A GRID BASED APPROACH FOR THE CONTROL AND RECALL OF THE PROPERTIES OF IEEE1394 AUDIO DEVICES

A thesis submitted in fulfilment of the
requirements for the degree of

MASTER OF SCIENCE

of

RHODES UNIVERSITY

by

PHILIP JAMES FOULKES

February 2008

Abstract

The control of modern audio studios is complex. Audio mixing desks have grown to the point where they contain thousands of parameters. The control surfaces of these devices do not reflect the routing and signal processing capabilities that the devices are capable of. Software audio mixing desk editors have been developed that allow for the remote control of these devices, but their graphical user interfaces retain the complexities of the audio mixing desk that they represent.

In this thesis, we propose a grid approach to audio mixing. The developed grid audio mixing desk editor represents an audio mixing desk as a series of graphical routing matrices. These routing matrices expose the various signal processing points and signal flows that exist within an audio mixing desk. The routing matrices allow for audio signals to be routed within the device, and allow for the device's parameters to be adjusted by selecting the appropriate signal processing points. With the use of the programming interfaces that are defined as part of the Studio Connections – Total Recall SDK, the audio mixing desk editor was integrated with compatible DAW applications to provide persistence of audio mixing desk parameter states.

Many audio studios currently use digital networks to connect audio devices together. Audio and control signals are patched between devices through the use of software patchbays that run on computers. We propose a double grid-based FireWire patchbay aimed to simplify the patching of signals between audio devices on a FireWire network. The FireWire patchbay was implemented in such a way such that it can host software device editors that are Studio Connections compatible. This has allowed software device editors to be associated with the devices that are represented on the FireWire patchbay, thus allowing for studio wide control from a single application. The double grid-based patchbay was implemented such that it can be hosted by compatible DAW applications. Through this, the double grid-based patchbay application is able to provide the DAW application with the state of the parameters of the devices in a studio, as well as the connections between them. The DAW application may save this state data to its native song files. This state data may be passed back to the double grid-based patchbay when the song file is reloaded at a later stage. This state data may then be used by the patchbay to restore the parameters of the patchbay and its device editors to a previous state. This restored state may then be transferred to the hardware devices being represented by the patchbay.

Acknowledgements

Many thanks go to my project supervisor, Professor Richard Foss, for continually guiding me throughout the project! Many thanks also go to my colleagues for their assistance and support in reaching the goals of the project.

To my family and friends, thank you for your support, encouragement, and for always believing in me.

Thank you to the departmental sponsors: Telkom SA, Business Connexion, Comverse SA, Verso Technologies, Stortech, Tellabs, Amatole, Mars Technologies, Bright Ideas Projects 39 and THRIP through the Telkom Centre of Excellence at Rhodes University for the financial and technical support they provided to undertake this research.

Table Of Contents

List Of Figures

List Of Tables

List Of Listings

Chapter 01

Introduction

1.1. Introduction

Routing audio within modern audio studios and setting up parameters in audio devices has become a complex task. Many current audio studios have done away with multiple analogue audio and MIDI cables and the use of hardware routing to connect audio devices in favour of digitally transmitted audio and software routing using high speed networking technologies such as FireWire coupled with mLAN [Foss, 2006].

Modern digital audio mixing desks are difficult to master. The routing and signal processing capabilities of these devices are not clear from the arrays of buttons and sliders presented on the control surfaces of the devices. Often features of these devices can only be accessed by navigating through a complex hierarchy of menus on a small LCD screen on the device.

Typically audio is routed from many audio producing devices (such as microphones and synthesisers) to a central audio mixing desk. Within the audio mixing desk the signals may be routed through many signal processing components (such as equalisers, dynamics and internal effects processors). The tone of the audio signals may be adjusted at various signal processing points as they flow through the audio mixing desk. The tone is adjusted by altering various parameters that belong to the signal processing components. After that, the signals may be routed onto various audio busses and through further signal processing components, and eventually out of the audio mixing desk. The audio signals are then typically routed to external devices for additional signal processing (for example, the signals may be routed to an external effects processor).

Software device editors have been created that allow for the representation and remote control of audio mixing desks. These software device editors usually represent a device in a graphical format that reflects the actual device and hence the complexities of such a layout. Due to the magnitude of these devices, their associated device editors have cluttered layouts with controls being layered. A grid approach to audio mixing has been developed. It aims to simplify the remote control of audio mixing desks by representing them as routing matrices. These routing matrices represent the signal flows and signal processing points that exist within an audio mixing desk, allowing these devices to be controlled in a more logical manner. The grid audio mixing desk editor uses XML as a configuration tool. Audio mixing desks are described with XML documents and these are used to configure the grid audio mixing desk editor to represent a specific audio mixing desk. This allows the editor to quickly adapt to different audio mixing desks.

In modern audio studios that use digital networks to connect devices together, audio signals are patched between audio devices using software patchbays. These devices have done away with hardware plugs to receive and transmit audio and control signals. They now use software plug abstractions. Software patchbays are used to expose the software plugs that exist on the audio devices, and to route signals between the devices. There are various kinds of software patchbays that exist. Each of these represents digital audio networks in a different manner. One such patchbay is the grid-based patchbay where the inputs and outputs of devices form a grid. A variation of the grid-based patchbay, a double grid-based patchbay, was developed with the aim of reducing the complexity of exploring devices, and patching audio and control signals between devices.

There is a need for closer integration of software and hardware in music production systems [Yamaha, 2005]. Normally, the setting up of audio software is done independently of the hardware. The Studio Connections initiative [Yamaha, 2005], of which phase one has been identified as Total Recall, has been introduced to offer a more convenient environment to make using audio software and hardware easier. The Total Recall phase allows Digital Audio Workstations (DAW's) to host software plug-ins. These plug-ins allow for the persistence of hardware device state to and from DAW native song files. Through the use of the Studio Connections – Total Recall SDK, the grid audio mixing desk editor was implemented in such a way to permit state persistence of audio mixing desks. This allows an audio mixing desk to be restored to a previous state. The double grid based-patchbay was implemented such that it can be hosted by compatible DAW applications, and such that it is able to host software device editors. These device editors may then be associated with devices that are on the audio network. The integration of these software entities has allowed for studio wide device control and recall to take place from a single software application: Devices may be edited with their associated software device editors, connections may be made between devices, and the state of the devices and their connections may be saved and restored through a DAW application.

Chapter 2 introduces audio mixing. It demonstrates the essence of audio mixing and it explores the use of the control surfaces of analogue and digital audio mixing desks. This chapter also explores software audio mixing desk editors that are used to remotely control audio mixing desks. The strengths and weaknesses of the different approaches to audio mixing are discussed.

Chapter 3 examines the fundamentals of grid patchbays and proposes an audio mixing desk editor layout based on the grid patchbay with the aim of simplifying the use of modern digital audio mixing desks.

Chapter 4 describes how the grid audio mixing desk editor (described in chapter 3) has been developed to be a generic audio mixing desk editor that is configurable via XML. XML elements and attributes have been defined that allow for audio mixing desks to be described. This chapter looks at how these XML element and attributes have been used to configure the editor.

Chapter 5 explores the Studio Connections project and how this project enabled the grid audio mixing desk editor to be integrated into DAW applications to allow for audio mixing desk parameter states to be saved and recalled.

Chapter 6 looks at various patching techniques used in audio studios. A hardware patchbay and various types of software patchbays are analysed. The strengths and weaknesses of the different software patchbays are compared. A FireWire patchbay is proposed with the aim of simplifying the process of patching signals between devices in audio studios.

Chapter 7 brings together the preceding parts of the project and proposes an application that allows for the total control and recall of the properties and interconnections of FireWire audio devices from within DAW applications.

Chapter 02

Hardware And Software Audio Mixing

The audio mixing desk is central to any audio studio. Its core function is to receive various audio signals, route them through internal signal processing components, merge the various audio signals into single mixed signals, and then assign the mixed signals to its outputs. Modern digital audio mixing desks are becoming complicated to use, as they use layering and assignable controls in order to perform audio mixing desk operations. Their control surfaces do not reflect the routing capabilities and signal processing control points that exist within the audio mixing desk.

Software audio mixing editors that expose the layered parameters of audio mixing desks have been created. These software editors allow for the emulation and remote control of audio mixing desks. While the editors do expose more of an audio mixing desk's functionality, they still remain complex as the control surfaces of these pieces of software usually reflect the control surfaces of audio mixing desks. The control surfaces of the software editors present a sound engineer with a dense, cluttered layout of controls.

In this chapter, we shall look at the core function of audio mixing, view the control surfaces of different audio mixing desks, have a look at the shortcomings of modern digital audio mixing desks and conclude with a discussion on software audio mixing desk editors. The Yamaha 01X Digital Mixing Studio [Yamaha, 2003a], Yamaha 03D Digital Mixing Console [Yamaha, 1997] and Yamaha 01V96 Digital Mixing Console [Yamaha, 2004a] were all used as part of this study to gain insight into the workings of digital audio mixing desks.

2.1. Audio Mixing And Audio Mixing Desks

The core function of an audio mixing desk is to:

- Receive various audio signals (these may be in analogue or digital form, and may come from devices such as microphones and CD players).
- Route the signals through internal signal processing components such as equalizers, dynamics processors and effects processors so as to shape and manipulate the sound.

- Merge the various audio signals into single mixed signals.
- Assign the mixed signals to the various outputs on the same audio mixing desk (the output signals present at these outputs may be in analogue or digital form). The outputs on the audio mixing desk may be connected to external devices such as effects units or power amplifiers for further signal manipulation.

Over the years, audio mixing desks have evolved. Early audio mixing desks were devices that were mere summing networks for a group of microphones that had a single output. Later, equalizers were added to each of the input channels, and then later still signal monitoring capabilities. As time passed, there was a demand for more outputs with these then added to the devices. The functionality of these devices has evolved to a point where audio mixing desks may be automated and controlled digitally. Although analogue audio mixing desks may only have a limited amount of automated parameters such as the volume faders and sub-grouping, digital audio mixing desks may have nearly all of their parameters automated. This allows for the parameter values to be stored, recalled and updated [Eargle, 2003].

In essence, within an audio mixing desk, audio signals are processed and routed at various key points. This can be seen in the block diagram for the Yamaha 01V96 Digital Mixing Console, shown in Figure 1:

1. The left hand side of the diagram shows the various audio signal sources. These signal sources enter the audio mixing desk via its analogue and digital inputs and can come from devices such as microphones and CD players.
2. The source signals are fed into the Input Patch block.
3. Within the Input Patch block, the signals may be patched through to the various input channels that the Yamaha 01V96 Digital Mixing Console has. On some audio mixing desks, the input signals are routed directly to the audio mixing desk's input channels, and it is not possible to select which input channels the incoming audio signals are routed to.
4. At each input channel, the signals may then be routed through various signal processing and sound shaping components such as the volume faders, equalisers and dynamics processors. This allows a sound engineer to manipulate each input signal to get a desired sound.
5. The manipulated signals from each input channel may then be patched through to various audio busses.
6. On each audio bus, the individual incoming signals are merged into a single mixed signal. The audio busses are shown on the block diagram by the vertical lines in the centre.

7. The mixed signals leave the audio busses and may then be routed through further signal processing and sound shaping components.
8. The manipulated signals are sent to the Output Patch block.
9. In the Output Patch block, the signals are patched through to the various physical analogue and digital outputs on the audio mixing desk. These analogue and digital outputs may be connected to devices such as external effects processors to allow for further signal manipulation, or they may be connected to power amplifiers that amplify the signals so that they may be played through loudspeakers.

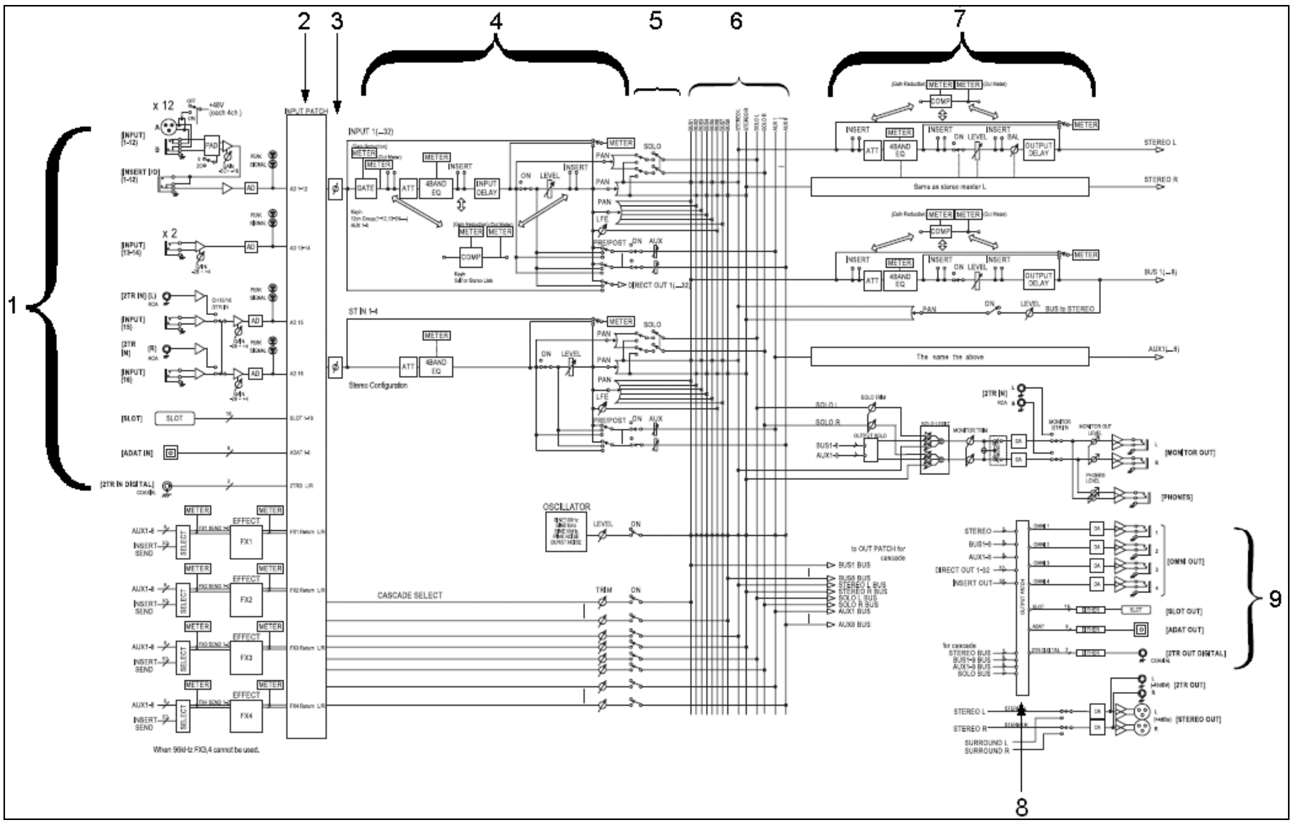


Figure 1: The block diagram for the Yamaha 01V96 Digital Mixing Console

As shown in this block diagram, audio mixing essentially involves a series of stages. Each stage usually involves:

- The processing and shaping of a source signal – for example, the individual signals that come from the audio mixing desk’s input channels are fed through an equaliser to have their tonal characteristics adjusted.
- The patching of a source audio signal through to a destination point – for example, the input channels’ signals could be patched through to the audio mixing desk’s internal audio busses to

be mixed together, or those mixed signals could be patched through to one of the audio mixing desk's outputs.

- The processing and shaping of the destination signal – for example, once the audio signals have been mixed together, the overall mixed signals may be processed and manipulated further by an equaliser.

The control surfaces of modern digital audio mixing desks may be much smaller than those of in-line consoles, but they still retain most of the operational capability and flexibility of larger consoles. These digital audio mixing desks layer channels. Analogue audio mixing desks have a lot of controls on their surface – all functions are shown all of the time. In contrast, digital audio mixing desks do not show all of their functions simultaneously. An engineer needs to acquire a high level of confidence before attempts are made at even the simplest of mixing sessions [Eargle, 2003].

Figure 2 shows the Mackie 32-8 Recording Console [Mackie, 2007]. It is a thirty two channel, eight bus analogue audio mixing desk. Each control on the surface of the device has a dedicated function. Each channel strip has controls that allow for adjustments of volume, equalization, aux send levels, panning, muting and bus sends, amongst others. All of the functionality of the audio mixing desk is shown all of the time.

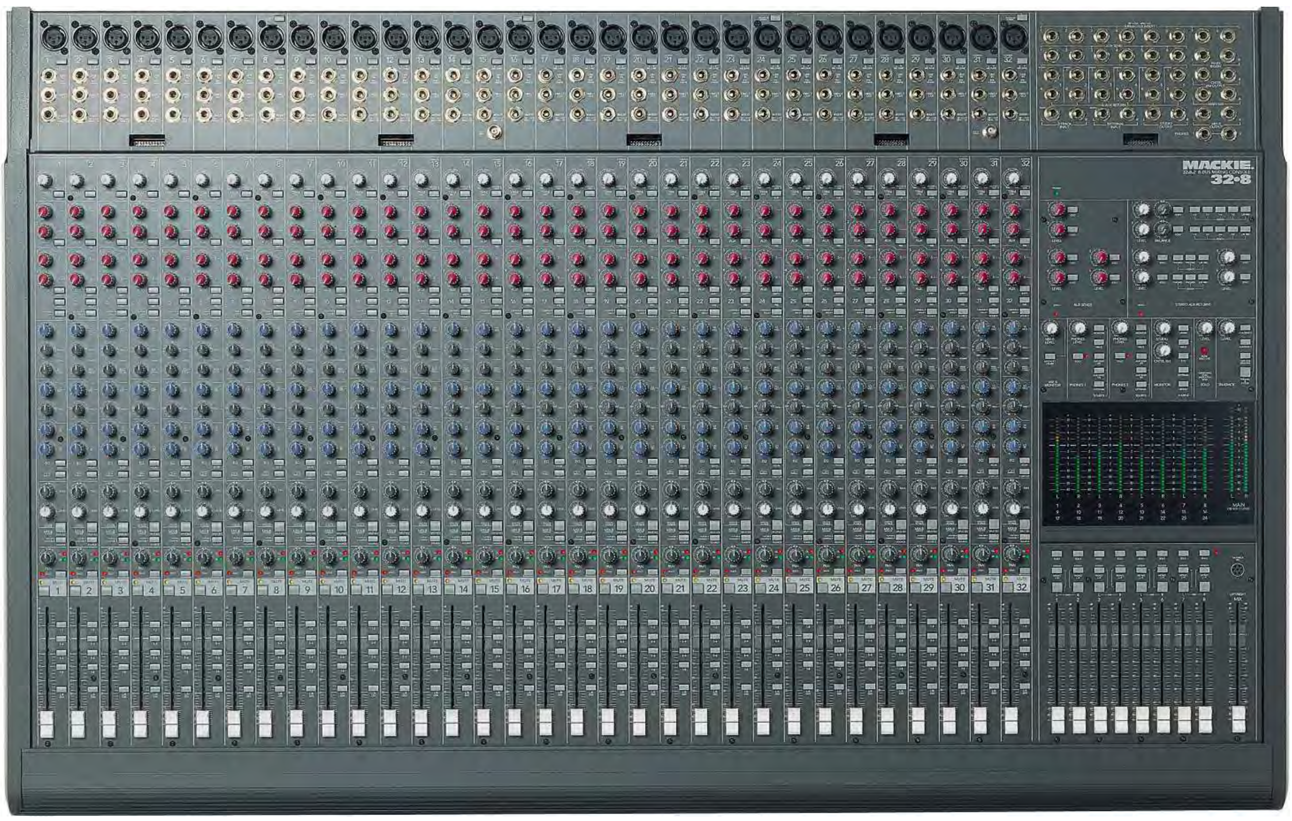


Figure 2: The Mackie 32-8 Recording Console

Figure 3 shows the Yamaha 01V96 Digital Mixing Console [Yamaha, 2004a]. This audio mixing desk is a forty channel, eighteen bus digital audio mixing desk. It is clear from this figure and the previous one that the control surfaces of these two audio mixing desks differ substantially. Even though the Yamaha 01V96 Digital Mixing Console has more functionality than the Mackie 32-8 Console, its control surface is much smaller. Most of the controls on its control surface do not have a dedicated function. Their function depends on the mode that the audio mixing desk is currently in. For example, the faders on the surface of the device may adjust the selected channel's input levels, the levels of the bus outs, or the aux send levels. The functionality of these faders depends on the mixer layer and fader mode that is currently selected.

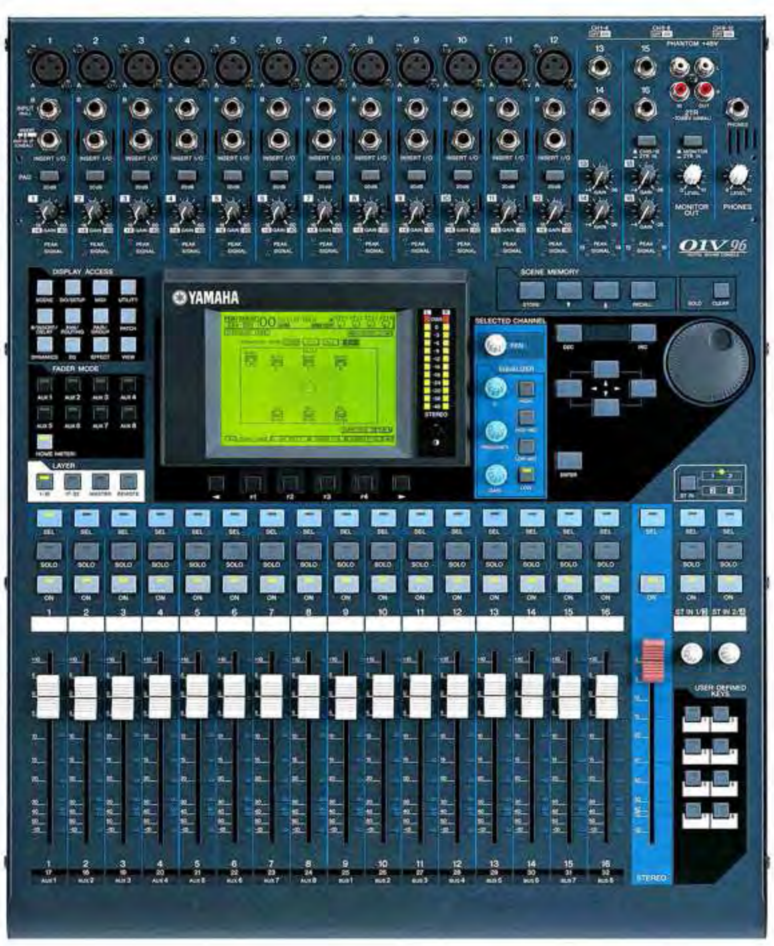


Figure 3: The Yamaha 01V96 Digital Mixing Console

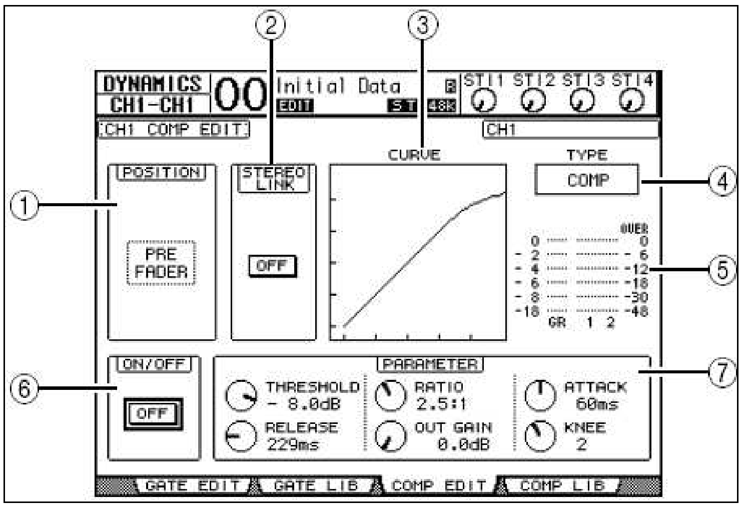
2.1.1. Shortcomings Of Modern Digital Audio Mixing Desks

Due to their feature richness and their compact control surfaces, modern digital audio mixing desks are becoming complicated to use. Their control surfaces (see Figure 3 above) do not reflect the routing capabilities and signals processing control points that may be found within the audio mixing desk. Discovering the routing capabilities of a particular audio mixing desk is often done by a sound engineer studying the block diagram (see Figure 1 above) of that audio mixing desk.

Once the capabilities of a digital audio mixing desk are known, navigating to the controls that allow for parameter adjustments and routing configurations to be changed can be a tedious job. There may be thousands of available parameters and routing configurations on a typical digital audio mixing desk, and these are all adjustable via a limited number of physical controls on the control surface of the device. Modern digital audio mixing desks typically present a sound engineer with a small LCD display that is used to display and adjust its parameters. The display shows the information required by a sound engineer.

