile pour les plugins latents, qui doivent mettre à jour leur port de sortie indiquant la latence de sorte que l'hôte puisse pré-calculer la latence»[11]. La latence est déclarée en échantillons, indépendamment de la fréquence d'échantillonnage, et est disponible sans compromettre le temps-réel.

Dans tous les cas, il est recommandé que l’hôte interroge le plugin sur sa latence après son activation car les paramètres utilisés pour créer l’instance peuvent influencer la latence de l’effet.

Lorsqu’on ajoute un plugin, en général il ajoute de la latence immédiatement. Il faut du temps pour que le signal passe à travers et soit traité avant d’être disponible en sortie. Cela mène la plupart du temps à des discontinuités de signal et des artéfacts audibles.

propriétaires effectuent des vérifications de licence possiblement longues, et sans fiabilité temps-réel. La latence ne peut ainsi être demandée qu’en réponse à une notification par la fonction de rappel.

Pour pallier ce défaut, LV2 va plus loin. Puisque la spécification est extensible, un mécanisme pour l'insertion sans clic a été ajouté ; il doit être supporté spécifiquement par l'hôte : un plugin est ajouté en mode court-circuité. Il peut pré-remplir ses tampons et commencer le traitement alors que le signal audible n'est pas traité. Remettre ensuite le plugin en circuit peut se faire avec une transition fluide sans clic. Il est important que le plugin lui-même fournisse ce mécanisme ; l'hôte ne peut savoir quelle est la manière correcte de désactiver ou réactiver le traitement : cela peut-être un fondu enchainé, ou encore accroître un paramètre en interne à un rythme spécifique, ou encore une transition entre deux états internes. Puisque les plugins LV2 exposent leur latence avec un port de contrôle en sortie classique, une abstraction élégante de ce mécanisme est possible bien qu'il impose une complexité accrue de l'hôte pour gérer les changement éventuels de latence.

Troisième partie

Architecture et modules

6 Vue d'ensemble

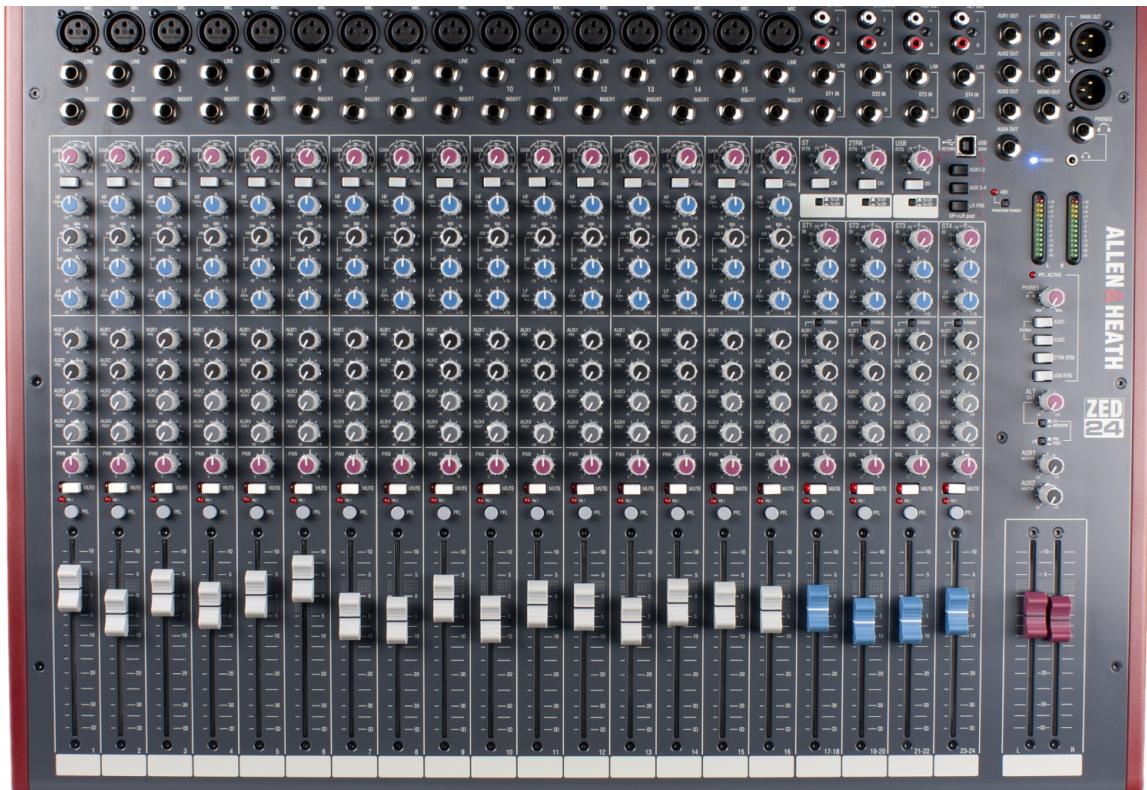


Figure fr-9 – Table de mixage Allen & Heath ZED-24.

Pour décrire et construire les mécanismes de la compensation de latence, on a besoin de modéliser toutes les composantes impliquées, et la première étape du processus de modélisation est d'identifier les structures communes dans le plan de conception.

Une station audio-numérique distingue conceptuellement les pistes et les bus. Les

pistes sont utilisées pour enregistrer et jouer du contenu, alors que les bus sont utilisés pour sommer et mixer. Une piste alimente en général un ou plusieurs bus.

À un niveau élémentaire, toutes ces données qui transitent dans le système représentent une *voie* (channel-strip), une boîte noire avec une entrée son et une sortie son qui peuvent être connectées arbitrairement. Le terme « voie » a une connotation historique venant des tables de mixages analogiques où le signal circule de haut en bas dans chaque voie qui comportait le traitement (en général l'égalisation et la balance). Dans une table de mixage, de multiples voies sont accolées horizontalement ce qui explique le terme de voie (par analogie avec les voies de circulation qui subdivisent une route). Voir la figure *fr-9*.

Ardour suit cette convention : l'objet interne de base est le **Stripable**, bien que dans le cas d'Ardour ce ne soit qu'un objet virtuel fournissant un état générique et une abstraction utile pour divers buts de l'interface utilisateur (sélection, tri) dans laquelle ce *Stripable* est représenté comme une voie ou une bande. Une voie peut aussi être un objet de seul contrôle, qui n'effectue aucun traitement de données réel (par exemple les VCA⁸).

7 Architecture

Une voie qui traite des données est mieux décrite par le terme **Route** : un objet abstrait qui possède les mécanismes de transmission de signal communs aux bus et

⁸Voltage Controlled Amplifier, ou amplificateur commandé en tension : un contrôle de gain qui ne fait que contrôler indirectement le niveau d'autres atténuateurs.

pistes.

Les *Routes* ont aussi bien des **Ports** d'entrée que de sortie. Bien qu'à l'origine les voies aient eu une entrée monophonique et une sortie stéréophonique, la route abstraite n'est pas ainsi limitée. La motivation du terme « route » vient de sa fonctionnalité principale : décrire le transport, le *routage* de l'audio ou du MIDI d'un point à un autre. La figure *fr-10* illustre la brique fondamentale qu'est l'objet *Route*.

Un bus est essentiellement une *Route*. Alors que *Route* désigne le modèle abstrait qui décrit les fonctionnalités et les interfaces, le terme «bus» permet de préciser la sémantique en fonction de l'usage. Comparée à un bus, une piste a des fonctionnalités supplémentaires correspondant à ses capacités d'enregistrement et de lecture.

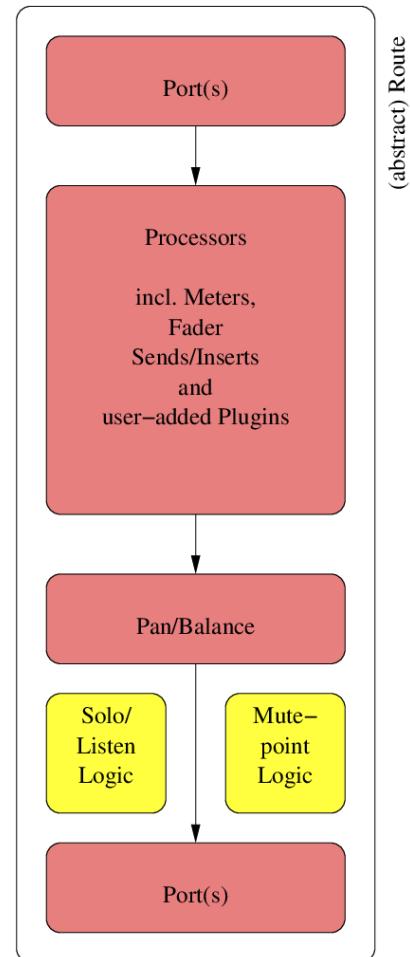


Figure fr-10 – Abstraction conceptuelle de base de l'objet *Route*

Terminologie orientée objet pour les classes :

Lorsque l'on écrit « Foo *est-un* Bar », alors Foo est une classe dérivée de Bar.

Si « Foo *a-un* Bar », alors Foo a au moins un membre qui est de type Bar.

De même pour « Foo *a-des* Bar ».

Forts de cette terminologie, nous pouvons définir un ensemble d'objets de base dans

une station audio-numérique :

- Une Route *est-un* Stripable ;
- Un Track *est-une* Route ;
- Une Route *a-un* Port d'E/S ;
- Une Route *a-un* Processor
- Un Track *est-une* Route qui *a-des* DiskProcessors (lecture/écriture).

Une *Route* est atomique, indivisible. Le traitement du signal à l'intérieur de la Route est linéaire et ne peut être interrompu de l'extérieur. Si une *Route* dépend de la sortie d'une autre *Route*, elle doit attendre que cette dernière ait terminé son traitement.

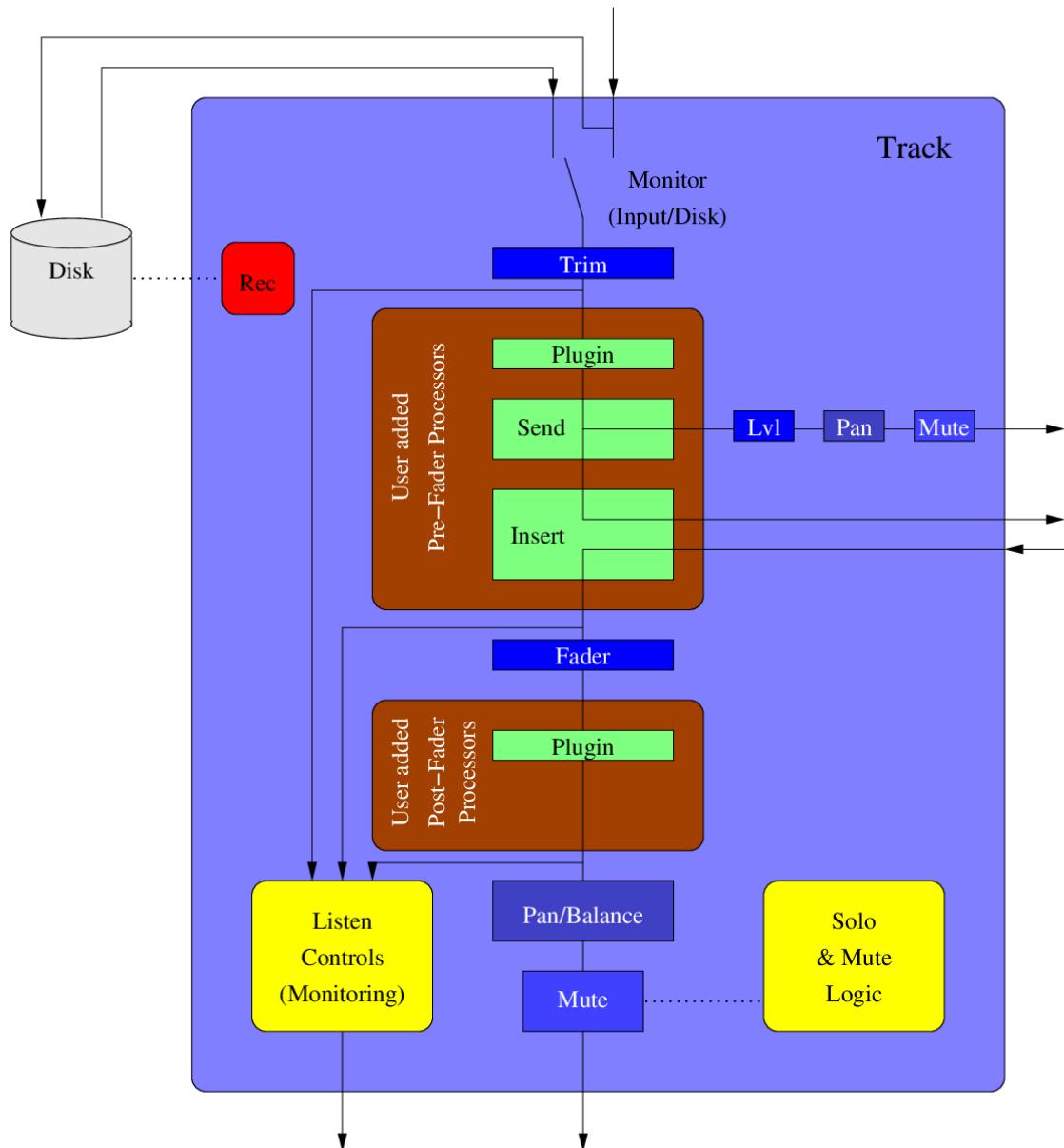


Figure fr-11 – Ancienne conception d'un objet piste *Track* (Ardour 3-5). Les *Processors* en vert ont été ajoutés par l'utilisateur. Le bleu indique les étapes d'ajustement du gain qui requièrent des *Processors* en interne. On note que les entrées/sorties disques ainsi que l'écoutte de contrôle sont traitées à part avec un aiguillage à l'entrée de la piste. Cette architecture-ci ne se prête pas à la compensation de latence.

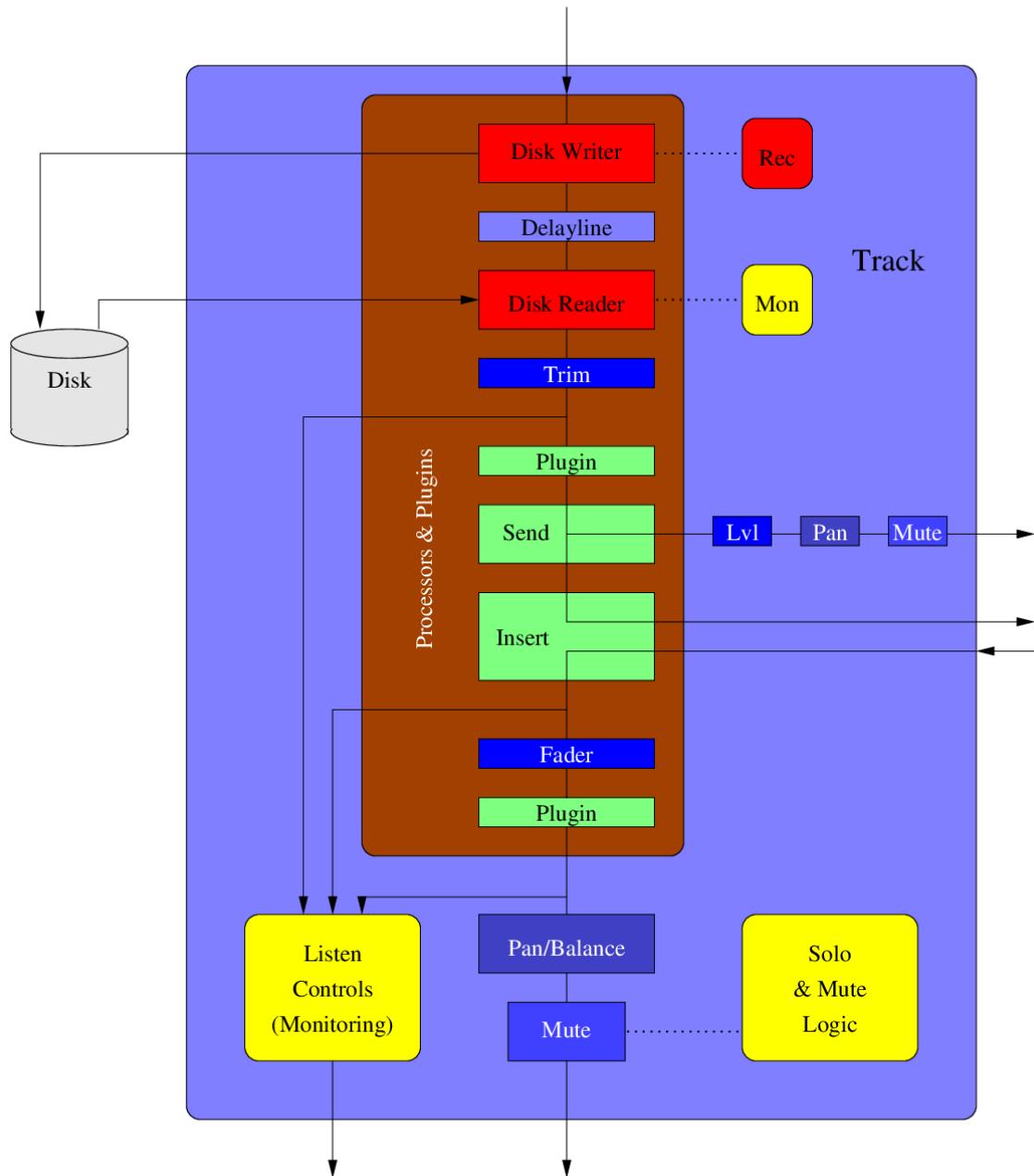


Figure fr-12 – Abstraction révisée d'un objet *Track*, où tout est géré par des *Processors*. Il n'y a pas d'ordre strict, les processeurs peuvent être librement réordonnés avec quelques contraintes. Le vert indique les processeurs optionnels ajoutés par l'utilisateur (les plugins). Une *Route* (un bus) n'a pas les *DiskProcessors* en rouge.

8 Structure interne d'une *Route*

De l'extérieur, une *Route* est un objet opaque. Dans le cas qui nous concerne de la latence, le *Port* d'entrée d'une route annonce sa latence de reproduction, et son *Port* de sortie annonce la latence de capture. Dans la *Route*, tout traitement du signal est effectué par des **Processors**, qui peuvent être intégrés ou modulaires (des plugins).

Un processeur est presque comme une mini-route avec des entrées et des sorties, et peut être latent ; en revanche la chaîne de processeurs est toujours linéaire. Un aspect intéressant de l'architecture, motivé par la pré-écoute de contrôle⁹ (cue-monitoring), est que les entrées/sorties disque ont été transformées en *Processors* génériques (comparer les figures *fr-11* et *fr-12*). Cela améliore l'uniformité du traitement à l'intérieur d'une route, et permet une séparation interne propre des latences d'entrée et de sortie ainsi qu'un alignement des paramètres de contrôle et des événements d'automation.

Un *Processor* est décrit comme suit :

- Un Processor *est-un* Automatable
- Un Automatable *a-un* AutomationControl
- Un AutomationControl *est-un* Parameter
- Un AutomationControl *a-une* liste de ControlEvents.

La classe **Automatable** est une abstraction des fonctionnalités communes à tous les

⁹La pré-écoute de contrôle est la possibilité d'entendre et d'enregistrer les entrées tout en écoutant les données lues sur le disque au même moment sur une même piste.

paramètres de contrôle dépendant du temps. Par exemple, la gestion de la transition arrêt/marche de la tête de lecture active une passe d'écriture d'automatisation. Cette classe garde une liste de tous les paramètres d'un processeur donné.

Un **AutomationControl** est en premier lieu un **Parameter**, qui décrit l'intervalle du contrôle (minimum, maximum, valeur par défaut), son unité (dB, Hz, etc.) et les propriétés du paramètre comme l'échelle (linéaire ou logarithmique), l'interpolation (aucune, linéaire, logarithmique, exponentielle), et la granularité (entier, booléen, virgule flottante). Le Paramètre peut aussi définir un intervalle spécifique avec une liste de graduations nommées.

Un *AutomationControl* englobe un objet Paramètre et y ajoute des fonctionnalités communes à tous les paramètres de contrôle : le mode d'automatisation (lecture, écriture, loquet, relatif, manuel) et une liste d'évènement ; cette liste est essentiellement une liste de couples horodatage/valeur. Cet objet fournit aussi de quoi modifier, évaluer et interpoler la valeur du paramètre à n'importe quel instant donné.

Un *AutomationControl* peut aussi dépendre (être «esclave») d'autres contrôles, ce qui est modélisé par une super-classe :

- Un SlavableAutomationControl *est-un* AutomationControl
- Un Automatable *est-un* Slavable
- Un Automatable *a-un* SlavableControlList

Ardour accepte les **SlavableControlLists** tant imbriquées que chainées. Un simple contrôle peut être esclave de plusieurs autres *SlavableControlLists* (imbrication) ; dans

le même temps un **SlavableAutomationControl** peut contrôler d'autres esclaves (chainage). Chaque *SlavableAutomationControl* mémorise sa propre liste de maîtres, et l'*Automatable* de niveau le plus haut conserve une liste de relations pour des raisons de maintenance.

9 Gestion des évènements d'automatisation

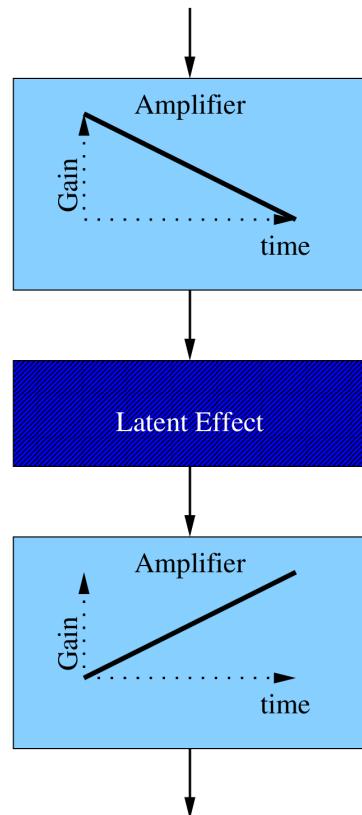


Figure fr-13 – Des processeurs automatisés dans une chaîne avec un plugin latent. Les effets des deux amplificateurs s'annulent l'un l'autre, cependant la latence doit être prise en considération pour appliquer l'automatisation.

Il y a deux manières fondamentalement différentes de gérer les contrôles :

- **Évaluation continue** : Les valeurs du paramètre de contrôle sont calculées à une certaine fréquence (souvent appelée le «k-rate»).
- **Évaluation par événements** : Les paramètres de contrôles sont uniquement évalués lorsqu'il y a un changement.

L'évaluation continue a l'avantage de produire des résultats fiables en particuliers lorsqu'on interpole ou accroît les paramètres automatisés. Elle dépend par contre d'un graphe d'évaluation des contrôles qui doit être synchronisé avec le graphe d'exécution audio et ne s'adapte pas bien à l'interaction avec des événements externes. L'excédent de charge induit est par ailleurs couteux. Un exemple ayant une fréquence d'évaluation est CSound pour lequel « les réglages habituels définissent la fréquence de contrôle entre un dixième et un centième de la fréquence d'échantillonnage»[12].

Le système de contrôle d'Ardour est basé sur des événements. Cela encourage la promotion du MIDI comme un vecteur de premier plan qui peut faire office lui-même de contrôle d'automatisation (par les événements MIDI Control Change). Les paramètres sont évalués de manière paresseuse, à la demande, et la chaîne de dépendances est implicite. Il n'y a pas de graphe dédié. Puisqu'il n'y a pas de fréquence de contrôle, les événements peuvent être aussi denses que nécessaire ce qui permet une automatisation à l'échantillon près.

Certains standards de plugins permettent de transmettre au plugin une liste complète des événements pour le cycle de traitement en cours. C'est aussi vrai de tous les

processeurs internes d’Ardour : par exemple les amplificateurs pour le contrôle du gain des atténuateurs, ou encore les processeurs de balance (panner). Il est aussi courant de fournir les évènements MIDI aux plugins sous forme de liste complète avant d’appeler le plugin pour les traiter. Dans le cas où le plugin ne supporte pas ce mode, et si l’évènement d’automatisation n’est pas aligné avec le cycle de traitement, le cycle du plugin en particulier peut être scindé : par exemple dans un cycle de 1024 échantillons, un égaliseur particulier peut être exécuté pour 300 échantillons, puis ses valeurs de contrôle mises à jour, et enfin les derniers 724 échantillons traités. L’implémentation d’Ardour ne découpe pas le cycle pour l’interpolation mais uniquement pour certains évènements précis de sorte à atteindre la valeur cible spécifiée par l’évènement. En revanche la valeur de contrôle est toujours interpolée au moins une fois à chaque cycle.

Puisque les données d’automatisation sont horodatées, elles doivent être décalées au long de la chaîne de traitement de la route en fonction du contexte local. Un exemple est visible à la figure *fr-13*. C’est une action locale, abstraite de l’alignement global par les ports d’entrée/sortie des routes. À l’intérieur d’une route donnée on retrouve les mêmes concepts qu’au niveau global : latence de capture depuis le port d’entrée, et latence de reproduction jusqu’au port de sortie.

Suite du propos

Les chapitres précédents proposent un tour d’horizon des concepts de base et une esquisse des recherches.

Les langages de programmation C et C++ utilisent l'anglais pour leur syntaxe, et le code source d'Ardour utilise aussi l'anglais comme langage principal. Des règles pour le choix des noms de variables et de classes sont données dans le guide stylistique du code[13], et de nombreux noms ne sont pas traduisibles sans introduire d'ambigüité ou de faux-sens. Dans le texte précédent divers termes comme Stripable, Slavable or AutomationControl, qui sont à la fois des noms communs anglais et des classes C++ ont été laissés sans traduction.

Pour aller plus loin dans les détails techniques il est plus aisé de continuer en anglais, d'autant que les divers exemples de code ne sont évidemment pas traduisibles.

La traduction française s'attache à décrire la motivation et les éléments constitutifs majeurs afin d'apporter une introduction générale, une vue d'ensemble et préparer la transition vers l'anglais. C'est essentiellement une traduction des sections 1 à 9.

L'on poursuit en anglais avec les sections 10 et 11 qui décrivent les derniers blocs conceptuels de l'architecture que sont *Port* et **Portengine**. Ceux-ci sont littéralement implémentés par les classes C++ éponymes.

Un *Port* est l'abstraction de la capacité d'interconnecter des tampons autant à l'intérieur de la station audio-numérique que pour les communications avec l'extérieur. Cependant, les fonctionnalités sont réellement fournies par *Portengine*. Le portrait de l'architecture est terminé par l'exposé de la hiérarchie des classes représentant les processeurs d'entrée/sortie (voir la figure 17 de la section 13).

La partie IV concerne les algorithmes et leur implémentation. En particulier elle

établit le graphe de traitement (process graph), et le calcul de la latence dans la section 16 ; des précisions sont données avec des exemples de code dans la section 18.

La thèse se conclut avec les perspectives sur les variations de vitesse et la synchronisation à un signal d'horodatage et leurs incidences sur l'alignement de contenu compensé en latence.

Part I

On Latency and Digital Audio

1 Introduction

1.1 Digital Audio

Sound as perceived by humans is a purely analogue phenomena: A longitudinal pressure wave propagating in a medium. It has been and described in fields of physics and spawned interdisciplinary research on acoustics to study the properties of mechanical waves as well as the auditory sensations evoked by them on humans.

One of the key properties of analogue audio is that it is a continuous signal. This is true for an actual pressure wave of sound in air, as well for an electric replica of the signal: At every point in time there is a value which represents the signal at the given instant: pressure or voltage or current.

Digital representations of audio are non-continuous. The data is sampled at given intervals and converted into a stream of numbers. As opposed to pure analogue signal processing, digital audio is represented as discrete samples in time.

The smallest possible granularity of an audio-signal in the digital domain is one sample, which at a sampling rate of 48000 samples per second corresponds to $20\mu\text{s}$.

While it may seem that this granular signal when converted back to analogue would result in a discontinuous stair-step signal this is not the case. According to the Nyquist-Shannon sampling theorem[1] there is a unique transformation between the analogue signal and the digital representation for the sequence of discrete audio samples if the bandlimit is no greater than half the sampling rate.

While there are technical motivations for choosing high sampling rates¹⁰, a sampling

¹⁰for example: passband filters for analogue to digital and digital to analogue converters

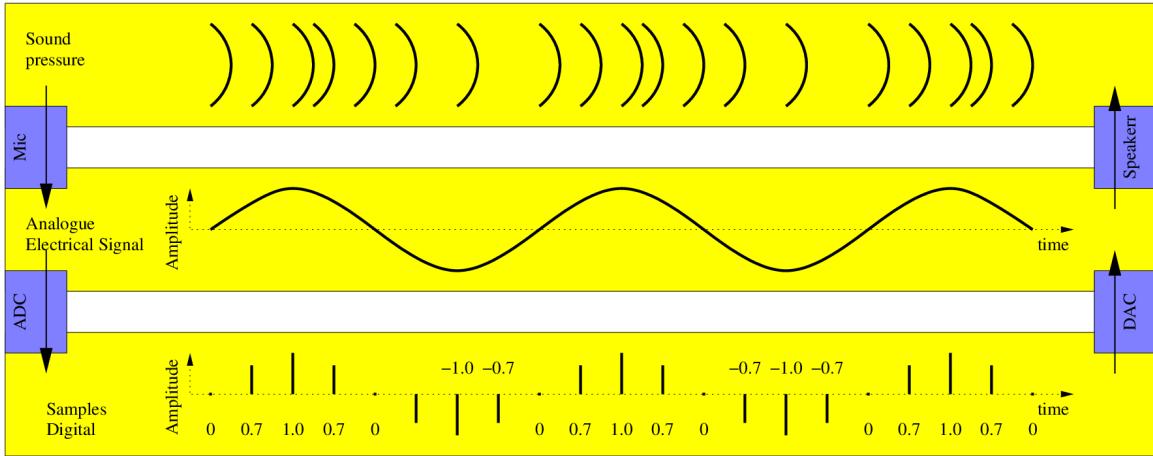


Figure 1 – Audio Signal, from sound-pressure wave, continuous analogue electrical signal and discrete digital signal representation.

rate of at least twice the upper limit of the audible spectrum does cover all use-cases for audio production in general. The upper limit of human hearing has been established at 20kHz, an assertion backed by nearly a century of experimental data[14, 15, 16], which leads to sampling rate of just over 40k samples per second (commonly 44.1kHz for CD, or 48kHz for DVD)¹¹.

1.2 Latency and Latency Compensation

One side-effect of analogue to digital and digital to analogue conversion is that it requires time.

An Analogue-to-Digital (ADC) converter samples the signal at discrete intervals (e.g. the 20 μ s mentioned above). The value is only available after the conversion. By the time the digital representation is available, the signal has already passed. Likewise converting it back with a Digital-to-Analogue converter (DAC) adds another delay.

¹¹Note that too high sampling rates are considered harmful, “ultra-sonics may produce distortions in analogue amplifiers and cause aliasing artefacts in the audible spectrum”[2].

Compared to a true analogue bypass of the signal, any digitally sampled and reconstructed signal will be late.

When combining a signal with a delayed replica of itself, various effects can occur. If the signal is 180° out of phase, it cancels completely resulting in silence as exemplified in Fig. 2. The generalization of effect of this is called comb-filtering: various frequencies in the spectrum cancel each other out, while others pass through amplified.