For example, on the Yamaha 01V96 Digital Mixing Console, a sound engineer might require information about a dynamics processor for the currently selected channel. This will be the only information that is displayed since the LCD display can only display a limited amount of information. Figure 4 shows the LCD display of the Yamaha 01V96 Digital Mixing Console. Here it is currently displaying the compressor’s edit page for the currently selected channel. The annotations in the figure are explained below:

1. A parameter to allow for the positions of the compressor to be set.
2. A parameter to turn the stereo link on and off.
3. A graph showing the current compressor curve.
4. A field to display the type of compressor that is currently selected.
5. Meters that are used to indicate the levels of the post-compressor signals and the amount of gain reduction.
6. A parameter that allows the dynamics processor to be turned on and off.
7. Parameters that allow for the compressor’s characteristics to be changed.


DYNAMICS CH1-CH1 00 Initial Data

CH1 COMP EDIT

CH1

POSITION: PRE FADER

STEREO LINK: OFF

TYPE: COMP

ON/OFF: OFF

THRESHOLD - 8.0dB

RATIO 2.5:1

ATTACK 60ms

RELEASE 229ms

OUT GAIN 0.0dB

KNEE 2

Figure 4: The Yamaha 01V96 Digital Mixing Console 'Dynamics | Comp Edit' page

In Figure 4, only eleven different elements relating to the dynamics processor for a particular channel are being shown. Adjusting a large number of different parameters that may belong to different channels can require that a sound engineer continually navigate through a hierarchy of pages and parameter controls on the LCD display of the device to locate the required parameter controls.

On the Yamaha 01V96 Digital Mixing Console, input, input insert, effect, cascade, output, output insert, direct out and 2TR output patching all happen via pages that are similar in nature to the In Patch page shown in Figure 5. Sound engineers are capable of configuring patching by navigating the cursor to the desired destination point and selecting the source for that destination point. The signal sources are show by annotation 1 in Figure 5 and the signal destination points are shown by annotation 2. Sound engineers are able to select which signal sources are patched through to specific signal destination points. For example, the patch point shown at the top left of the diagram indicates that the signal from the first analogue input is being routed to the first input channel.

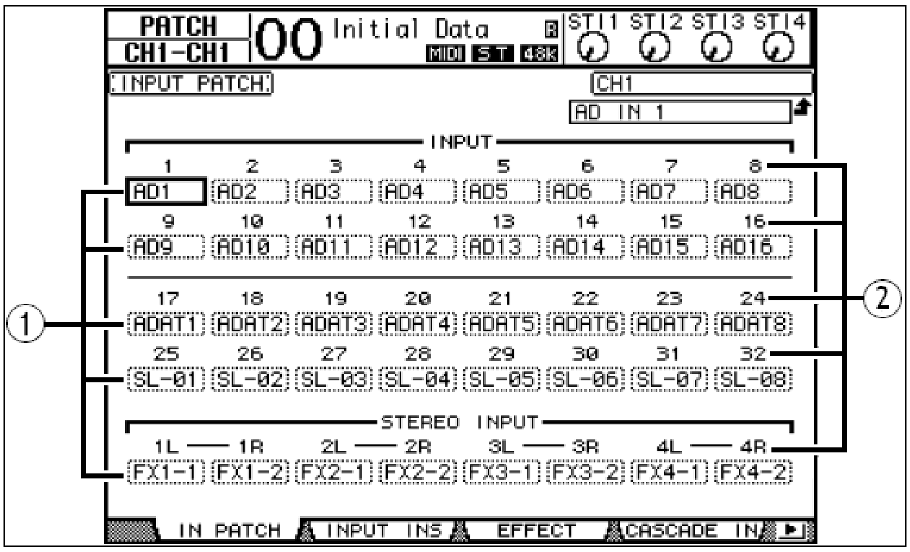

```
PATCH    | 00 Initial Data           ST I1 ST I2 ST I3 ST I4
CH1-CH1  |          MIDI ST 48k
INPUT PATCH                                  CH1
                                             AD IN 1
------------------------- INPUT -------------------------
  1      2      3      4      5      6      7      8
 AD1    AD2    AD3    AD4    AD5    AD6    AD7    AD8
  9     10     11     12     13     14     15     16
 AD9    AD10   AD11   AD12   AD13   AD14   AD15   AD16
 17     18     19     20     21     22     23     24
 ADAT1  ADAT2  ADAT3  ADAT4  ADAT5  ADAT6  ADAT7  ADAT8
 25     26     27     28     29     30     31     32
 SL-01  SL-02  SL-03  SL-04  SL-05  SL-06  SL-07  SL-08
--------------------- STEREO INPUT ----------------------
 1L --- 1R     2L --- 2R     3L --- 3R     4L --- 4R
 FX1-1  FX1-2  FX2-1  FX2-2  FX3-1  FX3-2  FX4-1  FX4-2
 IN PATCH | INPUT INS | EFFECT | CASCADE IN
```

Figure 5: The Yamaha 01V96 Digital Mixing Console 'In Patch' page

On the Yamaha 01V96 Digital Mixing Console, navigating to required parameter controls and adjusting their values on the LCD display of the device may be performed via the Data Entry Section, which is shown in Figure 6. The annotations in the diagram are shown below:

1. The Parameter wheel: The Parameter wheel is a rotary control and is used to adjust the parameter that is currently selected by the cursor. If, for example, the Threshold parameter of the dynamics processor shown in Figure 4 is currently selected, turning the wheel clockwise will increase it value. Turning the wheel anti-clockwise will decrease its value.
2. Enter button: The Enter button is used to turn buttons on and off and to confirm the values of edited parameters. For example, the signal sources shown in Figure 5 are adjusted by turning the Parameter wheel. Once the required signal source is shown as being patched through to the signal destination point, it is confirmed by pressing the Enter button.

3. Dec and Inc buttons: The Dec and Inc buttons are used to decrease or increase the value of the parameter selected by the cursor.
4. Left, Right, Up and Down cursor buttons: The cursor buttons are used to navigate the cursor around the LCD display to select parameters for adjustment. They are used to move the cursor to parameters that are adjacent to the parameter that is currently selected by the cursor.

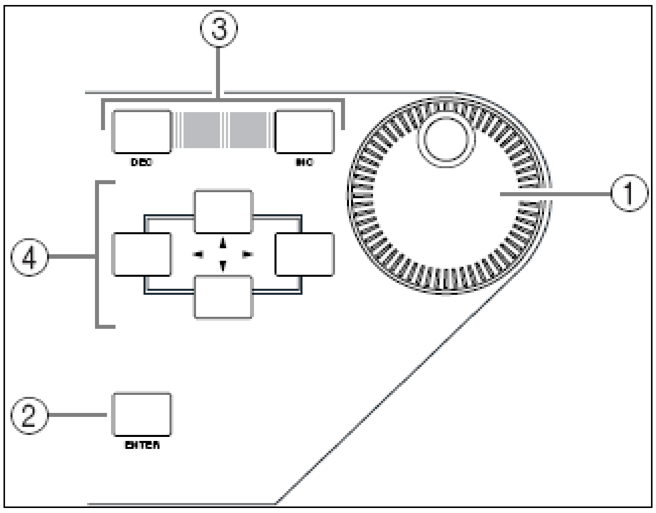


Figure 6: The Yamaha 01V96 Digital Mixing Console Data Entry Section

The Yamaha 03D Digital Mixing Console displays its parameters on its LCD display, allows for parameter control navigation, as well as for parameter adjustments to take place in a similar way to the Yamaha 01V96 Digital Mixing Console. The LCD display for the Yamaha 01X Digital Mixing Studio is shown in Figure 7. This figure shows the page that allows for the dynamics processor of the currently selected channel to be edited. This digital audio mixing desk has a much smaller LCD display than the two previously mentioned digital audio mixing desks. For the dynamics processor, it displays seven parameters at a time. The maximum number of different parameters it may display at once is eight. This gives an even more limited view of the audio mixing desk’s configuration. The parameters displayed on the LCD display are adjusted with the knobs shown below the parameter. This allows for the displayed parameters to be adjusted quicker as there is no need to initially navigate to the required parameter with cursor keys.

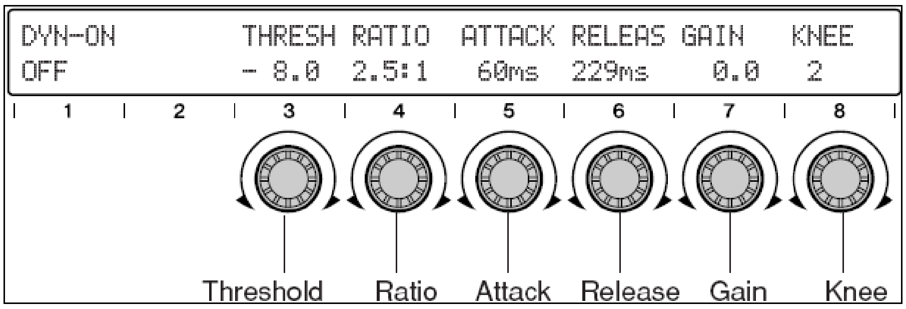

DYN-ON THRESH RATIO ATTACK RELEAS GAIN KNEE
OFF - 8.0 2.5:1 60ms 229ms 0.0 2
1 2 3 4 5 6 7 8
Threshold
Ratio
Attack
Release
Gain
Knee

Figure 7: The Yamaha 01X Digital Mixing Studio dynamics editing page

Adjusting parameters via the LCD display of a digital audio mixing desk can become a tedious task. If the parameters of signal processing components other than those currently being displayed need to be adjusted, there is a continuous need to navigate to the pages that allow for the parameter adjustments as the LCD display of the devices is small and only shows a limited amount of information. Once the correct page is located, there may be a need to navigate to the correct control on the page to allow the required parameter to be adjusted. Access to parameters may be slow. In contrast, locating parameters on a typical analogue audio mixing desk is a case of locating the required physical control on the control surface, and adjusting it.

2.1.2. Configuring A Digital Audio Mixing Desk Via Its Control Surface

The complexity of digital audio mixing is shown by way of an example for the Yamaha 01V96 Digital Mixing Console: assume that there is a microphone plugged into the first analogue input of the Yamaha 01V96 Digital Mixing Console. The signal from that input needs to be routed to the seventeenth input channel. The tone of the signal needs to be altered by routing it through the input channel’s equaliser and by adjusting the equalisation parameters to shape the sound. The signal needs to be routed through the dynamics processor to apply some compression to the signal. The parameters of the compressor need to be adjusted to compress the audio signal appropriately. The signal then needs to be routed onto the fourth auxiliary bus where it is to be mixed with the other signals present on that audio bus. The mixed signal from the fourth auxiliary bus needs to be routed to the first slot output located on the mini-YGDAI (Yamaha General Digital Audio Interface) card of the device. The Yamaha 01V96 Digital Mixing Console allows for a mini-YGDAI I/O card to be slotted into the device. These cards offer AD/DA conversions, and various analogue I/O options and digital I/O interfaces. The MY8-mLAN card, for example, allows the audio mixing desk to receive and transmit audio signals to and from an mLAN network.

In order to set up the audio mixing desk via its control surface in such a way that it performs the above routing and signal processing, a sound engineer would have to:

- Route the input signal from the first analogue input to the seventeenth input channel by:
    - Pressing the ‘17-32’ Layer button.
    - Pressing the ‘Patch’ button.
    - Selecting the ‘In Patch’ page on the LCD display.
    - Pressing the ‘Sel’ button for the seventeenth input channel.
    - Turning the parameter wheel until ‘AD1’ is shown as being routed through to the seventeenth input channel on the ‘In Patch’ page.
    - Pressing the ‘Enter’ button in order to make the patch between the analogue input and the input channel.
- Route the audio signal from the seventeenth input channel to the fourth auxiliary bus by:
    - Pressing the ‘Aux 4’ ‘Fader Mode’ button.
    - Pressing the ‘Enter’ button to make sure that the seventeenth input channel is routed through to the fourth auxiliary bus.
    - Raising the level of the first fader on the control surface to the desired level. This specifies how much of the input audio signal is sent to the fourth auxiliary bus. Here it is mixed with any other signals present on that audio bus.
- Route the audio signal present on the fourth auxiliary bus to the first mini-YGDAI slot output by:
    - Pressing the ‘Patch’ button.
    - Selecting the ‘Out Patch’ page on the LCD display.
    - Navigating to the first slot output patch with the cursor keys.
    - Selecting ‘Aux 4’ so that it is patched through to the first slot output by turning the parameter wheel until this is shown.
    - Pressing the ‘Enter’ button to make the patch between the fourth auxiliary bus and the first slot output.
- Route the incoming audio signal through the equaliser by:
    - Pressing the ‘EQ’ button.
    - Selecting the ‘EQ Edit’ page on the LCD display.
    - Turn the equaliser on by navigating to the ‘EQ On’ ‘On’ button with the cursor buttons and pressing the ‘Enter’ button in order to switch the equaliser on.

    - Navigating to the desired equalisation parameters with the cursor buttons and adjusting them with the parameter wheel until the desired sound is heard.
- Route the audio signal through the dynamics processor by:
    - Pressing the ‘Dynamics’ button.
    - Pressing the ‘Comp Edit’ button.
    - Turning the compressor on by navigating to the ‘On/Off’ ‘On’ button with the cursor buttons and turning the compressor on by pressing the ‘Enter’ button.
    - Navigating to the desired compressor parameters with the cursor buttons and adjusting the parameters via the parameter wheel until the desired sound is heard.

From this example it is apparent that configuring a digital audio mixing desk via its control surface can be a complicated task. The majority of the parameters that belong to the digital audio mixing desk are shown via a set of hierarchical pages on the LCD display of the device. Each time a different parameter is required, a sound engineer may have to navigate to the control that allows for the adjustment of the parameter.

If the parameter is located on the same page as the one currently being displayed, the parameter is selected by moving the cursor over the parameter via the set of cursor buttons. If, for example, the ‘Attack’ control shown in Figure 4 was currently selected, and a sound engineer wishes to turn the dynamics processor on with the ‘On/Off’ button, s/he would have to navigate the cursor to the left three times using the appropriate cursor key until the appropriate control is selected.

When locating parameter controls that belong to channels other than the channel that is currently selected, sound engineers first have to select the desired channel, and then press the button that represents the component that the parameter belongs to. Depending on the required parameter, it may be immediately visible and selected, or may require that a sound engineer navigate through further sub pages. Once the required sub page is located, the required parameter control may need navigating to via the cursor buttons.

Sound engineers have to gain a firm understanding of these digital audio mixing desks and their functionality in order to be proficient at using them to their maximum capacity.

2.2. Audio Mixing Desk Editors

An audio mixing desk editor is software that is used to emulate and remotely control an audio mixing desk. When the controls on a software audio mixing desk editor are adjusted, the editor instructs the associated audio mixing desk to adjust the corresponding parameter. If, for example, one of the graphical faders on the audio mixing desk editor is moved, the motorized fader on the audio mixing desk will move in sync with the graphical fader. On the audio mixing desk, other parameters may not have physical controls to represent them or the controls do not physically move, but their values may be shown as being adjusted on the LCD display of the audio mixing desk.

Controllers on the audio mixing desk can also control a software editor's controls. When the parameters on an automated audio mixing desk are being adjusted via the controls on the audio mixing desk, it may instruct an associated audio mixing desk editor to adjust its graphical representations of these parameters and so reflect the state that the parameters are in. For instance, if a fader on an audio mixing desk is adjusted, the corresponding graphical fader on the audio mixing desk editor will move as well.

This mechanism allows a software audio mixing desk editor to be in sync with the represented audio mixing desk at all times. It emulates the represented audio mixing desk. This mechanism allows for audio mixing desk state persistence. Because the software audio mixing desk editor is in the same state as the audio mixing desk at all times, it is capable of saving the state of the parameters of the audio mixing desk. This allows the saved state of those parameters to be recalled at a later stage, thus allowing the audio mixing desk's parameters to be brought back to their previous states.

Communication between an audio mixing desk and its software counterpart takes place with a control protocol, such as MIDI. MIDI is a control protocol that is used by devices to send messages to each other [MIDI Manufacturers Association, 2007]. MIDI can be used for applications that need to transmit and receive real-time performance controls, timing references, as well as universal or device specific parameters and data. In the audio mixing world, MIDI can be used to recall snapshots of an audio mixing desk's settings and can be used to adjust its various parameters.

This process is shown diagrammatically in Figure 8. When parameters are adjusted on a MIDI software audio mixing desk editor (annotation 1 in the figure), it sends out MIDI messages that specify which parameters have been adjusted, and the new values of those parameters (annotation 2 in the figure). The associated audio mixing desk is able to pick up these MIDI messages and is able to determine the new values of the adjusted parameters (annotation 3 in the figure). Every time a parameter on an automated MIDI audio mixing desk is adjusted (annotation A in the figure), it will send out a MIDI message that specifies the new value of that specific parameter (annotation B in the figure). The corresponding software audio mixing desk editor is capable of picking up that MIDI message, determining which parameter has been adjusted, and the value that the parameter is now set to (annotation C in the figure). It is therefore able to update its representation of that parameter.

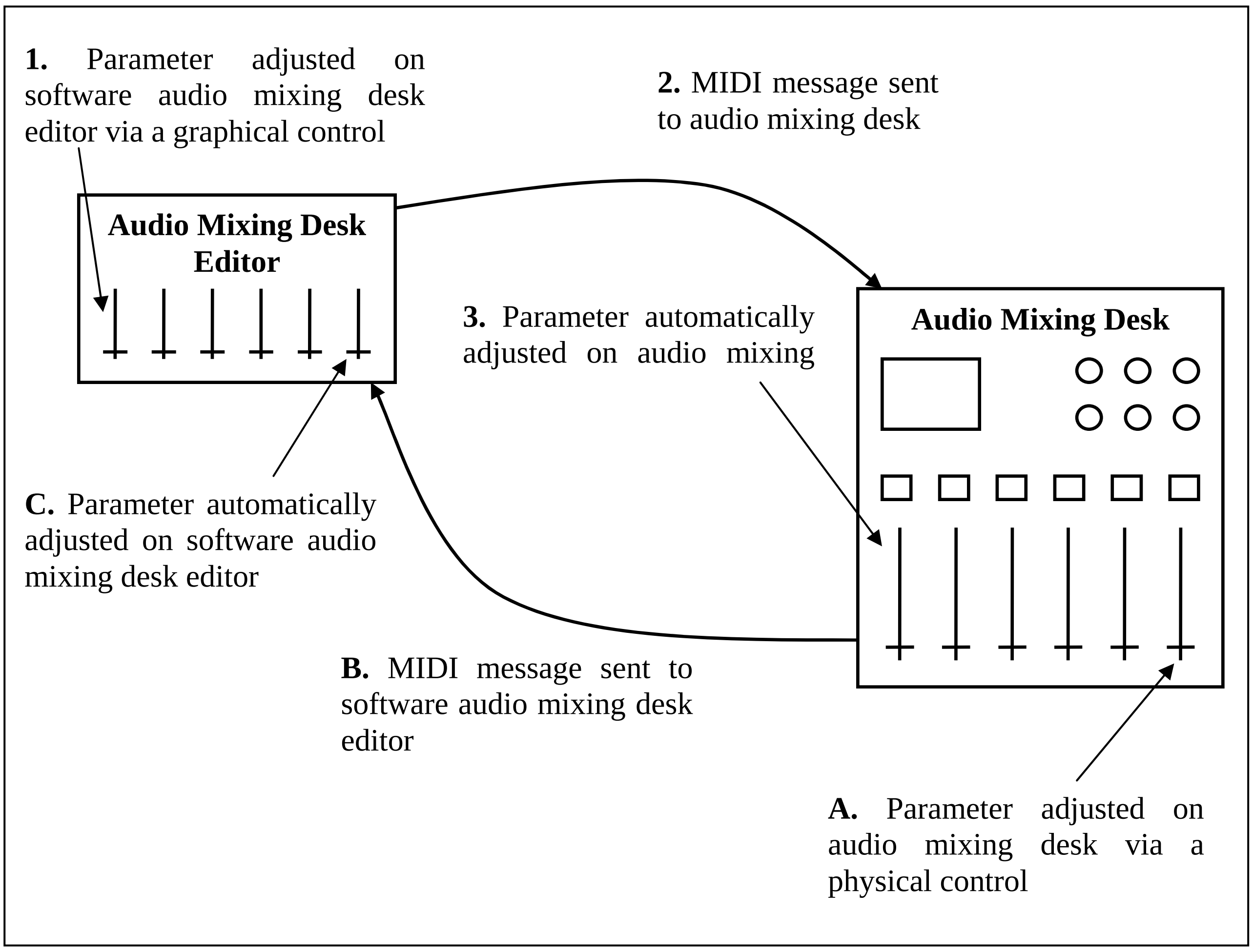

1. Parameter adjusted on software audio mixing desk editor via a graphical control
2. MIDI message sent to audio mixing desk
Audio Mixing Desk Editor
3. Parameter automatically adjusted on audio mixing
Audio Mixing Desk
C. Parameter automatically adjusted on software audio mixing desk editor
B. MIDI message sent to software audio mixing desk editor
A. Parameter adjusted on audio mixing desk via a physical control

Figure 8: Using MIDI messages to automat audio mixing desks

Typical audio mixing desk editors reflect the control surfaces of their represented audio mixing desks and hence the complexities associated with the layout of controls. The next two sections will discuss two such audio mixing desk editors. The 01V96 Editor and The Visualizer for 03D.

2.2.1. 01V96 Editor

The 01V96 Editor [Yamaha, 2004b] is an audio mixing desk editor that is designed to represent and control the Yamaha 01V06 Digital Mixing Console. The primary window of the 01V96 Editor is shown in Figure 9. From this, it is clear that the software editor presents the user with a user interface that reflects the control surface of a typical audio mixing desk.

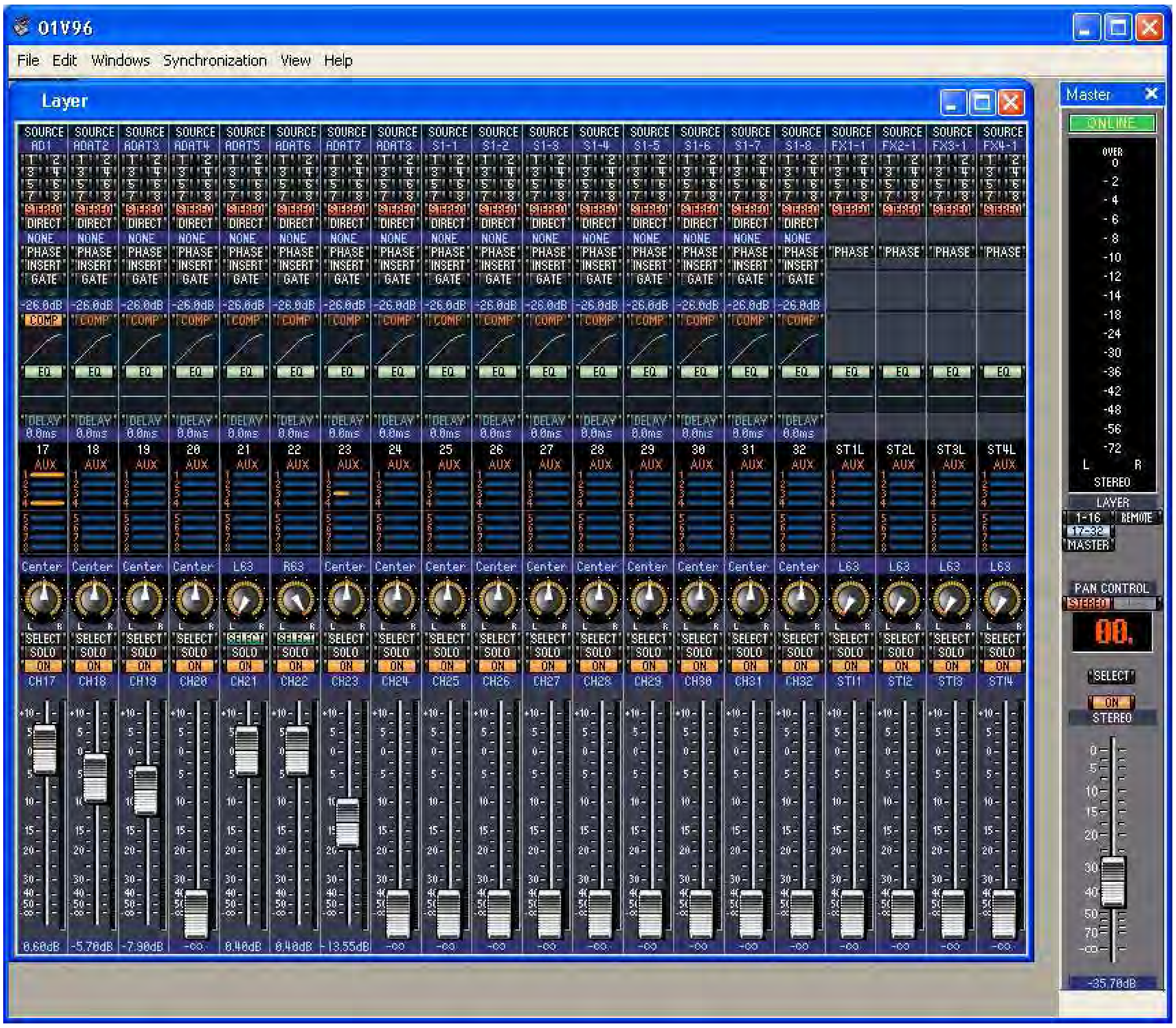


Figure 9: The 01V96 Editor primary window

The Yamaha 01V96 Digital Mixing Console layers its channels. The functionality of the controls on each channel strip depends on the layer that is selected. It is possible to select only one layer at a time. This device has four different layers. There are layers to represent input channels one to sixteen, input channels seventeen to thirty two, the master channels, and a remote layer that allows for the control of external equipment like Digital Audio Workstations (DAW) and other MIDI devices. This approach conceals the parameters of the channel strips that are not part of the selected layer. In order to adjust these parameters, the selected layer has to be changed. This layering technique is replicated on the 01V96 Editor software. Before sound engineers are able to adjust parameters of a non-selected layer via the primary window of the editor software, they have to select the required layer and then are able to make adjustments to the parameters.

However, the software counterpart does have a number of advantages over the hardware device that it represents.

The parameters of signal processing components, such as the equalisers and dynamics processors, on the Yamaha 01V96 Digital Mixing Console can only be viewed one channel at a time due to the limited size of the LCD display. Navigating to the required parameters often involves a tedious sequence of button pushes to locate the actual parameter. With the 01V96 Editor, it is possible to view the current state of the equalisers and dynamics processors with the aid of the curves shown for each channel on the main display. Each channel strip also provides access to the more common parameters that belong to each channel, such as volume, panning, aux sends, and parameters that allow the equaliser and dynamics processors to be turned on and off.

Parameter editing is also made easy with the aid of the Selected Channel window. The Selected Channel window allows for the editing of parameters that belong to the currently selected channel. This window is shown in Figure 10. It presents the user with a clear and uncluttered view of the selected channel’s available parameters and allows for easy editing of these parameters. Navigating to this window either involves selecting the desired channel first followed by selecting the Selected Channel menu from the Windows menu, or by pressing Ctrl + 3, or by right-clicking anywhere on the desired channel strip and selecting the ‘Open’ menu.

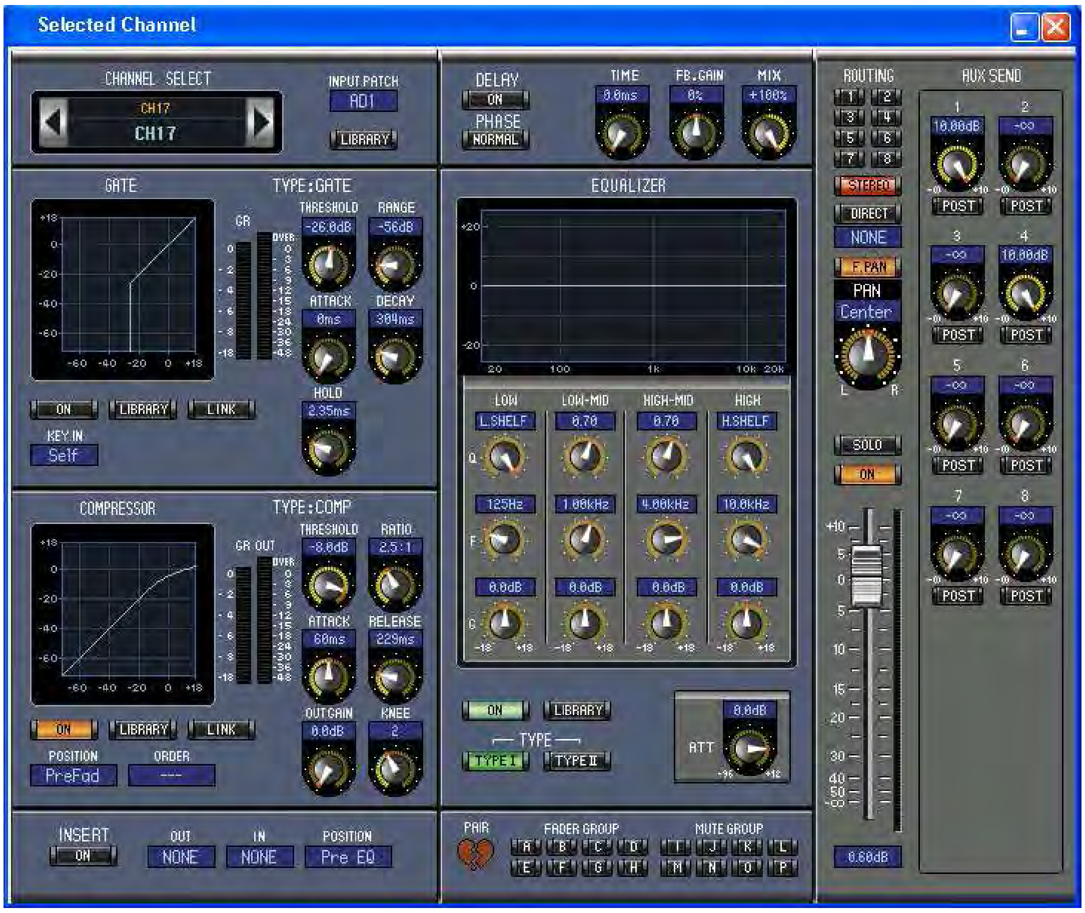


Figure 10: 01V96 Editor Selected Channel window

Routing audio within the Yamaha 01V96 Digital Mixing Console is made possible by the Patch Editor window, the use of the buttons and sliders available on each channel strip as well as the buttons and knobs available on the Selected Channel window. The Patch Editor window is shown in Figure 11. This window provides easier access to the patching facilities when compared to using the controls on the surface of the 01V96 Digital Mixing Console. The Patch Editor window allows for patching to happen simply but is limited in that it only allows patching for inputs, outputs, inserts, effects, and direct outs. It does not provide a holistic view of the current audio routing configuration. Patching is performed by simply selecting or de-selecting the cross points on the routing grid between the desired signal source point and signal destination point. If, for example, there exists a need to route the audio signal coming in on the first analogue input channel to the seventeenth input channel, a sound engineer would select the point on the grid where the ‘AD IN 1’ label intersects the ‘CH 17’ label. This will instruct the audio mixing desk to make a patch between the two points. Similarly, if the patch point between the two points needs to be broken, a sound engineer would de-select the cross points between the two labels.

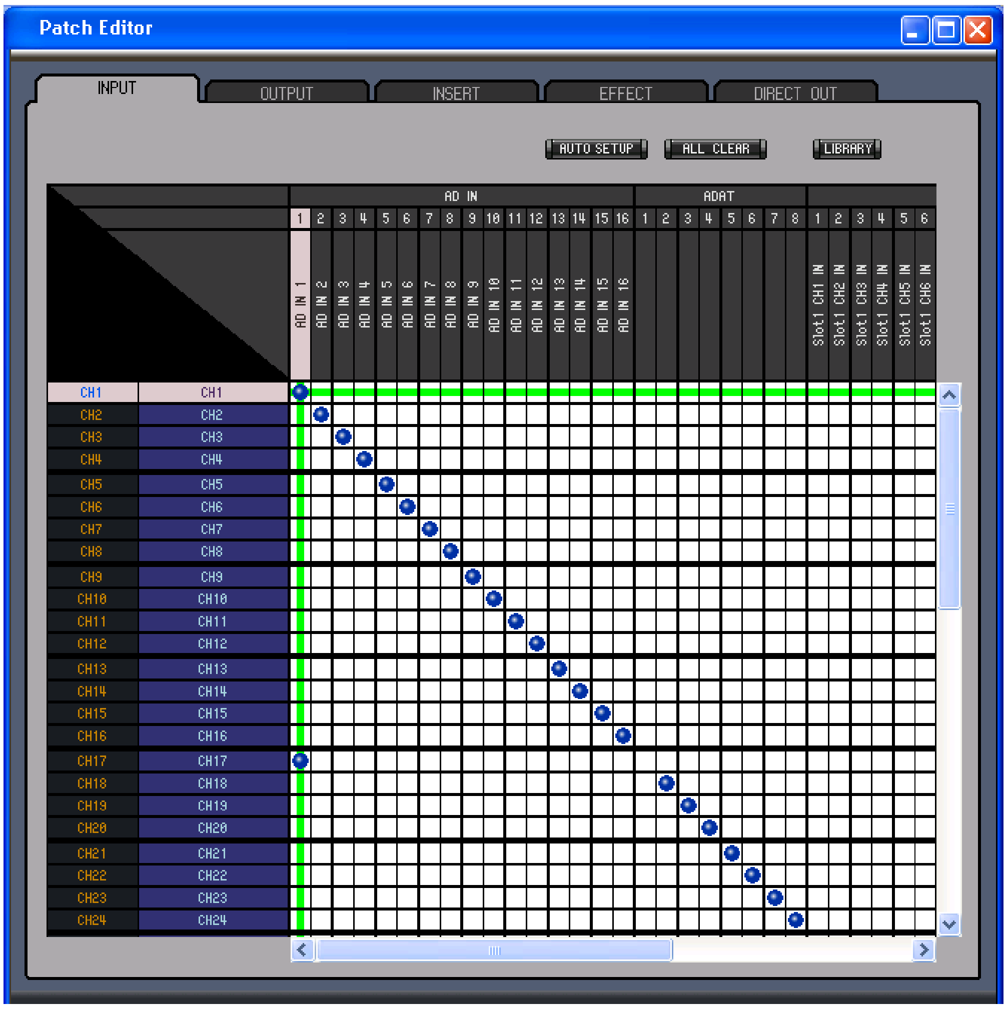


Figure 11: 01V96 Editor Patch Editor window

The primary window of the 01V96 Editor displays a large number of parameters and provides quicker access to them. This does however make the display look cluttered which can be quite intimidating. This device editor still makes use of layering and thus does not display all of its parameters simultaneously. Some of the controls are very small and this can make parameter adjustments inaccurate. The Patch Editor window has a busy layout and as a result detracts the user from its simplicity and capability.

2.2.2. C-Mexx Visualizer for Yamaha 03D

The C-Mexx Visualizer for Yamaha 03D [C-Mexx, 1998] was developed to represent and remotely control the Yamaha 03D Digital Mixing Console. The primary window of the C-Mexx Visualizer for Yamaha 03D is shown in Figure 12. As with the primary window of the 01V96 Editor (see Figure 9), this primary window also reflects the control surface of a typical audio mixing desk.

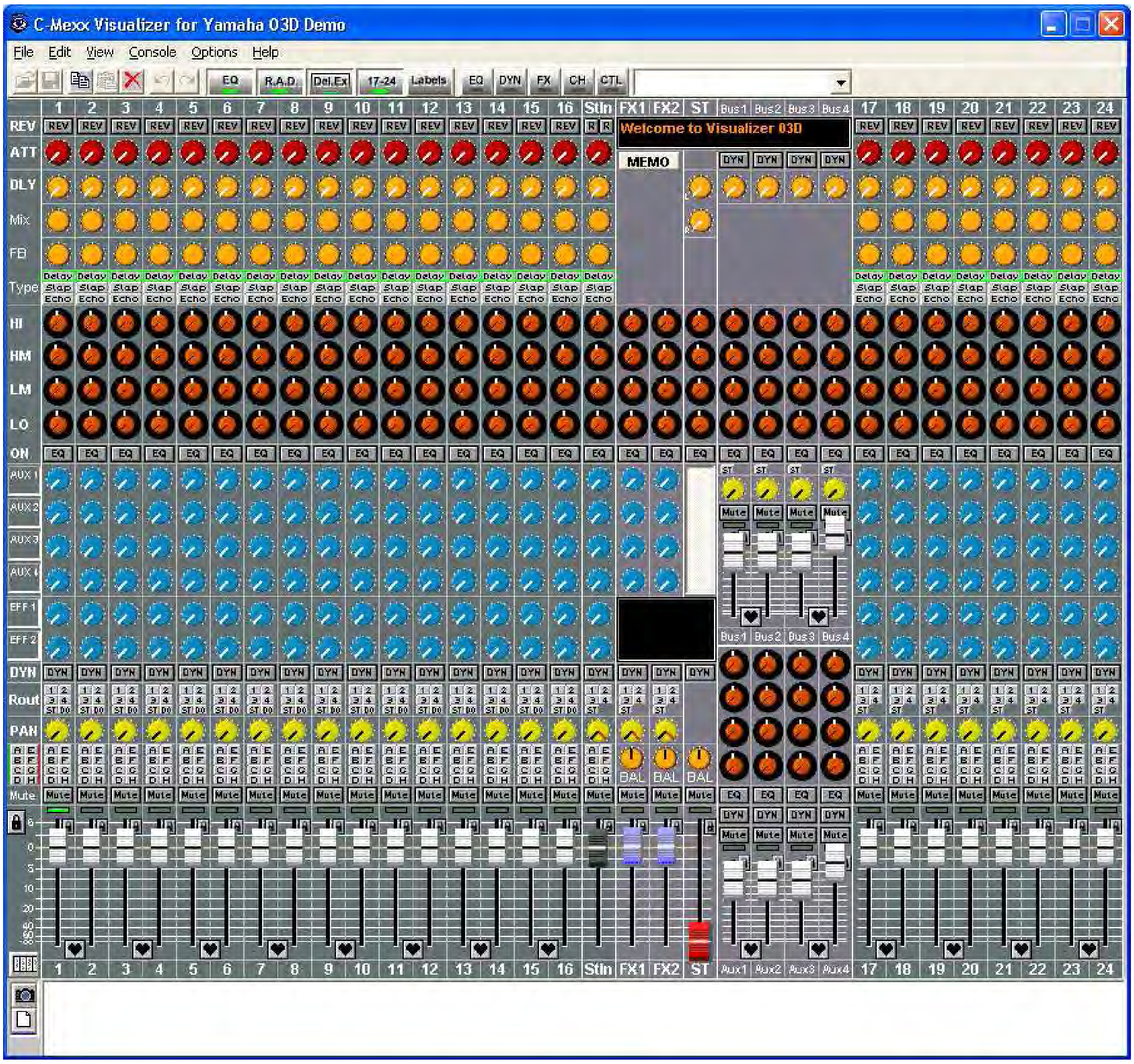

C-Mexx Visualizer for Yamaha 03D Demo
File Edit View Console Options Help
EQ R.A.D. Del.Ex 17-24 Labels EQ DYN FX CH CTL
Welcome to Visualizer 03D
MEMO
REV ATT DLY Mix FB Type HI HM LM LO ON AUX1 AUX2 AUX3 AUX4 EFF1 EFF2 DYN Rout PAN Mute
StIn FX1 FX2 ST Bus1 Bus2 Bus3 Bus4 Aux1 Aux2 Aux3 Aux4

Figure 12: The C-Mexx Visualizer for 03D Primary window

The Yamaha 03D Digital Mixing Console has two different mixing layers which are only visible one at a time. The Visualizer for Yamaha 03D displays both of these mixing layers simultaneously. This is advantageous in that there is no need to switch between mixing layers each time a parameter of a different mixing layer needs to be adjusted. This window also allows the state of common controls to be viewed for all input and output channels in the same way that the primary window of the 01V96 Editor does.

On the Yamaha 03D Digital Mixing Console, only a limited number of parameter controls are visible on the LCD display at any one time and often only a single channel at a time. Signal processing components, like the equalisers and dynamics processors, are displayed one channel at a time. The Visualizer for 03D does allow for limited access to some of the parameters of the equalisers via the primary window. Complete access to the parameters of the equalisers, dynamics processors and effects processors occurs via editor windows, but only one channel at a time. Each signal processing component has its own dedicated window. These windows may be seen in Figure 13, Figure 14 and Figure 15.

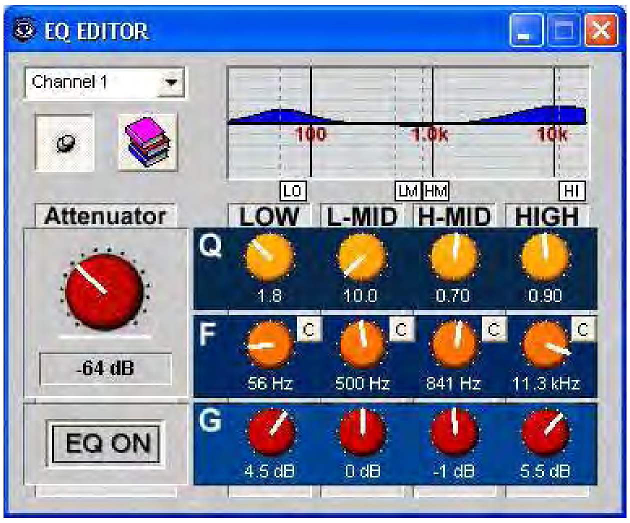


Figure 13: The Visualizer for 03D EQ Editor Window

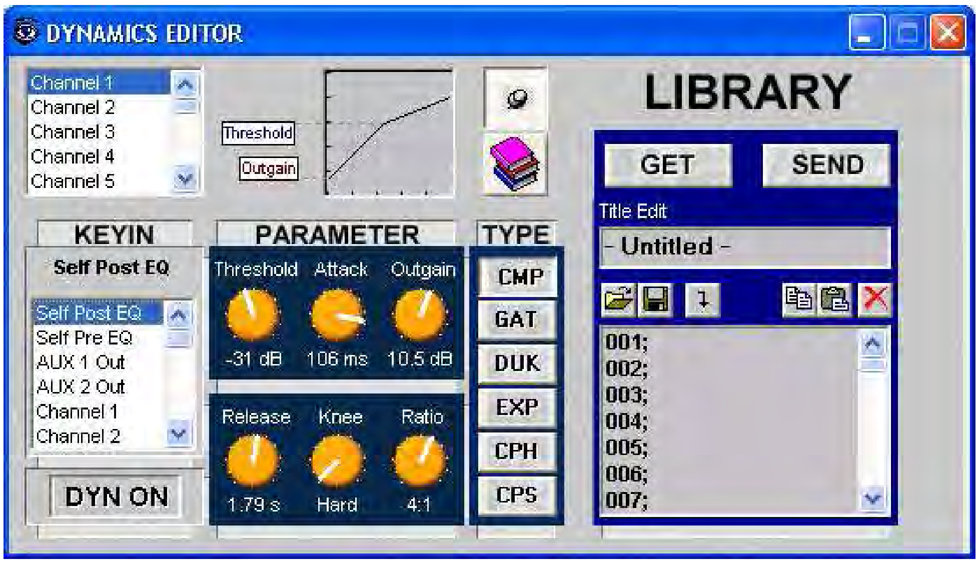
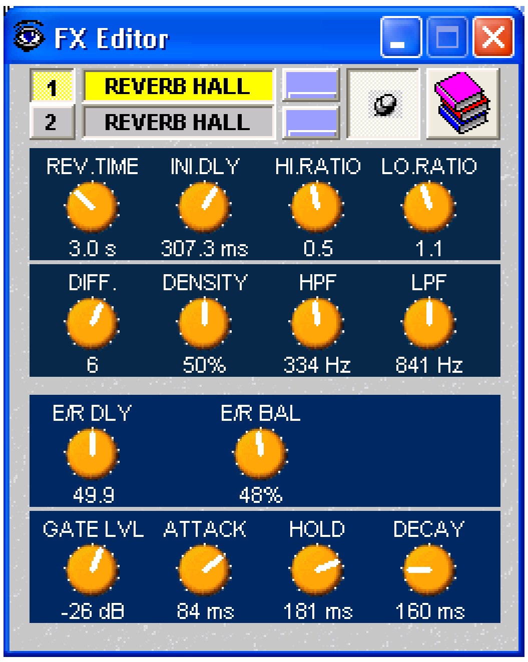


Figure 14: The Visualizer for 03D Dynamics Editor Window



1 REVERB HALL
2 REVERB HALL

REV.TIME	INI.DLY	HI.RATIO	LO.RATIO
3.0 s	307.3 ms	0.5	1.1
DIFF.	DENSITY	HPF	LPF
6	50%	334 Hz	841 Hz
E/R DLY	E/R BAL		
49.9	48%		
GATE LVL	ATTACK	HOLD	DECAY
-26 dB	84 ms	181 ms	160 ms

Figure 15: The Visualizer for 03D FX Editor Window

Whereas the 01V96 Editor has a Selected Channel window (see Figure 10) that displays all the available parameters for the selected channel, the Visualizer for 03D has a different window for the different signal processing components for the selected channel. It is difficult to get an overall view of a specific channel as many windows need to be opened to display all of its parameters. But each window does display its parameters in a clear, logical and uncluttered manner.

The primary window of the Visualizer for 03D also suffers from the fact that it is cluttered with numerous controls. It does, however, provide controls for a lot of functionality that belongs to each channel. It is clear from the primary window which functionality is available for each channel.

The primary window allows for audio routing to be configured. It presents the user with toggle buttons to route audio signals to the busses and rotary potentiometers to allow signals to be sent to the auxiliary and effects busses. This does provide a clearer view of the audio mixing desk's internal routing configurations, but still does not provide the current routing configuration in a clear and logical manner.

2.2.3. Configuring An Audio Mixing Desk Via A Software Editor

We shall return to the example given in section 2.1.2, but this time configuring the Yamaha 01V96 Digital mixing Console using the 01V96 Editor software. A sound engineer would have to perform the following steps in order to set up the device as in the example of section 2.1.2:

- Route the input signal from the first analogue input to seventeenth input channel by:
  - Selecting the ‘Windows’ menu and then selecting the ‘Patch Editor’ sub menu or by pressing Ctrl + 5.
  - Selecting the ‘Input’ tab on the Patch Editor window.
  - Selecting the cross point on the routing grid where ‘AD IN 1’ source signal label intercepts ‘CH 17’ destination point label.
- Route the audio signal present on the fourth auxiliary bus to the first slot output by:
  - Selecting the ‘Output’ tab on the Patch Editor window.
  - Selecting the cross point on the routing grid where the ‘AUX 4’ source signal label intercepts ‘Slot 1 CH 1 OUT’ destination point label.
- Route the audio signal from the seventeenth input channel to the fourth auxiliary bus by:
  - Selecting the primary window of the device editor.
  - Selecting mixing layer 17-32 by pressing the ‘17-32’ ‘Layer’ button.
  - Patching the signal from the seventeenth input channel through to the fourth auxiliary bus by selecting the auxiliary bus number.
  - Setting the fourth auxiliary bus send level by dragging the fader of the fourth auxiliary send of the seventeenth channel to the desired level.
- Route the audio signal through the equaliser by:
  - Right-clicking on the channel strip for the seventeenth input channel.
  - Selecting the ‘Open CH17’ menu to display the Selected Channel window for the seventeenth input channel.
  - Turing the equaliser on by selecting the ‘On’ button under the ‘Equaliser’ section.
  - Adjusting the parameters of the equaliser by turning the graphical potentiometers until the desired sound is heard.
- Route the audio signal through the dynamics processor by:
  - Turing the compressor on by selecting the ‘On’ button under the ‘Compressor’ section.
  - Adjusting the parameters of the compressor by turning the graphical potentiometers until the desired sound is heard.

Configuring the Yamaha 01V96 Digital Mixing Console via the 01V96 Editor software has several advantages over configuring it via the control surface of the device as shown by the examples given in sections 2.1.2 and 2.2.3. Because a computer monitor provides less space constraints than the LCD display located on the device, there is less need to navigate between different screens to get to required parameters. It is easier to access required parameter controls that belong to channels as these may be displayed on the primary window of the device editor or may be displayed by selecting the required channel strip and requesting to open the Selected Channel window. The Selected Channel window is able to display a lot of the functionality that is available at the selected channel, compared to only being able to display a limited amount of the functionality on the LCD display on the hardware device. Navigating to the controls that allow for the parameter adjustments is done with a computer mouse, so accessing the parameters is much quicker as there is no need to navigate through the controls one at a time until the required parameter is located.

With the 01V96 Editor software, creating patches in the input and output patch blocks of the audio mixing desk may be done by selecting the cross points on routing grids where the signal source and signal destination points intersect. Selecting the signal sources for channels may also be done by selecting the source signal for a specific channel at the top of the channel strip. Via the control surface of the device, sound engineers have to navigate to the required signal destination points, select the source signal by turning the parameter wheel until the required source is displayed as being patched through to the destination point, and then pressing the ‘Enter’ button to create the patch. Configuring patch points via the control surface of the audio mixing desk requires significantly more steps than configuring them via the routing grids of the 01V96 Editor software.

The primary window of the 01V96 Editor software has a more traditional analogue audio mixing desk look-and-feel to it. However, due to the feature richness of the Yamaha 01V96 Digital Mixing Console, the primary window is cluttered with many controls, some of which are small making parameters adjustments difficult.

2.3. Summary

This chapter introduced the concept of audio mixing at a high level. The Yamaha 01X Digital Mixing Studio, Yamaha 03D Digital Mixing Console and Yamaha 01V96 Digital Mixing Console were used as part of this study to gain insight into the workings of audio mixing desks. From using these devices it is apparent they are complicated devices to use via their control surfaces. Software audio mixing desk editors have been created that allow for the emulation and remote control of automated audio mixing desks, but these device editors tend to reflect a typical audio mixing desk.

From using the 01V96 Editor and The Visualizer for 03D, it is apparent that these software audio mixing desk editors have an audio mixing desk look-and-feel, thus incorporating the complexities of this kind of layout. Although the audio mixing desk editors expose more of the audio mixing desk’s functionality in one screen (similar to how analogue audio mixing desks display all their functionality all the time) this produces displays that are cluttered with many controls, some of which are small and difficult to use. These pieces of software do, however, allow for parameter editing of specific signal processing components via dedicated windows which provide clear, logical, and uncluttered control layouts. The 01V96 Editor provides audio routing capabilities through a pair of grid patch bays as well as via the buttons and sliders presented on its primary window. The Visualizer for 03D provides its routing functionality through the use of toggle buttons and rotary potentiometers on its primary window. These layouts have a disjointed feel because configuring the internal routing of these devices happens via many different types of controls. This also makes it difficult to gain insight into the internal routing configurations of the audio mixing desk.