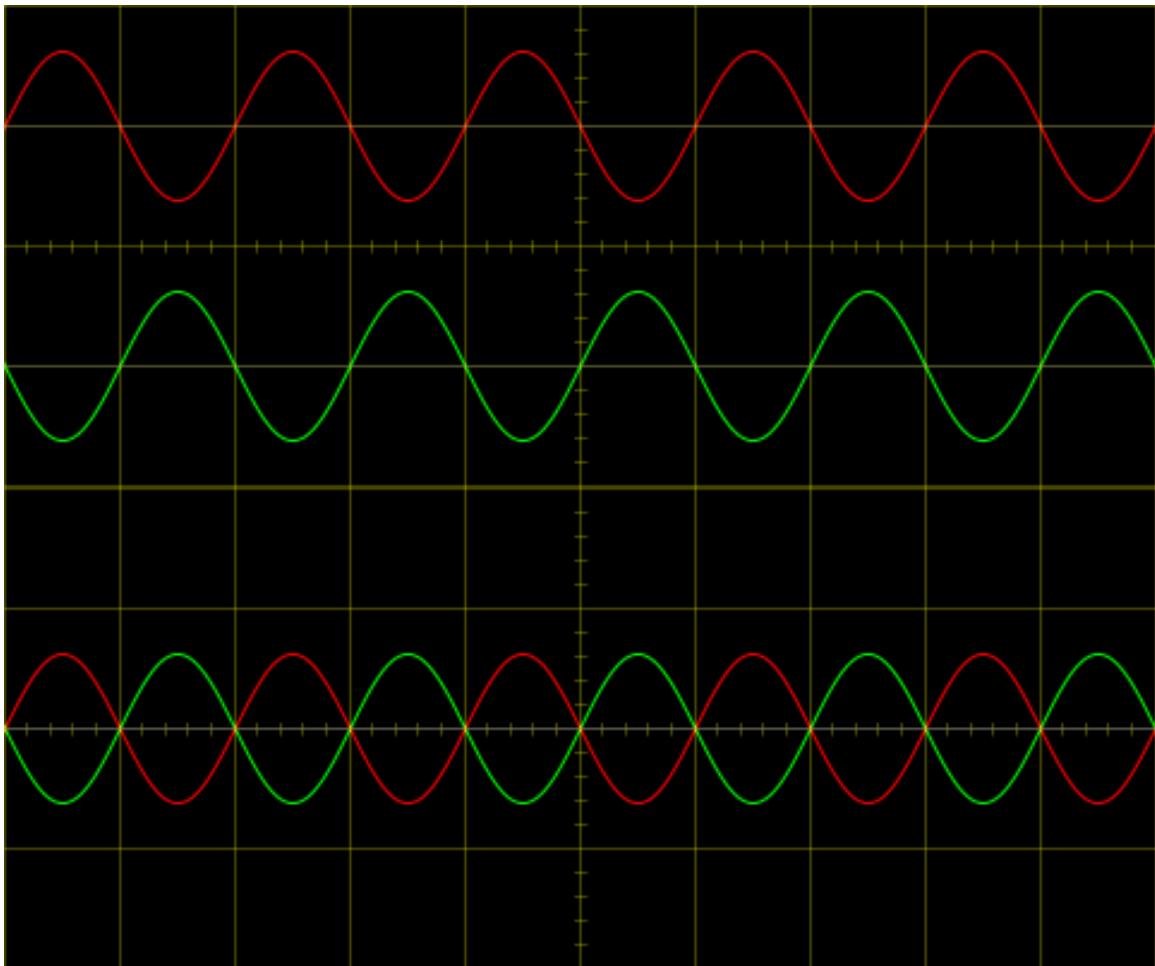


Figure 2 – Two sine 180 degree shifted sine-waves, when added result in silence.

If the delay between the two signals is large (or a non-constant ratio exists) it results in amplitude modulation, audible *beating*. Even larger delays are perceived as

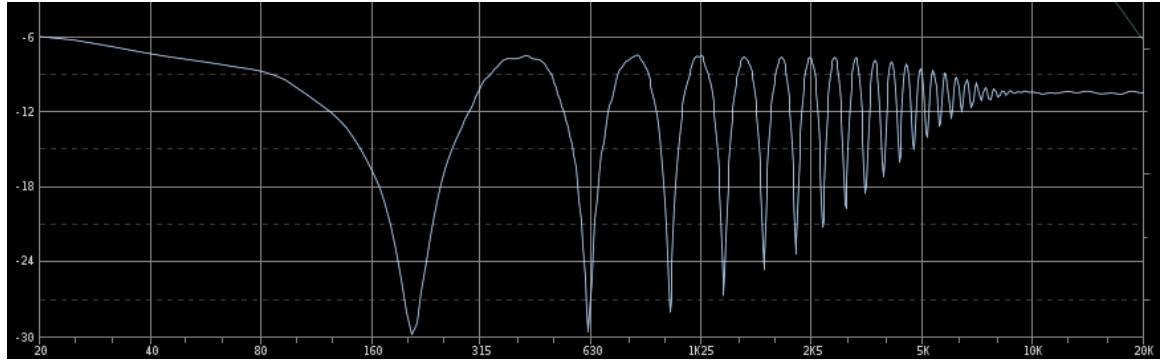


Figure 3 – Pink Noise spectrum, sampled at 48kHz combined with a 116 sample delayed signal of itself. Without the delay it would be a flat line at about -6dBFS. The visual shape of the signal with periodic dips motivated the name comb-filter.

< 10ms	Events sound simultaneous
10–20ms	Threshold of coincidence
> 20ms	perceived as two separate events.

Table 1 – Rule of thumb for audio latency.

echo. As rule of thumb: Humans can distinguish delays larger than 10ms as separate events (corresponding to a frequency of 100Hz)[17, 18], although this depends on the type of signal and its fundamental frequency. Professionally trained musicians often have an increased sense for timing accuracy as well as fluctuations thereof on different time-scales[19]. The perceived simultaneity is often subjective. For Audio/Video sync the timing tolerance has a plateau of undetectability of +25 ms (audio is early) and -100 ms (audio is late) has been identified[20].

Mixing signals with different delays results in effects that are usually unwanted during music-production.

If the delay is much smaller than the frequency of the limit of human hearing and the signal is sampled at high enough rate, the effect can be neglected. Pure hardware solutions can operate at sub-milli-second delay and their short response

time is the reason why many high-end professional systems are implemented using dedicated DSP¹² electronic circuits and are not built with generic PCs. However, a major advantage of a general purpose computer is its versatility. While dedicated hardware is usually limited to a single purpose, a PC offers flexibility.

While the theoretical background goes back into the 1930's[21], digital audio for music and composition was pioneered in the 1950s and various artists have experimented with it throughout the 60's and 70's[22]. General music production has seen a major shift from analogue into the digital domain since the early 80's[23, 24], but it was not until the late 1990's that major production and post-production work-flow followed.

With the beginning of the 21st century computers sufficiently powerful to process audio in real-time became widely accessible. Digital audio workstations moved out of the professional niche allowing average home users to access tools for authoring, recording and production.

The average PC stack consists of a chain of various components, from audio-interface, busses (PCI, USB) to CPU south-bridge. The modern computer architecture is optimized for bandwidth as opposed to throughput[25]. This constraint favours to process large blocks of data rather than a single sample at a time¹³. While processing data in blocks decreases overall system resource utilization and optimizes overall performance, it introduces a minimum response time.

The incurred delay is small, but not negligible. For example 64 samples, 48k samples per second, one period input, one period output:

$$2 \cdot 64[\text{samples}] / 48000[\text{samples/second}] = 2.66[\text{ms}]$$

To put this into perspective, this is the same amount of time it takes for sound to travel 90cm in air or the average distance from a piano to the ear.

¹²Digital Signal Processing

¹³SIMD: single instruction, multiple data (eg. SSE), PCI bus arbitration, burst-data-transfers

A generic, non-optimised PC will usually have longer latencies, for example the default scheduling-granularity on Microsoft Windows systems is 16ms per time-slice. The technical term for this delay is *Latency*.

Latency is a system's reaction time to a given stimulus.

In the context of audio it is usually divided into capture latency and playback latency:

- *Capture latency*: the time necessary for the digitized audio to be available for digital processing. Usually it is one audio period.
- *Playback latency*: the time necessary for the digitized audio to be processed and delivered out of the processing chain. At best it is one audio period.

But this division is an implementation detail and at first of no great interest¹⁴. What really matters is the combination of both. The round-trip latency is the time necessary for a certain audio event to be captured, processed and played back.

It is important to note that processing latency on PC systems is a matter of choice: it can be lowered within the limits imposed only by the hardware (audio-device, CPU- and bus-speed) and audio driver. Lower latencies increase the load on the computer-system because it needs to process the audio in smaller chunks which arrive much more frequently. The lower the latency, the more likely the system will fail to meet its processing deadline which results in a *x-run* (short for buffer over-run and buffer under-run) leaving its merry trail of clicks, pops and crackles in the audio-stream.

Low-latency is not always a desirable feature. It comes with a couple of drawbacks: the most prominent is increased power-consumption because the CPU needs to process many small chunks of audio-data, it is constantly active and can not enter power-saving

¹⁴We will later find that the distinction is important when recording and playing back, aligning audio.

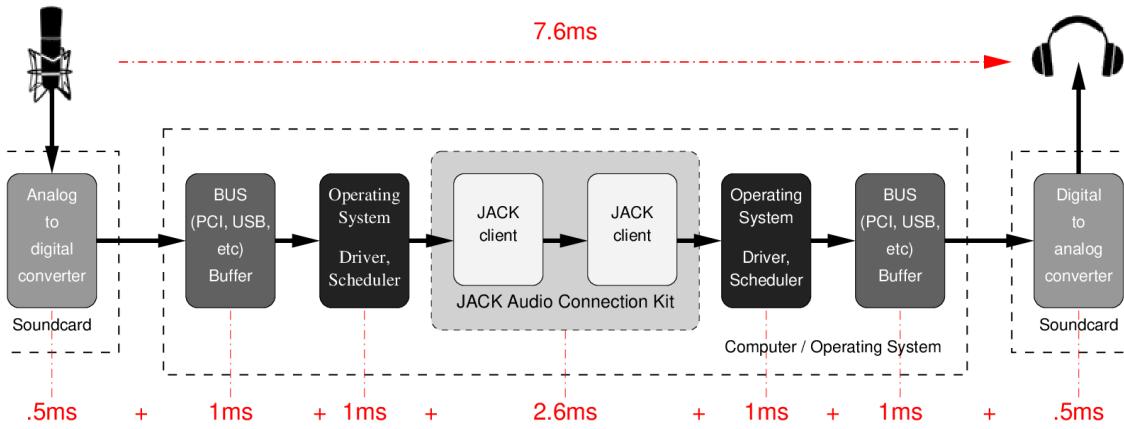


Figure 4 – Latency chain. The numbers are an example for a typical PC. With professional gear and an optimized system the total round-trip latency is usually lower. The important point is that latency is always additive and a sum of many independent factors.

mode (think fan-noise). Furthermore, if more than one application (sound-processor) is involved in processing the sound, each of these needs to run for a short and well defined time for each audio-cycle which results in a much higher system-load and an increased chance of x-runs.

Yet there are a few situations where a low latency is really important, because they require very quick response from the computer.

- *Playing virtual instruments*: a large delay between the pressing of the keys and the sound the instrument produces will throw-off the timing of most instrumentalists.
- *Software audio monitoring*: if a singer is hearing her own voice through two different paths, her head bones and headphones, large latencies can be disturbing.
- *Live-effects*: this case is similar to playing virtual instruments: instead of virtual-instruments or synthesizers it is about real-instruments and effects processing. Low latency is important when using the computer as effect-rack (e.g. guitar effects)—also precise synchronization may be important if you manually trigger

sound effects like delays.

- *Live-mixing*: some sound engineers use a computer for mixing live performances.

Basically that is a combination of the above: monitoring on stage, effect-processing and EQ. It is actually more complicated since one not only wants low latency (audio should not lag too much behind the performance), but exact low-latency (minimal jitter) for delay-lines between speaker in front and back.

In many other cases — such as playback, recording, overdubbing, mixing, mastering, etc. — latency is not important. It can be relatively large and easily be *compensated* for. To explain that statement: During mixing or mastering one does not care if it takes 10ms or 100ms between the instant of the key-press press that triggers playback and sound coming from the speaker. The same is true when recording with a count-in when using a metronome.

However, when tracking it is important that the sound that is currently being played back is internally aligned with the sound that is being recorded.

This is where latency-compensation comes into play. There are two possibilities to compensate for latency in a Digital Audio Workstation (DAW):

- *read-ahead*: the DAW starts playing early (relative to the playhead), so that when the sound arrives at the speakers a short time later, it is exactly aligned with the material that is being recorded.
- *write-behind*: to the same effect, the incoming audio can be delayed to realign it with the audio being played back.

1.3 Signal Routing

During music production, sound sources pass through various stages. These can be explicit connections, for example a microphone to a pre-amplifier to a reverb unit

which are wired by the sound-engineer; or implicit connections inside an effect box itself, which are opaque to the engineer. Explicit signal connections are usually done using a *patchbay* which allows to arbitrarily route source to different destinations. Some patchbays allow to combine signals, but the common way to merge signals is using an audio-mixer which provides control over the individual signal-levels. An important aspect is sub-grouping, which means combining several related sources onto a *bus* in order to share control and effect settings, for example a 'drum' bus that combines the individual microphones of a drum-set into a final stereo mix.

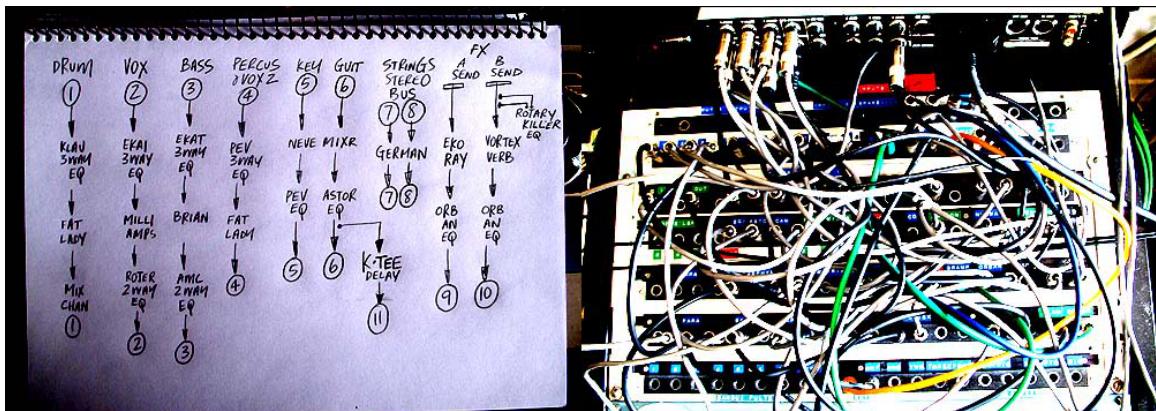


Figure 5 – A typical analogue audio patchbay and routing diagram (source ekadek.com)

Signal to bus mapping is by no means a unique one to one connection. For example all drum-microphones can be combined at various levels into a bus for common volume control while some microphones (eg. Cymbals) are also connected to a second bus for reverb and echo. The effective routing can be a complex network of connections.

1.4 Latency compensation in anywhere-to-anywhere routing systems

When signals takes different paths and are eventually combined acoustically, it must be assured that they are time-aligned. Differences as low as one audio-sample can

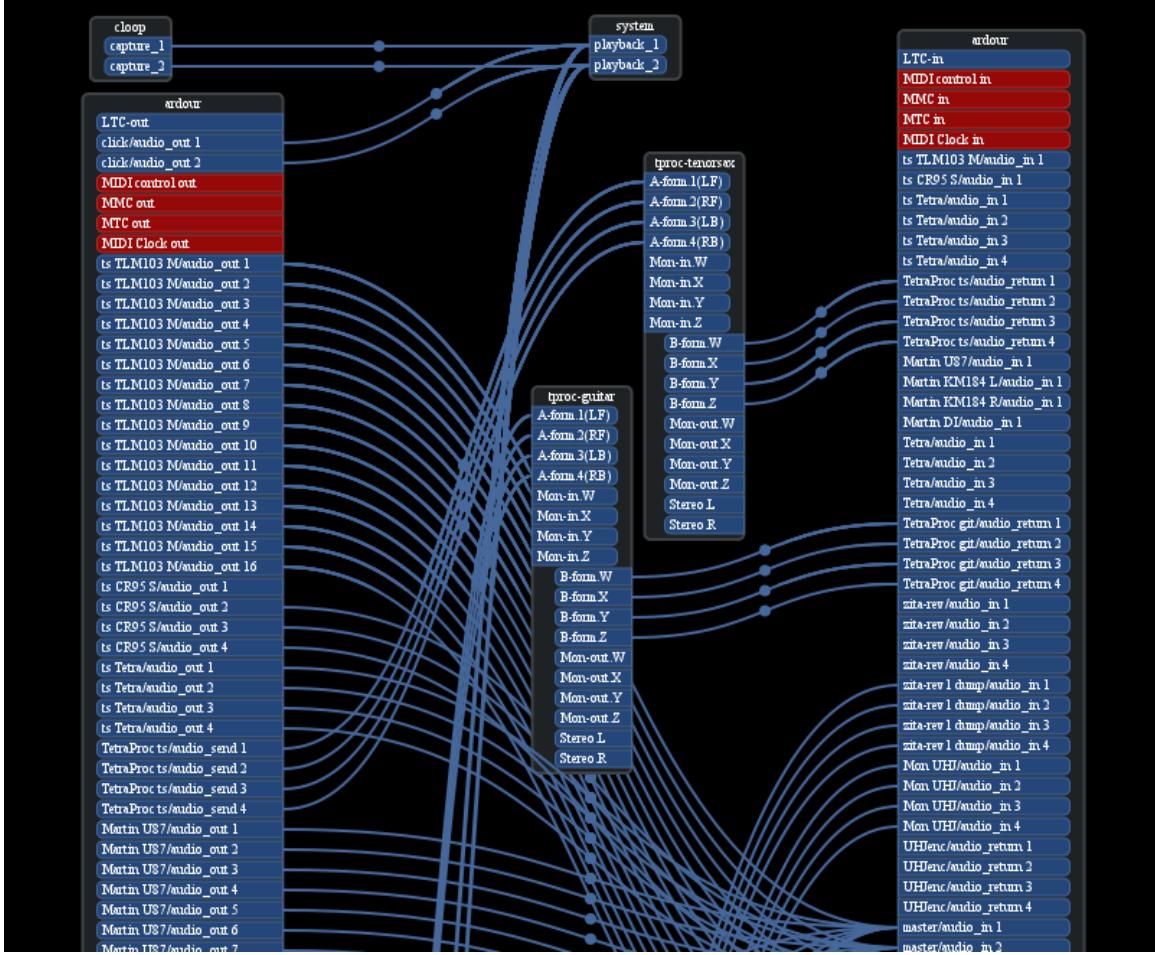


Figure 6 – Digital routing - here jack port connections visualized using ‘patchage’

produce unwanted effects such as phase-cancellation or comb-filtering. As can be seen in Fig. 3, larger differences are even more prominent.

Combining a signal with itself $50\mu\text{s}$ delayed (about 10 samples @192kHz) results in a comb-filter cancellation at approximately 10kHz. (Note: this delay would be equivalent to simply placing one speaker 1.7cm behind another speaker that is playing the same sound[3].)

In a purely analogue system this is usually of no concern. Electrical signals travel at the speed-of-light (or rather close to that), which is several orders of magnitude higher

than the audible-spectrum. In the digital domain however, the smallest granularity is one audio-sample (e.g. $20\mu\text{s}$ at 48kSPS). Digital effects often introduce a latency. For example an Equalizer usually requires one or two audio-samples to respond, or a limiter reads ahead to detect the signal level, 5ms is common for RMS averaging loudness. Most digital effects do have a fixed latency and announce this latency to the DAW which hosts the effect-plugin.

A common example recording guitar and vocals with multiple microphones and treating the recorded sound with multiple effects that introduce different signal latency during processing before summing the result the sound for playback.

Signals taking different routes from source to destination must be aligned to avoid audible oddities. To circumvent this problem many hardware-based solution limit routing possibilities or only allow effects to be added to certain busses in the overall audio-signal chain. In case of hardware pedal-boards the effect chain is also usually fixed without any routing capabilities. In order to reduce routing complexity many DAWs also have a limited number of aux-send/return effect channels and most DAW do not allow to output individual track or busses and have single entry and exit points.

In a professional post-production system, aligning audio to an external timecode (for example video) adds an additional layer of complexity. There are basically three possible sync-points:

- *Align Tracks* Individual tracks or even track types (MIDI, audio) outputs are aligned. This provides for flexible signal routing, but possibly increases total latency.
- *Align subgroups / busses*: only align busses after summing.
- *Master out*: all signals are aligned to a single target - the main output.

There is a fourth option which is to use all of the above and regress on demand as connections are made. This variant is the subject of the thesis: Allowing flexible

anywhere-to-anywhere routing, including individual track-outputs, while minimizing overall latency.

1.5 State of the art

Currently most professional tools solve this issue either with dedicated vendor-provided hardware or by arbitrarily limiting routing options available to the user. The approaches and trade-offs are vendor and implementation specific.

In most cases latency compensation is also reduced to plugin delay compensation (PDC), which applies to a tracks on playback only. e.g. “The compensation algorithm offsets each track by the required amount”[4], send/return routing is ignored, “Return tracks are not compensated” [4]. This is not unlike the approach that Ardour 3 took.

Another product requires paths to be linear and is even further constrained: “Automatic PDC does not work for Plugins using multiple outputs”[5].

At least one vendor only offers PDC for high-end/pro product-lines and requires explicit user-actions to enable it[6].

In order to align the capture-path, a common suggestion is to simply record with small buffer-sizes to mitigate the issue during recording or use dedicated vendor provided hardware[6].

Many DAWs also limit the possibility of effect insertion-points (post-fader only), while others lack concepts like busses. Information provided in user-manuals is usually sparse and the topic is discussed in web-forum along with various workarounds to manually add delay-lines in places.

A large part of the non-professional or *pro-sume* applications do not compensate for latency at all.

2 Research and Development

2.1 Background, motivation and a short history of Ardour

The case for this research is to implement professional latency compensation for the Ardour DAW and related features to make Ardour even more suitable for professional production and post-production.

One of the main concepts of Ardour is to facilitate anywhere-to-anywhere routing, a feature introduced by the JACK Audio Connection Kit for inter-application routing and not very common in other DAWs.

Ardour was started in 1999 by Paul Davis as a Hard Disk Recorder under a free-software license. The availability of the source-code and liberties provided by its GPL licensing makes it an ideal candidate for research.

Ardour5 (and earlier versions) has rudimentary support for aligning audio-tracks, during recording and playback, but aux-sends as well as live input latency and bus-latency are not taken into account. This has lead to recurring bug-reports dating back to 2005 (Ardour 0.99) and feature-requests to add latency-compensation.

Ardour 2 was concerned with a plain audio work-flow, a shortcoming that has lead to many requests to make it useful outside the field of classical audio-engineering. The author of this thesis is particularly interested in film-soundtracks and related workflow.

The development of Ardour 3 started in 2009 and by the time it was released in 2013 it gained various new features. The most prominent one are MIDI-tracks, but the step to version 3 was substantial: Large parts of the update — such as parallel DSP execution — are not directly visible to the user. Others such as the Matrix-style patching/routing, an overhauled export system, dedicated monitoring section and means to synchronise with external timecode sources paved the way towards adoption in the professional post-production houses.

During that time, the author contributed large parts to the timecode clock-chasing algorithms, implemented a video-timeline and support infrastructure and reworked signal-level measurement according to established standards. Ardour 3.5 fulfils many criteria for film and movie soundtrack production[26], and since its release it was already used to record, edit and mix full-feature films and various short-films[27].

Despite the fast-paced feature-packed development which lead to Ardour 3, proper latency compensation was still a major missing feature that remains to be addressed.

While a large part of the development for Ardour 3 was far from trivial and many complicated mechanisms were implemented based on discussion and basic planning, it became clear that proper latency compensation can not be done in the same way. The primary style of development is to progress confined dedicated modules or individual parts.

Latency-compensation on the other hand touches various major places across the complete software:

- audio-engine back-end (I/O port)
- transport synchronization (session transport, external timecode and vari-speed)
- audio and MIDI tracks (read-ahead, write-behind, buffers and caches)
- effect-plugin latency interface
- signal routing, side-chains and changes to the routing system
- process execution graph

Each of these parts is complicated module by itself and the interaction is rather complex. During the prototyping stage it became clear that proper latency compensation requires extensive planning which motivated this thesis.

2.2 Prototyping

In 2013–2014 during the late 3.x development cycle a prototype was developed to integrate bus as well as effect and MIDI-synthesizer latency into the process-graph of the DAW. This was a predominantly experimental playground and a research project to investigate feasibility and identify required building blocks as well as components that need to be re-designed.

The mechanism requires analysing the data-flow graph to find a solution that introduces minimal additional latency. All signals align to a common reference time, the global transport control or playhead. Signal paths that have no dependency on each other can be processed in parallel and handled atomically. Yet, even independent signal-paths need to be aligned in time, hence the process-graph is only partially useful to calculate latencies.

A mechanism to align all tracks relative to each other, as well as a constraint to align at least one track in absolute terms, is required to assure synchronicity.

Aligning independent paths is achieved by delaying signals with a shorter path to the overall worst-case delay. However if one of the signal is not a live-input, but read and played back from disk, the read-pointer can be shifted and a read-ahead mechanism applied. Instead of delaying incoming audio further to align it with pre-recorded tracks, read-ahead results in a lower overall system latency.

In order to compute the latency-graph one must be able to answer the following two questions for every node (audio port) in the graph:

- how long has it been since the data read from a given port arrived at the edge of the graph (capture)?
- how long will it be until the data written to a given port arrives at the edge of the graph (playback)?

Depending on the routing, there may not be a single unique answer to these

questions, but a minimum/maximum range of latencies for each direction when ports with different latencies are connected.

As we were presented in Fig. 4 the overall latency is a combination of systemic and internal latencies and absolute alignment also requires tools to quantify external latencies. Outside the process graph, systemic latencies may differ per device in case multiple devices are used. It is also not uncommon to have different buffering for capture and playback in the driver or configuration.

The routing is represented as a directed acyclic graph which is topologically ordered and afterwards walked in both directions to calculate capture and playback latencies. Eventually delay-lines are added to balance the latency of all nodes according to the chosen sync points.

A system has been prototyped for the Ardour 3 DAW as proof-of-concept which comprises systemic hardware as well as track and bus audio latency compensation, using the master bus as common sync point. See Fig. 7.

Many building-blocks that came to be during prototyping — for example audio/MIDI delay lines — can be used for the final implementation, yet the purpose of the prototype was predominantly to investigate and identify problem areas.

The initial proof-of-concept implementation lead to a development of many prerequisites and re-design in preparation for a complete implementation. Ardour 4 moved away from JACK as mandatory backend; this refactoring also included abstracting all audio I/O and hardware latency calculations into a dedicated built-in port-engine. Ardour 4.x also includes a built-in tool to measure and calibrate systemic audio and MIDI latencies. It was during the 4.x development cycle that many internal implementation blocks were updated to properly facilitate time alignment.

Various external tools and utilities came to be during the prototyping stage, some of which are described in the Appendix. Most notably plugins to simulate latency as well as measurement tools for validation. Another major addition was adding support

for scripting which allows to introspect and directly access internals. This turned out to become invaluable for debugging and fixing edge-cases.

A major step to facilitate latency compensation was separating the track disk-I/O objects into dedicated processors, a task which was undertaken in late Ardour 5 development process.

It was not until late Ardour 5 in 2017 that actual implementation of full-graph latency compensation and overall integration commenced.

A complete system including MIDI-tracks as well as generic-data (automation) and various sync points is to be described in part IV and the current git/master branch of upcoming Ardour 6 already features it.

2.3 Free Software & Collaborations

Ardour is licensed under the GNU General Public License[28] and has attracted a number of developers over its 17 year long history. The full list of which is available online at[29] (the author of this thesis is listed as username *x42*).

Many of the concepts that are presented in this thesis came to be during discussion between the original author of Ardour (Paul Davis) and the author of this thesis who is lead-developer and currently the driving force behind Ardour development since 2012.

While Paul laid a solid architectural abstraction early on, major refactoring and redesign was undertaken over the course of this thesis in recent years. Elements that are original work developed by the author include the approach to switch to a “processor only” design, described in later parts of this thesis. The driving factor behind this was to provide latency compensated processor automation as well as correctly aligned loop playback. Flexible plugin pin connections inside the processor chain and latency compensated side-chains was a result. Another major redesign is the novel approach to support vari-speeding at engine level.

Work on many other parts are not conceptual, but rely on correct detailed implementation. The concepts of ports, aux-send and inserts are traditional and common to all analogue mixing-desks. The basic structure of those has been available in Ardour early on, yet over the course of the thesis many implementation details regarding those have been changed, in particular ownership of ports and controls.

Likewise the port-engine had prior-art in form of the JACK Audio Connection Kit, yet with slightly different semantics, in particular for synchronous vs asynchronous callbacks to update port latencies. The author was also involved in prior work to specify an extended latency API in JACK.

Ardour is a very large and complex project. Yet many part are self-contained and development processes can be orthogonal. Separate parts are often developed independently e.g. Control Surfaces, Session Export, Plugin API support.

Notable active contributors to Ardour include David Robillard (drobillard), who designed and implemented MIDI support in Ardour as part of two Google Summer of Code sponsorships, he is also responsible for the LV2 plugin standard. Tim Mayberry (mojofunk) did the initial work of porting Ardour to the Windows operating system and refined many aspects of the graphical user-interface. Nick Mainsbridge (nmains) worked on the tempo-map and timeline-rulers amongst other aspects related to musical time as well as optimizations to waveform rendering.

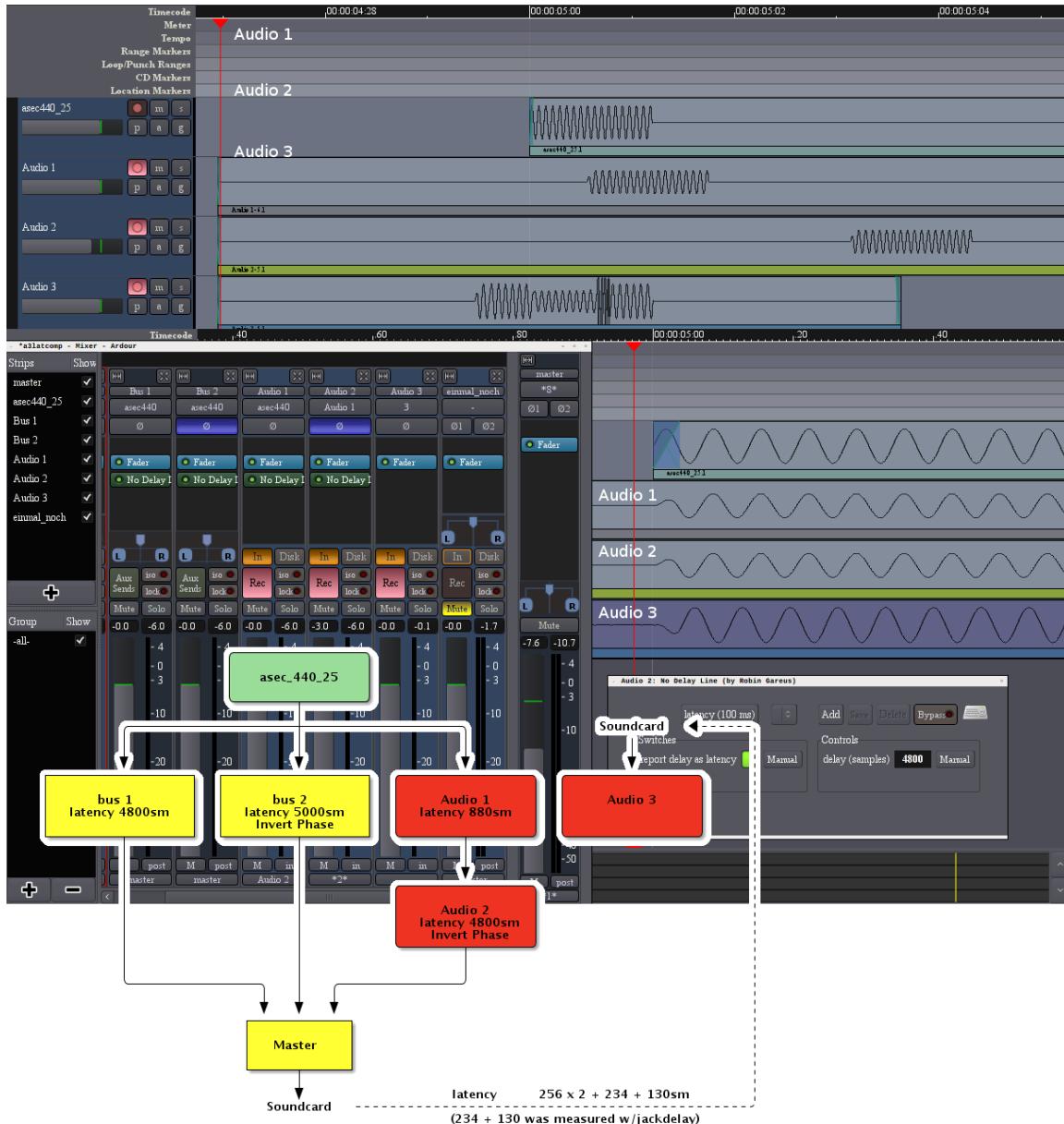


Figure 7 – Top: Previous state of Ardour 3 - before work on latency-compensation. **Bottom:** Proof of concept, prototype with proper latency compensation. The inset show the routing diagram. green: playback; red: recording tracks; yellow: busses

Part II

The Digital Audio Ecosystem

In order to compensate for latency, its quantity needs to be known. This chapter elaborates on the various aspects of latency source and methods to query or measure latency.