When comparing setting up the Yamaha 01V96 Digital Mixing Console via its control surface (see section 2.1.2) against setting it up via the 01V96 Editor software (see section 2.2.3), it is apparent that the software device editor provide much quicker access to the controls that allow for the manipulation of the audio mixing desk. The parameter controls are easily accessible, reducing the complexity of setting up the device as there is no need to navigate through the pages on the small LCD display of the device to locate required parameters.

Chapter 03

Grid Mixing

In the context of digital audio networking a software grid patch bay (or routing matrix) is often used to patch audio signals from their source points through to various destination points. Audio mixing is essentially a process of patching audio signals from one point to another within an audio mixing desk as well as adjusting parameters at these various points. This gave rise to the idea of representing audio mixing desks as visually simple, uncluttered routing grids. Signal source points are represented along the left-hand-side of the grid and signal destination points are represented along the top of the grid. The cross points on the grid represent the patch points that may exist between the signal source points and signal destination points.

In this chapter, we shall look at the fundamentals of grid patchbays and propose an editor layout that is based on a grid patchbay which simplifies the control of audio mixing desks. We shall then propose a customizable software audio mixing desk control surface.

3.1. Grid Patch-Bays

Figure 16 shows the Audio Mapping window of the EtherSound ESControl program [EtherSound, 2007]. This program allows for the management of EtherSound devices on an EtherSound network. The routing matrix shown in the diagram allows for audio routing between EtherSound devices to be configured. The labels along the bottom of the routing matrix show the EtherSound devices on the network and the outputs associated with these devices. The labels along the left-hand-side of the routing matrix show the EtherSound devices available on the network and the inputs associated with each of those devices. Configuring routing between the outputs and inputs of the EtherSound devices happens via the grid. A route is configured by selecting the cross point on the routing matrix where the output label of one EtherSound device intersects the input label of another EtherSound device. The active patches are shown on the routing matrix as green crosses.

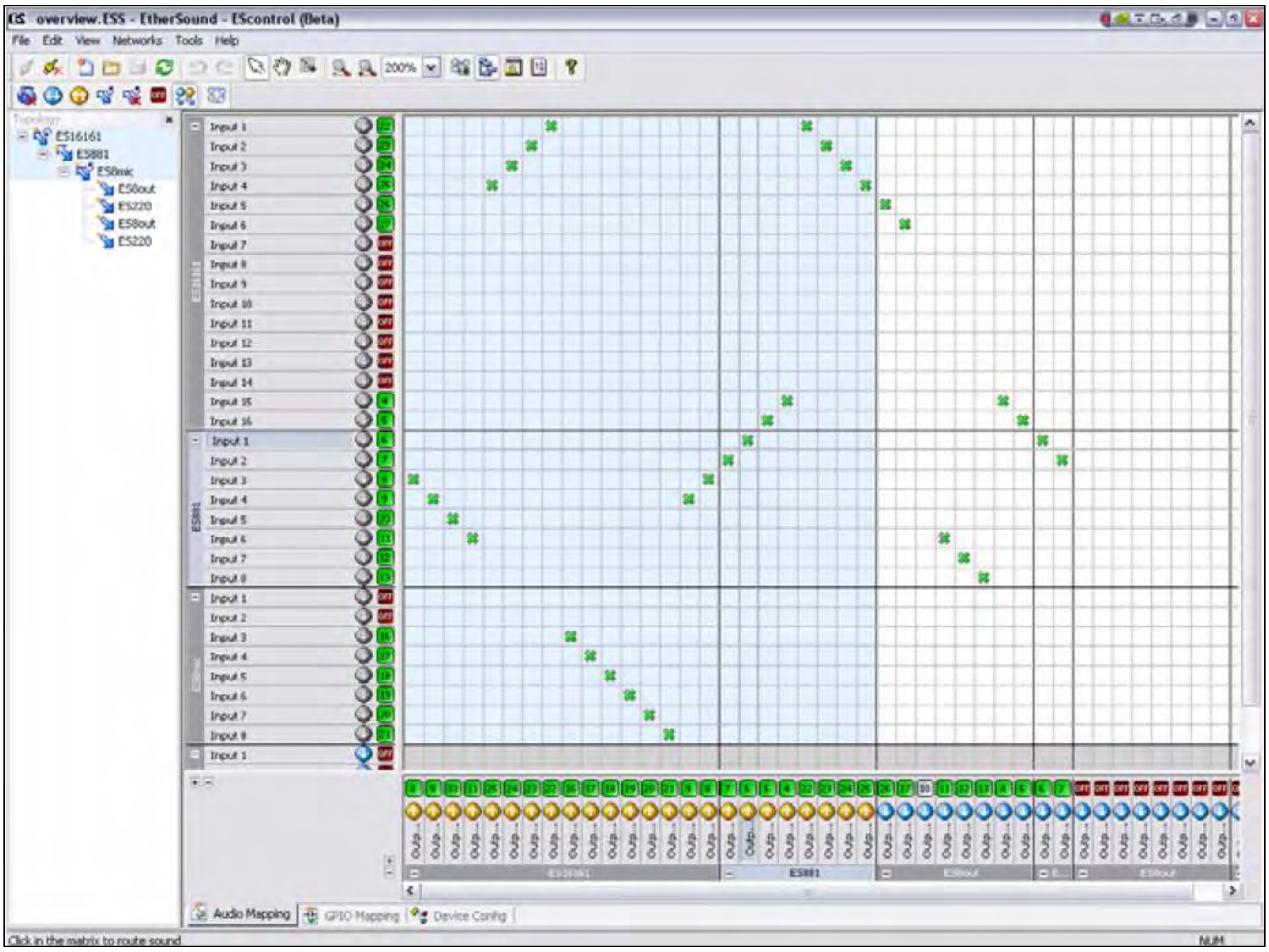


Figure 16: The EtherSound ESControl program

Figure 17 shows the Routing Matrix of the Otari mLAN Control Software [Otari, 2005]. This routing matrix is used to route audio signals from the outputs of Otari ND-20B units to the inputs of other Otari ND-20B units on an mLAN network. The left-hand-side of the routing matrix shows the devices along with their input channels. The top of the routing matrix shows the devices along with their output channels. As with the routing matrix of the EtherSound ESControl program, in order to change routing configurations between the Otari ND-20B units on the network, the cross points between the signal source points and the signal destination points on the grid are selected or de-selected to make or break connections between the devices respectively.

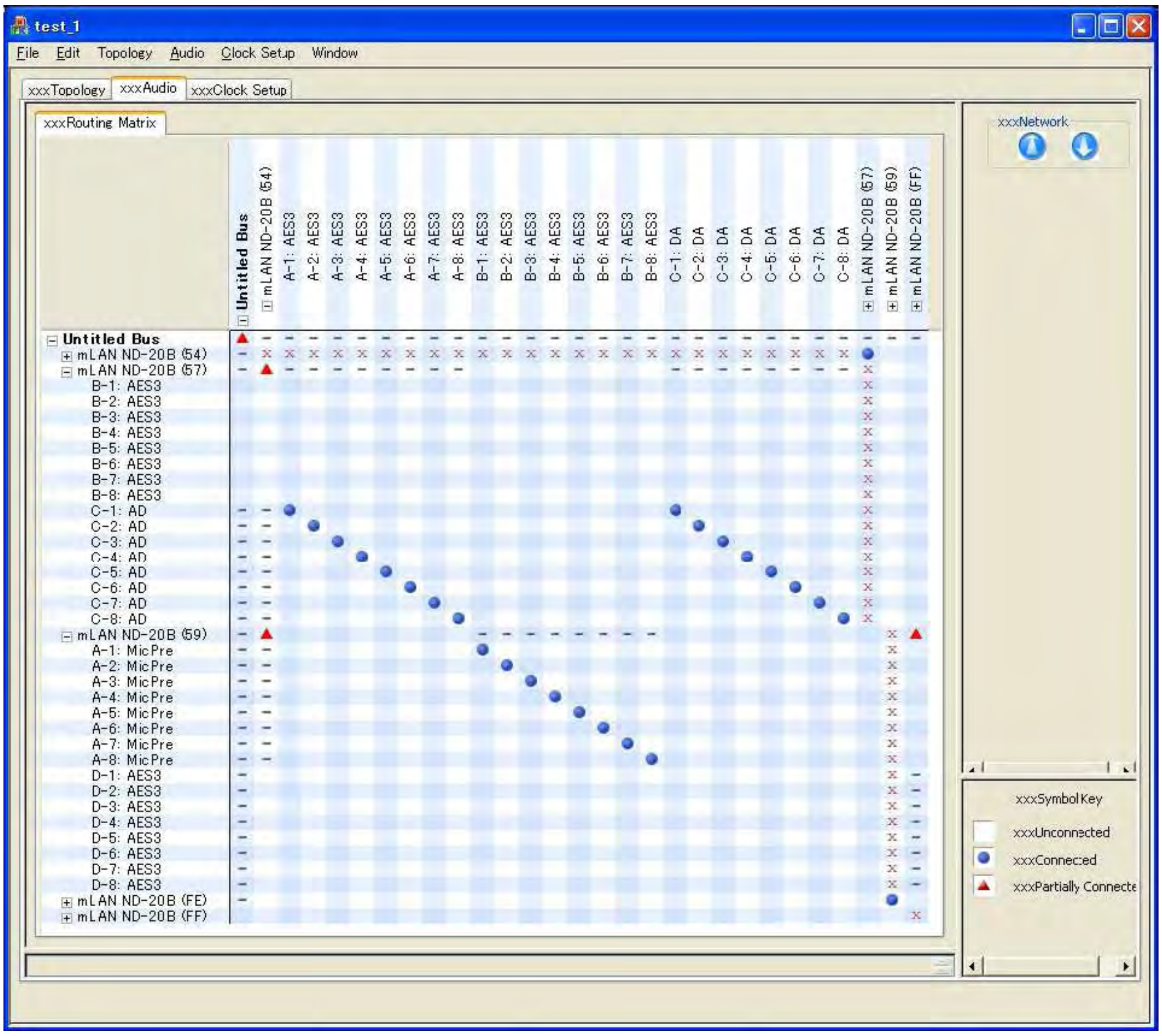


Figure 17: The Otari ND 20B mLAN Control Software Routing Matrix

These visually simple software user interfaces allow for dynamic routing configurations between signal source points and signal destination points to happen by simply selecting and de-selecting the cross points on the routing matrix. Such grid displays provide a clear layout of the routing configuration between devices.

3.2. Grid Mixing

Audio mixing console control essentially involves a series of:

- Routing audio signals from various audio sources within an audio mixing desk to various destination points within the same audio mixing desk.

- Adjusting parameters of signal processing components at the various signal processing points (signal source points, signal patch points and signal destination points) within the audio mixing desk to shape the audio as it travels through the device.

This can be seen in the block diagram for the Yamaha 01V96 Digital Mixing Console, shown in Figure 18. In a typical situation for this audio mixing desk, sound engineers would route the source audio signals from the analogue and digital inputs of the audio mixing desk (annotation 1 in the figure) to the destination input channels of the audio mixing desk (annotation 3 in the figure). The input patching happens in the input patch block (annotation 2 in the figure). The arriving signals are fed through signal shaping and manipulation components such as volume faders and equalisers (annotation 4 in the figure). These source signals are patched onto the audio busses (annotation 5 in the figure) within the audio mixing desk where they are mixed with the other signals present on the audio bus (annotation 6 in the figure). The signals from the audio busses are fed through signal shaping and manipulation components (annotation 7 in the figure). These source signals are patched (annotation 8 in the figure) through to the analogue and digital outputs (destination points) on the audio mixing desk (annotation 9 in the figure).

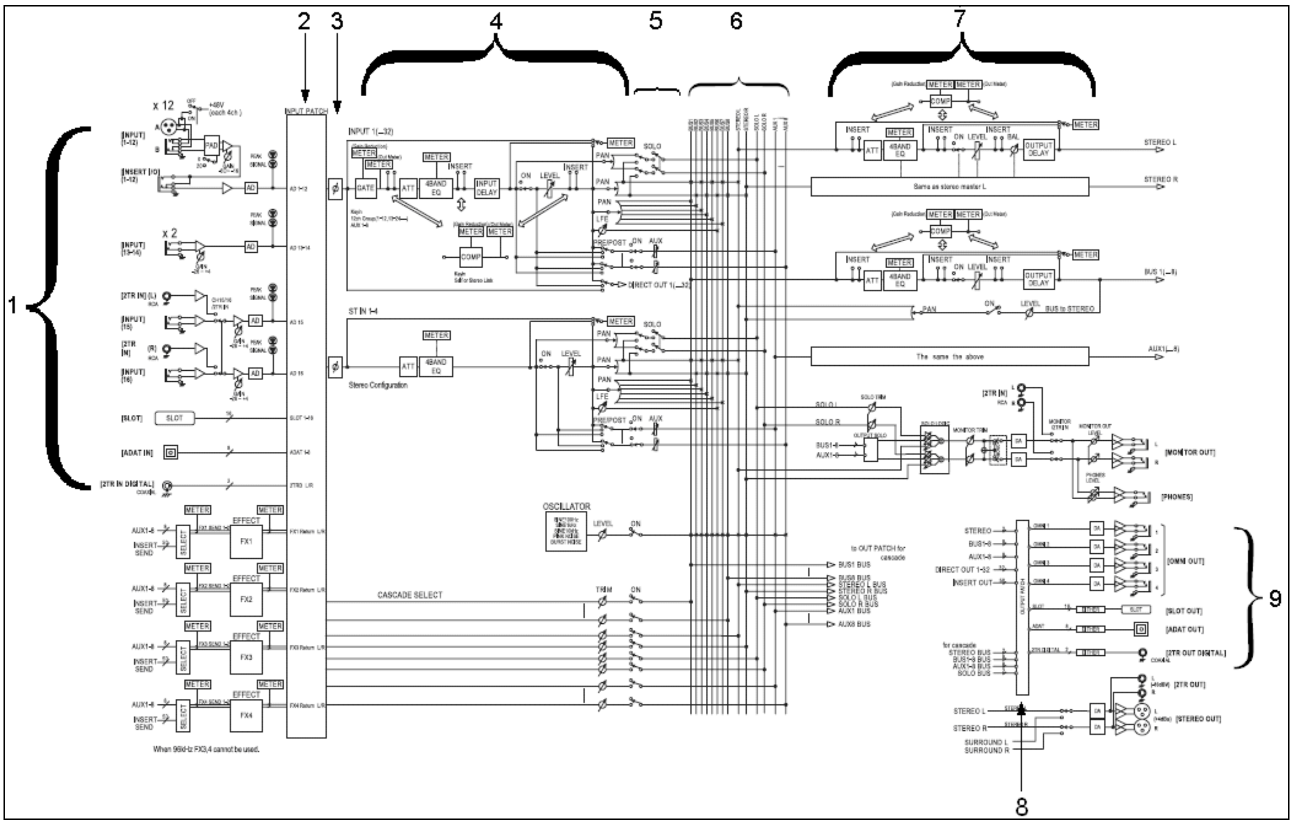


Figure 18: The block diagram for the Yamaha 01V96 Digital Mixing Console

When sound engineers determine the possible routing configurations and signal processing capabilities of a digital audio mixing desk, they would typically study the block diagram of the device. Once the routing and signal processing capabilities of the audio mixing desk are known sound engineers then have the onerous task of locating the controls that allow for parameter adjustments.

The Graphical User Interfaces (GUI's) of current software audio mixing desk editors reflect the control surfaces of typical audio mixing desks. The layout of controls do not give an insight into the routing configurations possible within the audio mixing desk being represented by the software and are often presented in a cluttered manner. The 01V96 Editor does provide a number of routing matrices that allow for patching to happen within the Yamaha 01V96 Digital Mixing Console but the routing is limited. They do not, for example, allow for audio signals to be routed onto the audio busses found within the audio mixing desk.

The fact that audio mixing desk control involves routing audio signals gave rise to the idea that audio mixing desks should be represented as simple graphical routing matrices. These should be similar in nature to that of the EtherSound ESControl and the Otari mLAN Control Software routing matrices shown in Figure 16 and Figure 17. Furthermore, the nature of the routing grids should mimic the signal paths of an audio mixing desk as they are depicted in the block diagram of the device. For example, in the block diagram for the Yamaha 01V96 Digital Mixing Console, the signals flow from the left-hand-side to the right-hand-side of the diagram in a series of stages. Usually input patching occurs, then audio signals are patched through to the audio busses with output patching then taking place. Each stage of the routing process should be represented by a set of routing matrices. The routing matrices should flow from the left to the right as the signals do in the block diagram. This should let sound engineers view the possible routing configurations and signal processing capabilities of the device as if looking at the block diagram of the device. The routing matrices should also allow for control over the parameters of the device

These routing matrices should have the following characteristics:

- The signal source points that exist within the audio mixing desk should be represented along the left-hand-side of the appropriate routing matrices with labels. Each label should give an indication as to which signal source point it is representing (for example, 'AD1' to represent the first analogue input).

- The signal destination points that exist within the audio mixing desk should be represented along the top of the appropriate routing matrices with labels. Each label should give an indication as to which signal destination point it is representing (for example, 'CH1' to represent the first input channel).
- The possible patch points that exist between the signal source points and signal destination points within the audio mixing desk should be represented as the cross points on the routing matrices.
- Creating and breaking patches within the audio mixing desk should be performed by selecting or de-selecting the appropriate cross points on the appropriate routing matrix, respectively.
- Adjusting parameters available at the signal processing points should be done by selecting the appropriate signal processing point and displaying graphical controls that represent the parameters available at the point.

An initial version of this idea was implemented and the product named the Matrix Mixer [Foss and Foulkes, 2006]. This implementation was created to represent and remotely control MIDI controllable audio mixing desks. The software was developed to be a generic audio mixing desk editor providing common audio mixing desk functionality. The initial implementation provided features that allowed for the following parameters to be adjusted: routing, volume, panning, muting, equalisation, dynamics and effects parameters. Each specific audio mixing desk is described using an XML document. The XML document describes the functionality of the specific audio mixing desk and the MIDI messages that are used to communicate with the audio mixing desk. When the software is initialised it configures itself based on the information in the selected XML document.

Functionality has since been added to:

- Represent more parameters.
- Allow for state persistence and bidirectional state transfer.
- Allow for custom control surfaces to be created by the user.
- Be Studio Connections Total Recall [Yamaha, 2005] compatible.

The initial system was developed to represent the Yamaha 01X Digital Mixing Studio but it has been further developed to represent the Yamaha 01V96 Digital Mixing Console as well.

3.2.1. Matrix Mixer Routing Configuration

Different views of the primary window of the Matrix Mixer are shown in Figure 19, Figure 20 and Figure 21. This specific instance is used to represent the Yamaha 01V96 Digital Mixing Console. Unlike the primary window of the 01V96 Editor and the primary window of The Visualizer for 03D, the Matrix Mixer presents the user with visually simple and uncluttered graphical routing matrices that represent the signal processing points that exist within an audio mixing desk.

In these figures, a red patch button indicates that an active patch exists between its corresponding source and destination signal processing point. An orange button indicates that an inactive patch exists between the corresponding source and destination point. A grey button indicates that an audio route between the corresponding points is not possible or not explicitly selectable.

Figure 19 shows one of the routing matrices that allows for patching to happen within the Yamaha 01V96 Digital Mixing Console’s input patch block (shown as annotation 2 in the block diagram for the device (see Figure 18)). This permits a sound engineer to instruct the audio mixing desk to route audio signals from the analogue and digital inputs of the audio mixing desk to its input channels. This happens by a sound engineer selecting the cross points on the routing matrix. The annotations in Figure 19 correspond to the annotations shown in the block diagram in Figure 18, and are explained below:

1. The labels along the left-hand-side of the routing matrix represent the analogue and digital inputs that the audio mixing desk has. For this stage of the patching process, these are the source signals.
2. The cross points on the routing matrix represent the possible patch points that may exist between the analogue and digital inputs and the input channels of the device. These represent the source signals being patched through to the destination points and may be selected or de-selected in order to make and break patches, respectively.
3. The labels along the top of the routing matrix represent the input channels that the audio mixing desk has. For this stage of the patching process these are the signal destination points.

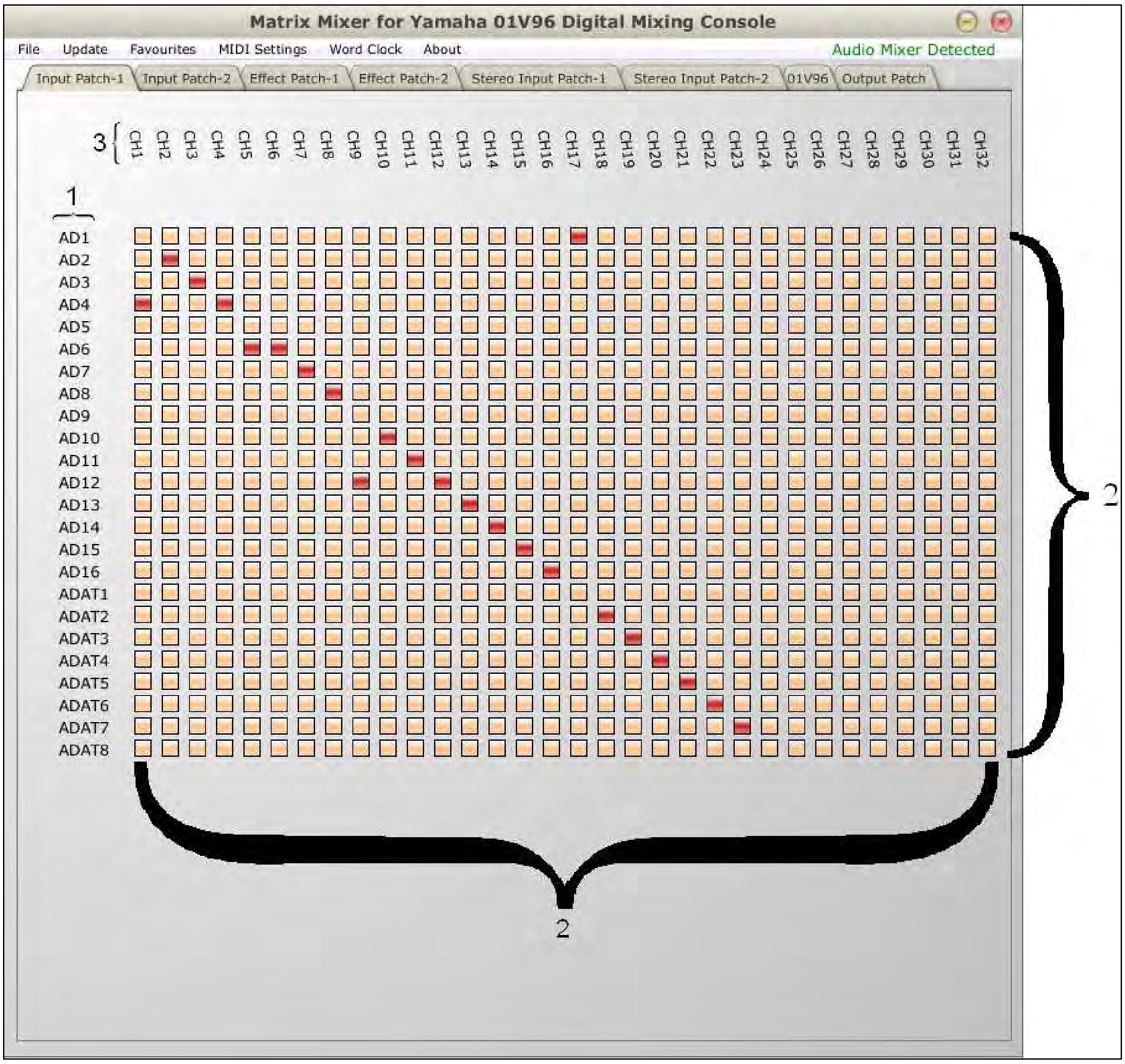

Matrix Mixer for Yamaha 01V96 Digital Mixing Console
File
Update
Favourites
MIDI Settings
Word Clock
About
Audio Mixer Detected
Input Patch-1
Input Patch-2
Effect Patch-1
Effect Patch-2
Stereo Input Patch-1
Stereo Input Patch-2
01V96
Output Patch
3
CH1 CH2 CH3 CH4 CH5 CH6 CH7 CH8 CH9 CH10 CH11 CH12 CH13 CH14 CH15 CH16 CH17 CH18 CH19 CH20 CH21 CH22 CH23 CH24 CH25 CH26 CH27 CH28 CH29 CH30 CH31 CH32
1
AD1 AD2 AD3 AD4 AD5 AD6 AD7 AD8 AD9 AD10 AD11 AD12 AD13 AD14 AD15 AD16 ADAT1 ADAT2 ADAT3 ADAT4 ADAT5 ADAT6 ADAT7 ADAT8
2
2

Figure 19: The Matrix Mixer for 01V96 primary window – input patching

Via the routing matrices that represent the input patching, sound engineers are able to view what inputs and input channels the audio mixing desk has as well as the possible routing configurations that may exist between these different source and destination points. They are also able to view the current routing configurations and perform routing by selecting and de-selecting the cross points on the matrices.

The main routing matrix of this instance of the Matrix Mixer (shown in Figure 20) allows for audio signals to be routed from the audio mixing desk's input channels to its various internal audio busses where the audio signals are mixed together into single signals. The annotations in the diagram correspond to the annotations on the block diagram for the Yamaha 01V96 Digital Mixing Console (see Figure 18) and are explained below:

3. The labels along the left-hand-side of the routing matrix represent the audio mixing desk’s input channels. In Figure 19, they represent the signal destination points. Here they are the signal sources for the next stage of the patching process.
5. The cross points on the routing matrix represent the patch points that may exist between the signal sources and the signal destination points. Here it is possible to instruct the associated audio mixing desk to route audio signals from the audio mixing desk’s input channels onto its audio busses by selecting the appropriate cross points.
6. The labels along the top of the grid represent the audio mixing desk’s internal audio busses. At this stage of the patching process they represent the signal destination points.

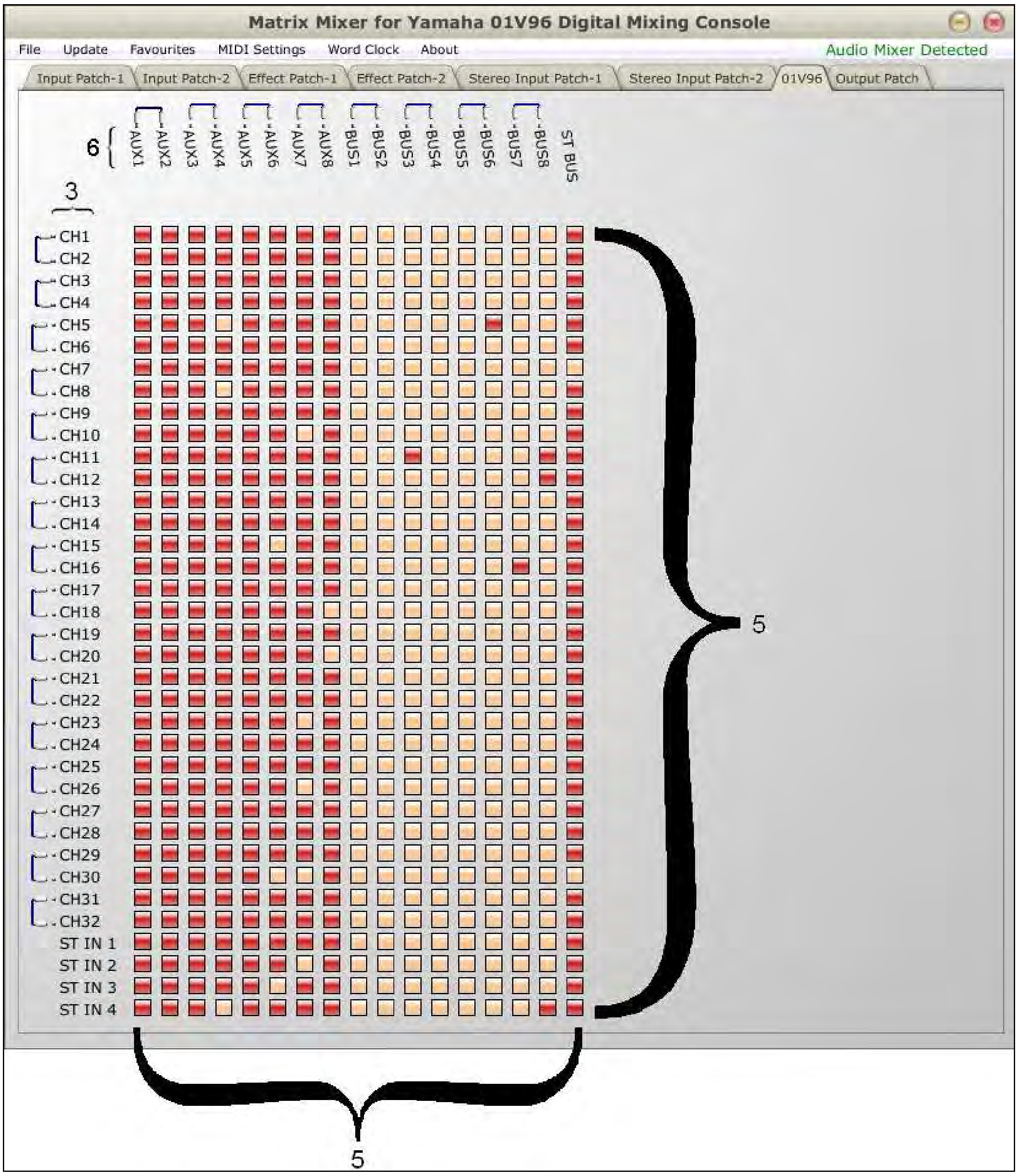


Figure 20: The Matrix Mixer for 01V96 primary window – bus sends

From this routing matrix, sound engineers are able to learn what input channels and audio busses the audio mixing desk has. They are also able to see what routing configurations are possible, the current routing configurations that are in place, as well as make routing changes by selecting and de-selecting the cross points on the routing matrix.

Figure 21 shows the routing matrix that allows for output patching to be performed within the Yamaha 01V96 Digital Mixing Console. This routing matrix allows for audio signals to be routed from the audio mixing desk’s audio busses to the audio mixing desk’s analogue and digital outputs. The annotations in the diagram correspond to the annotations in the block diagram for the device (see Figure 18) and are explained below:

6. The labels along the left-hand-side of the routing matrix represent the audio signals that have been mixed together into single signals on the audio busses of the audio mixing desk. At this stage of the patching process they are considered signal sources.
8. The cross points on this matrix represent the possible patch points that may exist between the signal source points and the signal destination points. It also shows the current routing configurations that exist between the signal source points and signal destination points.
9. The labels at the top of the routing matrix represent the outputs that exist on the audio mixing desk. These are the signal destination points.

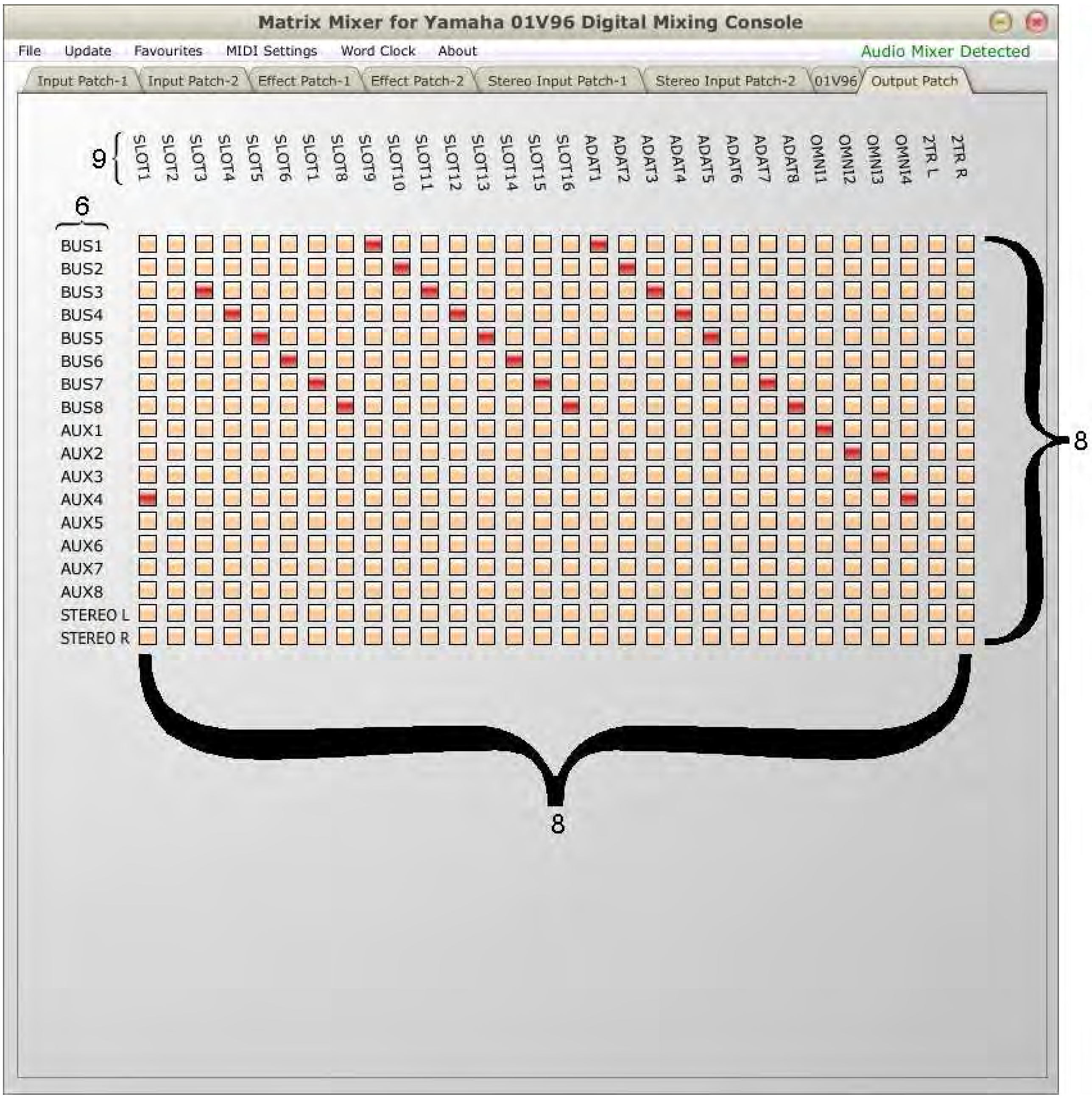


Figure 21: The Matrix Mixer for 01V96 primary window – output patching

From this routing matrix, sound engineers are able to learn what audio busses and outputs exist on the audio mixing desk. It also shows the possible and currently active patch points that exist within the audio mixing desk and enables sound engineers to change these configurations by selecting and de-selecting the cross points on the routing matrix.

From the routing matrix shown in Figure 20, it is also possible to pair adjacent odd and even channels. This allows the corresponding parameters of paired channels to work together. If, for example, the first input channel was paired with the second input channel and the volume fader of the first input channel was adjusted, the volume fader of the second input channel would move in tandem with the first channel’s volume fader. This allows two channels to be used together to create stereo channels. The Matrix Mixer allows sound engineers to pair channels by selecting the patch cable indicators to be found either to the left of the signal source labels or at the top of the signal destination labels.

Below is the functionality of the routing matrices of the Yamaha 01V96 Digital Mixing Console’s Matrix Mixer:

- **Input Patch routing matrices**: The first two routing matrices are used to patch the signals coming into the input patch block through to the audio mixing desk’s input channels.
- **Effect Patch routing matrices**: The second two routing matrices are used to patch audio signals through to the audio mixing desk’s internal effects units.
- **Stereo Input Patch routing matrices**: The next two routing matrices are used to patch audio signals through to the audio mixing desk’s stereo input channels.
- **01V96 routing matrix:** Once the input patching has taken place, the audio signals need to be routed from the input channels onto the internal audio busses. This is done via the ‘01V96’ routing matrix.
- **Output Patch routing matrix:** Finally, the last routing matrix is used to route the audio signals from the internal audio busses to the outputs of the audio mixing desk.

Each of the routing matrices allows a sound engineer to see the available signal source, destination and patch points that are available at various parts of the associated audio mixing desk. With the layout of the routing matrices, sound engineers would work from left to right setting up the signal flows within the associated audio mixing desk in a logical way as depicted in the block diagram of the device.

Representing the signal processing points of an audio mixing desk as labels and buttons takes up less space than the channel strips presented on a typical audio mixing desk editor. This then in turn has reduced the layering present on the actual audio mixing desk and its software editor. On the Matrix Mixer for Yamaha 01V96 Digital Mixing Console, all the input channels, audio busses and the patch points between these points are represented on a single routing grid. On the actual device and on the 01V96 Editor, the input channels and audio busses are represented on three different layers.

3.2.2. Adjusting Parameters

There are a number of points within an audio mixing desk at which the signal present may be manipulated (see annotation 4 and 7 in Figure 18). These signals are manipulated with various signal processing and sound shaping components. The input channels, for example, may have volume faders, equalisers and dynamics processors associated with them. These components allow for the manipulation of the audio signal present at each specific input channel.

Each signal processing component has adjustable parameters that belong to it. The Matrix Mixer allows for the manipulation of these parameters via graphical controls on a Parameter Adjust Window which is displayed by selecting the required signal processing point on the appropriate routing matrix. The Parameter Adjust Window only displays the graphical controls of the parameters that are available at the selected signal processing point. Figure 22, Figure 23, Figure 24 and Figure 25 show four different examples of Parameter Adjust Windows. From these figures, it can be seen that the window is constructed according to the parameters available at the signal processing point. The Parameter Adjust Window shown in Figure 22 is the window that is displayed when the first input channel is selected on the ‘01V96’ routing matrix. This channel is represented by the ‘CH1’ label and is a signal source point. These parameters correspond to the parameters shown as annotation 4 on the block diagram in Figure 18.

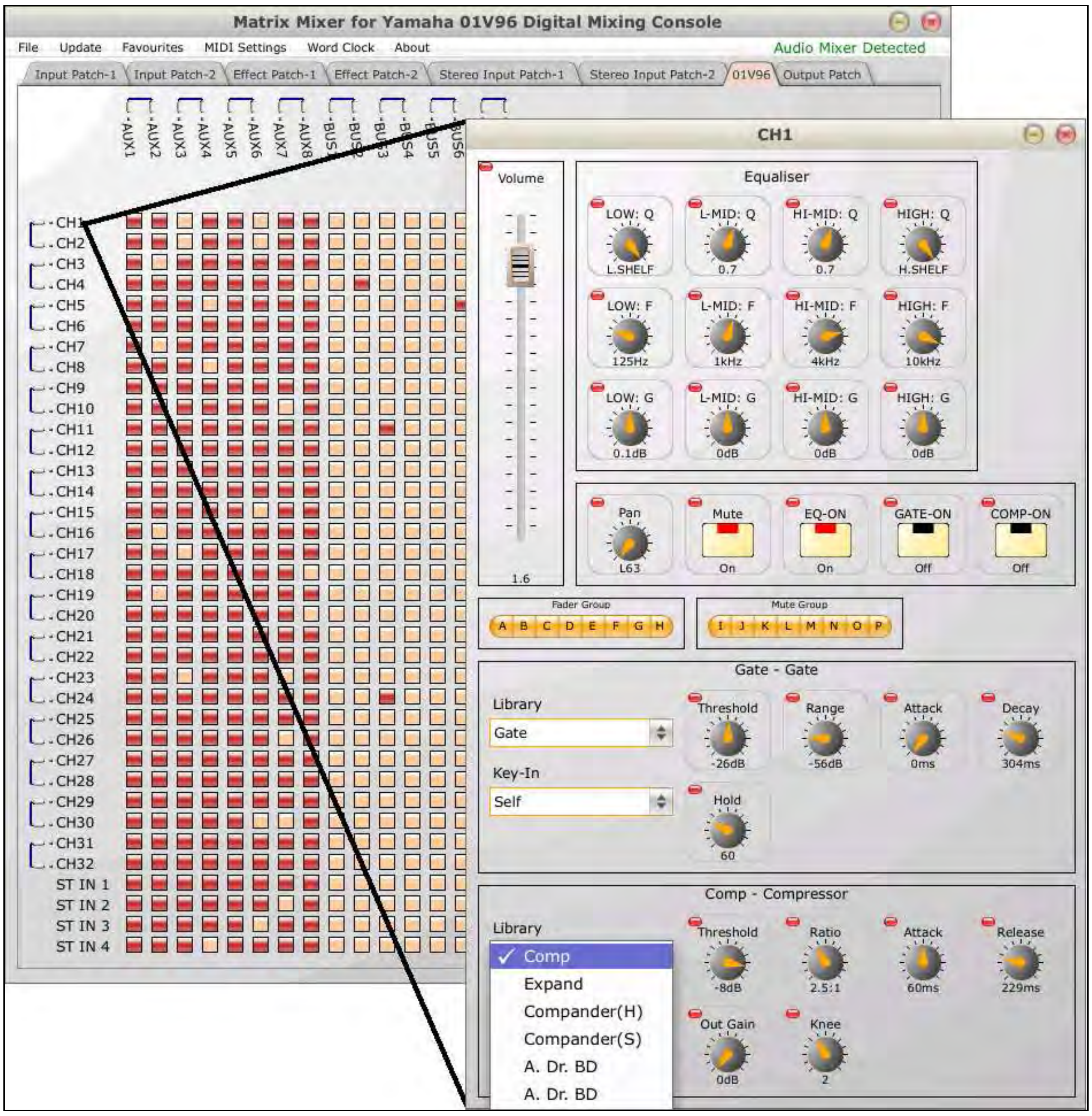


Figure 22: The Matrix Mixer Parameter Adjust window for 'CH1'

Via this Parameter Adjust Window shown for the first input channel of the Yamaha 01V96 Digital Mixing Console, a sound engineer is able to:

- Adjust the volume level of the signal present at the selected point. This is performed by moving the linear fader under the 'Volume' section.
- Adjust the equalisation parameters available at the selected signal processing point. These are changed by adjusting the graphical rotary potentiometers under the 'Equaliser' section.
- Adjust the pan parameter available at the signal processing point. This is performed by adjusting the rotary potentiometer under the 'Pan' section.
- Switch the mute parameter available at the signal processing point. The signal at this point may be muted and un-muted by selecting and deselecting the button under the 'Mute' section, respectively.

- Switch the equaliser bypass parameter. This parameter allows the signal present at the signal processing point to bypass the equaliser. This is performed by selecting and de-selecting the button under the ‘EQ-ON’ section.
- Switch the dynamics processor bypass parameters. These parameters allow the signal present at the signal processing point to bypass the available dynamics processors. This is performed by selected and de-selecting buttons under the ‘GATE-ON’ and ‘COMP-ON’ sections.
- Group parameters together. This allows parameters to be grouped to work together with parameters of the same kind. The Yamaha 01V96 Digital Mixing Console has various fader and mute groups to which faders and mute buttons may be added. All the parameters in a specific group work together. If, for example, the volume fader of the first and second input channels were both in fader group A and one of the those faders is moved, the other fader in the group will move relative to the fader being moved. The volume fader that is represented on the Parameter Adjust Window is added to a fader group by pressing one of the fader group buttons under the ‘Fader Group’ section. Similarly, the mute button being represented on the Parameter Adjust Window is added to a mute group by pressing one of the buttons under the ‘Mute Group’ section.
- Select a specific type of dynamics processor. This is done via a library of preset dynamics processors. Each of the dynamics processors shown under the ‘Gate - …’ and ‘Comp - …’ sections were selected from a library of present dynamics processors. The library of preset dynamics processor values is presented with a combo box from which a library title is selectable. Each library title is associated with a specific type of dynamics processor and a set of preset parameters. In the diagram, the ‘Comp’ library title represents a compressor and the ‘Expand’ library title represents an expander. Selecting these will set the dynamics processor to the type of dynamics processor the library title is representing. The ‘A. Dr. BD’ library title also represents a compressor, but the values of the compressor’s parameters are different when compared to the values of the parameters associated with ‘Comp’ library title. Once a library title has been selected, the parameters associated with the specific type of dynamics processor are displayed along with their preset values.
- Adjust parameters that allow the selected dynamics processor’s characteristics to be adjusted. These parameters are adjusted via the rotary potentiometers under the ‘Gate - …’ and ‘Comp - …’ sections. For dynamics processors that have a key-in parameter, the available values for this parameter are selectable from the combo box under the ‘Key-in’ section.

The window in Figure 23 is displayed when the patch point between the first input channel and the first auxiliary bus is selected. This point corresponds to annotation 5 in the block diagram (see Figure 18) and is considered as a signal patch point.

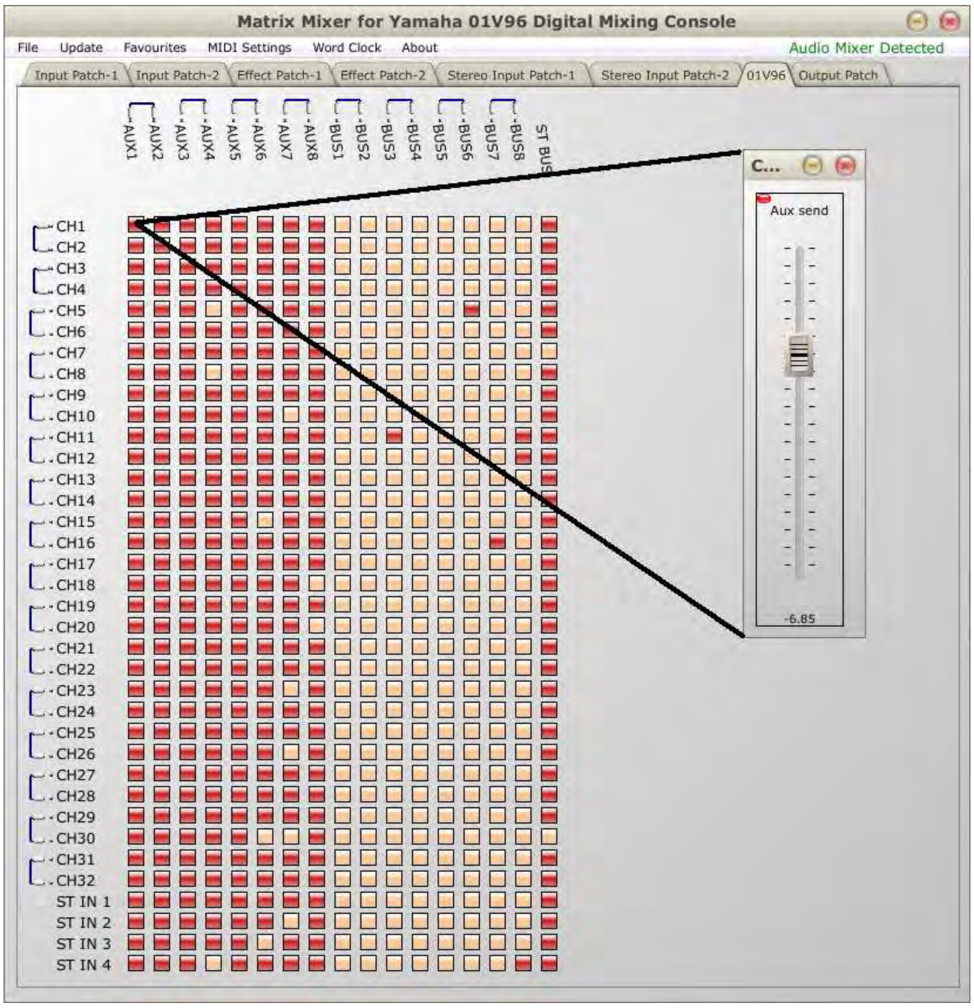


Figure 23: The Matrix Mixer Parameter Adjust window for ‘CH1-AUX1’

Via this Parameter Adjust Window a sound engineer is able to adjust the amount of signal present at the first input channel that is sent to the first auxiliary bus. This is done by adjusting the linear fader under the ‘Aux send’ section to the desired level.

Figure 24 shows the window that is displayed when the stereo bus signal destination point is selected. This signal destination point is represented with the ‘ST BUS’ label.