3 Sources of Latency

3.1 Sound propagation through air

Since sound is a mechanical perturbation in a fluid, it travels at comparatively slow speed of about 340ms^{-1} [30, 31]. This is not directly relevant to any latency compensation issues, but a proper microphone and speaker setup is a prerequisite.

It is however noteworthy that many musicians learn to accept a given fixed latency. In particular for mechanical instruments - such as pipe organs - where the mechanics themselves impose a latency. For comparison the sound from an acoustic guitar or piano ($\approx 50\text{cm}$ distance) has a latency of about $1 - 2\text{ms}$ due to the propagation time of the sound between the instrument and the ear, while a pipe organ can take up to a second to react.

The speed of sound is also relevant for large concert venues where multiple speaker arrays are present. Appropriate delays must be added to align the signal of various speakers with the sound coming from the stage.

3.2 Digital-to-Analogue and Analogue-to-Digital conversion

Electric signals travel fast (on the order of the speed of light), so their propagation time is negligible in this context, but the conversions between the analogue and digital

domain take a comparatively long time to perform. Conversion delay is usually below 1 ms and lower on professional equipment.

3.3 Computer Architecture

The main source of latency is the process cycle. While the number of samples which are processed per cycle is usually a user-choice, various constraints are imposed by both hardware as well as system architecture and audio driver.

First and foremost, cycle-times are usually power of two. This is not a hard constraint, but a direct consequence of the CPU and Bus interface, where address-spaces are managed in binary. There is often no benefit to choosing a number between two powers of two as the actual page which is transferred is rounded up to the next power of two. The CPU load to transfer — say 86 samples — is equal to that of 128 samples, except the deadline is shorter.

Typical consumer grade sound cards (e.g. HDA Intel) impose a lower limit of 32 samples per cycle, but even so, empirically the sound card cannot reliably read/write a stream of sub-millisecond chunks. Values above 256 samples per cycle are common in consumer-grade hardware. Professional equipment with lower chunk-sizes is available, but for all practical reasons cycles below 8 samples per period are unrealistic on general purpose PCs.

On a general purpose computer, there are various other factors which can interfere with low latency. On the hardware side these are mainly power-saving related: For example CPU frequency-scaling and bus-frequency scaling can have an impact on the minimum response time. Also poorly engineered video interfaces, wifi-interfaces, USB-devices or their device drivers can hog computer resources for a long time, preventing the audio interface from keeping up with the flow of data.

Most modern machines also feature system-management and hardware health checks (SMI: system management interrupts), which cannot be disabled safely. They can also

take a relatively long time to process¹⁵.

3.4 Operating System

The computer has to receive and transmit audio data at regular intervals, which involves many subsystems (Fig. 4).

The audio-interface signals the completion of a data-acquisition/playback cycle by raising an Interrupt (IRQ) signal. The operating system will have to continue the currently active task until a point where it can handle the interrupt. Many operating systems allow to prioritize tasks and can provide for a faster response time to dedicated processes. Real-time systems (e.g. Linux-rt) also allow to pre-empt other IRQ handlers to further lower the time required to handle a high-priority IRQ.

Switching tasks in the Operating System involves switching the process-context which in turn adds a further delay. A context-switch comes with a fixed cost, time required to save and restore registers and stack (linear with CPU speed), and a variable cost which depends on touched memory (Translation Lookaside Buffer flush, cache coherence). The impact is about 10-100 μs [7] per context-switch.

3.5 Digital Signal Processing

Many algorithms to process audio require context (past or future samples) to perform calculations. Incoming audio is buffered and only when sufficient context is collected, processing begins. This effectively delays the incoming signal.

¹⁵Mainboards which never send SMIs are preferable for reliable real-time audio, this is also a requirement for real-time stock trading systems, which have similar issues with latency.

3.6 Processing Period

Processing the signal, applying various effects or synthesizing sound triggered by an incoming event involves performing mathematical operations on the signal, which can be very demanding even for powerful machines.

Most CPUs designed since the mid 90's feature SIMD (Single Instruction, Multiple Data) instructions that allow to batch process large chunks of data efficiently. The most commonly known is the SSE (Streaming SIMD Extensions) instruction set, and various successors SSE2, SSE3, SSE4, AVX are meanwhile commonplace. SIMD instructions greatly increase performance when executing the same operation on multiple data objects: processing 32 samples at once will be a lot faster than 32 times processing a single sample.

Another contributing factor is overhead introduced by function calls. In many cases DSP is performed by many separate functions which cannot be inlined. Even though the overhead of a function call is on the nanoseconds scale on modern CPUs it can be relevant when processing buffers in small chunks.

The larger the block size, the less overall CPU time is required to process a given data-block¹⁶

The optimal size of that block depends on the processing-algorithm and performance/cost considerations. Common buffer-sizes range between 64 and 2048 samples per cycle.

This is usually the main cause of latency when using a computer, yet one that is predictable and can be optimized.

¹⁶While the relationship block-size to computing time is monotonic, it is not linear for small block-sizes, but approaches a linear relationship above 1k-8k samples.

4 Input/Output Latency Measurement

Because of the complex interaction of the separate parts mentioned section 3 the only reliable procedure to establish the latency of a system is to measure it. Notable exceptions to this are situations where a single vendor is responsible for all components in the complete system including hardware, firmware and software. Some Apple Macintosh® machines fall in this category for example.

It is not uncommon that the additional systemic latency is a function of the buffer-size. Figure 8 shows this effects for a USB device. Measurement is performed on a closed loop audio chain by emitting a test signal and capturing it again after a round trip through the whole chain.

The loop can be closed in different ways:

- Moving a speaker close to a microphone. This is rarely done, as air propagation latency is well known, so there is no need to measure it.
- Connecting the output of the audio interface to its input using a patch cable. This can be an analogue or a digital loop, depending on the nature of the input/output you use. A digital loop will not factor in the AD/DA converter latency.

Using a continuous signal of non-harmonic overtones allows to measure the actual latency with sub-sample accuracy by inspecting the relative phase of each of the test-tones. A tool called *jack delay*[8] which can reach accuracy of about 1/1000 of a sample was used to perform the measurement of a PreSonus® Audiobox 1818VSL (Fig. 8). The audio device calibration mechanism in the Ardour DAW is based on the jack delay code-base.

Knowing the precise round-trip latency is essential for studio recording work.

The closed loop described above corresponds to the situation of an audio-workstation playing back existing material or generating a metronome click, with a musician performing along this existing material and recording the performed instruments.

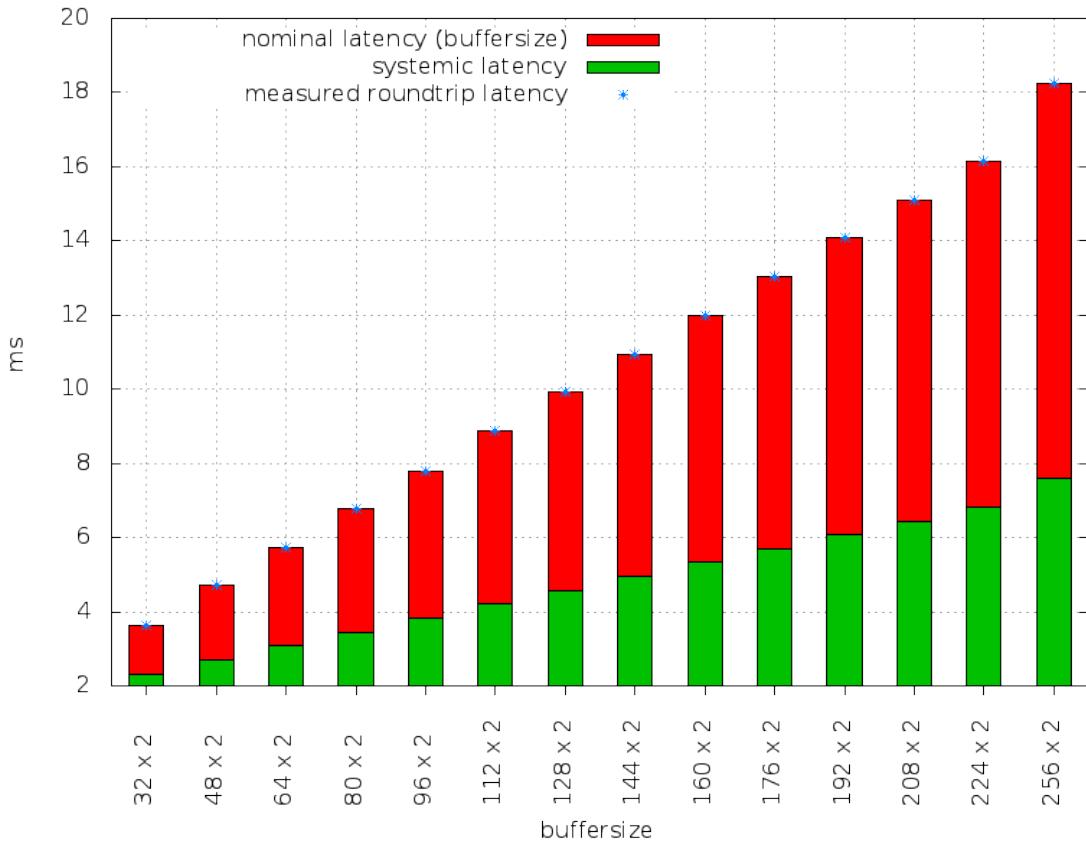


Figure 8 – Measured round-trip latency of a Presonus Audiobox VSL1818 at 48kHz sample-rate.

The newly captured audio will be one round-trip cycle late compared to the existing material that is being played back.

To further complicate issues, systemic latency may not be reproducible and vary every time the audio subsystem of a PC is re-initialized. This is mainly relevant for USB devices. The USB bus imposes further timing constraints (commands can be queued at most at 1ms intervals). The solution adopted by the Linux Kernel snd-usb module is add a ringbuffer to separate transfer and runtime buffers. This adds a fixed latency which differs every time the sound-device is opened. Source-code of other drivers is not freely accessible, but judging from measurements the same mechanism

is used for other devices. This is not a problem as long as the audio-device remains active and the data stream is retained after measuring the latency.

At the time of writing, the only operating system Audio-API which allows to directly query hardware latency (without explicitly measuring it) is CoreAudio on OSX. However, the accuracy of the reported values depends on the audio-interface and driver in question, and may not be accurate with external third-party gear.

If an interface has multiple I/O streams, for examples sound cards with additional MIDI I/O, the streams are commonly not aligned and separate systemic latencies apply for the individual streams. A notable exception to this is IEEE1394 (Firewire) which provides an isochronous data stream that allows to combine multiple streams into the same data packet.

Likewise when combining multiple clock-synchronized interfaces, the systemic latency may differ for each of the devices.

Subtracting the known, nominal latency from the measured round-trip delay allows to quantify the overhead of the system.

The key takeaway is that systemic latency is a variable that needs to be measured and calibrated at runtime.

5 Plug-ins

Most Digital Audio Workstations have a modular architecture. The actual signal-processing is not performed by the workstation itself, but provided by plugins.

The concept goes back to early 1960 Modular Synthesizers, where modules are not hard-wired, but connected together, usually with patch cords or a matrix patching system, which allows for creating complex patches. The digital equivalent of the individuals modules are plugins: Additional third party software loaded into the workstation (the plugin-host) intended to extend the environment.

Typically, plugins can be considered as virtual devices based on the analogy with hardware devices such as reverbs, compressors, delays, synthesizers, etc. that can be connected together in a modular way. This allows host manufacturers to focus on the conviviality and efficiency of their products while specialized manufacturers rather focus on the Digital Signal Processing part[9]. “Audio plugins usually provide their own user interface, which often contains GUI widgets that can be used to control and visualise the plugin’s audio parameters”[10].

A Processing algorithm in a plugin can introduce latency. The reasons for this are manifold and can be of intrinsic nature to the effect or be a result of a specific implementation.

Since Plugin DSP is optional and may be inserted anywhere in the signal-flow, it is the main source of complexity for latency-compensation.

5.1 Examples of Latent Plugins

Latent plugins fall in two main categories: those with a very short delay, which is not perceived as such but results in phasing, and those with longer block-size delays that results in echo-like artefacts.

The first category comprises equalizers, distortions, waveguides and similar timbre-shaping effects. While it is possible to implement many variants of these using latency-free digital signal processing, various models require a few audio-samples total delay to re-align the signal phase, or because of z^{-1} feedback implementations. Linear Phase Equalizers are one example: the signal is delayed to align the phase.

In the latter category there are effects which perform heavy processing and trade off CPU usage with buffering and effects that need to pre-filter or analyse audio before processing. The delays are usually on the order of 20Hz (2400 samples at a sample-rate of 48KSPS) to include low-frequency content.

An example for the latter is a lookahead limiter. This effect limits the peaks of an

audio signal usually with a smooth attack and release curve. If the effect responds too fast (short attack) and attenuates the signal rapidly, it will result audible artefacts. A lookahead limiter pre-buffers the audio signal for a given time-window. This buffer is used to detect the overall level before processing the audio, which results in a delayed response and smooth gain changes.

Also any effect that needs context will be latent. De-noisers, noise-filters or pitch-shifters fall into this category. They do require a chunk of content to operate on. The size of the required data is usually fixed and independent of the process-buffer-size.

Many effects that perform matrix-multiplication such as convolution based cabinet or speaker simulators use a fixed convolution-kernel for the implementation. By nature of this implementation the block-size is fixed and buffering is required to align it with the host's process block size. There are again implementation trade-offs and most convolution effects are latent.

The latency may not be a fixed value, but depend on setup/configuration and parameters. Plugins with a short feedback, feed-forward usually have a fixed sample-count latency, while processors that require to analyse audio often have a delay which is a fraction of the sample-rate. Only in rare cases the latency depends on the process-buffer-size.

A very common practice that requires latency compensation is *parallel compression*: a mixing technique where a dry signal is mixed with a heavily compressed wet version of the same signal. Unlike normal upward compression this preserves the onset and fast transients in the dry-path. However the delay introduced by the compressor, which is usually on a bus and may also be fed by multiple sources, needs to be compensated for.

5.2 Plugin Standards

For historical and commercial reasons a variety of plugin standards exist. From an abstract point of view they are very similar and provide identical functionality. However when looking into details, there are many differences on the technical level.

The most common plugin-standards are VST (Virtual Studio Technology, Steinberg), Audio Unit (Apple, OSX only) and LV2 (ISC licensed). A general discussion is beyond the scope of this thesis.

All standards provide means for a plugin to report its latency to the host, yet subtle differences exist:

- *Audio Units* report the latency in seconds (double precision floating point). It can be queried asynchronously after instantiating the plugin. The plugin can notify a host about changes of its latency using a callback ¹⁷
- *VST* allows plugins to have a fixed value (32bit integer) which is initialized when instantiating the plugin. The value called ‘initialDelay’ is reported in audio-samples.
- In *LV2*’s case the latency is a standard control output port, it need not be a fixed value. The LV2 standard explicitly specifies a process call with a zero-length buffer as latency-compute run: “As a special case, when sample_count is 0, the plugin should update any output ports that represent a single instant in time (e.g. control ports, but not audio ports). This is particularly useful for latent plugins, which should update their latency output port, so hosts can pre-roll plugins to compute latency.”[11]. In LV2 latency is reported in audio-samples, independent of the sample-rate and is available real-time-safe.

¹⁷During prototyping it was found that some proprietary plugins perform license checks when querying latency, which can take a long time and is not real-time safe. Hence latency should only be queried on demand or when receiving a callback notification.

In either case, the host is recommended to query the latency after activating the plugin since instantiation parameters may influence the effect's latency.

When adding a plugin, it usually is immediately latent. It takes time for the signal to pass through it and be processed before output is available, which usually leads to signal discontinuities and audible artefacts.

To alleviate these issues, LV2 goes a step further. Since the plugin standard is extensible, a mechanism for click-free insertion was being added that depends on dedicated host support: A plugin is added in bypassed-mode. The plugin can prefill its buffers and start processing internally while the signal is not processed. Unbypassing the plugin later allows for a smooth click-free transition. It is important for the plugin itself to provide this mechanism. The host cannot know the correct way to click-free bypass/un-bypass the DSP: the valid way may be a cross-fade or ramping up parameter internally at a specific rate or transition between internal states. Since LV2 plugins provide latency information as a normal control-output, this mechanism can be nicely abstracted, yet puts additional complexity to the host to deal with potential latency changes.

Part III

Architecture and Building Blocks

6 Overview

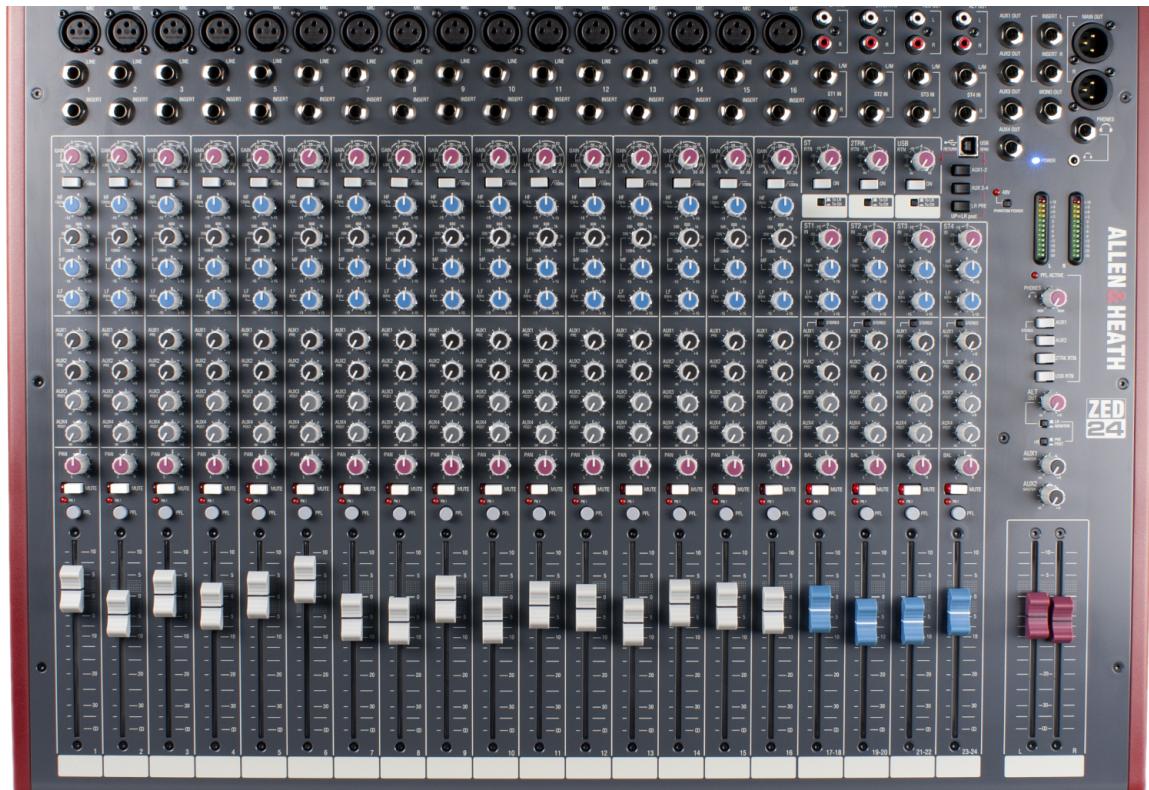


Figure 9 – Allen & Heath ZED-24 mixing desk.

In order to describe and build mechanisms for latency compensation one needs a model of all involved components. The first step in a modelling process is to identify common structures in the design.

A Digital Audio Workstation conceptually distinguishes tracks and busses. Tracks are used to record and play back media, while busses are used for summing and mixing.

A track usually feeds one or more busses.

At the basic level, all data passing through the system can be modelled as a channel-strip, a black-box with one audio-input and one audio-output which can be connected arbitrarily. The term *channel-strip* has a historical connotation from analogue mixing-desks where the signal flows top to bottom, includes processing, usually equalization and stereo-panning. On a mixing-desk, multiple channel-strips are physically arranged horizontally next to each other, which lead to the term strip. See Image 9.

Ardour follows that naming convention, the basic internal object is a **Stripable**. Although in Ardour's case this is merely a virtual object providing general state and abstraction for various user-interface purposes (selection, ordering) where it is also visually displayed as a strip. A mixer strip can also be a pure control-object, that performs no actual data-processing (e.g. VCA¹⁸).

7 Architecture

A strip that processes data is better described as a **Route**: an abstract object which includes mechanisms common to busses and tracks to route a signal.

Routes have both inputs and outputs **Ports**. While traditional channel-strips usually have a mono input and stereo-output, the abstract route is not limited to this. The term route is motivated by its main functionality: It is used to describe *routing* audio (or MIDI) from one place to another. Figure 10 illustrates the basic building block formed by a route object.

A bus is basically just a route. While “route” refers to the abstract model that describes functionality and interfaces, the term “bus” is used to provide semantics in context of usage. Compared to a bus, a track does have additional functionality to provide recording and playback capabilities.

¹⁸Voltage Controlled Amplifier: a gain-stage which indirectly controls another signal's gain.

Object oriented class terminology:

If we say “Foo *is-a* Bar” then Foo was derived from Bar.

If “Foo *has-a* Bar, then Foo has at least one member that is Bar.”

With this terminology we can define a basic set of objects in a DAW:

- A Route *is-a* Stripable
- A Track *is-a* Route
- A Route *has-a* I/O Port
- A Route *has-a* Processor
- A Track *is-a* Route that *has* Disk-I/O Processors

A route itself is atomic: signal processing inside the route is linear and cannot be interrupted from the outside. If a route depends on the output of another route, it needs to wait on the first one to complete processing.

Note: The description of Ardour internals in the following sections refers to the design and implementation of Ardour 6, unless stated otherwise.

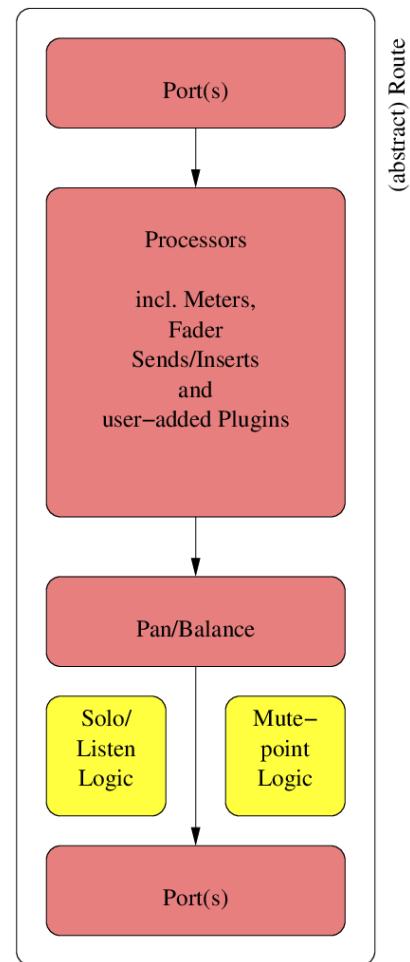


Figure 10 – Basic conceptual abstraction of the Route object

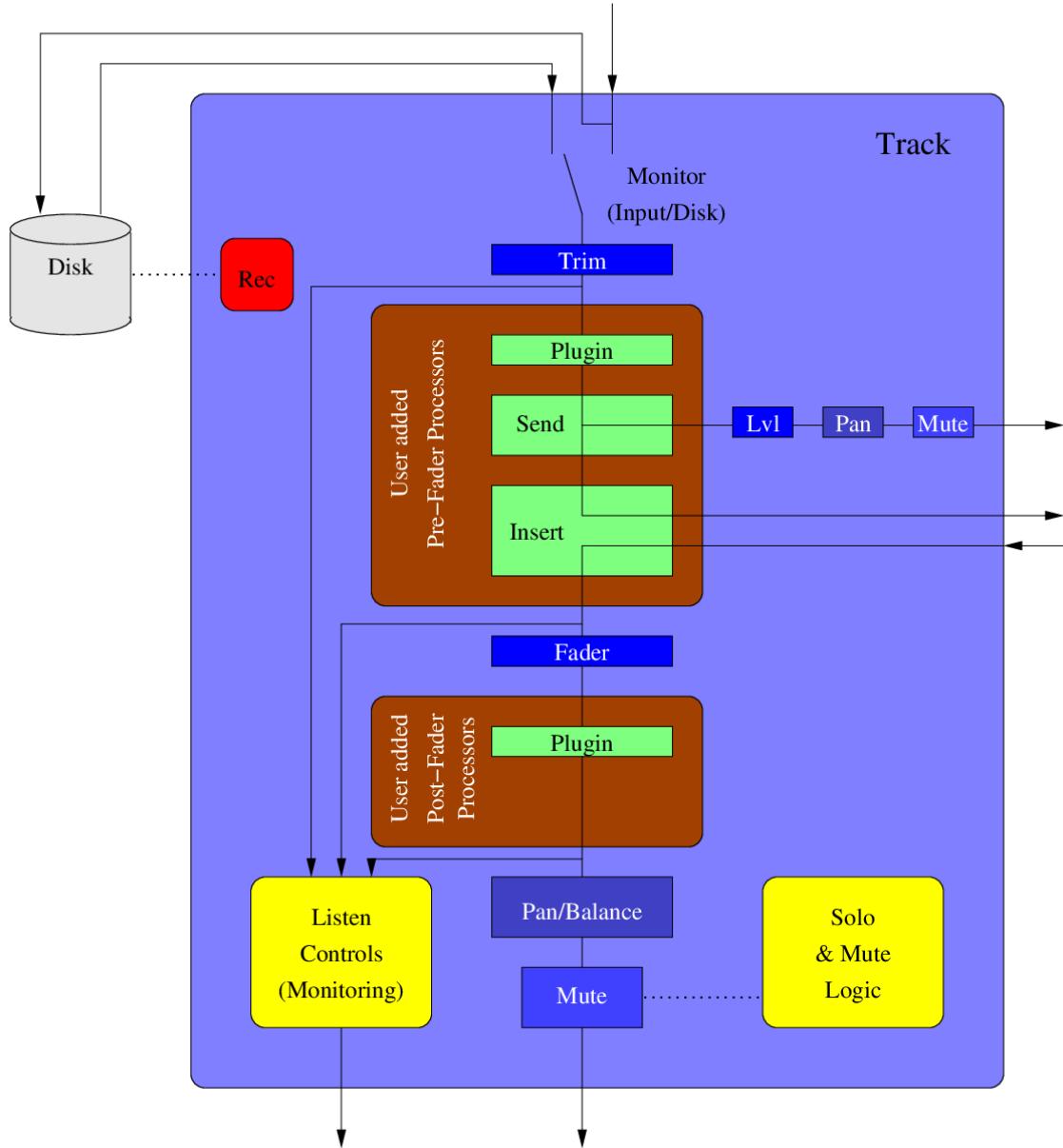


Figure 11 – Old (Ardour 3-5) conceptual abstraction of a track object. The green processors are user-added. Blue indicates gain-staging and internally required processors. Note that disk I/O and monitoring was special-cased with a switch at the track's input. This particular architecture does not lend itself for latency compensation.

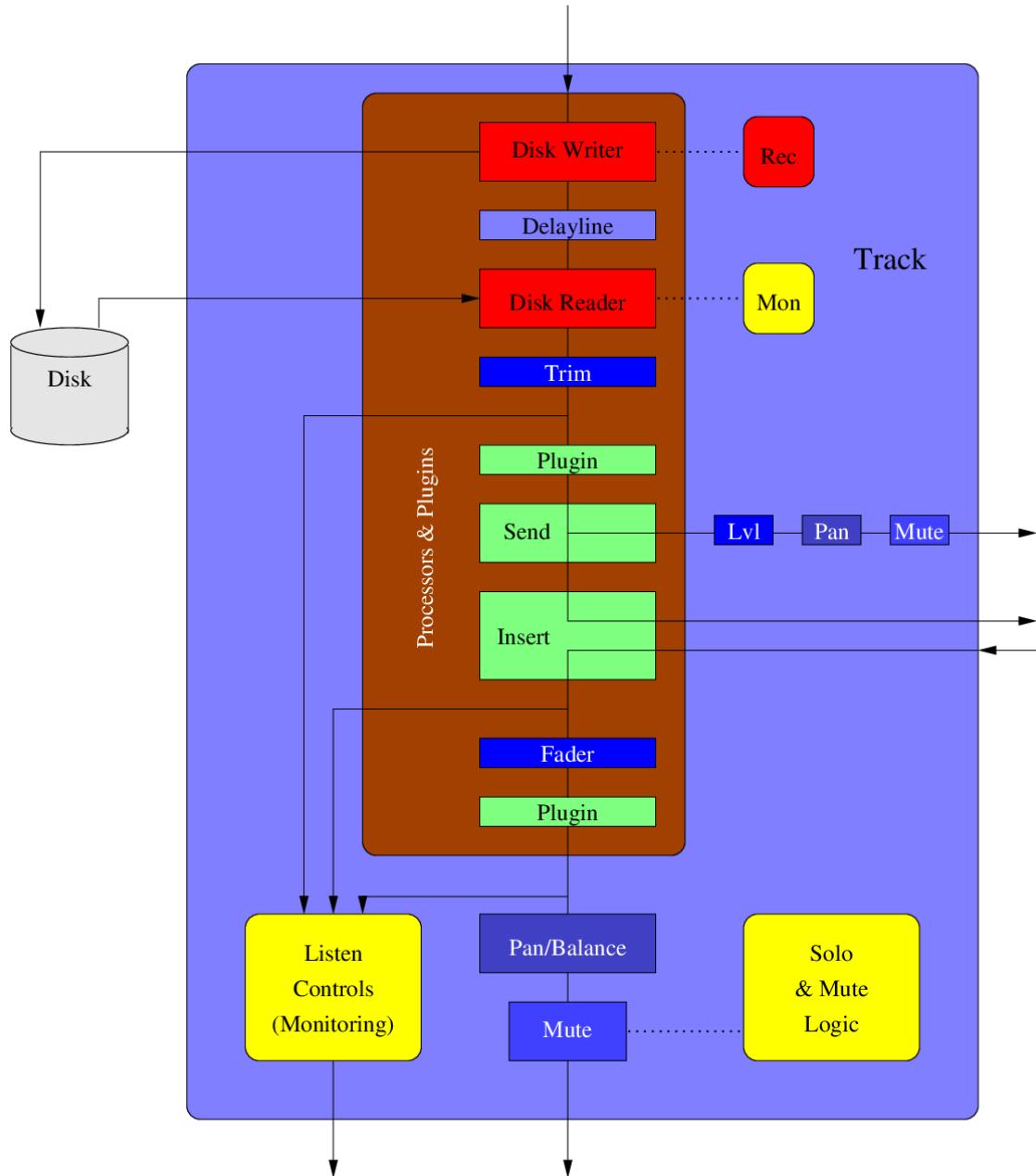


Figure 12 – Re-designed abstraction of a track object, where everything is a processor. There is no strict order, processors can be freely re-ordered with a few constraints. Green indicates optional user-added processors/plugins. A route (bus) does not have the red disk-I/O Processors.

8 Internal Route Architecture

From the outside the Route is an opaque object abstracted by ports. Inside the route all signal processing is performed by **Processors**, which can be plugins or built-in DSP. A processor is almost like a *mini-route*: it has inputs, outputs and is possibly latent. However, the processor chain is always linear. An interesting design aspect that was motivated by cue-monitoring¹⁹ was to turn disk-i/o into generic processors (compare Fig. 11 with Fig. 12). This improves overall consistency of processing inside the route and allows for a clean internal separation of input latency and output latency as well as alignment of control-parameters and automation events.

A Processors is described as follows:

- A Processor *is-an* Automatable
- An Automatable *has-a* Automation Control
- An Automation Control *has-a* Parameter
- An Automation Control *has-a* ControlEvent List

The class **Automatable** abstracts common functionality of time-dependent control-parameters. For example on the transitions of transport roll/stop, a new automation-write pass needs to be initiated. The *Automatable* instance also keeps a list of all parameters of a given processor.

An **automation control** is at first order used as a **parameter**, which describes control-ranges (min, max, default), unit of the control (dB, Hz, etc) and parameter-properties such as scale (linear, logarithmic), interpolation (discrete, linear, logarithmic, exponential) and granularity (integer, toggle, float). The Parameter can also define a specific range with annotated enumerated scale-marks.