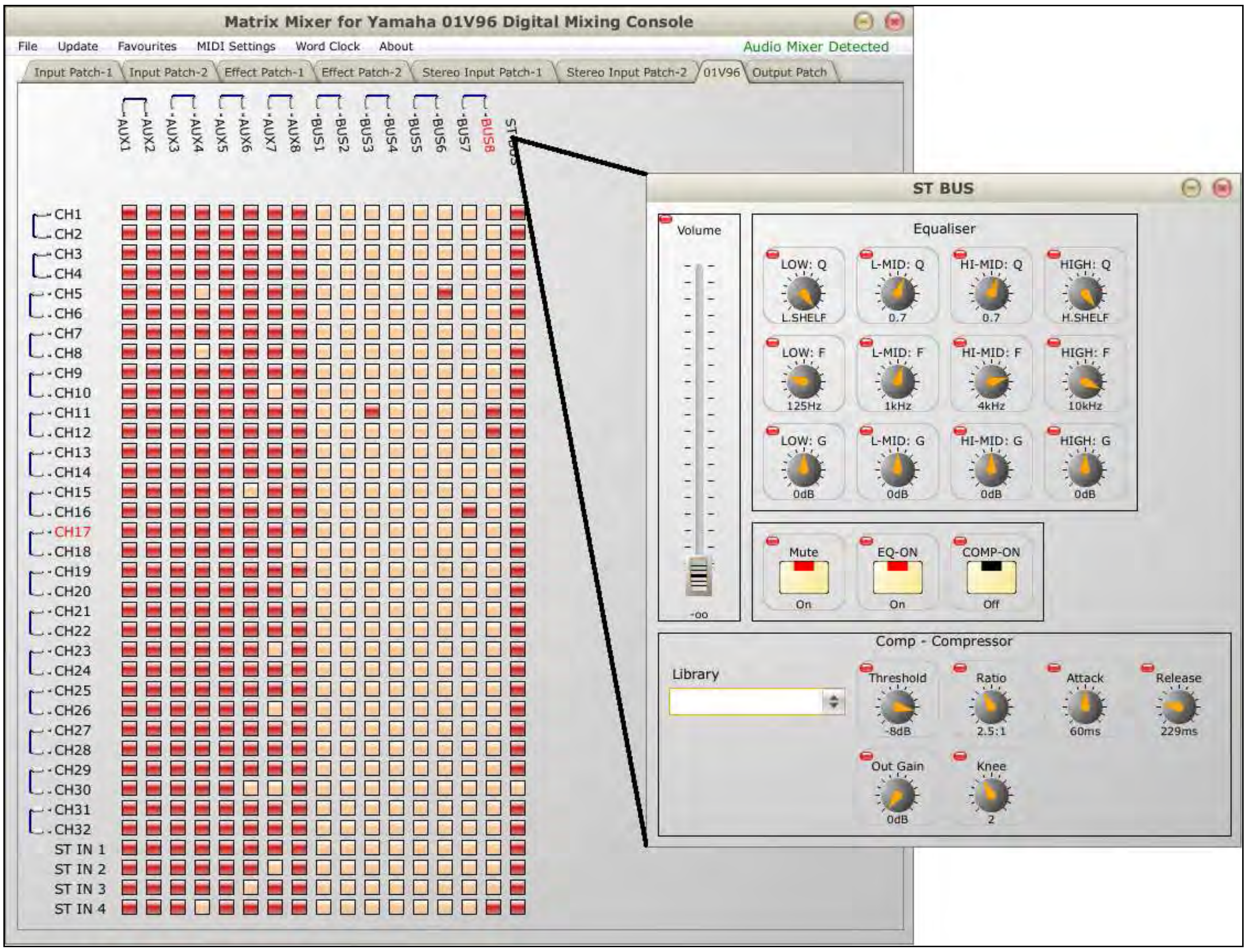

Matrix Mixer for Yamaha 01V96 Digital Mixing Console
File Update Favourites MIDI Settings Word Clock About
Audio Mixer Detected
Input Patch-1 Input Patch-2 Effect Patch-1 Effect Patch-2 Stereo Input Patch-1 Stereo Input Patch-2 01V96 Output Patch
AUX1 AUX2 AUX3 AUX4 AUX5 AUX6 AUX7 AUX8 BUS1 BUS2 BUS3 BUS4 BUS5 BUS6 BUS7 BUS8 ST BUS
CH1 CH2 CH3 CH4 CH5 CH6 CH7 CH8 CH9 CH10 CH11 CH12 CH13 CH14 CH15 CH16 CH17 CH18 CH19 CH20 CH21 CH22 CH23 CH24 CH25 CH26 CH27 CH28 CH29 CH30 CH31 CH32
ST IN 1 ST IN 2 ST IN 3 ST IN 4
ST BUS
Volume
-oo
Equaliser
LOW: Q L.SHELF
L-MID: Q 0.7
HI-MID: Q 0.7
HIGH: Q H.SHELF
LOW: F 125Hz
L-MID: F 1kHz
HI-MID: F 4kHz
HIGH: F 10kHz
LOW: G 0dB
L-MID: G 0dB
HI-MID: G 0dB
HIGH: G 0dB
Mute On
EQ-ON On
COMP-ON Off
Comp - Compressor
Library
Threshold -8dB
Ratio 2.5:1
Attack 60ms
Release 229ms
Out Gain 0dB
Knee 2

Figure 24: The Matrix Mixer Parameter Adjust window for 'ST BUS'

With this specific Parameter Adjust Window, a sound engineer can adjust the same parameters as described for the first input channel, except for panning and those parameters related to the gate dynamics processor.

Figure 25 shows a Parameter Adjust Window with an effects processor component on it. This is displayed when one of the effects processor signal processing points is selected.

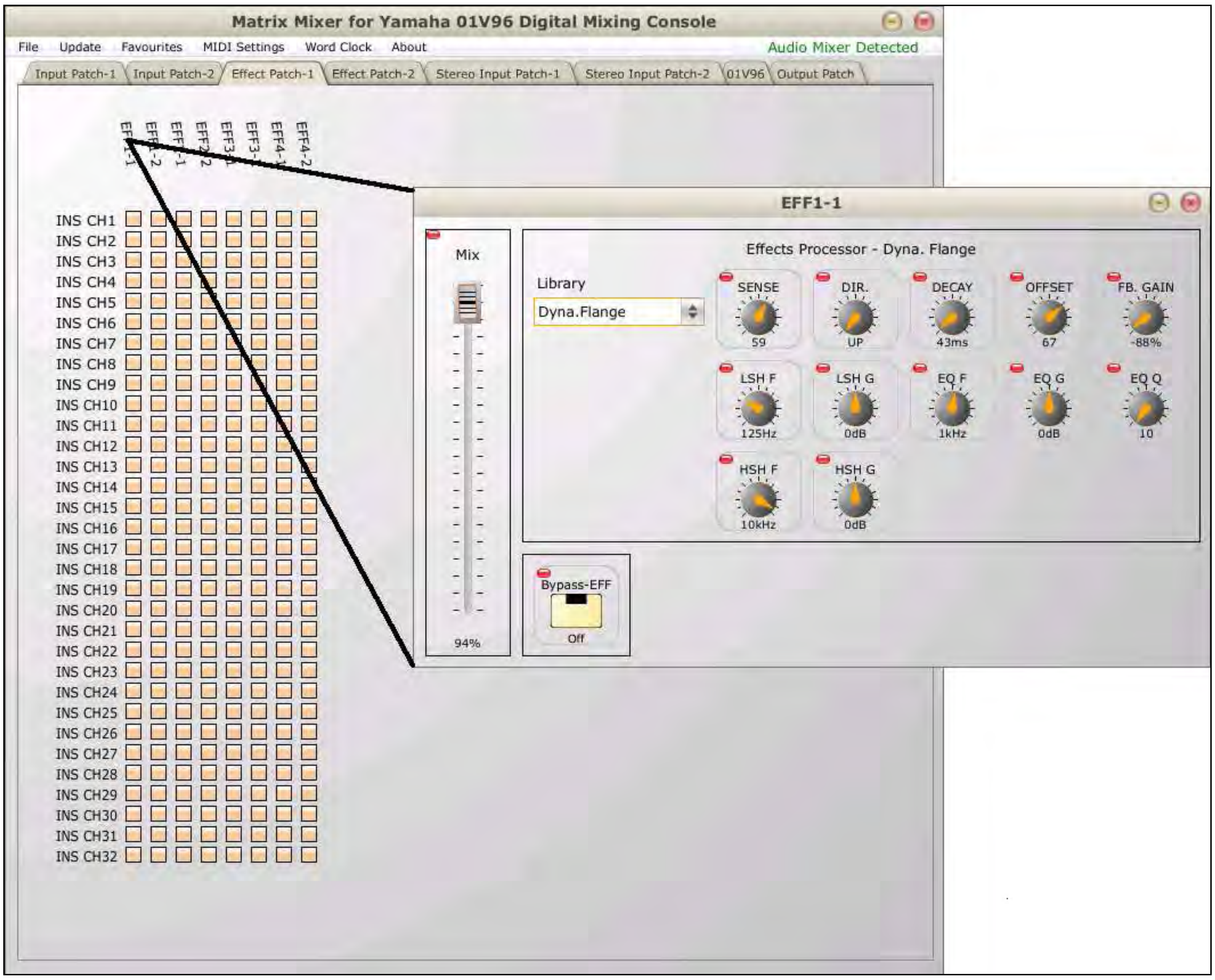


Figure 25: The Matrix Mixer Parameter Adjust window for 'EFF1-1'

Via this Parameter Adjust Window, parameter settings pertaining to the selected effects processor may be adjusted:

- On the Yamaha 01V96 Digital Mixing Console there are a number of types of effects. Each effects processor may be set to a specific type of effect. For example, there are reverb and echo effects. Each type of effect has a set of adjustable parameters associated with it. These can be seen in Figure 25 by the rotary potentiometers on the Parameter Adjust Window. A specific effect type is selected through the use of a library of preset effect settings. The library has a number of library titles. Each library title represents a specific type of effect with its specific set of parameters. A sound engineer wanting to change the type of effect that is selected would do so from the combo box under the 'Library' section. Once a selection is made, the parameters relating to the selected effect populate the Parameter Adjust Window. A sound engineer is then able to adjust the effect's parameters by adjusting the rotary potentiometers under the 'Effects Processor - …' section.

- Sound engineers are able to adjust the amount of balance between the amount of dry and wet signal that is available at the specific signal processing point. This is done via the linear fader under the ‘Mix’ section. Setting the parameter to 0% means that only the dry signal is heard and setting the parameter to 100% means that only the wet signal is heard.
- Sound engineers are able to select the incoming signal to bypass the effects processor. This is done by selecting the button under the ‘Bypass-EFF’ section.

As the graphical controls on the various Parameter Adjust Windows are adjusted, the Matrix Mixer sends out MIDI messages to the associated audio mixing desk instructing it to adjust the corresponding parameter. As parameters are adjusted on the audio mixing desk, the audio mixing desk sends out MIDI messages to the Matrix Mixer which it uses to update the graphical controls that represent the specific parameters. This is depicted in Figure 8.

3.2.3. Summary Of Grid Mixing

The routing matrices of the Matrix Mixer display the signal processing points that are available within the audio mixing desk. The layout of these matrices mimic the signal flows as depicted in the block diagram of the device and so alleviate the need to study such a diagram to figure out the routing capabilities of the audio mixing desk. Sound engineers are able to use the routing grids to configure routing between the various signal source and destination points. Parameters that belong to signal processing points are logically accessed by selecting the required signal processing points on the routing matrices. This displays the parameters available at the specific points allowing them to be adjusted.

3.3. State Persistence

As the graphical controls on the Matrix Mixer are adjusted and as the Matrix Mixer receives MIDI messages from the associated audio mixing desk, the Matrix Mixer keeps track of the parameter changes. This allows the Matrix Mixer to emulate the associated audio mixing desk and allows for the state of each of the parameters of the associated audio mixing desk to be saved to a file. This allows the state of the Matrix Mixer and therefore the state of the associated audio mixing desk to be restored to a previous state. This functionality is further explained in Chapter 05.

3.4. Configuring An Audio Mixing Desk Via The Matrix Mixer

If we consider the example introduced in sections 2.1.2 and 2.2.3 but this time we set up the Yamaha 01V96 Digital Mixing Console using the Matrix Mixer a sound engineer would perform the following steps in order to achieve the required setup:

- Route the input signal from the first analogue input to the seventeenth input channel by:
    - Selecting the ‘Input Patch-1’ tab on the primary window.
    - Selecting the cross point between the ‘AD1’ audio source point label and the ‘CH17’ audio destination point label.
- Route the audio signal present on the fourth auxiliary bus to the first slot output by:
    - Selecting the ‘Output Patch’ tab on the primary window.
    - Selecting the cross point between the ‘AUX4’ audio source point label and the ‘SLOT1’ audio destination point label.
- Route the audio signal to the fourth auxiliary bus by:
    - Selecting the ‘01V96’ tab on the primary window.
    - Selecting the cross point between the ‘CH17’ audio source point label and the ‘AUX4’ audio destination point label.
    - Right-clicking on the same cross point to display the Parameter Adjust Window.
    - Raising the graphical fader of the auxiliary send to the desired level.
- Route the audio signal through the equaliser by:
    - Right-clicking the ‘CH17’ audio source point label on the primary window to display the Parameter Adjust Window for that particular signal processing point.
    - Turning the equaliser on by clicking the ‘EQ-ON’ button.
    - Adjusting the parameters of the equaliser by turning the potentiometers under the ‘Equaliser’ section until the desired sound is heard.
- Route the audio signal through the dynamics processor by:
    - Turning the compressor on by clicking the ‘COMP-ON’ button.
    - Adjusting the parameters of the compressor by turning the potentiometers under the ‘Comp – Compressor’ section until the desired sound is heard.

From the example used throughout the last two chapters, it can be seen that it has become progressively easier to manipulate the Yamaha 01V96 Digital Mixing Console. Given below is a set of tables that each give the number of actions that a sound engineer would have to perform in order to set up the Yamaha 01V96 Digital Mixing Console as described in the example.

Table 1 shows the number of actions that a sound engineer would have to perform in order to patch the first analogue input to the seventeenth input channel. Setting up the input patch via the control surface of the device involves navigating through sets of pages and components on the LCD display of the device. With the software audio mixing desk editors creating the input patch is done via a set of routing matrices. As can be seen, the use of the routing matrices reduces the number of steps needed to configure the input patch.

Yamaha 01V96 Digital Mixing Console control surface		**01V96 Editor control surface**	**Matrix Mixer control surface**
Button pushes	**Parameter wheel turns**	**Mouse clicks**	**Mouse clicks**
4	16	4	2

Table 1: Configuring an input patch within the Yamaha 01V96 Digital Mixing Console

Table 2 shows the number of actions that a sound engineer would have to perform in order to patch the signal from the fourth auxiliary bus to the first mini-YGDAI slot output. Via the control surface of the device, a sound engineer has to cycle through the signal sources on the LCD display until the required one is located. With the software device editors a sound engineer has to locate the correct routing matrix and select the cross point between the signal source and signal destination point. This requires fewer actions.

Yamaha 01V96 Digital Mixing Console control surface		**01V96 Editor control surface**	**Matrix Mixer control surface**
Button pushes	**Parameter wheel moves**	**Mouse clicks**	**Mouse clicks**
3	12	2	2

Table 2: Configuring an output patch within the Yamaha 01V96 Digital Mixing Console

Table 3 shows the number of actions that a sound engineer would have to perform in order to patch the signal from the seventeenth input channel through to the fourth auxiliary bus. Here, once again, the software audio mixing desk editors require fewer actions by the sound engineer. This is due to the fact that the controls for these parameters are immediately available from the primary windows of the device editors.

Yamaha 01V96 Digital Mixing Console control surface		**01V96 Editor control surface**		**Matrix Mixer control surface**	
Button pushes	**Faders moved**	**Mouse clicks**	**Fader moves**	**Mouse clicks**	**Fader moves**
4	1	2	1	3	1

Table 3: Patching through to an auxiliary bus on the Yamaha 01V96 Digital Mixing Console

Table 4 shows the number of actions a sound engineer would have to perform in order to set up the equaliser and dynamics processor as required in the example. Controlling the equaliser and dynamics processor via the control surface of the device requires more moves as there is a need to navigate to each of the required controls with the cursor buttons. Through the software audio mixing desk editors, the number of actions is reduced as each of the parameter controls is accessible with the mouse cursor.

Yamaha 01V96 Digital Mixing Console control surface		**01V96 Editor control surface**		**Matrix Mixer control surface**	
Button pushes	**Potentiometer moves**	**Mouse clicks**	**Potentiometer moves**	**Mouse clicks**	**Potentiometer moves**
14	*	4	*	3	*
* These will be the same for each approach if the same equalisation and dynamics parameters are adjusted by the same amounts					

Table 4: Adjusting the equaliser and dynamics processor of the seventeenth input channel

In the above example, both the 01V96 Editor and the Matrix Mixer offer similar values when comparing the number of actions a sound engineer would have to perform to set the device up in similar configurations. The primary window of the 01V96 Editor presents a lot of the functionality of its associated audio mixing desk simultaneously thus creating displays that are cluttered with many controls. The Matrix Mixer only displays signal processing points available within the associated audio mixing desk. This allows a sound engineer to display the required controls by selecting the required signal processing points. This creates a display that is not cluttered with different types of controls.

3.5. A Customisable Software Audio Mixing Control Surface

Modern audio mixing desks have continued to grow to the point where they can contain thousands of parameters. An audio mixing desk such as the Yamaha 01V96 Digital Mixing Console is a forty channel, eighteen bus audio mixing desk. Most of its channels have volume, pan, mute, attenuation, delay, equalisation and dynamics parameters available to them. This is over twenty five different parameters for each channel.

Some audio mixing desks have a dedicated control for each parameter (see Figure 2). This can lead to the control surface of the audio mixing desk becoming cluttered with vast arrays of various faders, potentiometers and buttons. This approach does, however, provide for easier access to the controls. Each channel strip contains the controls that are to be used with that particular channel. Some digital audio mixing desks have done away with having a dedicated control for each of their parameters in favour of assignable controls (see Figure 3). The function of the controls depends on the current mode that the audio mixing desk is in.

This presents two situations: Sound engineers are presented with a large number of controls - most of which only occasionally need adjusting - or they have to search to find the specific control they are looking for.

Software audio mixing desk editors like the 01V96 Editor and The Visualizer for 03D provide the user with a more traditional analogue audio mixing desk look-and-feel and as such their graphical displays become cluttered with controls. The Matrix Mixer’s approach to audio mixing presents sound engineers with the various internal signal processing points that are found within an audio mixing desk. This approach allows for simplified routing configurations to take place and for parameter adjustments to happen by selecting the required control points. While this approach gains from being able to instantaneously see the potential and current routing configurations of an audio mixing desk, it hides the parameters of the signal processing points away from sound engineers. Sound engineers have to select a signal processing point before they are able to see and adjust the parameters that are available at a specific point.

This problem was solved by implementing a customisable audio mixing desk control surface as part of the Matrix Mixer. This idea came from a hardware MIDI controller know as the Mawzer [Mawzer]. The Mawzer is a hardware MIDI controller that has a customisable control surface. The control surface is made up of a collection of pluggable modules that are plugged into the surface of the device allowing for the construction of a custom control surface. This allows for control surfaces to be built up based on the specific needs. The main unit of the Mawzer is shown in Figure 26. This figure shows the Mawzer without any modules plugged into it. The bottom panel of the main unit consists of a number of jack sockets into which the pluggable modules can be plugged.

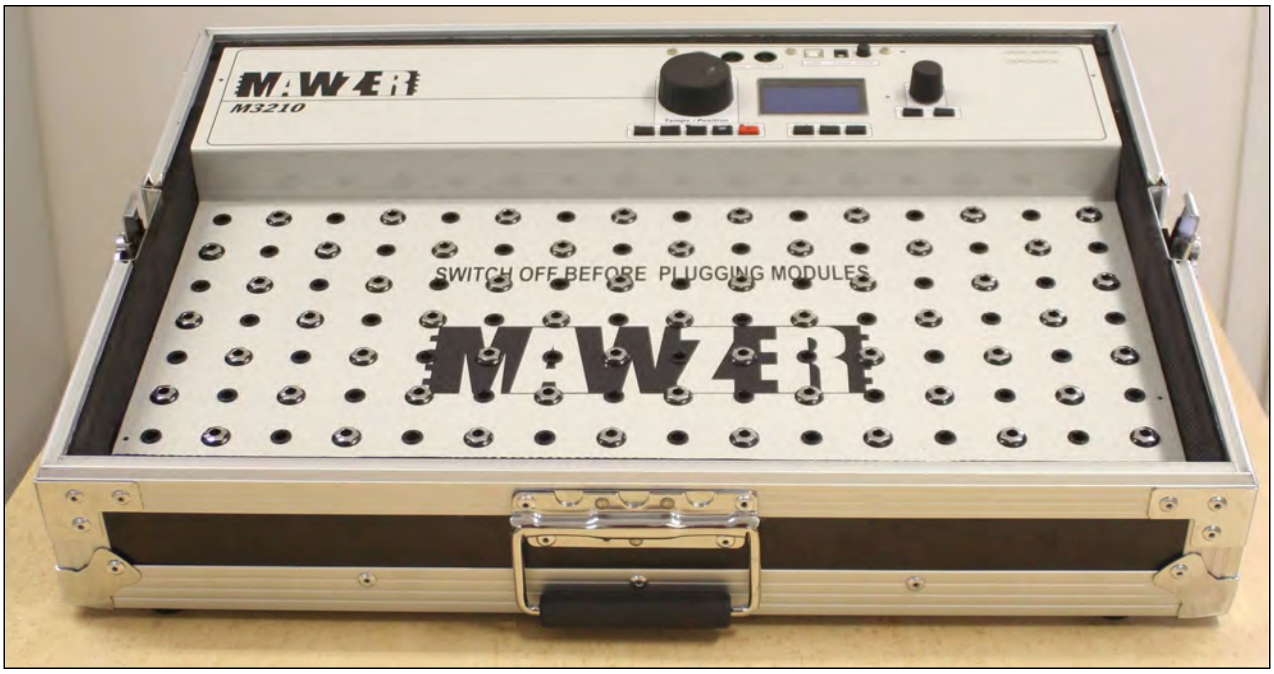


Figure 26: The Mawzer with an empty control surface

Figure 27 shows some of the pluggable modules that are available to be plugged into the customisable control surface of the Mawzer. From the side views of these modules, it is evident they have jack plugs on the back of them. This allows these modules to be added or removed from the surface of the device depending on the specific needs of the user. There are various different modules available which have different controls on them. These include modules that are made up of faders, potentiometers and push buttons.

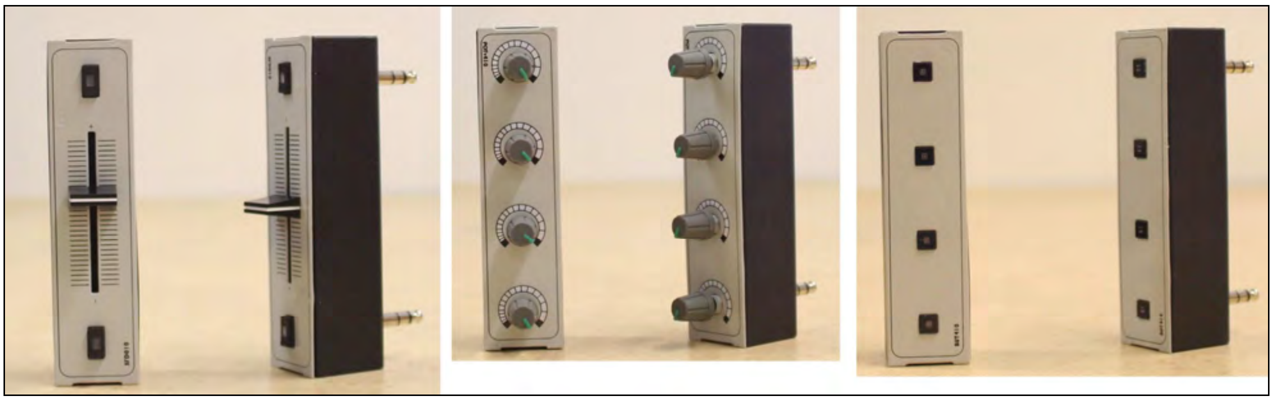


Figure 27: The Mawzer's pluggable modules

Following the Mawzer concept, a customisable control surface has been created for the Matrix Mixer. This idea brings to the software world what the Mawzer brings to the hardware world. The idea fuses the power and simplicity of the grid audio mixing approach provided by the Matrix Mixer with the traditional analogue audio mixing desk look-and-feel. This allows sound engineers to create control surfaces that suit their individual audio mixing needs. As such, there is no need for unused controls to be displayed.

The custom control surface that was implemented as part of the Matrix Mixer is shown on the right-hand-side of the routing grid in Figure 28. It is possible to build up a custom control surface by dragging controls from a Parameter Adjust Window (shown on the left-hand-side of Figure 28 over the routing matrix) onto the customisable custom control surface. The Matrix Mixer allows for its graphical faders, potentiometers and push buttons to be dragged to the custom control surface. This allows for controls that are frequently used to be viewed and adjusted quickly as they are part of the Matrix Mixer’s primary window. It also reduces the amount of control-clutter that is evident on the analogue style audio mixing desk editors.