¹⁹Cue-monitoring is the ability to hear and record input while listening to data playing back from disk on the same track at the same time.

The *automation control* wraps a parameter object and adds additional functionality common to all control-parameters: automation-mode (read, write, latch, trim, manual) and a **control event list**. This event list is basically just a list of timestamp/value pairs. The *automation control* object also provides means to modify, evaluate and interpolate the parameter value for any given time.

An *automation control* can also be dependent on (“slaved to”) other controls. This is modelled by a super-class:

- A SlavableAutomationControl *is-a* AutomationControl
- An Automatable *is-a* Slavable
- An Automatable *has-a* SlavableControlList

Ardour provides for both nested and chained slavable control-lists. A single control can be slaved to multiple other *slavable automation controls* (nesting) and at the same time a slaved automation control can control other control slaves (chaining). Each *slavable automation control* maintains its own list of control masters, the top-level automatable keeps a list of relationships for maintenance reasons.

9 Automation Event Handling

There are two fundamentally different ways to handle controls:

- **Continuous evaluation:** Control parameter values are calculated at a specific control-rate (often called k-rate).
- **Event based evaluation:** Control parameters are only evaluated when a change occurs.

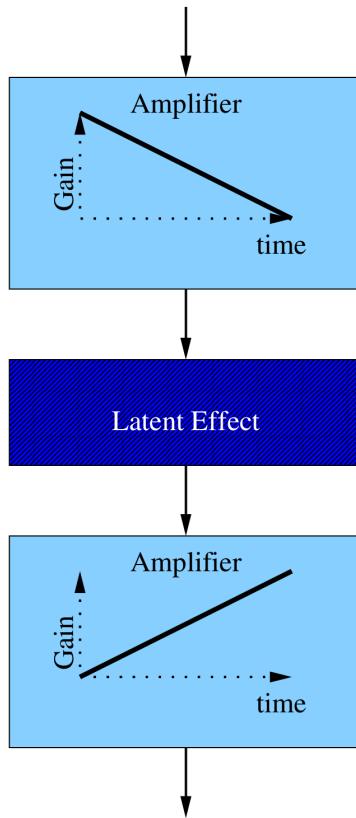


Figure 13 – Automated Processors in a chain with a latent plugin. The effect of the two Amplifiers cancel each other out, however the latency needs to be taken into account when applying automation.

Continuous evaluation has the benefit that it produces reliable results, in particular when interpolating or ramping automated parameters. It does however depend on a graph to evaluate the controls, which needs to be synchronized with the audio execution graph and does not lend itself to interaction with outside events. The added overhead is also expensive. One example system using a dedicated control-rate is csound where “usual settings have the control rate ranging from one-tenth to one-hundredth of the sampling rate”[12, 32].

Ardour’s control-system is event-based. This facilitates MIDI to be a first-class

citizen and become an automation control itself (MIDI Control Change Events). Parameters are evaluated lazily, on demand and the dependency chain is implicit. There is no dedicated control graph. Since there is no dedicated control-rate, events can be as dense as needed, which also provides for sample-accurate automation.

Some plugin standards allow to pass a complete list of events for the current process-cycle to the plugin. This is also true for all internal processors in Ardour: e.g. amplifier for fader-gain control or panners. It's also common to pass MIDI events to a plugin as complete event list before calling the plugin to process them. For plugins that do not support this mode, and in case an automation event does not align with the process-cycle, the cycle of an individual processor can be split: e.g. in a 1024 sample cycle, an individual equalizer is can be executed for 300 samples, the control values updated and the remaining 724 sample be processed. Ardour's implementation does not split the process for interpolation, but only for dedicated events in order to reach the target value specified by the event, however the control-value is still interpolated at least every cycle.

Since automation-data is timestamped, it needs to be offset inside the route's processor-chain depending on local context. An example is shown in Fig. 13. This is a local effect and abstracted from the complete overall alignment by the routes input and output ports. Inside a single route the same concepts as for the global level apply: capture latency from the input-port and playback latency of the output.

10 Ports

A Port abstracts the concepts of connecting buffers inside the DAW, as well as connections for communication with the outside.

The Port itself however does not provide actual functionality for either. Its main purpose is to set policy for possible connections. Ports are mono-directional. A Port

is either an input-port or an output-port and one can only directionally connect them. An analogy where this is enforced physically would be USB Type A/B ports (Fig. 14).



Figure 14 – A USB Type A and Type B port. The ports physically impose a structured connections.

Ports do have properties, which are set when registering (creating) the port: The data-type that can be handled by the port (Audio or MIDI) the direction of the port (Input or Output) and additional flags listed in Table 2.

is Input	The port can receive data (signal-source, writeable)
is Output	Data can be read from the port (signal-sink, readable)
is Terminal	<p>The port cannot be connected to another port.</p> <p>e.g synthesizers which receive MIDI, but will not pass the data on; or audio generators producing data that does not originate from an other port.</p>
is Physical	The port represents actual hardware I/O (those ports are usually also terminal)

Table 2 – Port Flags

Note that the input/output semantics are described from the connection point-of-view. One connects output-ports to other input-ports. An output port's signal is *read* and *written* to an other input.

Inside the route the read/write semantics are reversed. The input port provides data for the route to process (read). From the route's perspective the input (top) of

the route is also an input-port: The port receives data that was written into it from the outside (another port). The data is provided for the route to process. At the end of the route, data is written to an output-port, which in turn can feed one or move input-ports (Fig. 15).

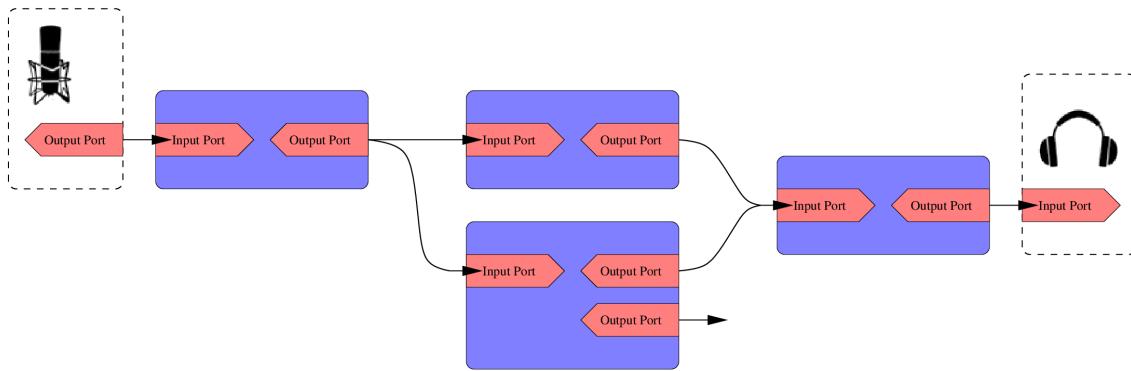


Figure 15 – Port directions: Data is read from an output, processed and written to an input for others to process. Connections are directional from input to output.

The actual payload that is abstracted by the port are audio or midi data-buffers. This adds another set of constraints since only ports of the same data-type can be connected to each other. The port-object itself does not manage connections, but it does allow one to query ports that it is connected to. It also provides an interface to connect and disconnect the ports to/from other ports. The payload buffers are not owned by the port either, the port only provides an interface that facilitates retrieving them. Connection management and data-handling are provided by a port-engine.

A port does however have additional meta-data that describes the object which provides the port:

- **Name:** A unique port-name to identify the port
- **Pretty Name:** A human readable name, which can have semantic meaning

- **Private latency:** The latency of the object which owns the port, usually a route (internal use).
- **Public latency:** Alignment of the port in the full-graph (computed, cached).
- **Buffer offset:** A counter or byte-offset for partial processing of port-data

Ports allow to completely abstract the internals of the object which provides the port(s), usually a route, as black-box.

The public-latency values of a port are separate values for capture and playback, a concept that is motivated and elaborated on in sections 16.2 and 16.3.

The port also offers an API to synchronize internal-buffers with port-buffers. This is particularly useful for MIDI or event-based data to be collected and flushed once at the beginning and end of each cycle.

11 Portengine

The port-engine is global unique instance managing ports. It provides mechanisms for all port operations:

- register port
- un-register port
- connect port(s)
- disconnect port(s)
- list ports
- list port-connections per port
- query port-connections between port

- query public latencies of ports
- set public latencies of ports
- handle and provide access to port-buffers
- combine (sum) port buffers

The port-engine has complete information about all ports and the connection-network between them. It is responsible for handling connection changes and managing port-buffers.

For output ports the engine only needs to provide a buffer for the port owner to write data into. This data could come from a physical input (e.g. a socket on the sound card may be connected to a microphone) or be a route's output port that receives playback data from disk. It can later be read by connected input ports.

If an input is only connected to a single output port, the output's buffer can simply be re-used. The port-engine does not have to actually copy the data (read from output, write to input), but can use a zero-copy mechanism and share the buffer.

In other cases input-ports need a dedicated buffer. It will contain silence if the port is not connected or data mixed down from multiple connected output ports.

It is important to note that the port-engine only provides mechanisms and not policy. The engine is only concerned with data-buffers and passing data around. It does not align or delay buffers: “There is no processing in the wire connecting ports”.

The port-engine does however provide mechanisms that enable clients to align their port buffers. The port-engine computes and sets port-latencies. Whenever port-connections change, the engine traverses all ports, queries their private port-latency, adds them up on the path of connection and asks a port to update its public port-latency.

The actual mechanism is implemented as a callback. The engine only sets latencies for ports provided by the engine itself (usually physical ports) and does not modify

the port-latencies of client ports. When an update is required because a port’s latency changes or when connections change, the engine sends notification by emitting a signal. This is handled by a callback (either the port itself or the owner of the port) which in turn updates the port’s latency-information.

Latency updates need to happen in strict order: for input ports down the stream, for output-stream in reverse-order from the end of the chain to the beginning.

In Ardour’s case, the port-engine can be provided by JACK or internally by one of Ardour’s Audio/MIDI-backends.

12 Aux Sends and Busses

As a final major build-block aux-sends need to be considered. As we described in section 7, Routes are monolithic objects which are processed atomically and can be described as black-box with a dedicated input and output.

There are however cases were it makes sense to tap-off a signal in the middle of processing. A common use-case is a auxiliary-send feeing a reverb-bus. “Listen” controls to monitoring a signal at a specific point inside the route’s processor chain fall into that category, as do side-chains.

The special case of such sends is that they tap-off (“send”) the signal in the middle of a route’s processor-chain and make it available inside another route’s processor-chain (“return”) in the same process cycle: Fig. 16.

Aux-sends are not exposed as ports, they are truly internal to Ardour.

Port-inserts, which send the current data to an output port and read-back the data from an input port (see Fig. 12) do have (at least) 1 cycle delay. These inserts are provided to send data to an external signal-processor and can only return in the next cycle. aux-sends do not have this limitation.

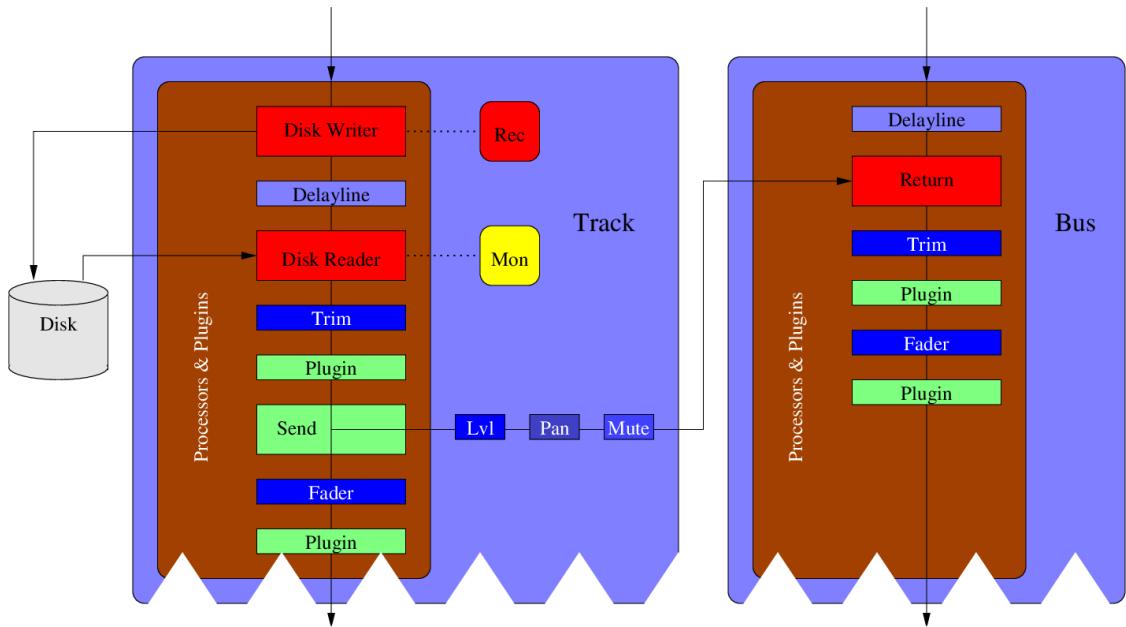


Figure 16 – Track with an Aux-Send feeding a Bus.

13 IO-Processors and Deliveries

Ports (section 10) provide a very low-level interface and represent individual, monophonic channels. It thus makes sense to build a more complex representation of ports to provide higher-level functionality and allow to group ports. In Ardour the **IO** object serves this purpose and allows to create stereo-ports from two mono ports. Any other combination of ports is possible, including mixed data-types of audio and MIDI.

An IO-object *has-a* Port and provides common methods to operate on those port-sets: connect all ports, clear/silence buffers of all ports and set port-latencies. IO objects can also be marked as inactive to skip actual input or output. The number of ports in an IO-object is dynamic: Ports can be added or removed from the IO at runtime.

The IO is still very basic and for internal use in Ardour it would be consistent to

abstract it further. **IOProcessor** does that: It uses a default processor API (like any plugin) and maps the processor-i/o pins to backend-ports. An IO-processor has one or two IO elements. It can be input only, output only or bi-directional (send/return). This configuration is set at instantiation time.

The IOProcessor is further extended for special-cases. The most notable is the **Delivery** object which includes balance or panorama controls.

- A Delivery *is-a* IOProcessor which *is-a* Processor.
- An IOProcessor *has-a* Port
- A Delivery *has-a* Panner

Abstracting the IO using the processor API has the distinct advantage that controlling its parameters can be handled like for any other processor.

Other classes derived from the basic IOProcessor are displayed in Fig. 17. Most notably there is the “Send” which was mentioned before. It *has-a* Delayline, which can delay signals being sent before writing it to a port.

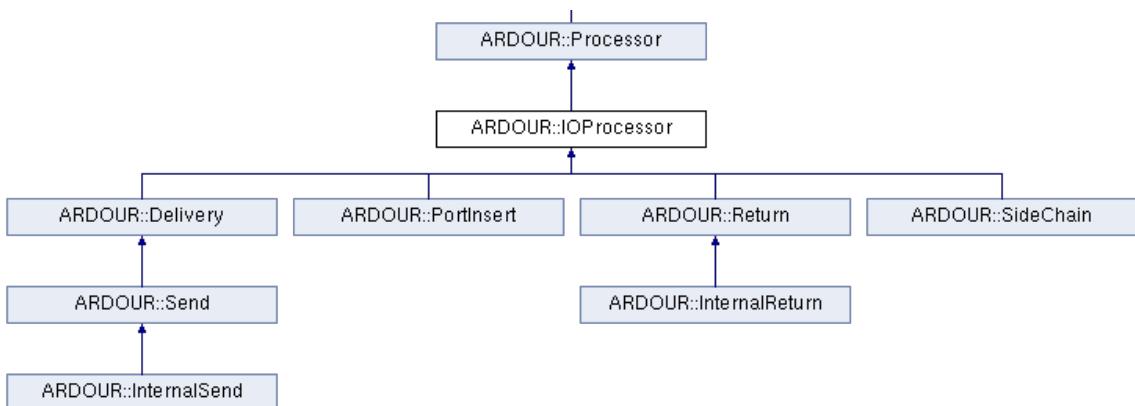


Figure 17 – IOProcessor class inheritance diagram. (generated by Doxygen, Ardour 6.0-pre0-53-g94cce9e06)

The earlier definition from section 7 can hence be refined: “A Route *has-a* I/O Port” to be more accurate:

- For input: A Route *has-a* IO which *has-a* Port
- For output: A Route *has-a* Delivery which *is-a* IOProcessor which *has-a* IO which *has-a* Port

Part IV

Latency Compensation

14 Synchronization

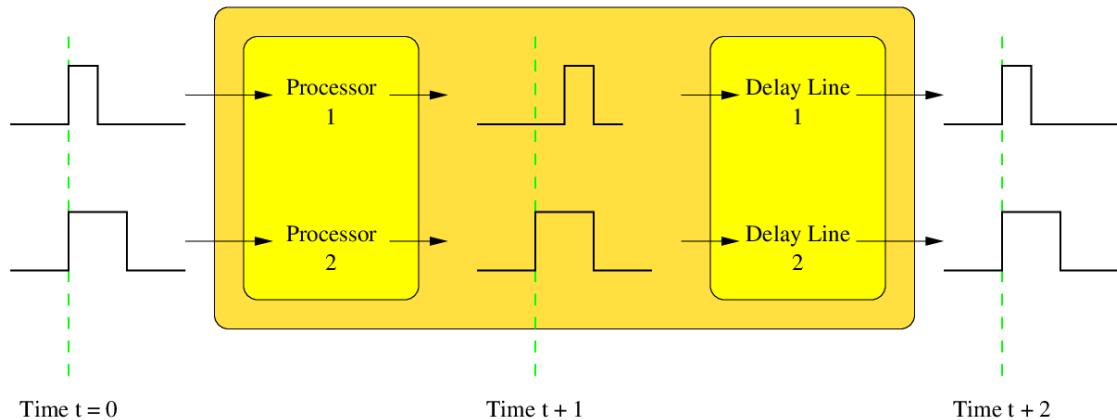


Figure 18 – Synchronization: Two separate simultaneous events take different paths with different delays. They need to be re-aligned in order to remain synchronized at the output.

Separate events that take place simultaneously must also be synchronous after passing through different routes, see Fig. 18. The main goal of latency compensation in Digital Audio Workstations is to retain the relative timing of events.

This alone however is not always sufficient. Further constraints are imposed depending on use-case:

For live-performances it is important to minimise the total delay between paths in order to allow musicians to interact and respond to the performance in time. The absolute relationship to a given wall-clock is not important in this case.

For public-address (PA) systems, the delay needs to exactly match the speaker-distance measured in the speed-of-sound. The constraint is similar as mentioned above.

Yet total processing delay must not exceed this distance.

When working with additional media, for example video or additional playback tracks, absolute time becomes relevant. The events need not only be synchronized to themselves, but also to an external clock or reference point in time. This is also relevant with distributed recording, e.g. during film-production multiple recorders may be used to capture the same scene; or for broadcasting where multiple facilities and OB (open-broadcast) vehicles are used to track a single event. In those cases clock and timecode distribution is used.

If the total processing delay is larger than zero, only one point in the process chain can reference an absolute timecode.

15 Time Alignment

Time alignment of various signals can be achieved by different means. Since latency is always additive, the longest signal path with the worst latency must be determined and signals on a shorter path delayed to align the result.

A common routing scenario is use multiple tracks, which in turn feed busses that are eventually combined to a single master-bus. A simple example is given in Fig. 19.

To address various special requirements further synchronization points may be relevant. Additional constraints on latency compensation can be imposed by requiring to align all track outputs, all bus outputs or both. This is necessary for stem-export²⁰ of multiple tracks. While this alignment scenario is not common for audio/video production, it is useful for multi-speaker performance in media-art performances, or theatre sound where individual tracks or subgroups (busses) directly feed dedicated loudspeakers. Imposing additional alignment requirements will at best yield no

²⁰A stem export creates one file per track. This is particularly useful for interoperability to export each track into another DAW or project.

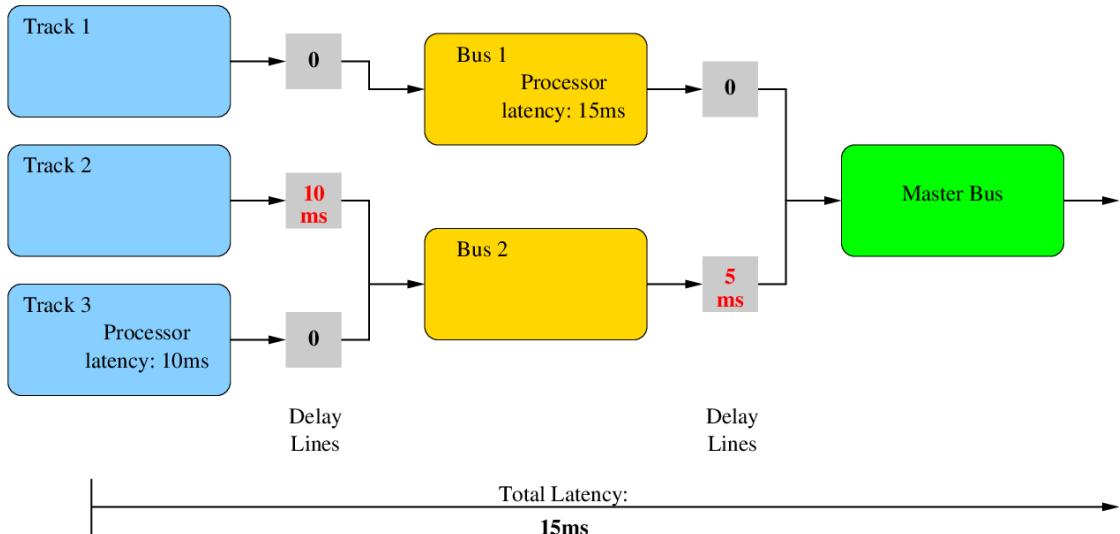


Figure 19 – Routing 3 tracks, via 2 effect busses, to master out without additional time alignment constraints.

additional latency, but usually increases total system latency. Compare Fig. 19 with 20.

The most common approach is to align all signals at a single point only: the output.

Output alignment has various advantages. The time when the signal reaches the speakers is what matters for human perception. This becomes even more relevant when synchronizing external playback machines, such as video playback for soundtrack creation. The timecode of a given picture on the screen needs to coincide with the audio relevant for the given video-frame.

For a pure audio workflow this is equally practical. During recording, the resulting audio at the output is a reference point when playing along to a metronome or click-track. The output is also well what matters for the final master.

However from a mathematical standpoint the choice is arbitrary. The time-reference used internally could be any point, for example input-aligned, and all other parts, including clock-displays, and metronome clicks be offset accordingly.

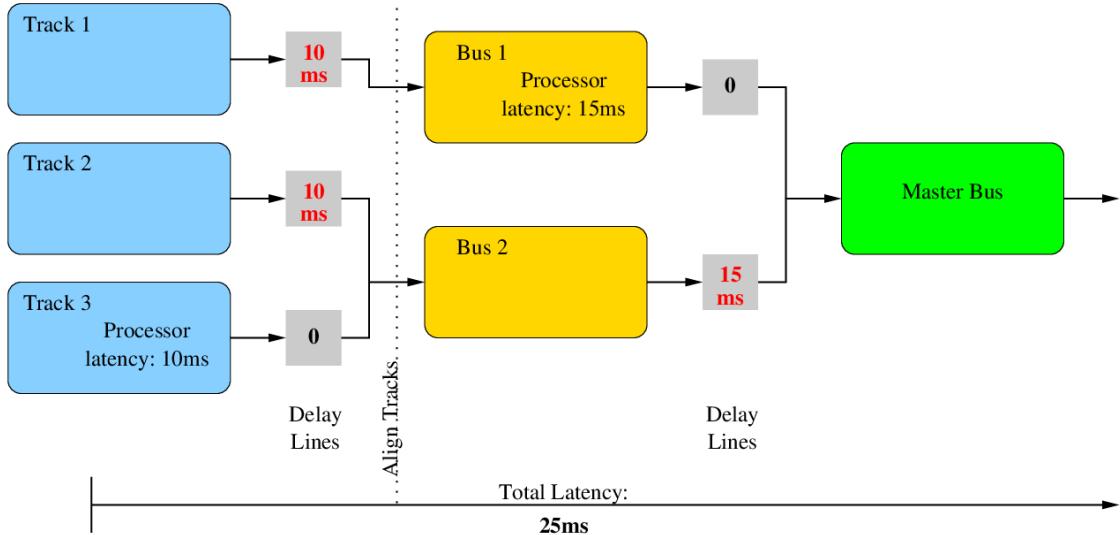


Figure 20 – Figure 19 with additional constraint to align all track outputs results in increased total latency.

Yet there are additional considerations to take into account to influence the choice of a common alignment point:

- Adding latent effects to the graph or making port-connections which change the latency
- Timecode 00:00:00:00. The output needs to play the audible frame corresponding to zero
- Visual display of a moving playhead

If the internal position was aligned to input, latency would be pushed down the stream. When a latent effect is added, the overall worst-case latency increases. To guarantee overall alignment of the audible sample, the complete graph needs to be shifted backwards in time. In other words, the internal position would need to temporarily halt or jump backwards (replay) so that overall alignment is maintained. This results in an audible glitch.

If the internal-position is aligned to output, it is invariant to latency changes during playback. For Playback, only the sub-graph that is affected by the latency change needs to adjust. This can be click-free when mixing²¹.

The boundary condition at zero is less obvious: Assume a data-stream that contains latent effect with a delay of $\Delta T = 5$ and transport-position is aligned to output. If the playback transport starts rolling at time $T = 8$, the input to the latent effect must correspond to $T = 13$. Data from the *future* is used as input to the plugin. By the time the transport reaches $T = 13$ ($\Delta T = 5$ later), the corresponding sample becomes audible. During the initial period $8 \leq T < 13$ the output is silent.

When playing from disk, there is actual future data which can be read in advance, however that does not solve the issue to actually hear the sample at $T = 0$. The solution to this is rather straight forward: pre-roll. In above example the DAW performs a silent pre-roll, feeding latent plugins as needed, here for a duration of $T = 5$. The plugin receives input corresponding to $0 \leq T < 5$ while the transport is seemingly stopped and transport and clocks only start rolling after the pre-roll period.

This pre-roll countdown prior to beginning to roll also allows to activate individual *Routes* on demand, depending on their latency, until the signal trickled down to the output.

As side effect, using pre-roll also visually aligns the playhead and clocks. The clocks remain still until the first sample becomes audible and then start moving in sync. Likewise at the “stop” position. The playhead can simply stop and delay-lines or latent plugins be flushed.

These arguments favour aligning transport-position to the output. A remaining question is whether the reference time used as internal-time should include hardware output latencies. This becomes relevant if the physical latency can change, or if it

²¹The delay-lines to align worst-case latency are before the disk-reader. Only cue-monitoring live input are affected by overall latency changes.

is ambiguous. The former case is a non-issue in reality. When disconnecting the final master-output which is used as reference-point nothing would be sent to the loudspeakers in the first place. Overall alignment may still be relevant when doing a silent internal bounce, but changing port-connections is never click-free to begin with. Ambiguous output latencies is another issue. If the master-out is sent to two different destinations, each with an individual systemic latency. The output needs to align to the worst-case and delay the other signal.

These ambiguities can also happen internally during input e.g. when recording the output of the master-bus back onto a track (bouncing), or when a track A feeds another track B, and track B also receives external input. For recording Ardour hence offers a per track configuration how to align the data to be recorded depending in i/o connections or absolute time.

Earlier versions of Ardour (until version 5) used input as alignment and offset the clock and playhead position. This lead to visually jumping playhead position and added restrictions: latency differences due to monitoring were not taken into account while rolling. Latency changes were ignored and only synchronized at transport-stop (to prevent audible glitches mentioned above) and bus latency was not compensated at all. Furthermore, due to lack of explicit pre-roll the internal-position could become locally negative in the processor chain.

As side note, to avoid potential issues at 00:00:00:00 in various tools used during production, many movie projects start timecode at 01:00:00:00 to be on the safe side. This also has the added benefit of being able to use 01:59:55:00 for a 5 second synchronization beep.

16 Execution and Latency Graphs

Signal Processing in a Digital Audio Workstation from input to output must happen in a strict order. The signal-flow is modelled by an execution graph using vertices for processing tasks and edges for each dependency constraint. The topology is set by the use-case at hand. The causal relationship of output depending on input mandates a directed graph or dependency graph, where input of a given node depends on the output of a former.

Furthermore the graph must be free of cycles. If a given process would depend on output of any following processes, the circular dependencies would result in a feedback loop. This is not to be confused with audio feedback. While the input of a signal may depend on some previous value, they are never coincident. The execution graph describes the dependencies for a single point in time. Audible feedback can be produced by running a second iteration of the same graph, using the output of the first iteration as input for the second.

The execution order is given by a directed acyclic graph (DAG). Processing tasks, including input and output nodes as well as all effect-processors and signal-connections, are represented by vertices.

In simple cases like the one depicted in Figure 21 the graph edges directly represent the audio-signal flow.

More complex routing situation, particular latent routes including processors with external connections, such as sends can be described by inferring additional vertices. One example is an auxiliary send, followed by a latent effect, see Figure 22. The overall execution graph is not altered, the bus (yellow) still depends on the track to complete processing and the duplicate connection from the Track to the Bus is not relevant for process scheduling.

The same signal can take multiple paths with different run-times. For this reason the process-graph and latency-graph can differ.

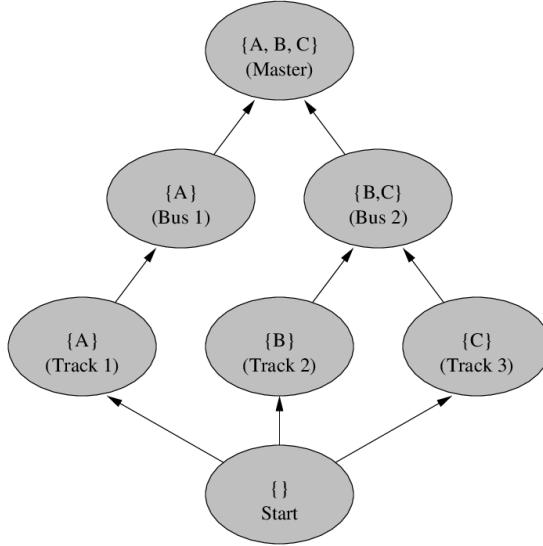


Figure 21 – Hasse diagram of the directed acyclic graph corresponding to the routing in Fig. 19, 20.

Inferring a dedicated process node for the delay-line, would allow to parallelize the graph and compute the delay-line and *Effect2* at the same time. This would however require to break up the atomicity of the route structure. Processors inside a route are always in linear order by design and hence can not alter the topology.

Every directed acyclic graph has a topological ordering. Vertices are ordered such that the head of an edge occurs earlier than the endpoint of an edge. The graph can be traversed from a starting point following all edges without looping back to the start.

In general, this ordering is not unique; a DAG has a unique topological ordering if and only if it has a directed path containing all the vertices, in which case the ordering is the same as the order in which the vertices appear in the path[33].

Calculating the graph is achieved by performing a topological sort. Ardour's implementation is using Kahn's[34] Algorithm that performs $\mathcal{O}(N)$. Once sorted, the DAG can be cast into a dependency graph to evaluate child vertices. If there is more

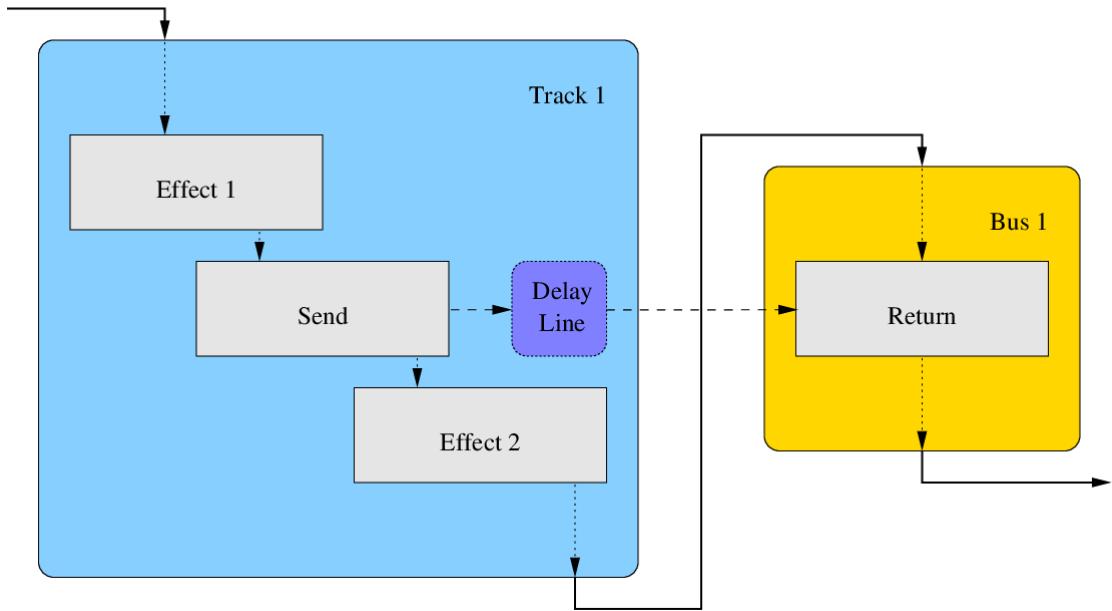


Figure 22 – An internal auxiliary send allowing for parallel signal flow.

than one unique solution to the graph, it means that it can be parallelized.

If the topological sorting fails, Ardour continues to use the old graph (if any) and notifies the user.

If the topological sort succeeds, the graph contains no loops and the activation list can be set up. Nodes that have no incoming edges serve as initial trigger, and nodes without outgoing edges act as indicator that processing has completed. Collecting these node lists is again a single iteration over all routes and $\mathcal{O}(N)$.

Ardour keeps two process-graph chains in memory. While computing the graph a prior process-graph can keep running, switch-over happens on a cycle-boundary. This also allows to continue processing without dropout in case a connection-changes results in an acyclic graph which cannot be processed. This is done in order to allow potential user mistakes when re-connecting ports, or to facilitate connection changes in *wrong* order (connect first, disconnect later) without interrupting the current signal-flow,

which is vital in live situations.

16.1 Routing and Feeds

In order to construct the DAG, various information is required. We assume that all route input-ports also feed all of the route’s output-ports. This is abstracted by the IO Object, which handles port-sets. As Ports are owned by routes, we only need to consider routes that are part of the process-graph, not ports itself. Hence the goal is to establish a graph representing the route-list.

To determine if there is an edge between two nodes, the required information that every route needs to answer is if it “has a feed from” another route.

An edge is created if there is a direct port-connection from one route’s output to another route’s input, or if there is an internal send/return between two routes.

In general this is straight forward, the route itself owns its input and output ports, while the port engine keeps track of connections between ports. For the internal structure a route provides an API for relations between routes: “feeds route” and “fed-by route” which effectively iterate over ports of the two objects and checks if any connections exist between them. The complexity is $\mathcal{O}(N^2)$.

Routes that are not connected to another route are terminal in the graph.

Note that edges are considered to be inside the DAW. External connections are not taken into account and are always terminal. Either they are explicit physical ports, that have the *is Terminal* flag set (see Table 2), or when used with the JACK port-engine, the nodes are part of JACK’s graph.

There is a subtle, but important distinction for send/return inserts as opposed to aux-sends here. An aux-can directly feeds another route and needs to be included in the feeds/fed-by list. Sends and inserts use ports and they are considered like normal in/out ports of the route.

This prevents a route inserting another route in the middle of its signal flow.

Sending to a route creates a dependency to the source, while a direct return creates a dependency on the target, which would result in a circular dependency. Port-Inserts are generally only useful to be connected to terminal nodes.

The graph only represents connections. Internal state such as monitoring or playback must not have any influence when constructing the DAG. Even if a track is momentarily playing and ignores all data from the input port, any upstream input connections need to be taken into account. A use-case for this punch-in/out recording. Punch-in engages recording at a specific time which changes monitoring to input. Punch-out later drops out of recording back into playback from disk at a given punch-out location.

16.2 Upstream/Capture Latency

Upstream latency is the time it takes from a terminal input to a given route. It is a property of every route object that can be calculated by adding information from edges in the connection-graph leading to the route-node.

For terminal nodes, boundary conditions of the graph have to be taken into account as well: “how long has it been since the data read from a given port arrived at the edge of the graph itself”. For physical ports this includes systemic input latency of the audio/MIDI hardware.

This is relevant for capture alignment when recording.

16.3 Downstream/Playback Latency

The downstream latency is defined by “how long will it be until the data written to a given port arrives at the edge of the graph” plus systemic output latency for terminal nodes outside the graph. This value is used when processing, applying effect automation or playing back data.

Since processing happens from input to output, the alignment for each route is computed by subtracting the downstream latency from the maximum total latency,

effectively pushing synchronization upstream.

16.4 Calculating Latencies

The goal is to set all capture and playback latencies for all ports. This in turn will allow a route to align incoming data to output and vice versa.

With a process-graph that provides a sorted execution-order for routes, the overall latency graph can be computed by walking the graph. Since capture and playback latencies are separate, setting the latencies is a two step process.

The first step is to set capture latency. This concerns data arriving at root nodes of the graph and hence is performed by walking the DAG in forward direction as described in the pseudocode in Listing 1.

```

for (each route in process-order) {
    /* Calculate the route's own latency */
    own_latency = sum_latency_of_route_processors ();

    /* Get the capture-latency (best/worst-case) of ports feeding the route */
    for (each output-port connected to the route input) {
        capture_latency = max (capture_latency, connected_port_capture_latency ())
    }

    set_capture_latency_of_route_input_ports (capture_latency)

    /* Propagate latency from outside the route to the route's output port */
    set_capture_latency_of_route_output_ports (own_latency + capture_latency);
}

```

Listing 1 – Set input port latencies

In order to set the playback latency the graph-sorted route-list needs to be walked in reverse as shown in Listing 2 starting at the output edge of the graph.

```

for (each route in reverse process-order) {
    /* Calculate the route's own latency */
    own_latency = sum_latency_of_route_processors ();

    /* Get the playback-latency (best/worst-case) of ports fed by this route */
    for (each input-port connected to the route output) {
        playback_latency = max (playback_latency, connected_ports_playback_latency ());

    }

    setPlaybackLatencyOfRouteOutputPorts (playback_latency);

    /* Propagate latency from outside the route to the route's input port */
    setPlaybackLatencyOfRouteInputPorts (own_latency + playback_latency);
}

```

Listing 2 – Set output port latencies

An step-by-step example of this algorithm on an example-routing situation displayed in Fig. 23 is shown in Tables 3 and 4 for capture and playback latency respectively. The final result is summarized in Table 5.

At the end of this exercise all port latencies have been set. Route latencies can be inferred via the IO of the route that owns the ports.

As was mentioned in section 11, the port-engine sends a notification, or invokes a callback-function whenever a port's latency changes or port-connections change. The port-engine itself does not know which route-input ports correspond to route-output ports. So when a port's latency changes, the engine can not infer which other port-latencies may change as a result.

Like we assumed above that “all route inputs feed all of the route's outputs”, the engine can only assume that all ports of a given client (here Ardour) depend on each other, and trigger an update of all ports owned by the client. The port-engine also

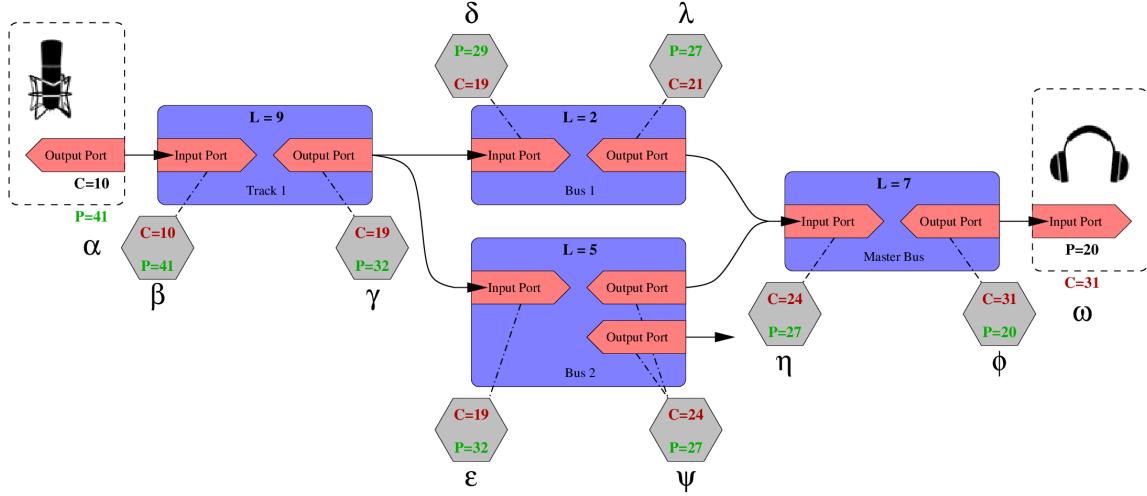


Figure 23 – Port capture and playback latency computation. Capture latency (red) increases downstream (left to right), Playback Latency (green) is propagated upstream (right to left).

has insufficient knowledge about internal routing inside the client and hence cannot assume a given order. As we've seen above the client's DAG needs to be used to calculate latencies.

This is why ports in Ardour's implementation maintain a separate set of private and public port latencies. Once the private latencies are updated as described in the pseudocode above, the actual port-latencies can be published as-is. Notifications from the portengine to update public port-latencies can simply use the pre-computed private latency without re-triggering a latency-update as long as the graph remains unchanged.

Another implementation detail that has been left out in the pseudocode is that port-latencies are a minimum/maximum pair. This allows to identify cases where two ports with different latencies are connected to a single port. Ardour always uses the maximum latency and requires routes to align processing to the worst-case. An example for this is given in Fig 23. The playback latency of port $\gamma = \max(\delta, \epsilon)$.

Iteration	Port								
	α	β	γ	δ	ϵ	λ	ψ	η	Φ
Start	10								
1.a	10 10								
1.b	10 10 19								
2.a	10 10 19 19 19								
2.b	10 10 19 19 19 21 24								
3.a	10 10 19 19 19 21 24 24								
3.b	10 10 19 19 19 21 24 24 31								
End	10 10 19 19 19 21 24 24 31 31								

Table 3 – Step by step port capture latency computation for routing in Fig. 23.

16.5 Send, Return and Insert Latency

Processors inside a route are always in linear order, so the route itself can be also be considered as an atomic object in the latency chain. However the latency of the route itself can depend on additional factors because signal-processors inside a route can include *aux-Sends*, *External Sends* and *Inserts*:

- **Aux-Send:** An internal connection from a certain position inside the processor-chain of one route to the input of another route, usually a Bus.
- **External Send:** A connection from a certain position inside the processor-chain of one route to some port outside of the DAW, a terminal-port.
- **Insert:** Send / Return pair using ports to connect to external gear.

With respect to calculating the process graph, internal aux-sends need to be treated like outputs, connecting the route to the target. While there can be multiple auxiliary sends, every send is from one route to another route, regardless of the number of

Iteration	Port									
	α	β	γ	δ	ϵ	λ	ψ	η	Φ	ω
Start										
1a										
1b										
2a										
2b										
3a										
3b										
End	41	41	32	29	32	27	27	27	20	20

Table 4 – Step by step port playback latency computation for routing in Fig. 23.

Direction	Port									
	α	β	γ	δ	ϵ	λ	ψ	η	Φ	ω
Capture	10	10	19	19	19	21	24	24	31	31
Playback	41	41	32	29	32	27	27	27	20	20

Table 5 – Complete list of playback and capture port latencies corresponding to routing in Fig. 23.

channels. This is modelled as a single edge in the graph alike a normal output to input connection.

External Sends are always outside the DAW by definition and hence connect to a terminal port. They should play no role in the graph computation and external port latency computation takes place outside the DAW by the client which provides the destination port (usually the engine). The only constraint is that the data is processed before the end of every cycle, which is always true for terminal outputs. However port-connections are not under the DAW’s control and hence this rule cannot

be enforced. The connection need to be taken into account when calculating the DAG to flag up potential circular dependencies. The latency of the output port likewise needs to be included in the route's own latency.

Inserts are treated like External Sends, regardless if the data is processed digitally by an external application or outboard analogue gear. The matching signal *Return* path is treated as an input and adds an additional constraint.

Since anywhere-to-anywhere routing is allowed, it is possible to not match the send/return pair, but connect the return-inputs to outputs of other routes or channels from an *External Sends*. This is a way to model side-chain processing, where the signal of one track or route is used to modify effect parameters of another track. In fact Ardour's side-chain send and return are just special-cased²² IOProcessors (Fig. 17)

Regarding graph ordering this adds another dependency on the connecting ports for the route that hosts the port-insert.

As was mentioned earlier, when calculating the process-graph, Ardour operates on a list of routes and uses a dedicated API provided by the route `Route::feeds (Route other_route)`. The implementation becomes trivial. To test port-connections, the route simply iterates over IOProcessors in the route's processor-chain, which includes sends, aux-sends and ports.

And by induction latency is not just a property that the route needs to compensate for, but individual processors need to abstract paths leading into, or out of the route. As an example consider the connection network in Fig. 24. An aux-send may contribute to the route's overall latency, with the route itself being ignorant about it.

If all three effects have no latency, the solution is simple: The delay-line can be set

²²Side-chain IO special-casing is needed to maintain the send/return pair which exists on different routes. If the side-chain's return-target is removed, the side-chain send is automatically removed as well.

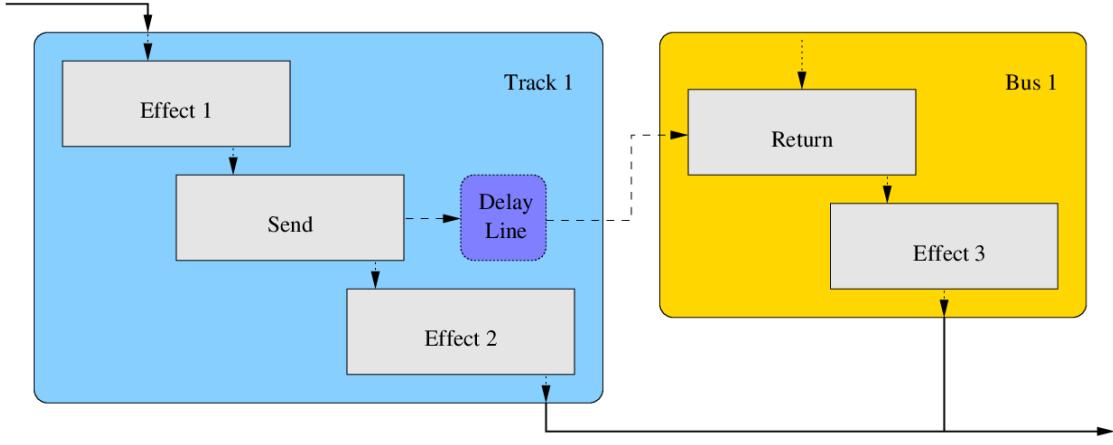


Figure 24 – A routing situation involving an aux-sends and output alignment.

to zero.

Note that Effect 1's latency plays no role, it would only shift the complete alignment (track 1's internal input delay-line is not displayed in this graph for the same reason).

If Effect 3's latency is zero, Effect 2's latency can be compensated by using the send's delay-line:

$$\text{delay} = \text{Latency}_{\text{Effect2}}$$

For brevity, Latency is denoted as \mathcal{L} .

A more generic solution that works for all cases as long as Effect 2's latency is smaller than the latency of Effect 3 can be formulated:

$$\text{delay} = \mathcal{L}_{\text{Effect3}} - \mathcal{L}_{\text{Effect2}}$$

As soon as Effect 3's latency begins to dominate, the signal needs to be delayed before it is fed into Effect 2. To handle this case each Send has a built-in delay-line for the “thru” path which is denoted as ‘ delay_2 ’.

A complete model including this additional delay-line is displayed in Fig. 25.

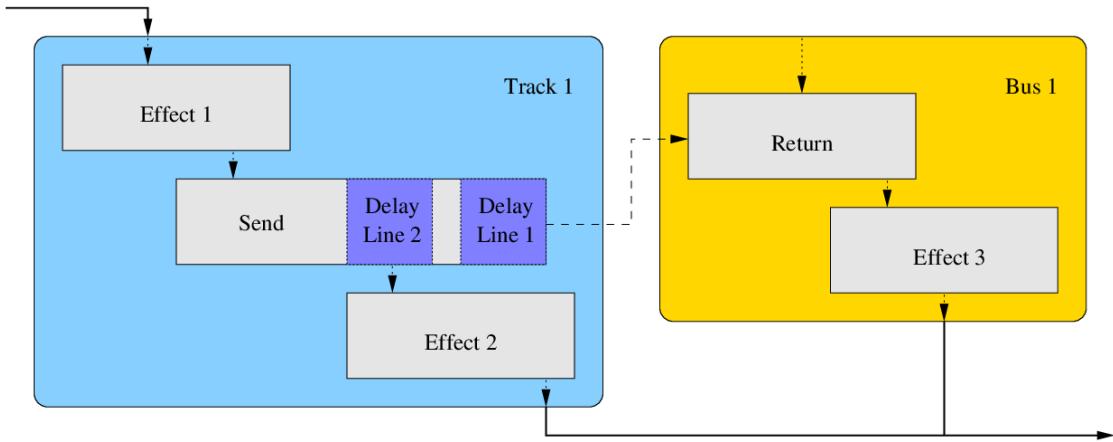


Figure 25 – Complete model of the Send with 2 internal delay-lines, one for the send-path, one for the thru-path

$$\text{delay}_1 = \begin{cases} \mathcal{L}_{\text{Effect3}} - \mathcal{L}_{\text{Effect2}}, & \forall \mathcal{L}_{\text{Effect3}} \geq \mathcal{L}_{\text{Effect2}} \\ 0, & \text{otherwise} \end{cases}$$

$$\text{delay}_2 = \begin{cases} 0, & \forall \mathcal{L}_{\text{Effect3}} \geq \mathcal{L}_{\text{Effect2}} \\ \mathcal{L}_{\text{Effect2}} - \mathcal{L}_{\text{Effect3}}, & \text{otherwise} \end{cases}$$

For balancing the delay in this isolated case it would not matter if the signal is delayed before, or after Effect 2. However for the overall alignment of Effect 2's parameters (in case they're automated), the delay-line needs to be inside the send, or directly after it, before the signal reaches Effect 2. When processing the processor-chain inside the route, automation data is offset by each processors' latency before running the next processor. In this case the send's latency does affect Effect 2 and hence the signal needs to be delayed accordingly, along with any control data.

If a send's latency on the 'thru' pass changes, the route's overall latency changes. This is not unlike plugin latency changing dynamically. There is a subtle difference

though: The send’s latency can only change while re-computing latencies. A plugin’s latency can change at any time.

The send implementation offers an API to set the latency of the send-path and the latency of the signal in the route separately. This is shown in Listing 3: `set_delay_in()`, `set_delay_out()`.

If either of them changes, the send updates the delay-lines. If the “thru” delay-line latency changes, the send emits a signal to notify the session to re-start latency computation. This can be seen in Listing 4. A track can have multiple sends, and a bus can be fed from multiple tracks. However aux-sends can only feed busses (Track → Bus), so no recursive latency-compensation cycles are necessary. Overall latency compensation can be performed as described in section 16.4, this will update all sent-internal delay-lines to compensate for latencies between tracks and busses. If any send thru-latency changes, the complete process needs to be repeated only once. Due to the constrained routing of aux-sends, the second iteration cannot result in changes of the thru-path latency.

16.6 Ambiguous Latency

It is possible to construct valid graphs that do not have a unique latency solution. Consider the example from the previous section 16.5, but add an additional connection to the bus’ input as displayed in Fig. 26.

As soon as Effect 3’s Latency becomes non-zero, latency cannot be compensated without counter-balancing its latency “in the wire”. The Port-engine however does not provide for this. The output on the right would have a latency-range with min/max differing by the Effect 3’s latency.

The routing situation in Fig. 26 is rather constructed. During production one would not mix port-connections and sends to a single target-bus simply because it can easily become unmanageable to maintain and one will lose the overview. A more

```

class Send {
    /* Set latency of the path on the source-side. This should only be called by Route::
       update_signal_latency */

    void set_delay_in (samplecnt_t);

    /* Set latency on the receiving side. This should only be called by InternalReturn::
       set_playback_offset (via Route::update_signal_latency) */

    void set_delay_out (samplecnt_t);

    /* called from set_delay_in() and set_delay_out() when the latency changes */
    void update_delaylines ();

    /* Delaylines */
    boost::shared_ptr<DelayLine> _send_delay;
    boost::shared_ptr<DelayLine> _thru_delay;
};

}

```

Listing 3 – Send Latency API

realistic scenario would use two sends as displayed in Fig. 27 which can also be latency compensated correctly.

In order to prevent ambiguous latency situations to occur, it can be argued that all busses should have no actual input ports and only receive a signal via send and return only. There are however good use-cases for only using busses with direct input (no send/return) for live-mixing. Furthermore this issue is not limited to busses. The same situation can be constructed with tracks. A track however may decouple the input and output path internally by monitoring the input and only align the disk-reader for playback to the output. Not accounting for cue-monitoring, separate capture and playback latencies can help to avoid this situation in real-world use-cases.

This was referred to earlier in section 15: the track's disk-writer processor can align to input, depending on i/o connections or absolute time. Cue monitoring both input

```

void Send::update_delaylines ()
{
    bool changed;
    if (_delay_out > _delay_in) {
        changed = _thru_delay->set_delay(_delay_out - _delay_in);
        _send_delay->set_delay(0);
    } else {
        changed = _thru_delay->set_delay(0);
        _send_delay->set_delay(_delay_in - _delay_out);
    }

    if (changed) {
        ChangedLatency (); /* Emit Signal */
    }
}

```

Listing 4 – Send Latency Implementation

and disk-playback however cannot be aligned in case there are ambiguous latencies for a route.

A work-around to the ambiguous routing situation is also possibly by adding another bus in place where the latency on the wire needs to be as depicted in Fig. 28.

16.7 Additional Graph Considerations

There is usually no unique solution to topologically sort the execution graph. For an example see Figure 29.

Since the DAW has no a-priori knowledge about the execution time of each individual route, nor information about per-CPU core utilization, there is no specific topological order to favour over another. The DAW however has information to additional internal state.

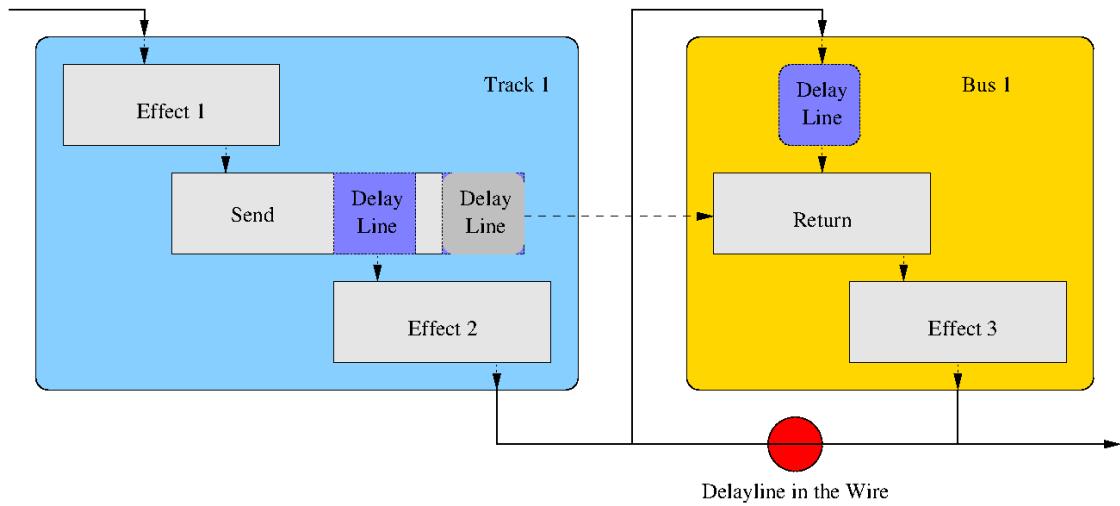


Figure 26 – A complex routing situation involving an aux-sends and output alignment resulting in ambiguous latencies.

Pre-sorting route nodes to favour non-record-enabled tracks over record-enabled tracks not only speeds up the topological sort, but also directly allows to record signals coming from other routes and reduces the randomness of the graph in case multiple solutions exist.

17 Processing

17.1 The Process Cycle

Processing is triggered by the audio/midi backend which is effectively tied to audio-hardware. Synchronization is usually provided by hardware IRQs, although this can differ between Operating Systems and OS provides APIs. The general schema however is common:

0. one-time setup: configure hardware, start audio-clock, and write silence to playback device to align capture + playback.

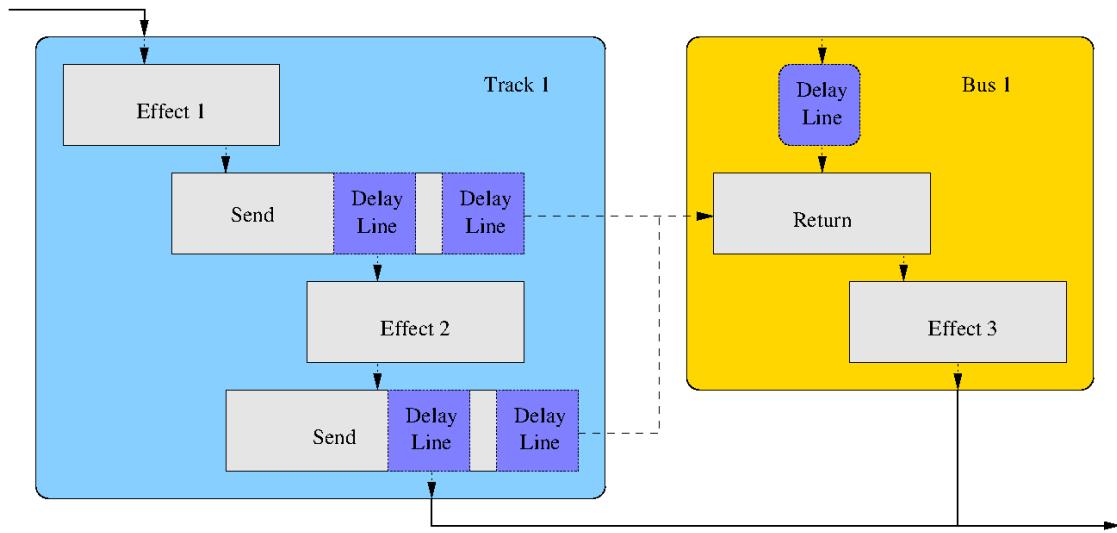


Figure 27 – Resolution to the ambiguous latency situation displayed in Fig. 26 by only relying on sends.

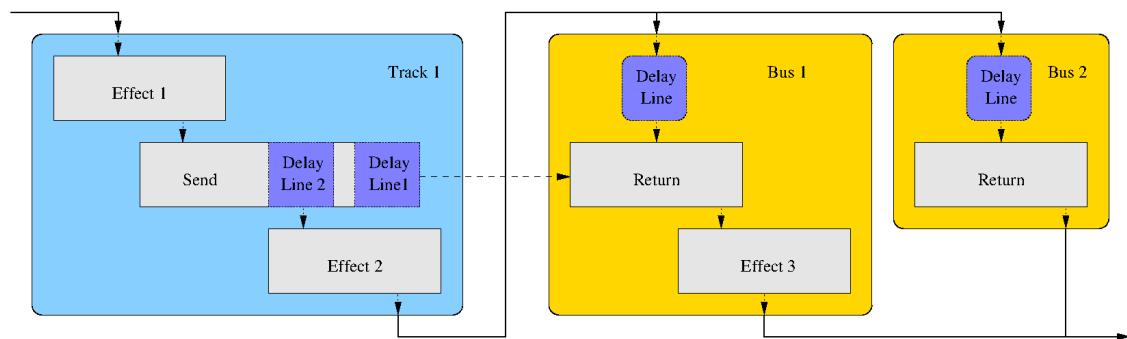
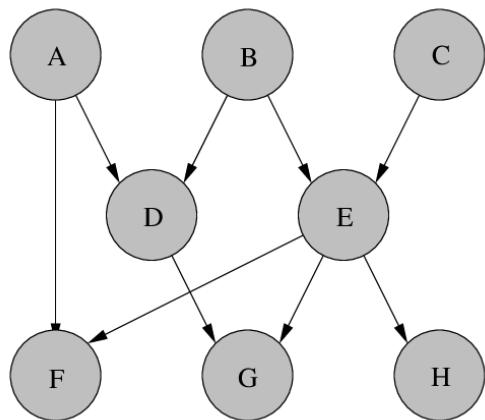


Figure 28 – Another resolution to the ambiguous latency situation displayed in Fig. 26 by adding a bus.

1. Wait for device until device's capture buffer is full.
2. Read data from capture device.
3. Process
4. Write processed data to playback device.



- A, B, C, D, E, F, G, H
(visual left-to-right, top-to-bottom)
- C, B, A, D, E, F, G, H
(fewest edges first)
- B, C, E, H, A, D, G, F
- A, C, B, D, E, H, G, F
- B, C, E, A, F, D, G, H
- A, B, D, C, E, F, H, G

Figure 29 – The shown directed acyclic graph can be topologically sorted in different ways

5. Go to Step 1.

Full duplex sound cards use the same clock for input and output. This is usually a hardware oscillator (crystal), or a similar clock that provides clocking for each audio sample or “word”. The technical term is “wordclock”. This means that for every audio-sample that is being captured, one sample is being played back.

Assuming a buffer-size of one sample: Step 1 above waits until one sample is available at the input. While this sample is being captured, one sample also needs to leave the device. This motivates the one-time setup. Without initially writing data in step 0, the device’s playback buffer would underflow after Step 1.

Processing the data must not take longer than the time corresponding to buffer-size, here, one sample. The resulting data needs to be written back to the device before the next playback sample is required.

The audio-engine, which handles interaction with the actual hardware or driver, abstracts this cycle to the DAW. In Ardour, the audio-engine (backend) *is-a* port-

Cycle	Buffer #					
2. Capture	A	B	C	D	E	F
3. Process		A	B	C	D	E
4. Playback			A	B	C	D
						$\xrightarrow{\text{time}}$

Table 6 – Audio Engine Process Cycle, round-trip latency is 2 buffers

engine. During step 2. Port-buffers of physical capture ports are filled (they are outputs, since they provide data), and the process-callback function is invoked to process the buffers (step 3). When processing is complete, the audio-engine writes data from playback port (inputs from the port-engine’s perspective) back to the playback device.

If the process callback takes longer than the nominal process-cycle time a dropout occurs. The device’s input buffer overflows and the device’s output buffer underflows, which leads to audible artefacts. These buffer *under/over-runs* are commonly called *x-run*. Recovering from a x-run usually requires re-initializing the device, starting at Step 0²³.

17.2 The Process Callback

The process-callback is the top-level entry into actual DAW processing. Port-buffers are prepared and ready to be processed. A simple and straight-forward approach is to process each route in turn, as displayed in Fig. 30.

The callback arriving from the audio-engine only provides the number of audio-samples that have been captured and are expected to be processed. In a DAW processing is not simply free-running, but aligned to an absolute time-reference.

²³Some device drivers, or APIs also provided dedicated x-run recovery methods which do not require to completely re-initialize the device.

Process without events:

- For each route in process graph order:
 - Get input-port buffer
 - For each processor in the route:
 - * Call processor's run() method
 - Flush processed buffer to output-port
-

Figure 30 – Simple process callback

```
Session::process (pframes_t n_samples)
{
    start_sample = transport_position;
    end_sample = transport_position + n_samples;

    for (each route in graph-ordered route_list) {
        route->process (n_samples, start_sample, end_sample);
    }
    transport_position += n_samples;
}
```

Listing 5 – Main process callback, transport rolling

Transport-state and timeline need to be taken into account. The pseudocode Listing 5 outlines an API for a rolling transport, passing transport-position along to every route.

The algorithm shown in Listing 5 assumes that the transport is rolling forward at speed = 1.0. The end-sample is offset by exactly n_samples and transport-position is incremented by n_samples after every cycle. Changes of transport-state from “transport-stopped” to “transport-rolling” or locating the transport to a different position need to happen synchronously. Likewise various events on the global timeline

itself, like loop-ranges, need to be handled in sync before processing.

In Ardour there are two classes of real-time events.

- **Immediate events** are handled as-soon as possible, usually at the beginning of every process-callback
- **Timestamped events** are handled at a given transport-position

In case an event falls in the middle of a process-cycle, the cycle needs to be split at the given position. This is very similar to the mechanism described in section 9 for processor automation, however it affects the complete process cycle. A routes is not aware of the split-cycles and cannot distinguish between processing N samples due to a cycle split, or processing N sample as nominal block-size. Since routes access port-buffers, splitting a cycle needs to offset the buffer-pointers. The buffer-offset property of the port (see section 10) is used for that purpose.