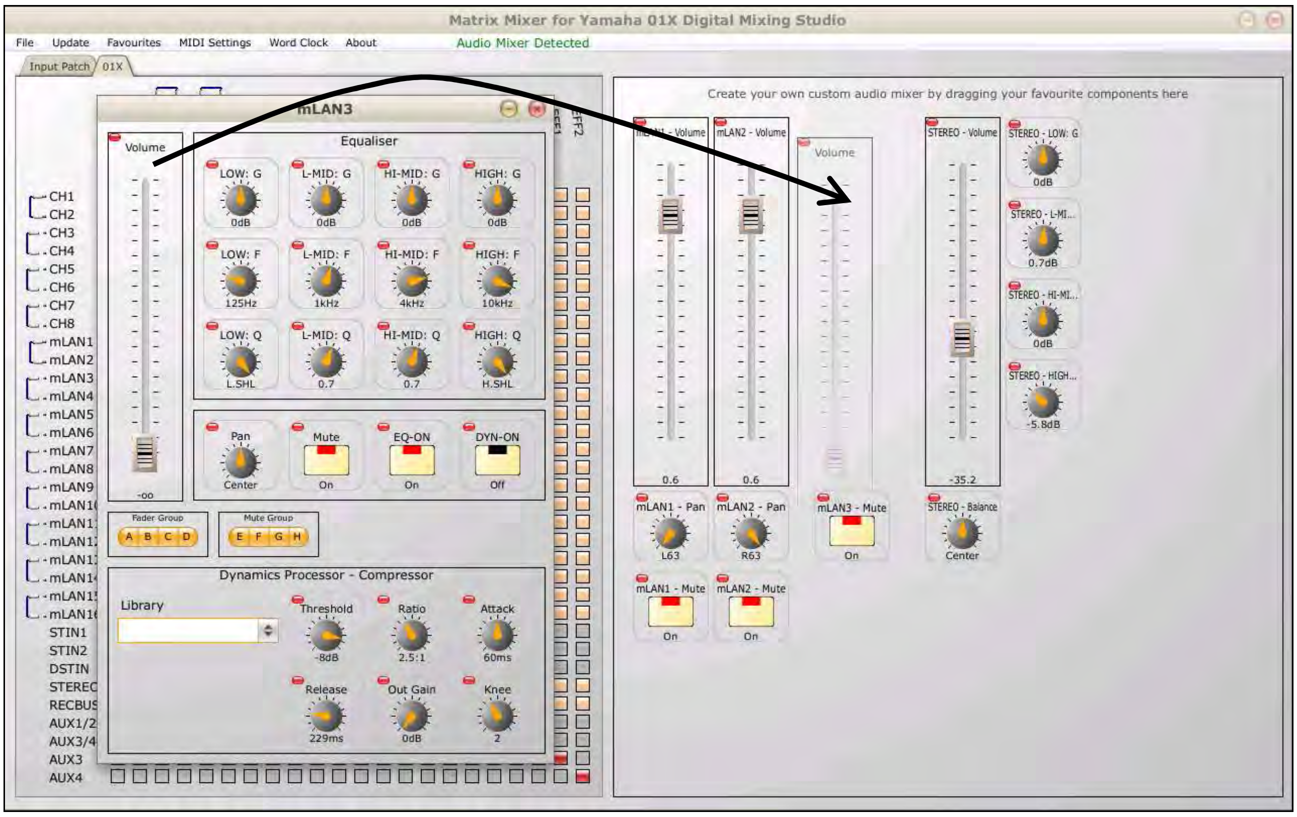

Matrix Mixer for Yamaha 01X Digital Mixing Studio
File
Update
Favourites
MIDI Settings
Word Clock
About
Audio Mixer Detected
Input Patch
01X
mLAN3
Volume
Equaliser
Dynamics Processor - Compressor
Library
Create your own custom audio mixer by dragging your favourite components here

Figure 28: The Matrix Mixer with its Custom Control Surface

In much the same way as adding controls to the custom control surface, controls can also be removed by simply dragging them off the surface and dropping them.

The Matrix Mixer allows for the current configuration of the customisable control surface to be saved to an external file. This file is an XML file which contains information allowing for the re-creation of the controls that are currently on display. This not only allows for custom control surfaces to be created, but also for them to be used again. This in turn permits the control surfaces to be used with other instances of the Matrix Mixer that represent the same audio mixing desk - a specific control surface is not limited to working with only one instance of the Matrix Mixer.

3.6. Summary

This chapter introduced the idea of software grid patchbays that are often used to patch audio signals between devices in digital audio networks. This idea has been used to successfully create a generic patchbay-based audio mixing editor that presents a sound engineer with all the signal processing points that exist within an audio mixing desk. This approach allows for routing configurations within audio mixing desks to be set up by selecting and de-selecting the cross points that exist on the routing matrices. It also allows for parameter adjustments to happen by selecting the signal processing points at which the required parameter exists.

The Matrix Mixer approach to mixing uses graphical routing matrices to display the signal processing points that are available within an audio mixing desk. On the block diagram for the Yamaha 01V96 Digital Mixing Console (see Figure 1) the audio signals flow from the left-hand-side to the right-hand-side of the diagram and at various points along the way get routed to various other points within the audio mixing desk. The Matrix Mixer mimics this signal flow with the way the routing grids are laid out and thus alleviates the need to study such diagrams.

Due to the control-clutter evident on analogue-style audio mixing desk editors as well the hidden nature of the controls on the Matrix Mixer, this chapter also proposed a customisable control surface that was implemented as part of the Matrix Mixer. This allows controls from Parameter Adjust Windows to be dragged onto the custom control surface in order to allow sound engineers to build up control surfaces that suit their individual audio mixing requirements.

Chapter 04

Describing Audio Mixing Desks With XML

There are a number of parameters that are common to most audio mixing desks. The Matrix Mixer is a generic grid MIDI audio mixing desk editor that was built and designed to represent and control these common audio mixing desk parameters. In order to facilitate its quick adaptation to multiple audio mixing desks, specific audio mixing desks are described using XML documents. The Matrix Mixer parses a specific XML document when it initialises itself, thus allowing it to build itself up to represent and control specific audio mixing desks.

In this chapter, we discuss how XML has been used as the configuration tool for the Matrix Mixer.

4.1. The Matrix Mixer Configuration Architecture

There are a number of parameters that are common to most audio mixing desks such as patching, volume, panning, muting, equalisation, dynamics and effects parameters. The Matrix Mixer was designed and developed to be a generic MIDI audio mixing desk editor. The intention at the outset was to develop a software audio mixing desk editor that was not tied down to work with a specific audio mixing desk. The Matrix Mixer was developed to represent and control these common audio mixing desk parameters. From the experience gained from using the Yamaha 03D Digital Mixing Console, the Yamaha 01X Digital Mixing Studio and the Yamaha 01V96 Digital Mixing Console these common parameters were defined as:

- Patching
- Volume and volume parameter grouping
- Muting and mute parameter grouping
- Panning
- Equalisation
- Dynamics processor parameters
- Channel pairing
- Effects processor parameters

There was a need to allow the Matrix Mixer to represent and control specific audio mixing desks quickly. It was decided that the Matrix Mixer should be configured to represent a specific audio mixing desk using an external configuration file. The Matrix Mixer parses a specific configuration file when it initialises itself.

From the configuration files the Matrix Mixer is able to determine:

- The various signal processing points (signal source, destination and patch points) that exist within an audio mixing desk.
- The specific parameters that are available at each of those signal processing points (for example, volume and equalisation parameters).
- The control messages that are used to communicate with the associated audio mixing desk.

The Matrix Mixer has been developed to work with the Yamaha audio mixing desks mentioned above. These audio mixing desks are remotely controlled via the MIDI protocol and so the Matrix Mixer has been developed to communicate using MIDI. The concept of configuring the Matrix Mixer via an external configuration file is not limited to MIDI controllable audio mixing desks. Audio mixing desks are fundamentally the same in terms of their functionality and architecture. Thus, this concept may be adapted to describe audio mixing desks that are controlled with protocols other than MIDI. The defined configuration file structure will remain fundamentally the same. It will just need to be adapted to describe the control messages used to communicate with non-MIDI controllable audio mixing desks.

4.2. XML

XML [Elliotte and Means, 2002] was chosen as the configuration file format for the Matrix Mixer. XML is a standard that is used to describe a generic syntax that is used to mark up data with simple, human-readable tags. All the data included in an XML document is represented as strings of text surrounded by tags that describe what the data is. The specific elements and tags that an XML document is made up of are not pre-defined. Users of XML are free to make up the tag and attribute names that suit their individual needs.

An XML document has a hierarchical structure and is described by its markup. Each document starts off with a root element. The root element may have any number of child elements associated with it, and each of those child elements may have further child elements associated with them, and so on. Each element may have any number of attributes associated with it as well. It is possible to see what elements are associated with other elements, and if the document is designed well, it is possible to determine the semantics of the document.

The following are some of the factors that drove the decision to use XML as the file format for the Matrix Mixer's configuration files:

- Its rapid growth and acceptance as a data-storage format.
- The availability of XML parsers.
- The ability to easily store and retrieve data from an XML document.
- The simplicity of its syntax and its readability.

The hierarchical structure of XML documents is ideal for describing audio mixing desks as they too have a hierarchical structure. Figure 29 shows the hierarchical structure of a simplified audio mixing desk. It has two inputs and an output. The inputs each have a volume controller and an equaliser. Each individual equaliser is made up of three equalisation bands. Each equalisation band has three equalisation parameters. The output has a volume controller associated with it.

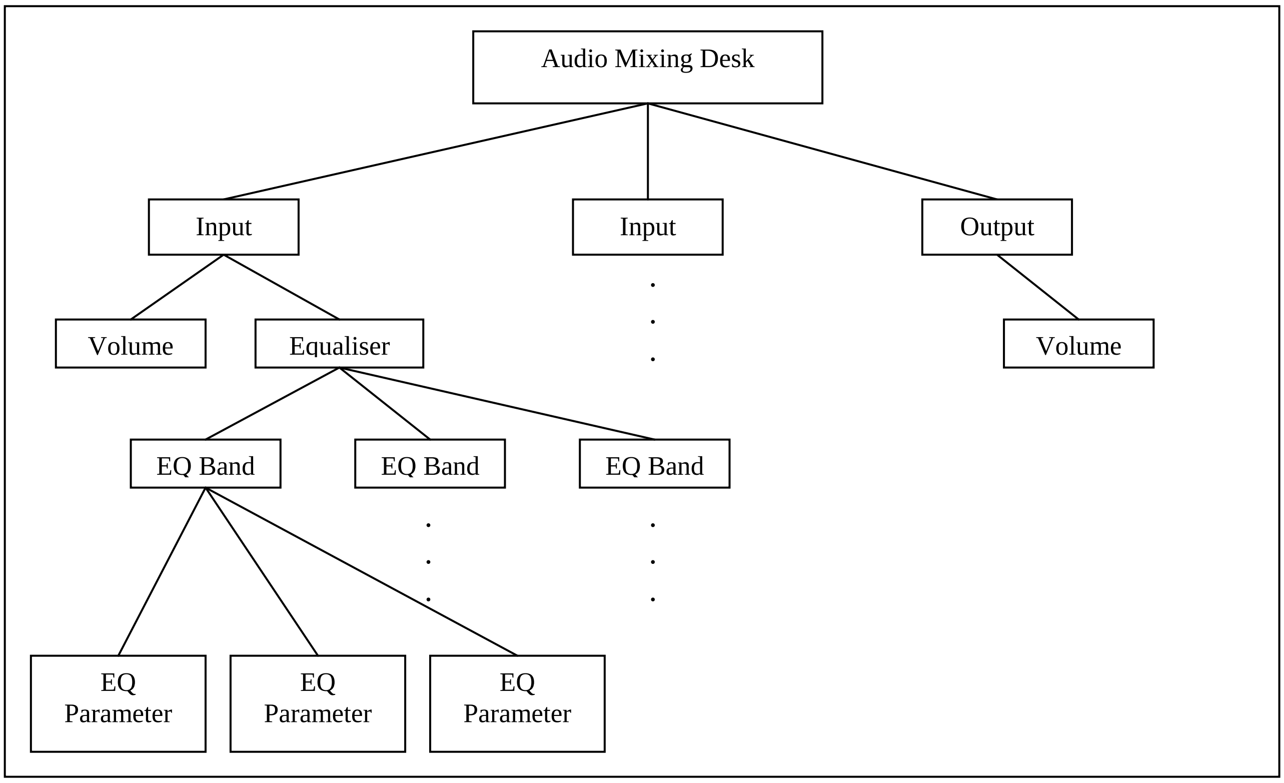


Figure 29: The typical hierarchical structure of an audio mixing desk

The structure shown in Figure 29 can be translated to a hierarchical XML document. Each of the individual components in the diagram can be described with an XML element. Properties associated with each of the components, such as the name of an input, can be described using an XML attribute. Listing 1 demonstrates how the structure in Figure 29 can be represented with an XML document. The root XML element of this XML document is `mixer`. It, along with all its child XML elements, describes the audio mixing desk. The `mixer` XML element has two child XML elements, `inputs` and `outputs`. These XML elements contain further child elements that describe the inputs and outputs of the audio mixing desk. Each `input` and `output` XML element has a `name` XML attribute. The value of this XML attribute is used to name the specific input or output. This pattern is continued until the hierarchical structure of the audio mixing desk has been described.

```
<mixer>
      <inputs>
            <input name=”CH1”>
                  <volumeParameter>
                  </volumeParameter>
                  <eq>
                        <eqBand>
                              <eqParameter>
                              </eqParameter>
                              ...
                        </eqBand>
                        ...
                  </eq>
            </input>
            <input name=”CH2”>
                  ...
            </input>
      </inputs>
      <outputs>
            <output name=”OUT”>
                  ...
            </output>
      </outputs>
</mixer>
```

Listing 1: A simplified XML audio mixing desk description

4.3. The Matrix Mixer Configuration File

During the development of the Matrix Mixer XML elements and attributes were defined to describe the common characteristics of audio mixing desks. Amongst others, XML elements and attributes have been defined to describe:

- The signal processing points (signal source, patch, and destination points) that exist within an audio mixing desk.
- The various parameters that may exist at each of these signal processing points (for example, volume and equalisation parameters).
- The MIDI messages used to control the parameters.

The core function of the Matrix Mixer's XML documents is to describe what features an audio mixing desk has. The Matrix Mixer determines how to represent and control the audio mixing desk described in a particular XML document.

Matrix Mixer XML documents start off with a root XML element called `mixer`. This element, along with all its child XML elements are used to describe an audio mixing desk for the Matrix Mixer. It has a single XML attribute, `name`. The value of `name` should be set to the name of the audio mixing desk being described by the XML document. The Matrix Mixer could, for example, use the value of the `name` attribute to name the primary window representing the audio mixing desk.

The `mixer` XML element may have a number of `select` child XML elements. Each `select` XML element is used to describe a routing grid. Visually, the Matrix Mixer represents each `select` XML element by adding a tab to its primary window. Each `select` XML element has an XML attribute called `name` that is used to name the specific routing grid. It is also possible to make one of the routing grids the default routing grid so that it is displayed when the Matrix Mixer is displayed. This is done by setting the specific `select` XML element's `default` XML attribute to `true`. Figure 30 shows:

- A Matrix Mixer XML document with two `select` XML elements.
- The Matrix Mixer's primary display after it has parsed the XML document.

```
<mixer name="01X">

      <select name="Input Patch">
      </select>

      <select name="01X" default="true">
      </select>

</mixer>
```

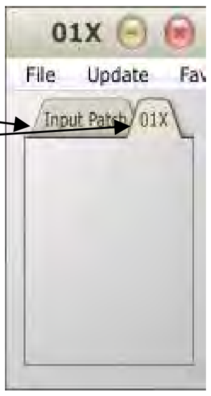


Figure 30: Representing the `select` XML element

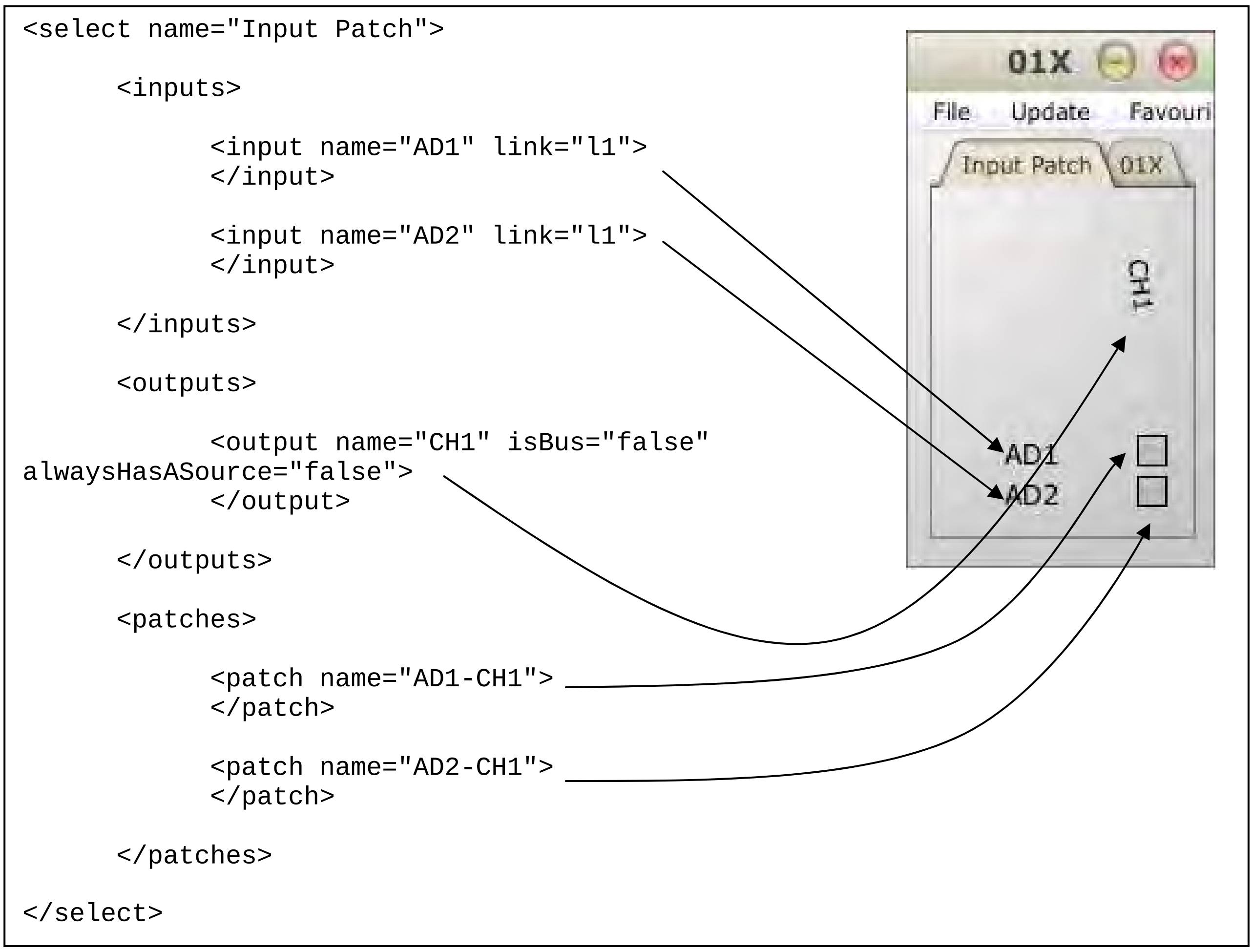
The Matrix Mixer for the Yamaha 01V96 Digital Mixing Console has eight different routing matrices. Each routing matrix represents a logical part of the routing process. For example, there are separate routing matrices to perform input patching, effects patching and output patching. In the

XML document that describes the Yamaha 01V96 Digital Mixing Console for the Matrix Mixer, each of these routing matrices is described with an individual `select` XML element.

4.3.1. Describing Signal Processing Points

Matrix Mixer XML documents may describe the signal processing points that exist within an audio mixing desk. These signal processing points are described using `input`, `output`, and `patch` XML elements. These elements are child XML elements of the `select` XML elements. This allows multiple routing grids to be described in the XML document, each with its own signal source, destination and patch points. Figure 31 shows:

- A portion of a Matrix Mixer XML document with `input`, `output` and `patch` XML elements.
- The Matrix Mixer's primary display after it has parsed the XML document.



```
<select name="Input Patch">

      <inputs>

            <input name="AD1" link="l1">
            </input>

            <input name="AD2" link="l1">
            </input>

      </inputs>

      <outputs>

            <output name="CH1" isBus="false"
alwaysHasASource="false">
            </output>

      </outputs>

      <patches>

            <patch name="AD1-CH1">
            </patch>

            <patch name="AD2-CH1">
            </patch>

      </patches>

</select>
```

Figure 31: Representing the `input`, `output` and `patch` XML elements

Each signal source point of a particular audio mixing desk is described using an `input` XML element. The Matrix Mixer represents each of these XML elements along the left-hand-side of the appropriate routing matrix. Each `input` XML element may have the following XML attributes:

- `name`: The value of the `name` XML attribute should be set to the name of the signal source point being described. In the figure above, the two signal processing points described above are two analogue inputs and have been named ‘AD1’ and ‘AD2’. It is the values of these XML attributes that are displayed on the left-hand-side of the appropriate routing matrix.
- `link`: The `link` XML attribute is used to link the specific signal processing point with other signal processing points. All signal processing points that have the same value for their `link` XML attribute are linked together. This allows for parameters available at different signal processing points to be shared. When a sound engineer wishes to display the Matrix Mixer’s Parameter Adjust Window by selecting a specific signal processing point, but no parameters have been defined for that specific point, the Matrix Mixer will display the parameters defined for a signal processing point that has parameters associated with it and has the same value for its `link` XML attribute.

Each signal destination point is described in the Matrix Mixer’s XML documents with an `output` XML element. The Matrix Mixer visually represents each `output` XML element along the top of the appropriate routing matrix, as seen in Figure 31. Each `output` XML element may have the following XML attributes:

- `name`: The value of the `name` XML attribute should be set to the name of the signal destination point being described. In the figure above, the signal processing point described is the first input channel of an audio mixing desk, and has been named ‘CH1’. It is the values of these XML attributes that are displayed on the top of the appropriate routing matrix.
- `link`: The `link` XML attribute is used to link the specific signal processing point with other signal processing points. Its functionality is that same as described for the `input` XML element.
- `isBus`: The value of the `isBus` XML attribute should be set to either `true` or `false`. If the signal destination point being described by the `output` XML element is an audio bus, this XML attribute’s value should be `true`. Otherwise it should be `false`. Audio busses may have many signal sources patched through to them simultaneously whereas an analogue output, for example, may at most have one signal source patched through to it. Setting this attribute appropriately ensures that the Matrix Mixer adheres to this rule.

- `alwaysHasASource`: This XML attribute has a value of either `true` or `false`. It specifies whether or not the specific signal destination point always has at least one source signal being patched through to it. If this XML attribute has a value of `true`, the Matrix Mixer will not allow the number of signal sources being patched through to this signal destination point to drop below one.

Each signal patch point is described within the Matrix Mixer's XML documents with a `patch` XML element. The Matrix Mixer visually represents the `patch` XML elements as cross points on the routing matrices of the Matrix Mixer. This can be seen in Figure 31 above. Each `patch` XML element has a `name` XML attribute that is used to name the patch point. The `patch` XML elements need to be ordered from the element that represents the first signal source being patched through to the first signal destination point to the last signal source being patched through to the last signal destination point.

4.3.2. Describing Channel Pairs

On some audio mixing desks it is possible to pair two adjacent odd and even mono channels together to work as a stereo pair. What this implies is that some of the parameters of the same type that belong to each of the paired channels will always reflect the same value. For example, if we paired input channel one and input channel two of an audio mixing desk together, and the fader of the first input channel is adjusted, the fader of the second input channel will be automatically and simultaneously adjusted as well. But, this may not be true for a pan parameter. A sound engineer may wish to send the signal present at the first input channel to the left stereo bus and the signal at the second input channel to right stereo bus in order to hear stereo sound. This would require that the pan parameters work independently of each other even though their parent channels are paired together.

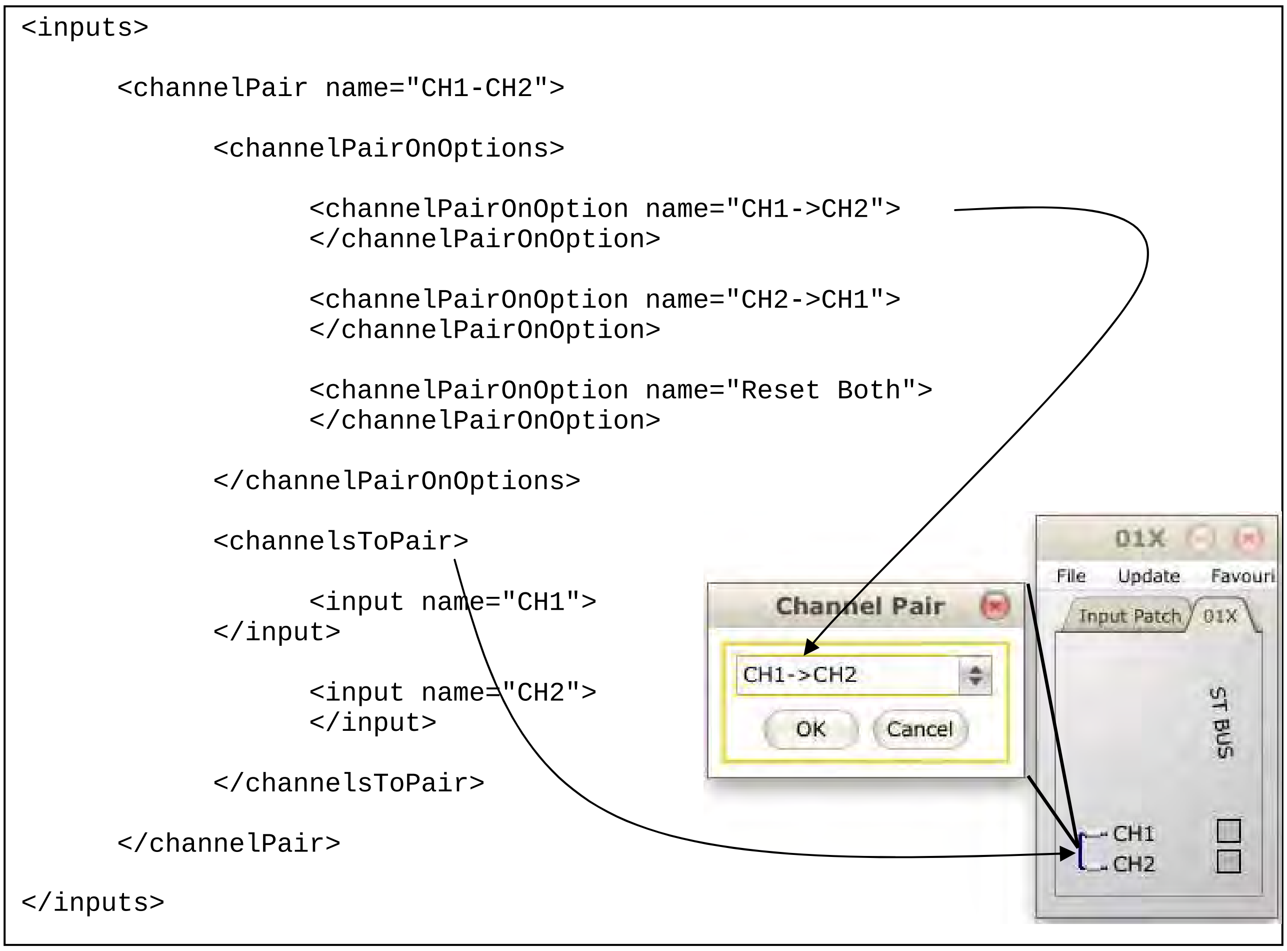
Before two channels are paired together, the corresponding parameters of each channel may have different values. When the channels are paired together, the corresponding parameters that work together need to have their values synchronised. There may be a number of options available to the sound engineer as to how the synchronisation takes place. For example, it could be possible to:

- Copy the parameter values from the odd channel across to the even channel.
- Copy the parameter values from the even channel across to the odd channel.