Apart from event-handling, the process-callback needs to handle various other conditions. A session may not use the internal transport-position, but depend on an external timecode source to provide speed and position. Before processing can commence, timecode readers need to be processed to set the transport-position for the current cycle.

Other special-cases that need to be handled inside the main process-callback include script-hooks, conditions for exporting (always rolls) and timecode generators.

For latency-compensation the most relevant variable is the “`transport_position`” i.e. “`start_sample`”.

As was outlined in section 15, the transition from “stopped” to “rolling” needs to be postponed by the worst-case latency. The session pre-rolls before the transport can start moving. This is achieved by processing an immediate real-time event “`SetTransportSpeed`”. On a transition from speed zero to non-zero, a pre-roll counter is initialized. As long as pre-roll is larger than zero, the `transport_position` will not

change and start_sample be adjusted to reflect the actual pre-roll position.

Each route may have a separate global alignment (playback latency) and hence the in-point varies. In the example given in Fig. 23 in section 16.4. Track 1 needs to start first, then Bus 2, followed by Bus 1 and finally when the signal has trickled down the chain, the Master-Bus will start processing. The actual offset between transport_position and start_sample is not a global offset, but needs to be calculated for each route using the session's per-roll count-down and route's playback-latency.

The route itself, which operates on ports, cannot split the process cycle internally since other routes may depend on data being present in the ports when they are invoked. i.e. A route cannot do a partial no-roll followed by a roll by itself, since each of those operations depends on flushing the port-buffer at the end, and the buffer for the next route in the chain would not be aligned.

It would be theoretically possible to push all required information to every route to do so, but it is much more practical to simply split the process-cycle in the session's main process-callback on every in-point. The main process-callback needs to walk the list of routes anyway. Route in-points are equivalent to the inverse route playback-latency and the route-list is guaranteed to be ordered in graph-order.

The process-callback will decrement the pre-roll counter by n_samples every cycle and split the cycle for every route, or when it reaches zero. The pseudocode in Listing 6 is almost an exact copy of the actual code from `libs/ardour/session_process.cc` ARDOUR::Session::process_with_events().

17.3 Parallel Execution

The pseudocode listing abstracted actual route processing by a simple `process_routes()` method which effectively iterates over all routes in graph-order. As we can see from earlier listings, e.g. Listing 5, the transport-position is passed directly to the route's process function. As opposed to timecode-readers, neither wall-clock time nor time

```

Session::process (pframes_t n_samples)
{
    // ...

    while (remaining_latency_preroll > 0) {
        samplecnt_t ns = min (n_samples, remaining_latency_preroll);
        for (each route in graph-ordered route_list) {
            samplecnt_t route_offset = route->playback_latency ();
            if (remaining_latency_preroll > route_offset + ns) {
                /* this route will no-roll for the complete pre-roll cycle */
                continue;
            }
            if (remaining_latency_preroll > route_offset) {
                /* route may need partial no-roll and partial roll from
                 * (transport_sample - remaining_latency_preroll) .. +ns.
                 * Shorten and split the process-cycle. */
                ns = std::min (ns, (remaining_latency_preroll - route_offset));
            } else {
                /* route will do a normal roll for the complete pre-roll cycle */
            }
        }
        process_routes (ns, transport_sample);

        remaining_latency_preroll -= ns;
        n_samples -= ns;
        if (n_samples == 0) {
            return;
        } else {
            engine.split_cycle (ns);
        }
    }
    // ...
}

```

Listing 6 – Main process callback, pre-roll processing when starting transport

relative to the audio-engine's process-callback is relevant when processing routes. This means that routes can be processed in parallel at any time during the process-cycle.

Technically this is implemented using a thread-pool and semaphores to synchronize them.

As previously described in Section 16, when calculating the process-graph, an initial trigger list is created: A list of graph-nodes that are not fed by another node (i.e. routes that are not connected to any input, or to external ports). Since the graph is acyclic, there is always at least one entry in the initial trigger list.

At the end of calculating the graph the number of nodes feeding any other node is counted and stored with every node. This number is called *reference-count*: "How many other nodes reference a given node". Furthermore the number of terminal nodes that have no outgoing edges is computed and stored as *finish-refcount*.

Processing commences with the initial trigger list. After processing a node, the dependency-counter of all dependent child-node is decremented. If it reaches zero, the thread which processed the current route can move on to this node. If not, the thread will *fall asleep* until further work can be done.

If a thread unlocks multiple routes by decrementing their dependency-counter to zero, it will push them onto the trigger-list and wake up other threads (if any) before continuing to process one.

If a process-thread reaches a terminal node that does not have any further dependent children, the number of remaining terminal-nodes is decremented. If this reaches zero, the graph was completely processed.

As an example consider the graph presented in section 16:

At the beginning the initial-trigger-list contains Track1, Track2 and Track3. Assuming there are two process threads, processing will start with the first two tracks. Also assume that Track2 does very heavy DSP and takes a long time to run. The resulting execution could take place as described in Table 7.

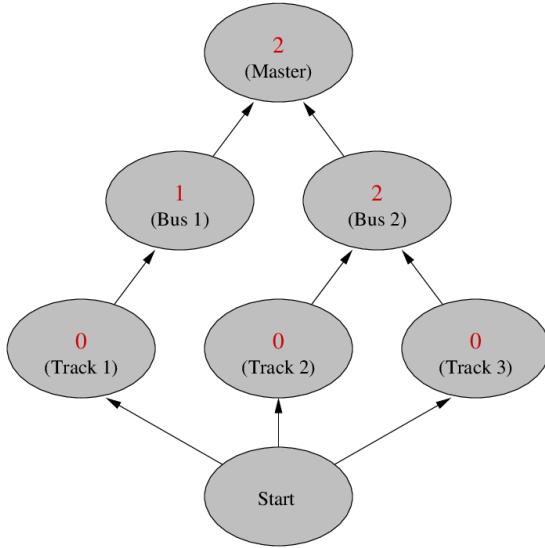


Figure 31 – Process-graph chain corresponding to the routing in Fig. 19, 20. The numbers in red are dependency counts: the reference count of a given node.

If Track2 processing would be less CPU intense, and complete earlier, more routes would be processed in parallel.

18 Route Processing

Processing inside the route was earlier described as simply iterating over all processors. While this is essentially correct, it is a rather incomplete description. Like the route itself, each processor inside the route take “start_sample” and “end_sample” parameters that correspond to the global transport-time, which is aligned to the output and is valid for the complete process-cycle (or sub-cycle). The route does have a second source of alignment: port-latencies.

Before the complete puzzle can be put together it is worth elaborating on two concepts relevant to overall route-processing.

A roll / no-roll distinction was already mentioned before. While it does not affect

Thread 1	Thread 2
Process Track 1	Process Track 2 (DSP heavy)
Processing Track 1 completed, decrement ref-count of Bus 1, ref-count is zero, move to Bus 1	
Process Bus 1	
Processing Bus 1 completed, decrement ref-count of Master-bus, ref-count is non-zero, check trigger-list, move to Track 3	
Process Track 3	
Processing Track 3 completed, decrement ref-count of Bus 2, ref-count is non-zero, check trigger-list, fall asleep	
(asleep)	Processing Track 2 completed, decrement ref-count of Bus 2, ref-count is zero move on to Bus 2
(asleep)	Process Bus 2
(asleep)	Processing Bus 2 completed, decrement ref-count of Master-bus, ref-count is zero move on to Master
(asleep)	Process Master-Bus
(asleep)	Processing Master completed, Master-bus is a terminal node, decrement finish-refcount. finish-refcount is zero, signal end of cycle.

Table 7 – Paralell processing of process-graph displayed in Fig. 31

the route it is relevant for the disk-reader and disk-writer processors. Due to real-time constraints those processors must not directly access the disk, and all data coming from and going to disk needs to be buffered. The actual work to read-from/write-to disk is performed in a low-priority background thread, aptly called “session-butler-thread”. The butler writes data into a ringbuffers for later use by the disk-reader and flushes data from the disk-writer’s ringbuffer back to disk during recording. When rolling, the route can summon the butler (notify the butler-thread) to handle pending data.

The method `Route:::roll ()` is special-cased for routes that have disk-i/o processors to set a “`need_butler = true`” flag for that purpose. Busses do not require this. Furthermore `Route:::roll ()` also verifies if the disk-reader should actually run in the given cycle. This parameter is set to false during pre-roll, while the route is still waiting. Also during no-roll, when the transport is stopped, the disk-reader needs not to be called.

Apart from roll / no-roll there are two more special cased methods:

`Route:::silent_roll ()` is used while synchronizing to timecode masters. As opposed to internal-transport, there cannot be a pre-roll period where the transport does not move while pre-roll is being counted down. Timecode masters always progress at some speed, usually wall-clock, and the route needs to follow. Tracks need to update disk-buffers along the way while not play audibly until alignment to the timecode master can be guaranteed.

`Route:::silence ()` handles the trivial case of simply silencing all output ports. It is useful during initial session setup, while loading and constructing routes. Existing routes may already be being processed. It is also a convenient method for deactivated routes to skip all processing.

Except for the last case, eventually `Route:::run_route ()` is invoked to perform processing. It is a short self explanatory method that can be copied almost literally from the actual codebase. Listing 7 includes some function-calls not directly relevant

for latency-compensation, left for the reader to explore further.

```
void Route::run_route (
    samplepos_t start_sample, samplepos_t end_sample, pframes_t n_samples,
    int declick, bool gain_automation_ok, bool run_disk_reader)
{
    BufferSet& bufs (_session.get_route_buffers (n_process_buffers()));

    /* read or map data from input-port to process buffers */
    fill_buffers_with_input (bufs, _input, n_samples);

    /* filter captured MIDI data */
    filter_input (bufs);
    /* append immediate messages to the first MIDI buffer */
    write_out_of_band_data (bufs, n_samples);

    /* run processors on buffers */
    process_output_buffers (bufs, start_sample, end_sample, n_samples, declick,
                           gain_automation_ok, run_disk_reader);

    /* map events (e.g. MIDI-CC) back to control-parameters */
    update_controls (bufs);

    /* write back process buffers to outputs, this also iterates
     * over all processors and flushes ports of Delivery and PortInsert instances */
    flush_processor_buffers_locked (n_samples);
}
```

Listing 7 – Route process function, this is a near literal copy from Ardour’s codebase.

18.1 Setting Processor Latencies

It is generally sufficient to invoke each processor's run() function simply by offsetting "start_sample" (and "end_sample") according to the processor's position in the route's signal-flow. The processor can look-up relevant automation-data using these absolute timestamps and inform plugins about the current transport-position.

While the offset calculation is performed by the route, there are however a good cases for storing latency offsets directly with every processor: Disk-I/O processors require absolute alignment. Offsets which are not relative to other route processors, but include the complete graph from input to output, are needed to facilitate read-ahead and write-behind semantics. Looking up actual port-latencies (upstream/downstream) latency by checking the port is expensive. Since the route is notified about changing connections and the session initiates a re-compute, the resulting capture and playback latencies can be cached until the next re-compute. So the route needs to store processor-alignment somewhere and keeping it with each processor is convenient, in particular since Ports are owned by IOProcessors. As shown in Listing 8, each processor stores 4 variables for both local and global alignment.

The raw source of `Route::update_signal_latency()` is a bit on the verbose side to reprint in full, particularly since various functions are abstracted by the IO object. The actual algorithm is described in Fig. 32.

Note that while each route does have a built-in delay-line, its latency is not reported and not taken into account for the route's own signal_latency. For tracks, this delay-line is ordered between the disk-writer and the disk-reader (see Fig. 12 in the section 7 architecture, The delay-line is only used to compensate for differences of capture to playback alignment. It does not alter capture latency of the disk-writer before it, nor does it change the playback latency of the disk-reader after it. For busses it is positioned before the Return (see Fig. 16 in section 12 and useful to align input-port to the return (e.g. Fig. 28).

```

class Processor {
/* ... */

public:
    /* return processor's own latency */
    virtual samplecnt_t signal_latency () const;
/* ... */

protected:
    /* relative to route */
    samplecnt_t input_latency;
    samplecnt_t output_latency;

    /* absolute alignment to session i/o */
    samplecnt_t capture_offset;
    samplecnt_t playback_offset;
/* ... */

```

Listing 8 – Processor latency variables

In either case the delay-line’s latency will not contribute to the overall graph latency compensation, it is effectively only used to align ports where min/max latency ranges differ.

18.2 Processing Route Buffers

Now that all information is in place, route-processing will be a piece of cake, well almost. The API that is provided to process the route buffers which invokes processors is given in Listing 9. It is very similar to that of `Route::run_route ()` described in section 18, and only has additional parameters for internal route-state.

The last three parameters are special cases coming from roll vs no-roll invocations: For the transition from no-roll to roll, the disk-playback needs to be declicked while input keeps flowing unmodified through the route. Gain-automation (fader, trim) is

Update Processor Latencies :

- $\mathcal{L}_{in} \leftarrow 0 ; \mathcal{L}_{out} \leftarrow 0 ; \mathcal{L}_{route} \leftarrow 0$
- $\mathcal{L}_{playback} \leftarrow$ output-port's playback latency
- $\mathcal{L}_{capture} \leftarrow$ input-port's playback latency
- Iterate over processors in reverse (last one first):
 - If the processor is a Send, set its delay-in to $(\mathcal{L}_{out} + \mathcal{L}_{playback})$
 - `processor.output_latency` $\leftarrow \mathcal{L}_{out}$
 - $\mathcal{L}_{out} \leftarrow \mathcal{L}_{out} + \text{processor.signal_latency}();$
- $\mathcal{L}_{route} \leftarrow \mathcal{L}_{out}$
- Iterate over processors in forward direction:
 - `processor.input_latency` $\leftarrow \mathcal{L}_{in}$
 - `processor.playback_offset` $\leftarrow \mathcal{L}_{route} - \mathcal{L}_{playback}$
 - `processor.capture_offset` $\leftarrow \mathcal{L}_{capture}$
 - $\mathcal{L}_{in} \leftarrow \mathcal{L}_{in} + \text{processor.signal_latency}();$
- Set the route's delay-line to delay for $(\text{upstream_playback_latency} - \text{downstream_playback_latency} - \text{signal_latency})$

Figure 32 – Algorithm to set per processor latencies, see also Listing 8.

skipped when not-rolling and can be disabled for certain monitoring situations. e.g. when recording new material, replacing existing audio or midi regions, the already recorded gain-automation curve will not apply to the new sound. Last but not least, the disk-reader will only be active when actually rolling (transport-speed is not zero, count-in is done and pre-roll for this track is complete).

The actual implementation of the `Route::process_output_buffers` function is rather long since it also handles edge-cases in the signal-flow: e.g. denormal-protection. Monitoring choices (silence buffers) and auditioning is special-cased. The function also

```

void
Route::process_output_buffers (BufferSet& bufs,
    samplepos_t start_sample, samplepos_t end_sample, pframes_t n_samples,
    int declick, bool gain_automation_ok, bool run_disk_reader)

```

Listing 9 – Processor latency variables

provides features for multi-channel polarity-inversion and automation-write-passes. Furthermore automation-lists are expanded and gain-curves for the current cycle are pre-computed at the beginning before invoking the actual processors.

The algorithm for processing the route’s buffers with latency-compensation is described in Fig. 33. The output of the route is aligned to route’s output-port playback latency.

Process Output Buffers (run processors):

- $\text{start_sample} \leftarrow \text{start_sample} + \mathcal{L}_{\text{route}} + \mathcal{L}_{\text{playback}}$
- $\text{end_sample} \leftarrow \text{end_sample} + \mathcal{L}_{\text{route}} + \mathcal{L}_{\text{playback}}$
- Iterate over processors:
 - `processor.run (bufs, start_sample, end_sample, n_samples, speed, ...);`
 - $\text{start_sample} \leftarrow \text{start_sample} - \text{processor.signal_latency} ()$
 - $\text{end_sample} \leftarrow \text{end_sample} - \text{processor.signal_latency} ()$

Figure 33 – Simplified router processor execution. Initially the sample is offset by the route’s own latency, which is the sum of all processor latencies. As the processors are called, the latency is reduced until after the last processor, `start_sample` aligns to the output.

There are two caveats: If a processor’s latency changes while it is being run, `start_sample` may become negative. Overall alignment and port-latencies cannot be updated from the process-callback itself. One solution is to query the proces-

sor.signal_latency () before invoking the processor, cache the value and only update it later during latency re-computation. In Ardour's case calling processor.run () with negative times is legal (they're clamped to zero if appropriate), so this is not damaging.

Seamless looping needs to be special-cased as well. If the offset start/end sample times cross the loop-boundary they needs to wrapped back the beginning of the loop. Likewise if subtracting the processor-latency crosses back over the loop-start. For time-aligned disk-i/o processors as well as automation events it will be necessary to split the processor.run() call for those cases. This is however trivial since buffers can be offset and no ports are directly involved in the process. In fact the same concept is used for sample-accurate automation inside Ardour's PluginInsert which may invoke the actual plugin in piecewise steps. IOProcessors that own ports are not sensitive to absolute time and need not be special-cased for seamless-looping.

18.3 Playback & Capture

Playback and capture are special processors because they perform disk input/output operations which are decoupled by a FIFO ringbuffer for real-time safety.

Recording is the easier part since it is always linear and the ringbuffer is fed in the real-time thread. It can be flushed to disk at any later time without special timing requirements. The only constraint is that the buffer must not overflow. Given that a floating-point mono audio-stream at 48kHz sample-rate is just 192 kBytes/sec and hard-disks are orders of magnitude faster²⁴, this is not an issue. While recording, the transport logic does not allow to locate the transport-position. This mainly to prevent accidents while actively recording. Loop-points as well as punch-in/out are known a-priori and can be handled.

²⁴An average USB2 hard-disk has a bandwidth of about 40 MByte/sec. Internal HDDs can reach 200 MBytes/sec, and SSD can write on the order 500-1000 MByte/sec. Ardour scales easily to 1000, or more tracks.

Playback on the other hand is often non-linear, and especially while editing, seeking is a common operation. The buffer feeding the disk-reader needs to be pre-filled and may not be ready by the time the disk-reader needs it.

18.4 Disk-writer - Write Behind

The disk-writer is designed to handle capture-alignment by itself. Recording always starts at a pre-defined position, usually the transport-position (audible sample) which is also indicated by the playhead in the editor GUI. As soon as the processor is invoked with a non-zero transport speed and its control-input is set to recording it marks the session's transport-sample as "capture-start-time". This is the position where the recorded region will eventually be positioned on the timeline. Also "first-recordable-sample", and, in case of loop-recording and dedicated punch-out, "last-recordable-sample" are set. First and last-recordable-sample positions define the record-range. Without dedicated a end time, last-recordable-sample is set to infinity.

Since latency is always positive and the session's transport-sample is aligned to output, the record-range will always be at, or after the global transport-sample.

If there is no pre-roll in a zero-latency session, the start_sample will match the session's current transport-sample and the disk-writer can immediately copy data from the input buffers to the capture-ringbuffer. In other cases the disk-writer will have to wait until it is invoked with a start/end sample range that passes first-recordable-sample. This is effectively a write-behind strategy.

Calculating overlap of the start/end range and the record-range is straight forward. There are four range overlap possibilities:

- Internal: The transport start/end range is smaller than record-range and complete enclosed by it $T_{start} \geq R_{start} \wedge T_{end} \leq R_{end}$: The complete input is written to the disk-ringbuffer.

```

capture_start_sample = _session.transport_sample ();
first_recordable_sample = _session.transport_sample () + capture_offset +
    playback_offset;

```

Listing 10 – Disk Writer Alignment, see Fig. 32 for definition of the offsets

- Start: The transport range start is earlier the record-start, and transport-end at, or before the record-range end $T_{start} < R_{start} \wedge T_{end} \leq R_{end}$: The first $R_{start} - T_{start}$ samples need to be skipped and writing to disk can begin.
- End: The transport range's end is after the record-end $T_{start} \geq R_{start} \wedge T_{end} > R_{end}$: Only the first $R_{end} - T_{start}$ samples need to be written.
- External: The record-range is shorter than the transport-range $T_{start} < R_{start} \wedge T_{end} > R_{end}$. Only $R_{end} - R_{start}$ samples after $R_{start} - T_{start}$ samples needs to be written (this can happen for very short loop-recording with a large buffer size).

When the route invokes the processor, the start/end sample positions are latency compensated and the corresponding data matches the given position. Since the incoming data is equally aligned to the output, the position is directly related to the playhead position, regardless of the signal-path prior to reaching any disk-writer. Thus placing the “first-recordable-sample” to “capture-start-time”, automatically aligns all inputs. The computation is given in Listing 10. Note that alignment of the processor inside the route’s processor-stream is handled by start/end sample, and the disk-writer only need to take the absolute alignment into account.

However this works only as long as the input to the disk-writer comes from an actual input-port or a track that aligns to output.

In case a track is used to record the output of the master-bus itself, and the track’s outputs are not connected to the same output as the master-bus, playback_offset will differ. A similar situation can occur when bouncing data from a track that is not

connected to any input. The disk-writer may need to pick up data from other tracks during pre-roll, even if the route itself is not yet rolling. This displayed in the following example:

no-input || Track 1 play → latency A $\xrightarrow{\text{port}}$ Track 2 capture → latency B $\xrightarrow{\text{port}}$ out.

Track 2 needs to start recording after pre-roll passed Track 1's threshold of latency A, regardless of its own pre-roll alignment.

The disk-reader distinguishes two different alignment modes that can be set automatically depending on I/O connections. In case there is no physical input in the path upstream to the source, the alignment specified in Listing 10 changes to

```
capture_start_sample = first_recordable_sample ← transport_sample
```

and recording only depends on start/end samples.

In case of Ardour, the choice of capture alignment is also exposed to the user in a track's GUI.

18.5 Disk-reader - Read Ahead

The disk-reader depends on data being available in a ringbuffer to read from and needs to know the absolute position of the data read from disk into the buffer. Since the ringbuffer is fed by the butler thread asynchronously, there is no guarantee that the required start/end range during run() is available.

When locating the playhead, the ringbuffer is filled starting at the transport-sample. Since this corresponds to the audible sample it is the earliest sample needed by the disk-reader and is marked as `playback_sample` in the disk-reader. Once the disk-reader's `run` is called with `start_sample` equal or larger than `playback_sample` it can play back. If the playback-latency is zero, the transport-sample always matches the playback-sample.

If the output latency is non-zero, the disk-reader processor always outputs future

data from disk, which is then delayed by latent processors or ports-latencies and will eventually arrive at the output at a time corresponding to the transport-sample. The disk-reader can initially skip data that is available in the ringbuffer. The only exception is pre-roll where the disk-processor may need to start directly at playback_sample.

After a locate operation is can become possible that only some disk-readers have sufficient data available in their ringbuffers, while others are still being re-filled. This can result in a partial playback of only some tracks, which is undesirable when editing or playing.

To prevent this condition, Ardour's main session process-callback verifies if all disk-readers can play, and a flag `DiskReader::set_no_disk_output (bool)` is set appropriately for all tracks. This allows disk-readers which do have available data to silently pop samples off the ringbuffer as needed and keep the playback-sampled aligned to to sample-start.

This ringbuffer implementation is however not appropriate for a latency compensated graph. Some disk-readers will need to start playback earlier than others and in some cases may need to jump backwards e.g. when rolling while a latent plugin is added.

The solution for this case is a random-access-ringbuffer that has no dedicated read-pointer and the absolute alignment of data in kept directly with the buffer. Such a RAM-ringbuffer would only have a write-pointer which marks the end of the available data and an alignment position which is kept in sync. The reader can access all valid data in the buffer at any position.

Since reading does not longer invalidate data from the buffer, this facilitates reverse playback as well as micro-locates in either direction. Ardour's disk-i/o buffers are on the order of seconds (default 10sec), keeping a history of 8k samples would be negligible, yet allow for major latency changes without the disk-reader needing to wait for a complete buffer-refill.

Lock-free implementation of such a ringbuffer are not trivial on modern CPU

architectures which only offer atomic CPU operations for 32bit integers: Both the write-pointer as well as the position of data in the buffer need to be set atomically. One solution would be a spin-lock (assuming update is fast), and another possibility is to use two halves of a 32bit word which can be updated atomically. The latter work-around is however not an option on a 64bit timeline.

18.6 Capturing Processor

The capturing-processor is a specialized disk-writer for raw unprocessed stem-export.

Ardour supports three export modes:

- Session export: Usually master-bus output is written to disk. But any combination of track or bus output can be selected.
- Stem export with processing: Any combination of track/bus output ports are written to separate files, one for each channel. This is useful as multi-channel interchange format.
- Stem export without processing: raw disk-reader data of selected tracks (or bus inputs) are exported to separate files.

Since Ardour aligns ports, both the master-out (default session export) and stem-export with processing are already aligned and their port-latency is known. Export only needs to initially skip the additional playback-latency coming from outside of the graph (if any).

However tapping off the data directly after the disk-reader will be offset by any downstream latencies in the route. The capturing-processor is basically a fixed delay-line that compensates for all latencies between a route's disk-reader and the final downstream output. The required delay is given by `disk_reader.output_latency() + $\mathcal{L}_{playback}$` .

This capturing processor has a dedicated output buffer which is used directly in the export-process. It is only added on demand, when configuring stem-export parameters and is otherwise not present.

18.7 Side-chains and Modulation

Another special-case are side-chain inputs directly feeding a plugin. They do not represent a new concept and are very similar to send/return pairs. In fact, Ardour uses IOProcessor Sends as source to feed a side-chain input. The return/receiver part is a IO which is owned by the PluginInsert processor. Setting the processor-latency of the PluginInsert can directly update relevant information for the side-chain target, which has the same latencies as the processor itself.

Since incoming data is available to the processor before it will evaluate processor control-data it can be used to modulate processor parameters if needed.

19 Vari-speed

Variable speed playback allows to slow down or speed up the playback rate. There are two ways to facilitate vari-speed playback. One is local to the disk-readers (and potentially disk-writers), the alternative is to use a global vari-speed stage directly at the inputs and outputs.

Up until Ardour 5 a local approach is used: The session's main process-callback computes the number of samples that need to be played according to playback speed and process-buffer-size. All disk-readers in the session use cubic interpolation to progress the playback_sample by given number of samples. This approach worked in Ardour because there was no proper latency compensation. Ardour 5 has no support for cue-monitoring, a track is either playing from disk or passing through input, never both which mitigated the issue to some extend.

When slaved to external timecode sources, the transport speed will usually be different in every process cycle to correct for drift. As opposed to a user setting a given speed, there is not a single fixed speed value that is valid for continuous cycles.

In order to use local vari-speed in context of latency-compensated processing, the speed of each disk-reader would need to be maintained locally, offset by each disk-reader's output latency and each disk-reader may have a different speed in the same process-cycle depending on its playback latency. Also with local vari-speed processing, every track in the session needs to resample the data from disk. There can be a considerable amount of disk-readers. Sessions with 64 or 128 tracks a common, a few hundred tracks are not unusual, and the number of resampling operations can become very CPU intense.

A simpler way is to resample port-data directly at the engine-level: up/down-sample all input ports at the beginning, and down/up-sample output ports using the inverse ratio at the end of the session's process cycle. The session itself is unaware of the speed-change, and only needs to handle transport speed of $\{-1, 0, +1\}$.

Live signals are upsampled on input and downsampled on output which is effectively a no-op. This approach also allows for variable-speed recording without dedicated support in the disk-writer. It further allows to resample synthesized MIDI instruments along with the audio. The local midi-event disk-reader only sped-up/slowed down the event emission. Resampling consistently changes the actual audible pitch.

In the case of Ardour, which has an abstract port concept as described in section 10, this is even easy to implement. The Port API does allow to completely abstract hardware speed from the engine. At cycle-start the engine can iterate over all ports, read the actual data from the port-engine and resample it into a local-buffer. DAW processing then just use this buffer instead of the port-engine provided one. At cycle-end the reverse action is performed.

Since the port API also abstracts port-latencies, they can be adjusted in a similar

manner. Some care has to be taken to multiply hardware latencies passed from the DAW to the port while dividing hardware latencies passed to the DAW by the playback-speed. Furthermore the latency of the resample-algorithm needs to be taken into account. Since Ardour uses samples as unit for latencies, all latencies inside the DAW, route and processor latencies, are not affected.

Resampling is CPU intense and does degrade the signal quality, this is particularly true for real-time resamplers that also need to support variable rates. While in many cases artefacts are below usual signal/noise ratios it is still prudent to avoid them whenever possible and skip resampling for the common case when the absolute speed equals 1.0.

Since resampling algorithms do need context, resampler-buffers need to be filled in every callback, even if no resampling occurs. This allows for immediate speed changes in the next cycle.

Timecode readers which do provide the target-speed need to be called before resampling and set the resampling ratio for ports.

Timecode generators (including the metronome/click) may be on either side, depending on the case at hand. In general timecode generators that support vari-speed basically just produce sped-up or slowed-down timecode which is not unlike resampling the data or multiplying the effective sample-time of events. Differences are either of a practical nature (e.g. pitch-shifting of metronome click) or related to timecode specification (e.g. LTC signal rise-times that can not be maintained while resampling).

Part V

Conclusion

The computer scientific approach to describe about how to approach and solve a problem is more than often at odds with the actual development processes.

This is particularly true for a continuously changing large codebase with contributions from a team of developers. The commercial side is often more interested in marketable features as opposed to algorithmic correctness and completeness.

While proper time alignment and basic plugin delay compensation is a prerequisite for any professional DAW, complete graph latency compensation with flexible multi-channel routing is unprecedented and a milestone. Particularly for the Ardour project which strived to provide this for a long time.

This thesis provides a very in-depth view of the inner workings of the Ardour 6 engine and presents concepts that can be applied to Audio Workstations in general.

Large parts that have been described in later sections are still evolving in the Ardour-codebase and this thesis aims to explain concepts in order to motivate re-factoring, serve as guideline, and progress development.

Especially in context of pro-audio engineering reliability is a key-factor. Writing real-time (reliable) audio software for general purpose operating systems requires adherence to principles[35] and requires strict attention. Having access to a documentation and a complete process description is a valuable assert while implementing, in particular for cases involving time. Drafts of this thesis already helped to end recurring confusing discussions “We can shift this forward in time”, “No, we need to shift all that backward in time.”; most of which originated from mixing different conceptual approaches in the middle of some deep development process.

Breaking the problem apart in dedicated confined blocks as presented in the archi-

tectural overview, is what allows to break down the complexity and make the complete project manageable.

This is particularly relevant for implementing latency compensation which affects nearly all major places in the backend, the central part of processing and covers many key-features behind the scenes. A vital part for of any large software project is getting the APIs right.

During prototyping and development various iterations have been unit-tested. The system as whole was hands-on testing by experienced audio-engineers throughout various stages and is being adopted by commercial vendors.

Ardour's quest for latency compensation started in 2008/09. The answer to "How hard can it be?" can be found in these pages.

Yet as any experienced software engineer who has released a large piece of end-user software into the wild can attest: It's a never-ending story. Development takes place openly at <https://github.com/ardour/ardour/>.

Part VI

Appendix

A Delaylines in Ardour

Two types of delaylines were developed for Ardour during the course of this thesis:

- DelayLine *is-a* Processor: a variable delayline using the processor API
- FixedDelay a stand-alone implementation operating on Buffers

Both cases can handle multi-channel audio and MIDI data.

A.1 ARDOUR::DelayLine

The ARDOUR::Delayline offers a processor API and can easily be used in a route. It maintains an internal buffer for real-time safe delay-changes and cross-fades for a smooth transition.

A second pending buffer is used for asynchronous delay-time changes, The method `DelayLine::set_delay` allocates a larger buffer if needed and switch-over happens seamlessly at next `run()` invocation. By default the processor is hidden and not displayed visibly in the signal-flow.

A.2 ARDOUR::FixedDelay

ARDOUR::FixedDelay on the other hand is a simple synchronous delay. It operates on dedicated buffers, which are pre-allocated in order to allow real-time safe operation. A maximum size (max delay-time in samples) needs to be specified when instantiating the delayline, but it can grow (or be shrunk) later if needed. There is no interpolation

or cross-fades when changing the delay and the delay-time is directly specified on every run () call.

In Ardour it used for the capture-processor during export and by the PluginInsert for bypassing latent plugins, or bypassing dedicated plugin-pins.

B Verification Tools

Looking at the git revisions of the Ardour codebase one will notice that after the building blocks were all in place, the actual implementation of the presented concepts was perhaps two weeks worth of effort. Followed by month and months of debugging, that is still ongoing work. In a project of about 650000 lines of code, it is not unexpected to have 5000 or more bugs. While various stages can be unit-tested, the vast majority of issues in Ardour are due to the overall complexity. It is therefore important to perform quality assurance and test the implementation on the complete system.

B.1 Bounce recording via physical cable

Vertias ex machina.

Closed loop testing is a very simple and the ultimate real-world test. Simply playing a signal from one track out to hardware, looping it back using a cable to the soundcard and re-recording it onto another track simulates a player, playing along to the original. A setup is depicted in Image 34.

Testing is a time-consuming task, that cannot easily be automated. Regardless many tests have been performed, using existing user-contributed sessions, as well as special constructed sessions to test edge-case.

The current development version of Ardour 6 holds up, although the implementation is not yet complete. Testing while developing does help to find issues early, and also

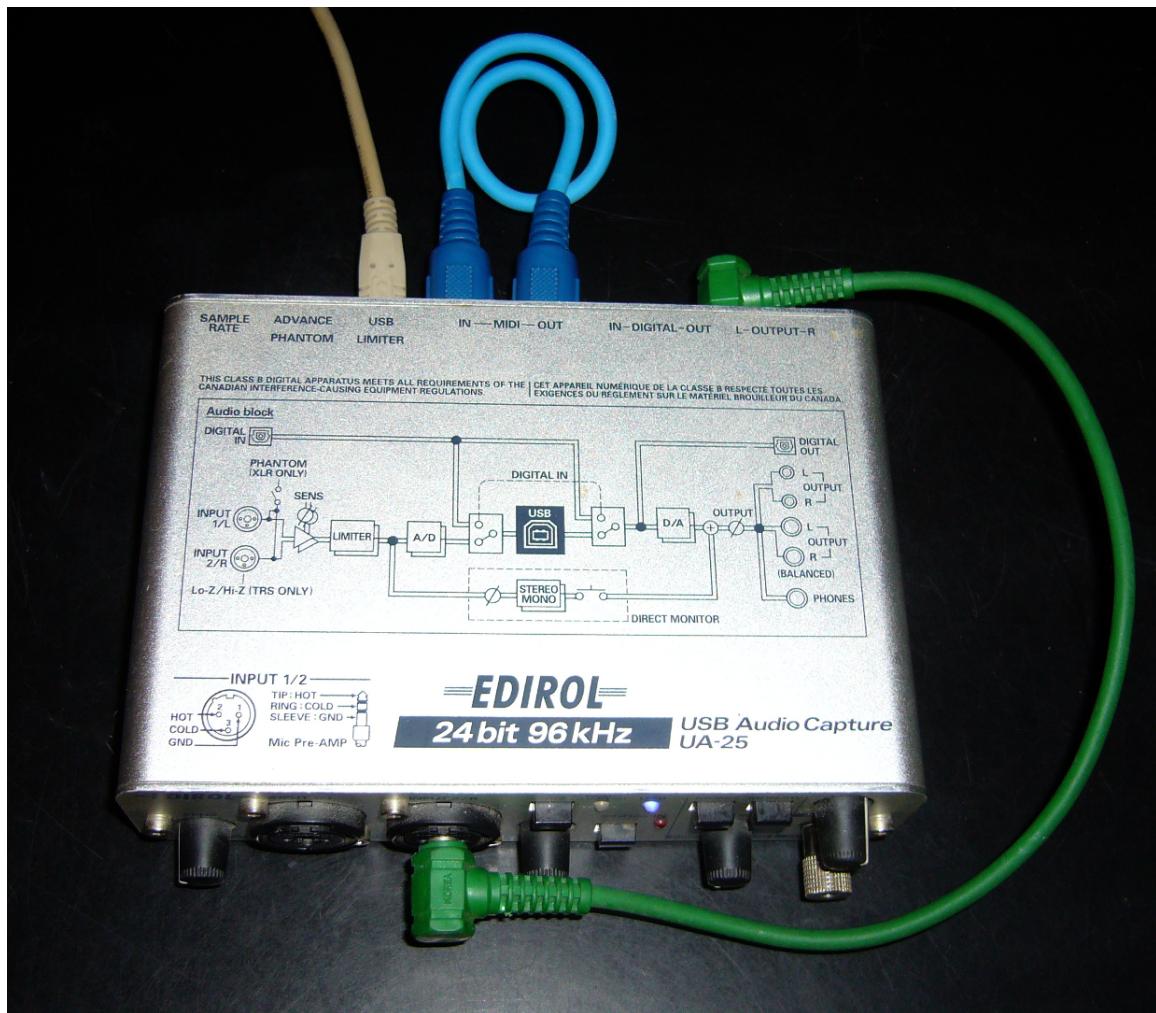


Figure 34 – Closed loop Audio + MIDI testing

reduces debugging complexity in the long run.

The same mechanism of a closed-loop is actually used by the audio-engine to calibrate systemic hardware latencies as mentioned in section 4. Yet in the measurement case no additional session or route processing is taken into account.

Bounce tests of Ardour 6 are being performed with both audio, MIDI and mixed signal flow.

B.2 Introspection

While it was not intended for debug-purposes, adding Lua scripting support to Ardour turned out to be a major time-safer for tracking down bugs and handling implementation edge-cases.

It also allows to directly verify if the internal state was set correctly. In this case port-latencies and delay-settings for various test-cases.

As opposed to adding `print()` statements to the source, scripting allows to query values on demand and in a real-time safe manner. The result can be formatted and directly compared to expected values.

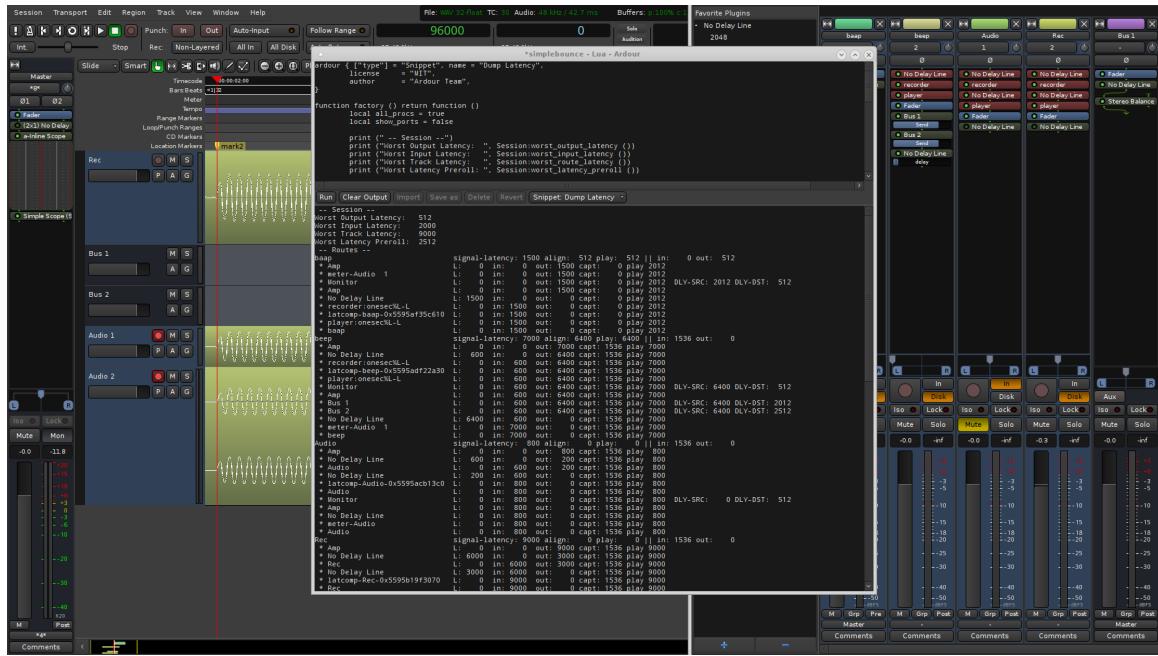


Figure 35 – A test-session, verifying alignment of bounce-processing with a Lua-script showing processor latency information (here: Ardour 6.0-pre0-148).

B.3 No Delay Line

Nodelay.lv2 is an audio delay-line plugin, with a variable, interpolated delay up to

192k samples. It can optionally report its delay as latency, in which case the effect should be transparent when used with hosts that implement latency compensation.

The plugin provides for a variable latency which can be changed at runtime. Since it's a plugin it can be inserted in any position in the signal flow and re-positioned on the fly.

It is useful as instrumentation tool and artificial latency [36], and has become invaluable in test-sessions to verify alignment.

The nodelay plugin was developed as part of work on this thesis during initial prototyping of latency-compensation in Ardour3.

B.4 First Non Zero Sample

For verifying export alignment, another small utility was developed that prints the time of the first non-zero audio-sample in an audio-file. Since it is rather ad-hoc it has never been published, yet it is short enough to print it verbatim here (Listing 11).

Tests are performed by placing a simple beep sound or similar sound at a given position on the timeline after an initial period of silence. When exporting or stem-exporting various tracks with different latencies, the resulting files are tested to have the first non-silent sample at the exact same position.

It's also useful to verify the inverse, with phase-cancellation all samples need to be zero.

B.5 Audio/MIDI Alignment Testing

Alignment if Audio and MIDI was tested with an analog signal tap as depicted in Fig. 36. A MIDI signal passing through is fed to the same audio-interface as sound-signal which can be recorded alongside the MIDI signal to verify alignment.

```
// gcc -o firstNZsample firstNZsample.c -lsndfile
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sndfile.h>

int main (int argc, char **argv) {
    SF_INFO nfo;
    SNDFILE* sndfile;
    if (argc < 2) {
        fprintf (stderr, "Usage: %s <audio-file>\n\n", argv[0]);
        return -1;
    }
    memset (&nfo, 0, sizeof (SF_INFO));
    if (0 == (sndfile = sf_open (argv[1], SFM_READ, &nfo))) {
        return -1;
    }
    float* buf = malloc (nfo.channels * sizeof (float));
    if (!buf) return -1;
    for (long int s = 0; s < nfo.frames; ++s) {
        sf_readf_float (sndfile, buf, 1);
        for (int c = 0; c < nfo.channels; ++c) {
            if (buf[c] != 0) {
                printf ("First non-zero sample: %ld (%d)\n", s, c + 1);
                goto end;
            }
        }
    }
    printf ("All samples are zero.\n");
end:
    sf_close (sndfile);
    free (buf);
    return 0;
}
```

Listing 11 – Inspect an audio-file and print its first non-zero sample

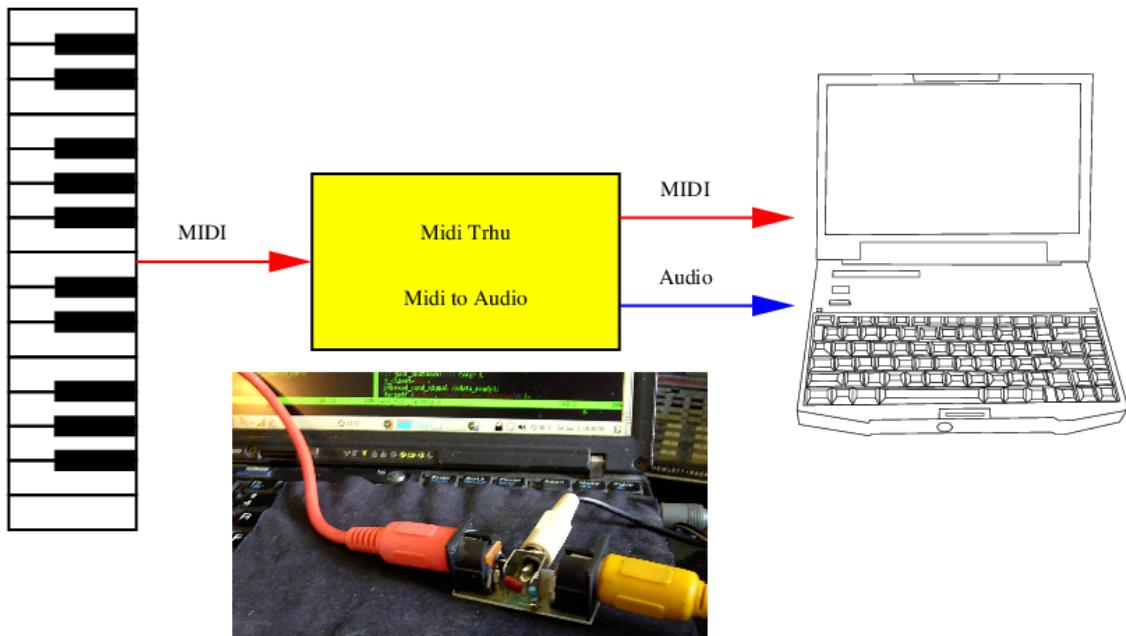


Figure 36 – Audio/MIDI alignment testing

C Audio Signal Measurement and Visualization

This part was previously published as paper at ICMC 2014

Some parts, in particular the oscilloscope and mixer/trigger came to be as diagnostic tools while prototyping latency-compensation

Many users of audio software at some time face problems that require reliable measurement.

An engineer reading and using audio level meters is comparable to a musician reading or writing sheet-music. Also, just like there are virtuoso musicians who can't read a single note, there are great sound-engineers who just go by their ears and produce great mixes and masters without ever looking at a single meter (Although this doesn't imply that the use of meters is the option for those who can't hear!).

C.1 On Measurements

As part of this thesis various audio-productions were carried out to verify the results and evaluate trade-off situations. The most important tool - particularly for verifying alignment - is an oscilloscope.

While there are fine hardware Scopes available, most software scopes are merely toys suitable for visualizing waveforms, unsuitable for any scientific measurements, see Section C.3.

The general lack of professional standardized equipment motivated the author to not only include various measurement tools directly in the Ardour DAW [37] as well as standalone implementation: [38]. A quality scope has been released at [39].

C.2 Introduction to Audio Signal Meters

Audio level meters are very powerful tools that are useful in every part of the production chain:

- When tracking, meters are used to ensure that input signals do not overload and maintain reasonable headroom.
- Meters offer a quick visual indication of activity when working with a large number of tracks.
- During mixing, meters provide a rough estimate of the loudness of each track.
- At the mastering stage, meters are used to check compliance with upstream level and loudness standards, and to optimise the dynamic range for a given medium.

Similarly for technical engineers, reliable measurement tools are indispensable for the quality assurance of audio-effects or any professional audio-equipment.

C.3 Meter Types and Standards

For historical and commercial reasons various measurement standards exist. They fall into three basic categories:

- Focus on **medium**: highlight digital number, or analogue level constraints.
- Focus on **message**: provide a general indication of loudness as perceived by humans.
- Focus on **interoperability**: strict specification for broadcast.

For in-depth information about metering standards, their history and practical use, please see [40] and [41].

Digital peak-meters

A Digital Peak Meter (DPM) displays the absolute maximum signal of the raw samples in the PCM signal (for a given time). It is commonly used when tracking to make sure the recorded audio never clips. To that end, DPMs are calibrated to 0dBFS (Decibels relative to Full Scale), or the maximum level that can be represented digitally in a given system. This value has no musical connection whatsoever and depends only on the properties of the signal chain or target medium. There are conventions for fall-off-time and peak-hold, but no exact specifications. Furthermore, DPMs operate on raw digital sample data which does not take inter-sample peaks into account, see section C.3.

RMS meters

An RMS (Root Mean Square) type meter is an averaging meter that looks at the energy in the signal. It provides a general indication of loudness as perceived by humans.

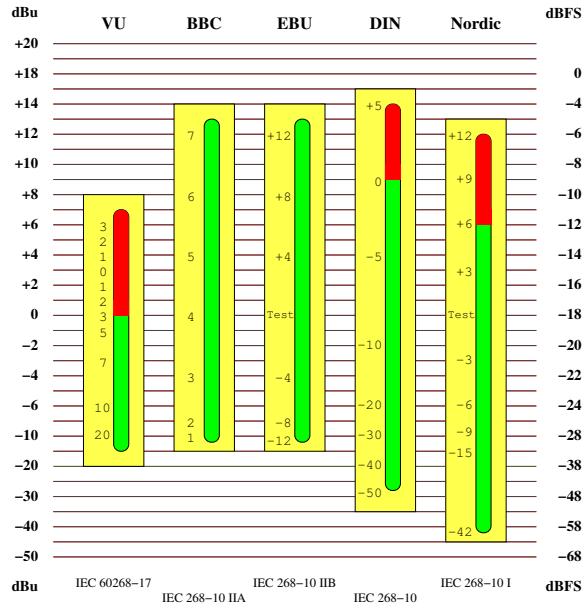


Figure 37 – Various meter alignment levels as specified by the IEC. Common reference level is 0dBu calibrated to -18dBFS for all types except for DIN, which aligns +9dBu to -9dBFS. dBu refers to voltage in an analogue system while dBFS to digital signal full-scale.

Bar-graph RMS meters often include an additional DPM indicator for practical reasons. The latter shows medium specifics and gives an indication of the crest-factor (peak-to-average power ratio) when compared to the RMS meter.

Similar to DPM's, there is no fixed standard regarding ballistics and alignment level for a general RMS meter, but various conventions do exist, most notably the K-system introduced by Bob Katz [42].

IEC PPMs

IEC (International Electrotechnical Commission) type Peak Programme Meters (PPM) are a mix between DPMs and RMS meters, created mainly for the purpose of interoperability. Many national and institutional varieties exist: European Broadcasting Union (EBU), British Broadcasting Corporation (BBC), Deutsche Industrie-Norm

(DIN),... [43].

These loudness and metering standards provide a common point of reference which is used by broadcasters in particular so that the interchange of material is uniform across their sphere of influence, regardless of the equipment used to play it back. See Fig. 37 for an overview of reference levels.

For home recording, there is no real need for this level of interoperability, and these meters are only strictly required when working in or with the broadcast industry. However, IEC-type meters have certain characteristics (rise-time, ballistics) that make them useful outside the context of broadcast.

Their specification is very exact [44], and consequently, there are no customisable parameters.

EBU R-128

The European Broadcast Union recommendation 128 is a rather new standard, that goes beyond the audio-levelling paradigm of PPMs.

It is based on the ITU-R BS.1770 loudness algorithm [45] which defines a weighting filter amongst other details to deal with multi-channel loudness measurements. To differentiate it from level measurement the ITU and EBU introduced a new term ‘LU’ (Loudness Unit) equivalent to one Decibel²⁵. The term ‘LUFS’ is then used to indicate Loudness Unit relative to full scale.

In addition to the average loudness of a programme the EBU recommends that the ‘Loudness Range’ and ‘Maximum True Peak Level’ be measured and used for the normalisation of audio signals [46].

The target level for audio is defined as -23 LUFS and the maximum permitted true-peak level of a programme during production shall be -1 dBTP.

²⁵the ITU specs uses ‘LKFS’, Loudness using the K-Filter, with respect to Full Scale, which is exactly identical to ‘LUFS’.



Figure 38 – Various meter-types from the meter.lv2 plugin bundle fed with a -18 dBFS 1 kHz sine wave. Note, bottom right depicts the stereo phase correlation meter of a mono signal.

The integrated loudness measurement is intended to quantify the average program loudness over an extended period of time, usually a complete song or an entire spoken-word feature. [47], [48].

Many implementations go beyond displaying range and include a history and

histogram of the Loudness Range in the visual readout. This addition comes at no extra cost because the algorithm to calculate the range mandates keeping track of a signal's history to some extent.

Three types of response should be provided by a loudness meter conforming to R-128:

- **Momentary** response. The mean squared level over a window of 400ms.
- **Short** term response. The average over 3 seconds.
- **Integrated** response. An average over an extended period.

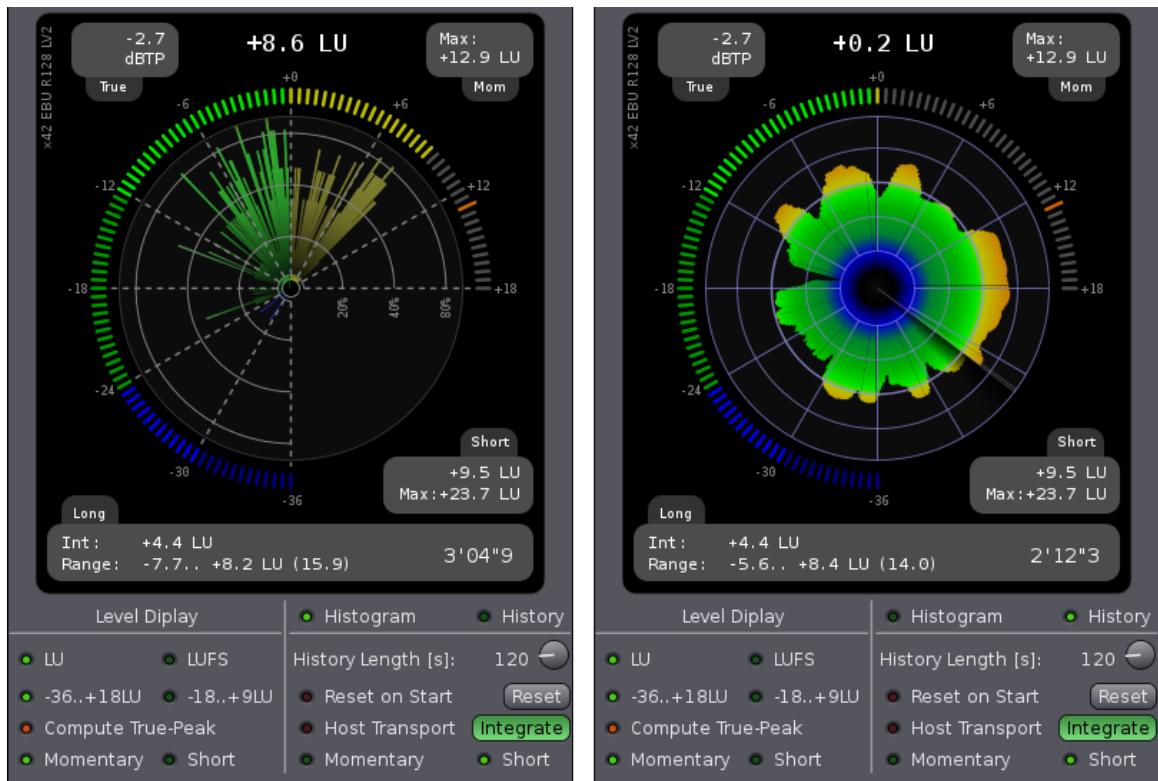


Figure 39 – EBU R-128 meter GUI with histogram (left) and history (right) view.

VU meters

Volume Unit (VU) meters are the dinosaurs (1939) amongst meters.

The VU-meter (intentionally) ‘slows’ measurement, averaging out peaks and troughs of short duration, and reflects more the perceived loudness of the material [49], and as such was intended to help program producers create consistent loudness amongst broadcast program elements.

In contrast to all the previously mentioned types, VU metes use a linear scale (in 1939 logarithmic amplifiers were physically large). The meter’s designers assumed that a recording medium with at least 10 dB headroom over 0 VU would be used and the ballistics were designed to “look good” with the spoken word.

Their specification is very strict (300ms rise-time, 1 - 1.5% overshoot, flat frequency response), but various national conventions exist for the 0VU alignment reference level. The most commonly used was standardised in 1942 in ASA C16-5-1942: “The reading shall be 0 VU for an AC voltage equal to 1.228 Volts RMS across a 600 Ohm resistance”²⁶

Phase Meters

A phase-meter shows the amount of phase difference in a pair of correlated signals. It allows the sound technician to adjust for optimal stereo and to diagnose mistakes such as an inverted signal. Furthermore it provides an indication of mono-compatibility, and possible phase-cancellation that takes place when a stereo-signal is mixed down to mono.

Stereo Phase Correlation Meters

Stereo Phase Correlation Meters are usually needle style meters, showing the phase from 0 to 180 degrees. There is no distinction between 90 and 270 degree phase-shifts

²⁶This corresponds to +4dBu

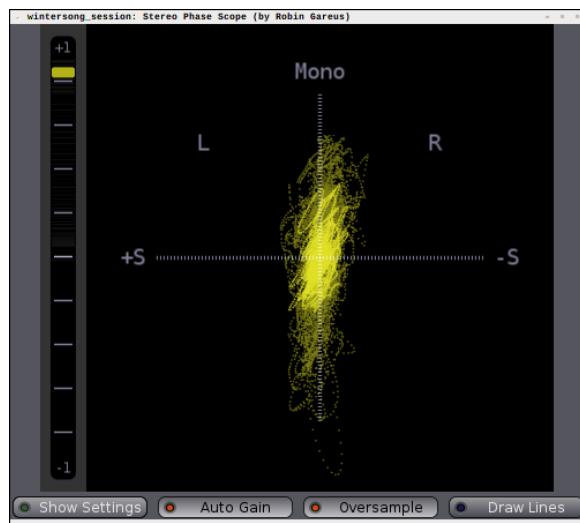


Figure 40 – Goniometer (Phase Scope)

since they produce the same amount of phase cancellation. The 0 point is sometimes labelled “+1”, and the 180 degree out-of-phase point “-1”.

Goniometer

A Goniometer plots the signal on a two-dimensional area so that the correlation between the two audio channels becomes visually apparent (example in Fig. 40). The principle is also known as Lissajous curves or X-Y mode in oscilloscopes. The goniometer proves useful because it provides very dense information in an analogue and surprisingly intuitive form: From the display, one can get a good feel for the audio levels for each channel, the amount of stereo and its compatibility as a mono signal, even to some degree what frequencies are contained in the signal. Experts may even be able to determine the probable arrangement of microphones when the signal was recorded.

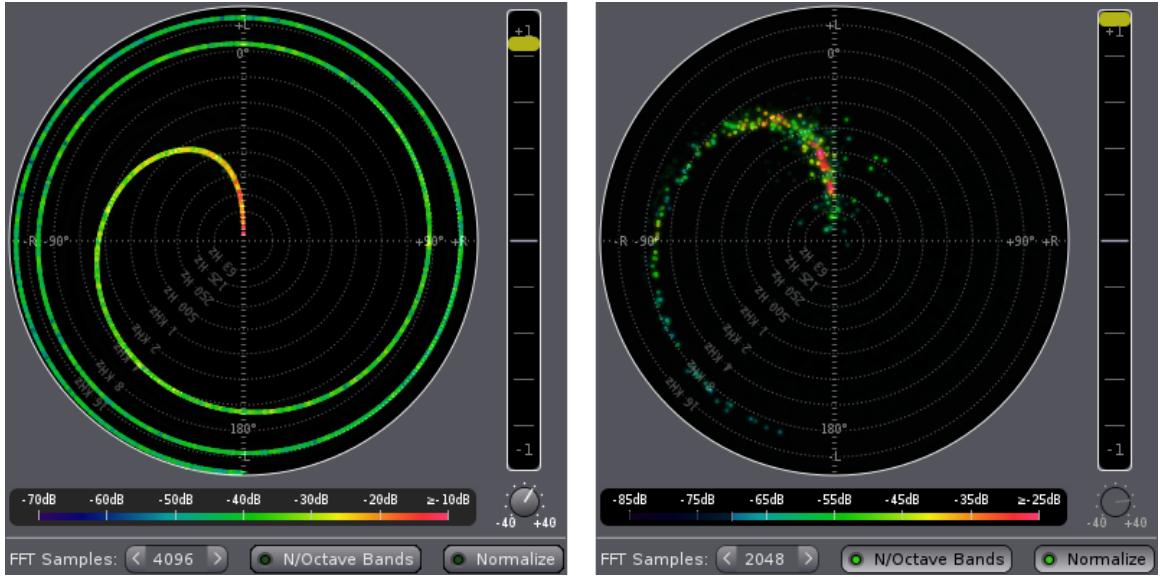


Figure 41 – Phase/Frequency Wheel. Left: pink noise, 48KSPS with right-channel delayed by 5 samples relative to left channel. Right: Digitalisation of a mono 1/2" tape reel with slight head misalignment.

Phase/Frequency Wheel

The Phase Wheel is an extrapolation of the Phase Meter. It displays the full 360 degree signal phase and separates the signal phase by frequency. It is a rather technical tool useful, for example, for aligning tape heads, see Fig. 41

Digital True-Peak Meters

A True-Peak Meter is a digital peak meter with additional data pre-processing. The audio-signal is up-sampled (usually by a factor of four [45]) to take inter-sample peaks into account. Even though the DPM uses an identical scale, true-peak meters use the unit dBTP (decibels relative to full scale, measured as a true-peak value – instead of dBFS). dBTP is identical to dBFS except that it may be larger than zero (full-scale) to indicate peaks.

Inter-sample peaks are not a problem while remaining in the digital domain, they can however introduce clipping artefacts or distortion once the signal is converted back to an analogue signal.

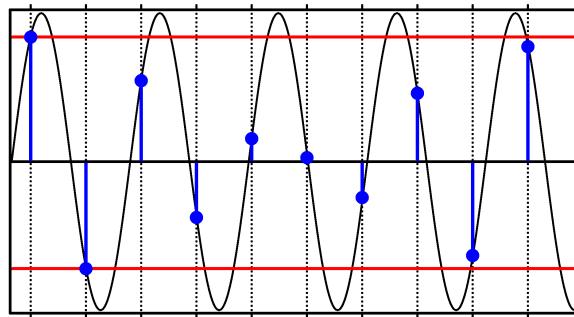


Figure 42 – Inter-sample peaks in a sine-wave. The red line (top and bottom) indicates the digital peak, the actual analogue sine-wave (black) corresponding to the sampled data (blue dot) exceeds this level.

floating point audio data	mathematical true peak value
.. 0 0 +1 +1 0 0 ..	+2.0982 dBTP
.. 0 0 +1 -1 0 0 ..	+0.7655 dBTP

Table 8 – True Peak calculations @ 44.1 KSPS, both examples correspond to 0dBFS.

Fig. 42 illustrates the issue. Inter-sample peaks are one of the important factors that necessitate the existence and usage of headroom in the various standards, Table 8 provides a few examples of where traditional meters will fail to detect clipping of the analogue signal.

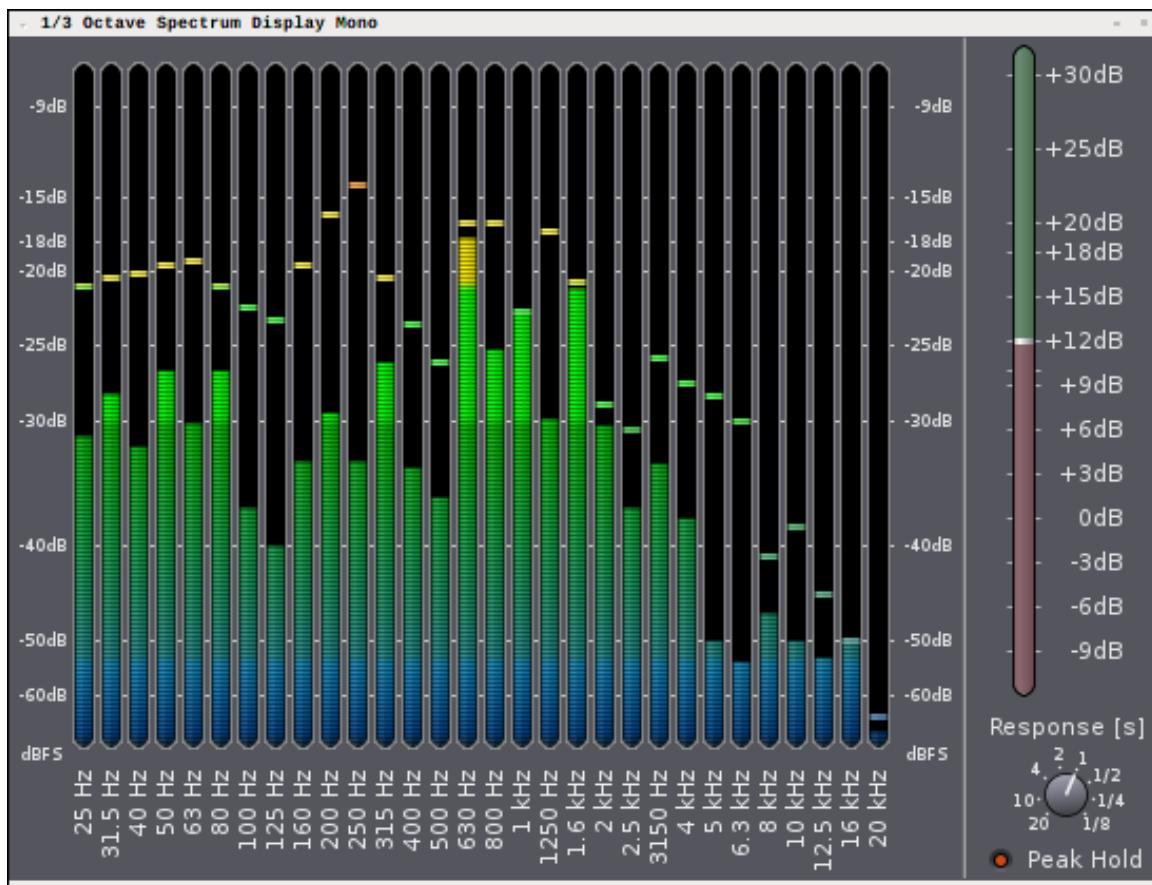


Figure 43 – 30 Band 1/3 octave spectrum analyser

Spectrum Analysers

Spectrum analysers measure the magnitude of an input signal versus frequency. By analysing the spectra of electrical signals, dominant frequency, power, distortion, harmonics, bandwidth, and other spectral components can be observed. These are not easily detectable in time domain waveforms.

Traditionally they are a combination of band-pass filters and an RMS signal level meter per band which measures the signal-power for a discrete frequency band of the spectrum. This is a simple form of a perceptual meter. A well known specification is a 1/3 octave 30-band spectrum analyser standardised in IEC 61260 [50]. Frequency

bands are spaced by octave which provides a flat readout for a pink-noise power spectrum, which is not unlike the human ear.

As with all IEC standards the specifications are very precise, yet within IEC61260 a number of variants are available to trade off implementation details. Three classes of quality are defined which differ in the filter-band attenuation (band overlap). Class 0 being the best, class 2 the worst acceptable. Furthermore two variants are offered regarding filter-frequency bands, base ten: $10^{\frac{x}{10}}$ and base two: $2^{\frac{x}{3}}$. The centre frequency in either case is 1KHz, with (at least) 13 bands above and 16 bands below.

In the digital domain various alternative implementations are possible, most notably FFT and signal convolution approaches²⁷.

FFT (Fast Fourier Transform, an implementation of the discrete Fourier transform) transforms an audio signal from the time into the frequency domain. In the basic common form frequency bands are equally spaced and operation mode produces a flat response for white noise.

For musical applications a variant called ‘perceptual analysers’ is widespread. The signal level or power is weighted depending on various factors. Perceptual analysers often feature averaging functions or make use of screen-persistence to improve readability. They also come with additional features such as numeric readout for average noise level and peak detection to mitigate effects introduced by variation in the actual display.

Oscilloscopes

The oscilloscope is the “jack of all trades” of electronic instrumentation tools. It produces a two-dimensional plot of one or more signals as a function of time.

It differs from a casual wave-form display, which is often found in audio-applications,

²⁷There are analogue designs to perform DFT techniques, but for all practical purposes they are inadequate and not comparable to digital signal processing.

in various subtle but important details: An oscilloscope allows reliable signal measurement and numeric readout. Digital wave-form displays on the other hand are operating on audio-samples - as opposed to a continuous audio-signal. Figure 44 illustrates this.

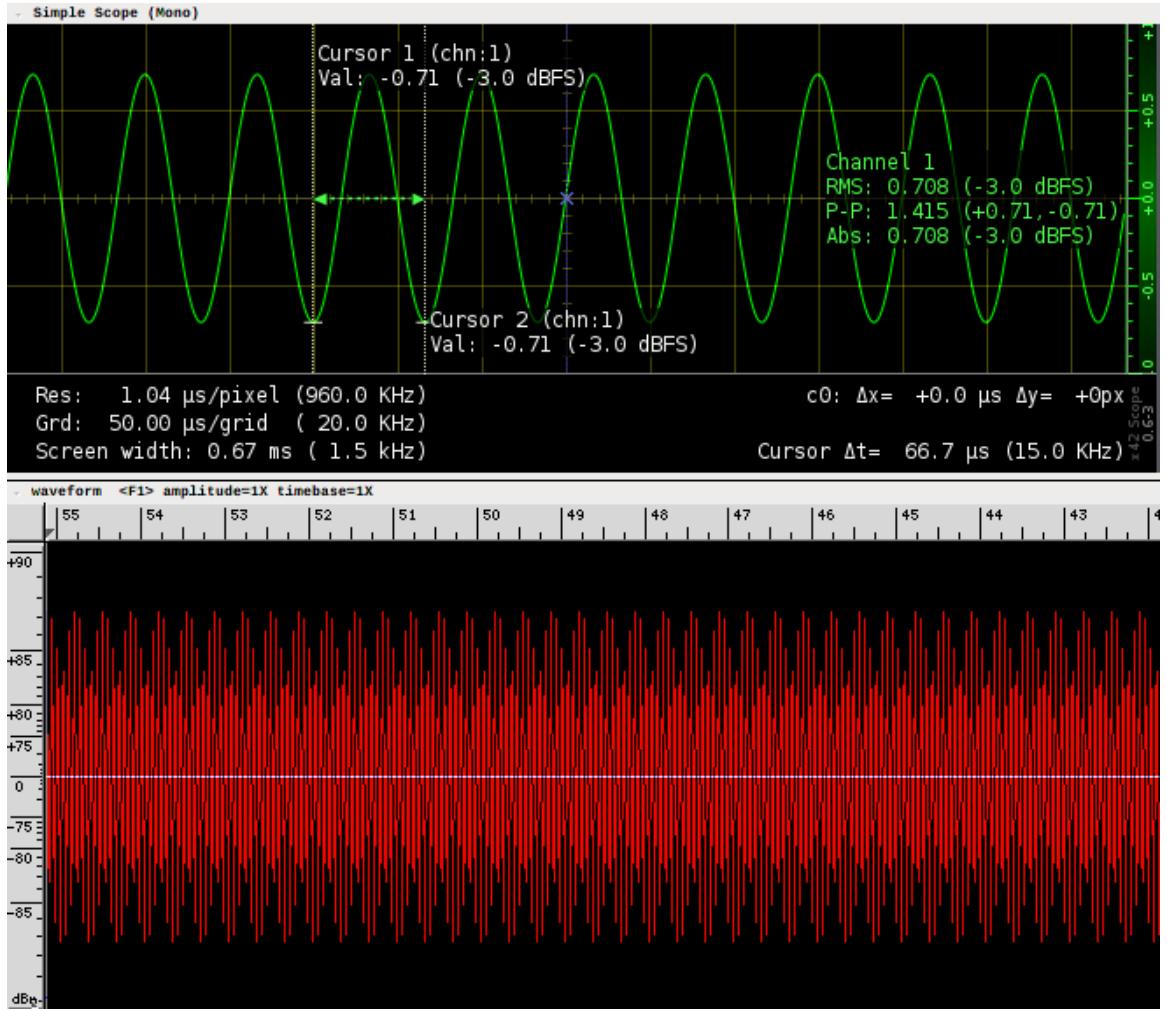


Figure 44 – 15KHz, -3dBFS sine wave sampled at 48KSPS. The Oscilloscope (top) up-samples the data to reproduce the signal. The wave-form display (bottom) displays raw sample data.

For an oscilloscope to be useful for engineering work it must be calibrated - for both

time and level, be able to produce an accurate readout of at least two channels and facilitate signal acquisition of particular events (triggering, signal history) [51].

C.4 Standardisation

The key point of measuring things is to be able to meaningfully compare readings from one meter to another or to a mathematically calculated value. A useful analogy here is inches and centimetres, there is a rigorous specification of what distance means. There are various standards and conventions, but there is no margin for error: One can rely on the centimetre.

Unfortunately the same rigour is not always applied to audio metering. On many products the included level meter mainly serves to enhance aesthetics, “make it look cool”, rather than provide a reliable measurement. This trend increased with the proliferation of digital audio plugins. Those meters are not completely without merit, they can be useful to distinguish the presence, or otherwise, of a signal, and most will place the signal-level in the right *ballpark*. There is nothing wrong with saying “the building is tall” but to say “the building is 324.1m high” is more meaningful. The problem in the audio-world is that many vendors add false numeric labels to the scale to convey the look of professionalism, which can be quite misleading.

In the audio sphere the most prominent standards are the IEC and ITU specifications: These specs are designed such that all meters which are compliant, even when using completely different implementations, will produce identical results.

The fundamental attributes that are specified for all meter types are:

- Alignment or Reference Level and Range
- Ballistics (rise/fall times, peak-hold, burst response)
- Frequency Response (filtering)

Standards (such as IEC, ITU, EBU,...) govern many details beyond that, from visual colour indication to operating temperatures, analogue characteristics, electrical safety guidelines, test-methods, down to electrostatic and magnetic interference robustness requirements....

C.5 Software Implementation

Meters.lv2

Meters.lv2 [38] is a set of audio plugins, licensed in terms of the GPLv2 [28], to provide professional audio-signal measurements according to various standards. It currently features needle style meters (mono and stereo variants) of the following

- IEC 60268-10 Type I / DIN
- IEC 60268-10 Type I / Nordic
- IEC 60268-10 Type IIa / BBC
- IEC 60268-10 Type IIb / EBU
- IEC 60268-17 / VU

An overview is given in Fig. 38. Furthermore it includes meter-types with various appropriate visualisations for:

- 30 Band 1/3 octave spectrum analyser according to IEC 61260 (see Fig. 43)
- Digital True-Peak Meter (4x Oversampling), Type II rise-time, 13.3dB/s falloff.
- EBU R128 Meter with Histogram and History (Fig. 39)
- K/RMS meter, K-20, K-14 and K-12 variants
- Stereo Phase Correlation Meter (Needle Display, bottom right in Fig. 38)

- Goniometer (Stereo Phase Scope) (Fig. 40)
- Phase/Frequency Wheel (Fig. 41)

There is no official standard for the Goniometer and Phase-Wheel, the display has been eye-matched by experienced sound engineers to follow similar corresponding hardware equivalents.

Particular care has been taken to make the given software implementation safe for professional use. Specifically realtime safety and robustness (e.g. protection against denormals or subnormal input). The graphical display makes use of hardware acceleration (openGL) to minimise CPU usage.

Sisco.lv2

Sisco.LV2 [39] implements a classic audio oscilloscope with variable time scale, triggering, cursors and numeric readout in LV2 plugin format. While it is feature complete for an audio-scope, it is rather *simplistic* compared to contemporary hardware oscilloscopes or similar endeavours by other authors [51].

The minimum grid resolution is 50 micro-seconds - or a 32 times oversampled signal. The maximum buffer-time is 15 seconds. Currently variants up to four channels are available.

The time-scale setting is the only parameter that directly affects data acquisition. All other parameters act on the display of the data only. The vertical axis displays floating-point audio-sample values with the unit [-1..+1]. The amplitude can be scaled by a factor of [-10..+10] (20dB), negative values will invert the polarity of the signal. The numeric readout is not affected by amplitude scaling. Channels can be offset horizontally and vertically. The offset applies to the display only and does not span multiple buffers (the data does not extend beyond the original display). This allows the display to be adjusted in ‘paused’ mode after sampling a signal.

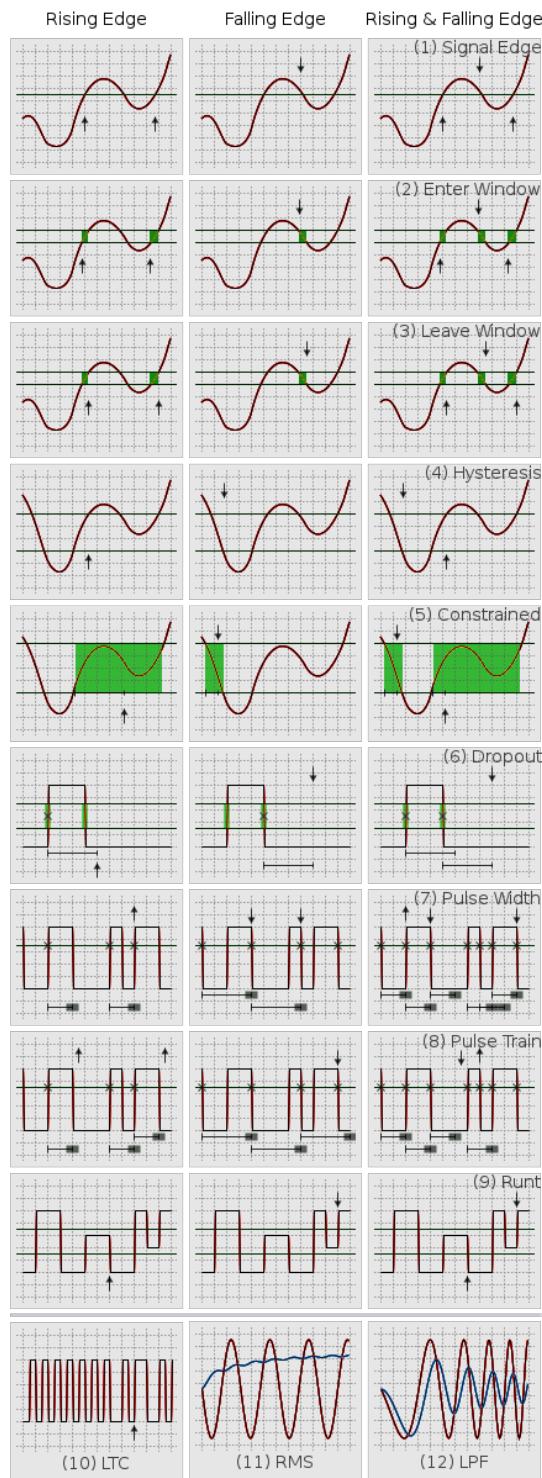


Figure 45 – Overview of trigger preprocessor modes available in “mixtri.lv2”. The arrow indicates trigger position.

#	Title	Description
1	Signal Edge	Signal passes ‘Level 1’
2	Enter Window	Signal enters a given range (Level 1, 2).
3	Leave Window	Signal leaves a given range (Level 1, 2).
4	Hysteresis	Signal crosses both min and max (Level 1,2) in the same direction without interruption.
5	Constrained	Signal remains within a give range for at least ‘Time 1’.
6	Drop-out	Signal does not pass through a given range for at least ‘Time 1’.
7	Pulse Width	Last edge-trigger occurred between min and max (Time 1,2) ago.
8	Pulse Train	No edge-trigger for a give time (max, Time 2), or more than one trigger since a give time (min, Time 1).
9	Runt	Fire if signal crosses 1st but not 2nd threshold.
10	LTC	Trigger on Linear Time Code sync word.
11	RMS	Calculate RMS, Integrate over ‘Time 1’ samples.
12	LPF	Low Pass Filter, 1.0/ ‘Time 1’ Hz

Table 9 – Description of trigger modes in Fig. 45.

The oscilloscope allows for visually hiding channels as well as freezing the current display buffer of each channel individually. Regardless of display, data-acquisition for every channel continues and the channel can be used for triggering.

The scope has three modes of operation:

- **No Triggering** The Scope runs free, with the display update-frequency depending

on audio-buffer-size and selected time-scale. For update-frequencies less than 10Hz a vertical bar of the current acquisition position is displayed. This bar separates recent data (to the left) and previously acquired data (to the right).

- **Single Sweep** Manually trigger acquisition using the push-button, honouring trigger settings. Acquires exactly one complete display buffer.
- **Continuous Triggering** Continuously triggered data acquisition with a fixed hold time between runs.

Advanced trigger modes are not directly included with the scope, but implemented as a standalone “trigger preprocessor” [52] plugin, see Fig. 45 and Table 9. Trigger-modes 1-5 concern analogue operation modes, modes 6-9 are concerned with measuring digital signals²⁸. Modes 10-12 are pre-processor modes rather than trigger modes. Apart from trigger and edge-mode selectors “mixtri.lv2” provides two level and two time control inputs for configuration.

²⁸Digital signal trigger modes are of limited use with a generic audio interface, but can be useful in combination with an adapter (e.g. Midi to Audio) or inside Jack (e.g. AMS or ingen control signals).

References

- [1] C. E. Shannon, “Communication in the presence of noise,” *Proceedings of the IRE*, vol. 37, ISBN 1, pp. 10–21, 1949.
- [2] C. Montgomery, “24/192 music downloads ...and why they make no sense,” <https://people.xiph.org/~xiphmont/demo/neil-young.html>, 2012.
- [3] M. Lester and J. Boley, “The effects of latency on live sound monitoring,” in *Proceedings of the 123rd Audio Engineering Society Convention*, 2007.
- [4] Ableton, “Ableton live 9.2 manual – latency and delay compensation overview,” <https://help.ableton.com/hc/en-us/articles/209072289-Latency-and-Delay-Compensation-overview>, [Online; accessed 10-October-2017].
- [5] I. L. Software, “Fl studio – mixer track properties,” https://www.image-line.com/support/FLHelp/html/mixer_trackprops.htm, [Online; accessed 8-October-2017].
- [6] Digidesign, “Protools, latency and delay compensation with host-based pro tools systems,” http://akmedia.digidesign.com/support/docs/Delay_Comp_PT_Host_Systems_33000.pdf, [Online; accessed 28-October-2015].
- [7] B. Sigoure, “How long does it take to make a context switch?” <http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.htm>, 2011.
- [8] F. Adriaensen, “Kokkini zita linux audio projects,” <http://kokkinizita.linuxaudio.org/linuxaudio/>, 2015.
- [9] V. Goudard and R. Müller, “Real-time audio plugin architectures,” <http://mdsp2.free.fr/ircam/pluginarch.pdf>, 2004, IRCAM.

- [10] M. A. Collins, *Professional Guide to Audio Plug-ins and Virtual Instruments*, 2003, ISBN 9780240517063.
- [11] D. Robillard *et al.*, “Core lv2 specification,” 2008, [Online; accessed 1-August-2017]. [Online]. Available: http://lv2plug.in/doc/html/group__lv2core.html
- [12] R. C. Boulanger, *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming*. MIT press, 2000.
- [13] P. Davis *et al.*, <https://ardour.org/styleguide.html>, 2002–2017.
- [14] H. Fletcher and W. A. Munson, “Loudness, its definition, measurement and calculation,” *Bell Labs Technical Journal*, vol. 12, ISBN 4, pp. 377–430, 1933.
- [15] F. Dunn, W. Hartmann, D. Campbell, and N. H. Fletcher, *Springer handbook of acoustics*. Springer, 2015.
- [16] S. Rosen and P. Howell, *Signals and systems for speech and hearing*. Brill, 2011, vol. 29.
- [17] S. Kim, W. L. Martens, and K. Walker, “Perception of simultaneity and detection of asynchrony between audio and structural vibration in multimodal music reproduction,” in *Audio Engineering Society Convention 120*, May 2006. [Online]. Available: <http://www.aes.org/e-lib/browse.cfm?elib=13629>
- [18] J. Vroomen and M. Keetels, “Perception of intersensory synchrony: a tutorial review,” *Attention, Perception, & Psychophysics*, vol. 72, ISBN 4, pp. 871–884, 2010.
- [19] O. Räsänen, E. Pulkkinen, T. Virtanen, M. Zollner, and H. Hennig, “Fluctuations of hi-hat timing and dynamics in a virtuoso drum track of a popular music recording,” 2015.

- [20] I. Recommendation, “1359, relative timing of sound and vision for broadcasting,” *International Telecommunication Union, Geneva, Switzerland*, 1998.
- [21] T. Fine, “The dawn of commercial digital recording,” *ARSC Journal*, vol. 39, ISBN 1, pp. 1–17, 2008.
- [22] J. Watkinson, “Digital audio recorders,” in *Audio Engineering Society Conference: 7th International Conference: Audio in Digital Times*. Audio Engineering Society, 1989.
- [23] ——, *An introduction to digital audio*. Taylor & Francis, 2002.
- [24] Wikipedia, “Digital recording — wikipedia, the free encyclopedia,” 2017, [Online; accessed 4-December-2017]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Digital_recording&oldid=811859781
- [25] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [26] R. Gareus, “Ardour3 - video integration,” in *Proceedings of the Linux Audio Conference 2012*, ISBN 978-1-105-62546-6, 2012.
- [27] L. Garrido and R. Gareus, “Made with ardour and xjadeo,” <http://xjadeo.sourceforge.net/main.html#made-with>, [Online; accessed 28-October-2017].
- [28] “Gnu general public license, version 2,” <https://www.gnu.org/licenses/old-licenses/gpl-2.0.html>, Free Software Foundation, June 1991.
- [29] “Ardour contributors,” <https://github.com/Ardour/ardour/graphs/contributors>.
- [30] K. Huang, C. Huang, Y. Li, and M.-S. Young, “High precision, fast ultrasonic thermometer based on measurement of the speed of sound in air,” *Review of scientific Instruments*, vol. 73, ISBN 11, pp. 4022–4027, 2002.

- [31] O. Cramer, “The variation of the specific heat ratio and the speed of sound in air with temperature, pressure, humidity, and co₂ concentration,” *The Journal of the Acoustical Society of America*, vol. 93, ISBN 5, pp. 2510–2516, 1993.
- [32] V. Lazzarini, S. Yi, J. Heintz, Ø. Brandtsegg, I. McCurdy *et al.*, *Csound: A Sound and Music Computing System*. Springer, 2016.
- [33] R. Sedgewick and K. Wayne, *Algorithms (4th ed.)*. Addison-Wesley, 2011, ISBN 9780132762564.
- [34] A. B. Kahn, “Topological sorting of large networks,” *Commun. ACM*, vol. 5, 1962. [Online]. Available: <http://doi.acm.org/10.1145/368996.369025>
- [35] R. Benica, “Real-time audio programming 101: time waits for nothing,” 2011. [Online]. Available: <http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothing>
- [36] R. Gareus, “nodelay.lv2 - audio delayline and instrumentation tool,” <http://x42-plugins.com/x42/x42-nodelay>, 2013.
- [37] P. Davis, <http://ardour.org/news/3.3.html>, <http://ardour.org/news/3.4.html>, 2013.
- [38] R. Gareus, “Meters.lv2 - audio plugin collection,” <https://github.com/x42/meters.lv2>, 2013.
- [39] ———, “Sisco.lv2 - Simple Scope,” <https://github.com/x42/sisco.lv2>, 2013.
- [40] E. Brixen, *Audio Metering: Measurements, Standards and Practice*, 2010, ISBN 0240814673.
- [41] J. Watkinson, *Art of Digital Audio*, 2000, ISBN 0240515870.

- [42] B. Katz, "How to make better recordings in the 21st century - an integrated approach to metering, monitoring, and leveling practices." <http://www.digido.com/how-to-make-better-recordings-part-2.html>, 2000, updated from the article published in the September 2000 issue of the AES Journal by Bob Katz.
- [43] Wikipedia, "Peak programme meter — wikipedia, the free encyclopedia," 2013, [Online; accessed 1-February-2014]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Peak_programme_meter&oldid=547296518
- [44] IEC, "IEC60268-10 sound system equipment – Part 10 Peak programme level meters," 1991, International Electrotechnical Commission.
- [45] ITU, "ITU BS.1770, Algorithms to measure audio programme loudness and true-peak audio level," <http://www.itu.int/rec/R-REC-BS.1770/en>, 2006, International Telecommunication Union.
- [46] EBU, "EBU R-128 – audio loudness normalisation & permitted maximum level," <https://tech.ebu.ch/docs/r/r128.pdf>, 2010, European Broadcast Union – Technical Committee.
- [47] F. Adriaensen, "Loudness measurement according to EBU R-128," in *Proceedings of the Linux Audio Conference 2011*, 2011.
- [48] EBU, "EBU TECH 3342 – Loudness Range: A measure to supplement loudness normalisation in accordance with EBU R-128," <https://tech.ebu.ch/docs/tech/tech3342.pdf>, 2011, European Broadcast Union – Technical Committee.
- [49] Wikipedia, "VU meter — wikipedia, the free encyclopedia," 2014, [Online; accessed 2-February-2014]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=VU_meter&oldid=588986200

- [50] IEC, “IEC61260 electroacoustics – Octave-band and fractional-octave-band filters,” 1995, International Electrotechnical Commission.
- [51] F. Adriaensen, “Design of an audio oscilloscope application,” in *Proceedings of the Linux Audio Conference 2013*, ISBN 9783-902949-00-4, 2013.
- [52] R. Gareus, “Mixtri(x).lv2 - matrix mixer and trigger processor,” <https://github.com/x42/mixtri.lv2>, 2013.

List of Tables

<i>fr-1</i>	Règle empirique pour la latence audio.	4
1	Rule of thumb for audio latency.	56
2	Port Flags	94
3	Step by step port capture latency computation for routing in Fig. 23.	117
4	Step by step port playback latency computation for routing in Fig. 23.	118
5	Complete list of playback and capture port latencies corresponding to routing in Fig. 23.	118
6	Audio Engine Process Cycle, round-trip latency is 2 buffers	128
7	Paralell processing of process-graph displayed in Fig. 31	135
8	True Peak calculations @ 44.1 KSPS, both examples correspond to 0dBFS.	169
9	Description of trigger modes in Fig. 45.	177

List of Figures

<i>fr-1</i> Un signal audio, de l'onde de pression sonore à la représentation numérique discrète en passant par le signal électrique continu	2
<i>fr-2</i> Deux sinusoïdes décalées de 180 degrés dont le cumul est un silence complet.	4
<i>fr-3</i> Spectre d'un bruit rose échantillonné à 48kHz combiné à une copie de lui-même retardée de 116 échantillons. Sans le retard on verrait un spectre constant vers environ –6dBFS. La forme du spectre avec des creux périodiques explique le nom de filtre en peigne.	5
<i>fr-4</i> Chaine de latence. Les valeurs données en exemple correspondent à un PC typique. Avec du matériel professionnel et un système optimisé, la latence totale d'aller-retour est souvent plus basse. Le point important est que la latence, toujours additive, est la somme de nombreux facteurs indépendants.	8
<i>fr-5</i> Une baie de brassage audio analogique typique avec son diagramme de routage (source ekadek.com)	12
<i>fr-6</i> Routage numérique — ici les connections des ports JACK sont visibles dans le logiciel ‘patchage’	13
<i>fr-7</i> En haut : état précédent d'Ardour 3, avant le travail sur la compensation de latence. En bas : prototype démontrant la faisabilité d'une compensation de latence correcte. L'encart montre le diagramme de routage (vert : lecture ; rouge : pistes d'enregistrement ; jaune : bus)	23
<i>fr-8</i> Latence d'aller-retour mesurée sur une Presonus Audiobox VSL1818 à une fréquence d'échantillonnage de 48kHz.	31
<i>fr-9</i> Table de mixage Allen & Heath ZED-24.	39
<i>fr-10</i> Abstraction conceptuelle de base de l'objet <i>Route</i>	41

<i>fr-11</i> Ancienne conception d'un objet piste <i>Track</i> (Ardour 3-5). Les <i>Processors</i> en vert ont été ajoutés par l'utilisateur. Le bleu indique les étapes d'ajustement du gain qui requièrent des <i>Processors</i> en interne. On note que les entrées/sorties disques ainsi que l'écoute de contrôle sont traitées à part avec un aiguillage à l'entrée de la piste. Cette architecture-ci ne se prête pas à la compensation de latence.	43
<i>fr-12</i> Abstraction révisée d'un objet <i>Track</i> , où tout est géré par des <i>Processors</i> . Il n'y a pas d'ordre strict, les processeurs peuvent être librement réordonnés avec quelques contraintes. Le vert indique les processeurs optionnels ajoutés par l'utilisateur (les plugins). Une <i>Route</i> (un bus) n'a pas les <i>DiskProcessors</i> en rouge.	44
<i>fr-13</i> Des processeurs automatisés dans une chaîne avec un plugin latent. Les effets des deux amplificateurs s'annulent l'un l'autre, cependant la latence doit être prise en considération pour appliquer l'automatisation.	47
1 Audio Signal, from sound-pressure wave, continuous analogue electrical signal and discrete digital signal representation.	54
2 Two sine 180 degree shifted sine-waves, when added result in silence.	55
3 Pink Noise spectrum, sampled at 48kHz combined with a 116 sample delayed signal of itself. Without the delay it would be a flat line at about -6dBFS. The visual shape of the signal with periodic dips motivated the name comb-filter.	56
4 Latency chain. The numbers are an example for a typical PC. With professional gear and an optimized system the total round-trip latency is usually lower. The important point is that latency is always additive and a sum of many independent factors.	59
5 A typical analogue audio patchbay and routing diagram (source ekadek.com)	61
6 Digital routing - here jack port connections visualized using 'patchage' . . .	62

7	Top: Previous state of Ardour 3 - before work on latency-compensation. Bottom: Proof of concept, prototype with proper latency compensation. The inset show the routing diagram. green: playback; red: recording tracks; yellow: busses	71
8	Measured round-trip latency of a Presonus Audiobox VSL1818 at 48kHz sample-rate.	78
9	Allen & Heath ZED-24 mixing desk.	85
10	Basic conceptual abstraction of the Route object	87
11	Old (Ardour 3-5) conceptual abstraction of a track object. The green processors are user-added. Blue indicates gain-staging and internally required processors. Note that disk I/O and monitoring was special-cased with a switch at the track's input. This particular architecture does not lend itself for latency compensation.	88
12	Re-designed abstraction of a track object, where everything is a processor. There is no strict order, processors can be freely re-ordered with a few constraints. Green indicates optional user-added processors/plugins. A route (bus) does not have the red disk-I/O Processors.	89
13	Automated Processors in a chain with a latent plugin. The effect of the two Amplifiers cancel each other out, however the latency needs to be taken into account when applying automation.	92
14	A USB Type A and Type B port. The ports physically impose a structured connections.	94
15	Port directions: Data is read from an output, processed and written to an input for others to process. Connections are directional from input to output.	95
16	Track with an Aux-Send feeding a Bus.	99
17	IOProcessor class inheritance diagram. (generated by Doxygen, Ardour 6.0-pre0-53-g94cce9e06)	100

18	Synchronization: Two separate simultaneous events take different paths with different delays. They need to be re-aligned in order to remain synchronized at the output.	103
19	Routing 3 tracks, via 2 effect busses, to master out without additional time alignment constraints.	105
20	Figure 19 with additional constraint to align all track outputs results in increased total latency.	106
21	Hasse diagram of the directed acyclic graph corresponding to the routing in Fig. 19, 20.	110
22	An internal auxiliary send allowing for parallel signal flow.	111
23	Port capture and playback latency computation. Capture latency (red) increases downstream (left to right), Playback Latency (green) is propagated upstream (right to left).	116
24	A routing situation involving an aux-sends and output alignment.	120
25	Complete model of the Send with 2 internal delay-lines, one for the send-path, one for the thru-path	121
26	A complex routing situation involving an aux-sends and output alignment resulting in ambiguous latencies.	125
27	Resolution to the ambiguous latency situation displayed in Fig. 26 by only relying on sends.	126
28	Another resolution to the ambiguous latency situation displayed in Fig. 26 by adding a bus.	126
29	The shown directed acyclic graph can be topologically sorted in different ways	127
30	Simple process callback	129
31	Process-graph chain corresponding to the routing in Fig. 19, 20. The numbers in red are dependency counts: the reference count of a given node.	134
32	Algorithm to set per processor latencies, see also Listing 8.	140

33	Simplified router processor execution. Initially the sample is offset by the route's own latency, which is the sum of all processor latencies. As the processors are called, the latency is reduced until after the last processor, start_sample aligns to the output.	141
34	Closed loop Audio + MIDI testing	155
35	A test-session, verifying alignment of bounce-processing with a Lua-script showing processor latency information (here: Ardour 6.0-pre0-148).	156
36	Audio/MIDI alignment testing	159
37	Various meter alignment levels as specified by the IEC. Common reference level is 0dBu calibrated to -18dBFS for all types except for DIN, which aligns +9dBu to -9dBFS. dBu refers to voltage in an analogue system while dBFS to digital signal full-scale.	162
38	Various meter-types from the meter.lv2 plugin bundle fed with a -18 dBFS 1 kHz sine wave. Note, bottom right depicts the stereo phase correlation meter of a mono signal.	164
39	EBU R-128 meter GUI with histogram (left) and history (right) view. . .	165
40	Goniometer (Phase Scope)	167
41	Phase/Frequency Wheel. Left: pink noise, 48KSPS with right-channel delayed by 5 samples relative to left channel. Right: Digitalisation of a mono 1/2" tape reel with slight head misalignment.	168
42	Inter-sample peaks in a sine-wave. The red line (top and bottom) indicates the digital peak, the actual analogue sine-wave (black) corresponding to the sampled data (blue dot) exceeds this level.	169
43	30 Band 1/3 octave spectrum analyser	170
44	15KHz, -3dBFS sine wave sampled at 48KSPS. The Oscilloscope (top) up-samples the data to reproduce the signal. The wave-form display (bottom) displays raw sample data.	172

- 45 Overview of trigger preprocessor modes available in “mixtri.lv2”. The arrow indicates trigger position. 176