- Reset both channels parameters to their default values.

The `channelPair` XML element is used to specify that two adjacent odd and even channels may be paired together. The child XML elements of the `channelPair` XML element are used to specify the pairing options available to the sound engineer and the channels that may be paired together. Figure 32 shows:

- A portion of a Matrix Mixer XML document that describes a channel pair.
- The corresponding Matrix Mixer after it has parsed the XML document.



```
<inputs>

      <channelPair name="CH1-CH2">

            <channelPairOnOptions>

                  <channelPairOnOption name="CH1->CH2">
                  </channelPairOnOption>

                  <channelPairOnOption name="CH2->CH1">
                  </channelPairOnOption>

                  <channelPairOnOption name="Reset Both">
                  </channelPairOnOption>

            </channelPairOnOptions>

            <channelsToPair>

                  <input name="CH1">
            </input>

                  <input name="CH2">
                  </input>

            </channelsToPair>

      </channelPair>

</inputs>
```

Figure 32: Representing `channelPair` XML elements

The `channelPair` XML element may be a child XML element to the `inputs` and `outputs` XML elements. It contains child XML elements that describe:

- The channels that may be paired. These channels are described using their usual `input` or `output` XML elements and are child XML elements to a `channelsToPair` element. The `channelsToPair` must have exactly two of the appropriate child elements. On the Matrix

Mixer, channels that may be paired are shown as having a cable like connector between their labels, as seen in Figure 32.

- The channel pair options available to a sound engineer. These are listed with the `channelPairOnOption` XML elements and are named with the `name` XML attribute. When two channels are unpaired, and a sound engineer selects the channel pair button, the channel pair options available to the sound engineer are displayed in a window. The sound engineer selects an appropriate channel pair option and clicks the 'OK' button and the associated audio mixing desk is instructed to pair the two channels.

In the above portion of XML, no control messages have yet been defined. Each of the channel pair options will have control messages associated with them and these are described in the Matrix Mixer XML documents (See section 4.3.3.2).

4.3.3. Describing Signal Processing Point Parameters

The `select`, `channelPair`, `input`, `output` and `patch` XML elements allow the routing matrices of the Matrix Mixer to be set up. At this point, none of the parameters available at the signal processing points have been defined. Only the signal processing points that exist within the audio mixing desk have been defined.

Each signal processing point (signal source, destination and patch points) may have various parameters associated with them. For example, the first input channel of the audio mixing desk may have volume and equalisation parameters associated with it. Each `input`, `output` and `patch` XML element may have a `parameters` XML element as a child element. The `parameters` XML element is used to describe the parameters associated with a specific signal processing point. Its individual child XML elements are used to describe the different parameters that the Matrix Mixer is able to represent. Omitting a `parameters` XML element instructs the Matrix Mixer that no parameters exist at the specific signal processing point.

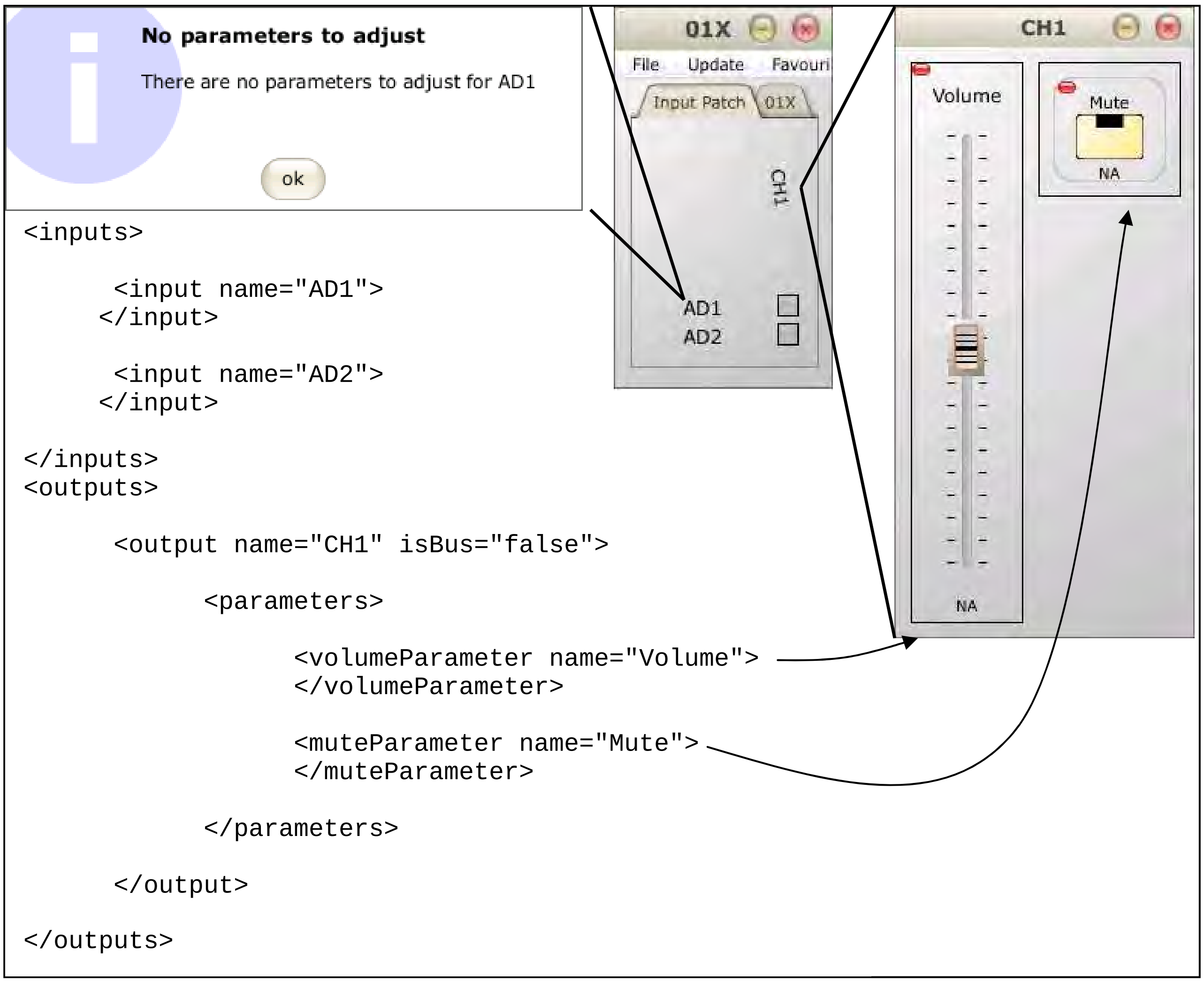
Essentially, there are two types of parameters: continuous and switch parameters. Continuous parameters are those that are represented with linear and rotary potentiometers, such as volume and pan parameters. Switch parameters have two states and are usually represented with buttons. These types of parameters include mute and bypass parameters.

In this section, we will look at how to describe parameters in the Matrix Mixer XML documents. A volume and mute parameter will be used as an example, unless otherwise stated. The principles shown here may be applied to all the other parameters that the Matrix Mixer is capable of representing.

A volume parameter is described with a `volumeParameter` XML element, and a mute parameter is described with a `muteParameter` XML element. In order to describe these parameters at a specific signal processing point, their corresponding XML elements are added as a child element to the `parameters` XML element.

Figure 33 shows:

- A portion of a Matrix Mixer XML document with no parameters defined for the signal source points, and a volume and mute parameter defined for the signal destination point (‘CH1’).
- The corresponding Matrix Mixer once it has parsed the XML document. Also shown are the windows that are displayed when the various signal processing points are selected.



```
<inputs>

      <input name="AD1">
     </input>

      <input name="AD2">
     </input>

</inputs>
<outputs>

      <output name="CH1" isBus="false">

            <parameters>

                  <volumeParameter name="Volume">
                  </volumeParameter>

                  <muteParameter name="Mute">
                  </muteParameter>

            </parameters>

      </output>

</outputs>
```

Figure 33: Representing `volumeParameter` and `muteParameter` XML elements

The `volumeParameter` XML element instructs the Matrix Mixer that a volume parameter exists at the specific signal processing point, and a `muteParameter` XML element instructs it that a mute parameter exists at the specific signal processing point. In the above diagram, when a sound engineer selects the ‘CH1’ signal destination point, a volume fader and mute button are displayed on a Parameter Adjust Window. These are defined as existing at that point in the Matrix Mixer XML document. At this point, none of the specific properties associated with these parameters (for example, the MIDI messages used to control the parameters, and the values that the parameter may take on) have been described, merely their existence.

As no parameters have been described for the signal source points (‘AD1’ and ‘AD2’) in the above diagram, selecting one of these signal processing points displays a window specifying this.

Similarly, there are XML elements to describe the following parameters in Matrix Mixer XML documents:

- Pan parameters
- Equalisation parameters
- Dynamics processor parameters
- Fader and mute group parameters
- Effects processor parameters
- Patch parameters – these are available at patch signal processing points only

The presence of the specific XML elements used to describe the parameters instructs the Matrix Mixer that the parameter exists at the specific signal processing point, as shown with the volume and mute parameter in Figure 33 above. Thus, the Matrix Mixer makes these parameters graphically available to a sound engineer. Each specific parameter has a pre-defined way that it is graphically represented. For example, a volume parameter is represented with a linear fader, and the mute parameter with a push button.

Each of the specific XML elements that represent the above mentioned parameters may have the following XML attributes:

- `name`: The `name` XML attribute allows a custom name to be associated with the specific parameter (such as 'volume', or 'aux send') which is displayed on the Parameter Adjust Window. This can be seen in Figure 33 above: The value of the `name` XML attribute for the `volumeParameter` XML element is set to 'Volume'. This value ('Volume') is displayed above the volume fader on the Parameter Adjust Window.
- `worksWithPairedChannel`: The `worksWithPairedChannel` XML attribute has a value of either `true` or `false`. If this XML attribute is set to `true`, it instructs the Matrix Mixer that the specific parameter should work simultaneously with the parameter of the same type that belongs to the paired channel.

Each of the specific parameter XML elements (for example, `volumeParameter` and `muteParameter`) may have a number of child XML elements that describe the properties of the specific parameter. XML elements have been defined to describe:

- The parameter change MIDI messages used to control the parameter.
- The parameter request MIDI message(s) used to request the value of the parameter.
- A description of the parameter.

- The display values that the parameter may take on.

We will discuss how the each of the above mentioned items are described within Matrix Mixer XML documents, but first we will discuss how MIDI messages are described in general.

4.3.3.1. Describing MIDI Messages

Within Matrix Mixer XML documents, it is possible to describe MIDI messages at various parts of the document. These may be described either as a single MIDI message, a collection of single MIDI messages, or a range of MIDI messages. Typically, single MIDI messages will be associated with switch parameters (such as a mute parameter), and a range of MIDI messages will be associated with a continuous parameter (such as a volume parameter). For example, a mute parameter has two states. It is either on or off. Each one of these states may be represented with a single MIDI message. A volume parameter may have 128 unique values that it may be set to. Each of those unique values may be represented with a unique MIDI message within a continuous range of MIDI messages.

The `midiMessages` XML element is used to describe the MIDI messages in Matrix Mixer XML documents. The nature of the MIDI messages described with this XML element may be one of two types, specified by the value of the `type` XML attribute. This XML attribute may either have a value of `single` or `range`. When this XML attribute is set to `single`, it instructs the Matrix Mixer that the `midiMessages` XML element is describing a single MIDI message or a collection of single MIDI messages. When the value of the `type` XML attribute is set to `range`, it instructs the Matrix Mixer that the `midiMessages` element is describing a range of MIDI messages.

4.3.3.1.1. Describing Single MIDI Messages

Listing 2 shows an example of a description of a single MIDI message. In this instance, the value of the `midiMessages type` XML attribute is set to `single`. The single MIDI message is described with the `value` XML attribute of the `midiMessage` XML element. The actual MIDI message being described (in this case a system exclusive MIDI message) is `0xF0 0x43 0x10 0x3E 0x7F 0x01 0x25 0x08 0x00 0x00 0x00 0x00 0x01 0xF7`.

```
<midiMessages type="single">
     <midiMessage value="0xF0 0x43 0x10 0x3E 0x7F 0x01 0x25 0x08
0x00 0x00 0x00 0x00 0x01 0xF7"/>
</midiMessages>
```

Listing 2: Describing a single MIDI message

Listing 3 shows an example of a `midiMessages` XML element that is used to describe more than one single MIDI message in a Matrix Mixer XML document. The value of its `type` XML attribute is set to `single`, as in the previous example. Each individual single MIDI message of the collection of MIDI messages is described by the `value` XML attribute of a `midiMessage` XML element. The individual `midiMessage` XML elements collectively describe the collection of MIDI messages.

```
<midiMessages type="single">

     <midiMessage value="0xF0 0x43 0x10 0x3E 0x0E 0x02 0x0D 0x00
0x00 0x00 0x00 0x00 0x0B 0xF7"/>

     <midiMessage value="0xF0 0x43 0x10 0x3E 0x0E 0x02 0x0D 0x00
0x01 0x00 0x00 0x00 0x0C 0xF7"/>

</midiMessages>
```

Listing 3: Describing more than one single MIDI message

4.3.3.1.2. Describing A Range Of MIDI Messages

Listing 4 shows how a range of MIDI messages may be described in Matrix Mixer XML documents. In this instance, the value of the `midiMessages type` XML attribute is set to `range`. When describing a range of MIDI messages, they are ordered from the MIDI message that sets a parameter to its lowest value, to the MIDI message that sets the parameter to its highest value.

```
<midiMessages type="range">
      <midiMessagesStartPart value="0xF0 0x43 0x10 0x3E 0x7F 0x01 0x1B 0x00
0x00 0x00 0x00"/>
      <midiMessagesVariablePart from="0x00 0x00" to="0x03 0x7F"/>
      <midiMessagesEndPart value="0xF7"/>
</midiMessages>
```

Listing 4: Describing a range of MIDI messages

When describing a range of MIDI messages, the `midiMessages` XML element consists of child XML elements that describe the different parts of the range of MIDI messages:

- `midiMessagesStartPart`
- `midiMessagesVariablePart`
- `midiMessagesEndPart`

These collectively describe the different parts that the MIDI messages are composed of:

- `midiMessagesStartPart`: When the `midiMessages` XML element is being used to describe a range of MIDI messages, its first child XML element has to be `midiMessagesStartPart`. The `value` XML attribute of this XML element is used to describe the initial part of a MIDI message that remains constant over the entire range of MIDI messages. The set of bytes in this part of the MIDI message may define things such as the type of MIDI message (in this example it is a system exclusive MIDI message), the manufacturer ID, and the specific parameter that the MIDI message belongs to. In the listing above, the initial bytes of the range of MIDI messages are `0xF0 0x43 0x10 0x3E 0x7F 0x01 0x1B 0x00 0x00 0x00 0x00`.
- `midiMessagesVariablePart`: Following the `midiMessagesStartPart` XML element are one or more `midiMessagesVariablePart` XML elements. These XML elements collectively describe the part of the MIDI messages that represents the unique values of a parameter. Each of these XML elements' `from` XML attribute describes the first set of bytes in the range of bytes, and its `to` XML attribute describes the last set of bytes in the range. It is assumed that the bytes increment or decrement by one each time a parameter is increased or decreased by one respectively. In Listing 4 above the range of bytes starts at `0x00 0x00` and ends at `0x03 0x7F`.

In some instances it is necessary to describe a range of bytes with a break in continuity. This may be described with the aid of further `midiMessagesVariablePart` XML elements. Each `midiMessagesVariablePart` is used to describe a sub range of bytes in the range of MIDI messages. Listing 5 shows an example of this. In this range of MIDI messages, the first variable part byte will be `0x2C`, the second `0x00`, the third `0x01` and so forth.

```
<midiMessagesVariablePart from="0x2C" to="0x2C"/>
<midiMessagesVariablePart from="0x00" to="0x29"/>
```

Listing 5: Describing a non-continuous range of MIDI messages

- The `midiMessagesEndPart`: This XML element optionally follows the `midiMessagesVariablePart` element(s). The `value` XML attribute of this XML element is used to describe the last part that a MIDI message may be made up of. This part of the MIDI message remains constant across the entire range of possible MIDI messages being described. In this example, the `midiMessagesEndPart value` XML attribute is set to `0xF7`, denoting the end of a system exclusive MIDI message. If the controls of an audio mixing desk were being controlled with MIDI control messages, where the size of the MIDI messages are known, the `midiMessagesEndPart` is omitted as there is no byte to indicate the end of the MIDI message.

From the information contained within these elements, the Matrix Mixer is able to construct MIDI messages as and when needed. If, for example, the Matrix Mixer required the first MIDI message from the range of MIDI messages described in Listing 4, it will create it from the different parts that were described:

- It will calculate the required bytes from the variable part of the MIDI message. In this instance, it is the first set of bytes, so the required bytes are `0x00 0x00`.
- The value calculated above is then concatenated together with the start and end parts of the MIDI message.
  - `0xF0 0x43 0x10 0x3E 0x7F 0x01 0x1B 0x00 0x00 0x00 0x00` is concatenated together with `0x00 0x00` which is concatenated together with `0xF7`.
- The resulting MIDI message is `0xF0 0x43 0x10 0x3E 0x7F 0x01 0x1B 0x00 0x00 0x00 0x00` **`0x00 0x00`** `0xF7`.

If, for example, the Matrix Mixer now required the third MIDI message from the range of MIDI messages, it would:

- Calculate the required bytes from the variable part of the MIDI message. In this instance, the bytes are incremented by three, and the resulting bytes are `0x00 0x02`.
- Concatenate the value calculated above with the start and end parts of the MIDI message.
  - `0xF0 0x43 0x10 0x3E 0x7F 0x01 0x1B 0x00 0x00 0x00 0x00` is concatenated together with `0x00 0x02` which is concatenated together with `0xF7`.
- Use the resulting MIDI message, which is `0xF0 0x43 0x10 0x3E 0x7F 0x01 0x1B 0x00 0x00 0x00 0x00` **`0x00 0x02`** `0xF7`.

4.3.3.1.3. Describing Groups Of MIDI Messages

When some parameters are adjusted on an audio mixing desk, for each individual adjustment of those parameters two or more MIDI messages are transmitted by the audio mixing desk. This may be true of parameters that represent stereo channels for instance. On the Yamaha 01V96 Digital Mixing Console, each time the volume fader of the stereo output channel is raised by one, it sends out two MIDI messages to represent the state of the channel. In Matrix Mixer XML documents, it is possible to group MIDI messages together. This allows the Matrix Mixer to send two or more MIDI messages to its associated audio mixing desk each time a parameter is adjusted by one increment. Listing 6 shows how the `midiMessages` XML elements may be grouped together with `midiMessagesGroup` XML elements. In this example, the `midiMessagesGroup` XML element contains two `midiMessages` XML elements which are of type `range`. When the Matrix Mixer requires the first MIDI message within a group, it will, for each `midiMessages` XML element generate the first MIDI message of each range.

```
<midiMessagesGroups>
      <midiMessagesGroup>
            <midiMessages type="range">
                  ...
            </midiMessages>

            <midiMessages type="range">
                  ...
            </midiMessages>
      </midiMessagesGroup>
</midiMessagesGroups>
```

Listing 6: Describing groups of MIDI messages

4.3.3.2. Describing The Parameter Change MIDI Messages

Each parameter of an audio mixing desk may have a number of parameter change MIDI messages associated with it. These allow for the remote control of the specific parameter. These MIDI messages are sent to an audio mixing desk when the graphical controls representing the parameters are adjusted in order to instruct the audio mixing desk to adjust the represented parameters. The same parameter change MIDI messages are transmitted in the opposite direction as well. When parameters are adjusted via the control surface of the audio mixing desk, the audio mixing desk sends out parameter change MIDI messages. The Matrix Mixer is able to receive these and use them to update its graphical representations of the parameters.

The parameter change MIDI messages associated with each parameter may be described in the Matrix Mixer XML documents. These MIDI messages:

- Are sent to the associated audio mixing desk when the graphical controls on the Matrix Mixer are adjusted.
- Are used to match up incoming parameter change MIDI messages with their respective parameters.

Listing 7 shows an example `parameterChange` XML element that is used to describe the parameter change MIDI messages associated with a continuous parameter. Each specific continuous parameter XML element (such as `volumeParameter`) may have a `parameterChange` XML element as a direct XML child element. The child XML elements to each `parameterChange` XML element are used to describe the actual MIDI messages.

```
<volumeParameter name="Volume">
      <parameterChange>
            <midiMessagesGroups>
                  <midiMessagesGroup>
                        <midiMessages type="range">
                              <midiMessagesStartPart value="0xF0
0x43 0x10 0x3E 0x7F 0x01 0x1B 0x00 0x00 0x00 0x00"/>
                              <midiMessagesVariablePart from="0x00 0x00"
to="0x03 0x7F"/>
                              <midiMessagesEndPart value="0xF7"/>
                        </midiMessages>
                  </midiMessagesGroup>
            </midiMessagesGroups>
      </parameterChange>
</volumeParameter>
```

Listing 7: An example `parameterChange` XML element used for a continuous parameter

Listing 8 shows an example of an `on` and `off` XML element that is used to describe the parameter change MIDI messages associated with a switch parameter. Each specific switch parameter XML element (such as `muteParameter`) may have an `on` and `off` XML element as direct XML child elements. The `on` XML element is used to describe the parameter change MIDI messages associated with the specific parameter when it is turned on. The `off` XML element is used to describe the parameter change MIDI messages associated with the specific parameter when it is turned off. The child XML elements to the `on` and `off` XML elements are used to describe the actual MIDI messages.

```
<on>
     <parameterChange>
           <midiMessagesGroups>
                <midiMessagesGroup>
                    <midiMessages type="single">
                         <midiMessage value="0xF0 0x43 0x10 0x3E 0x7F 0x01
0x25 0x08 0x01 0x00 0x00 0x00 0x01 0xF7"/>
                    </midiMessages>
               </midiMessagesGroup>
          </midiMessagesGroups>
     </parameterChange>
</on>
<off>
     <parameterChange>
           <midiMessagesGroups>
                <midiMessagesGroup>
                     <midiMessages type="single">
                          <midiMessage value="0xF0 0x43 0x10 0x3E 0x7F 0x01
0x25 0x08 0x01 0x00 0x00 0x00 0x00 0xF7"/>
                     </midiMessages>
      </midiMessagesGroup>
            </midiMessagesGroups>
     </parameterChange>
</off>
```

Listing 8: Example `on` and `off` XML elements used for a switch parameter

In this example, the actual parameter change MIDI messages being described are single MIDI messages. It is also possible to associate a range of MIDI messages with the `on` and `off` XML elements. If a range of MIDI messages is associated with the `on` and `off` XML elements, and the parameter that is associated with those MIDI messages is turned either on or off, each MIDI message that is part of the related range of MIDI messages is generated by the Matrix Mixer and sent across to the associated audio mixing desk. Usually, there will only be a few MIDI messages in the range.

Figure 34 shows an example of how the Matrix Mixer uses the parameter change MIDI messages described in Matrix Mixer XML documents to communicate with its associated audio mixing desk. Internally, signal processing point and parameter objects are assigned unique ID’s, and the graphical controls that represent the parameters are assigned the ID of the parameter they represent.

Every time a parameter is adjusted on the Matrix Mixer (annotation 1 in the figure), the parameter object with the same ID as the graphical control is located and the parameter change MIDI message that represents the new value of that parameter is generated by the Matrix Mixer (annotation 2 in the figure). The generated parameter change MIDI message is sent across to the associated audio mixing desk and the parameter is adjusted (annotation 3 in the figure).

When a parameter is adjusted via the control surface of the audio mixing desk (annotation A in the figure), it sends out parameter change MIDI messages representing the new values of the adjusted parameter (annotation B in the figure). The Matrix Mixer receives these MIDI messages and is able to use them to match them up to the parameter change MIDI messages described in the Matrix Mixer XML documents (annotation C in the figure). Once the corresponding parameter object is located, the graphical control with the same ID as the object that was located is adjusted to represent the new value (annotation D in the diagram).

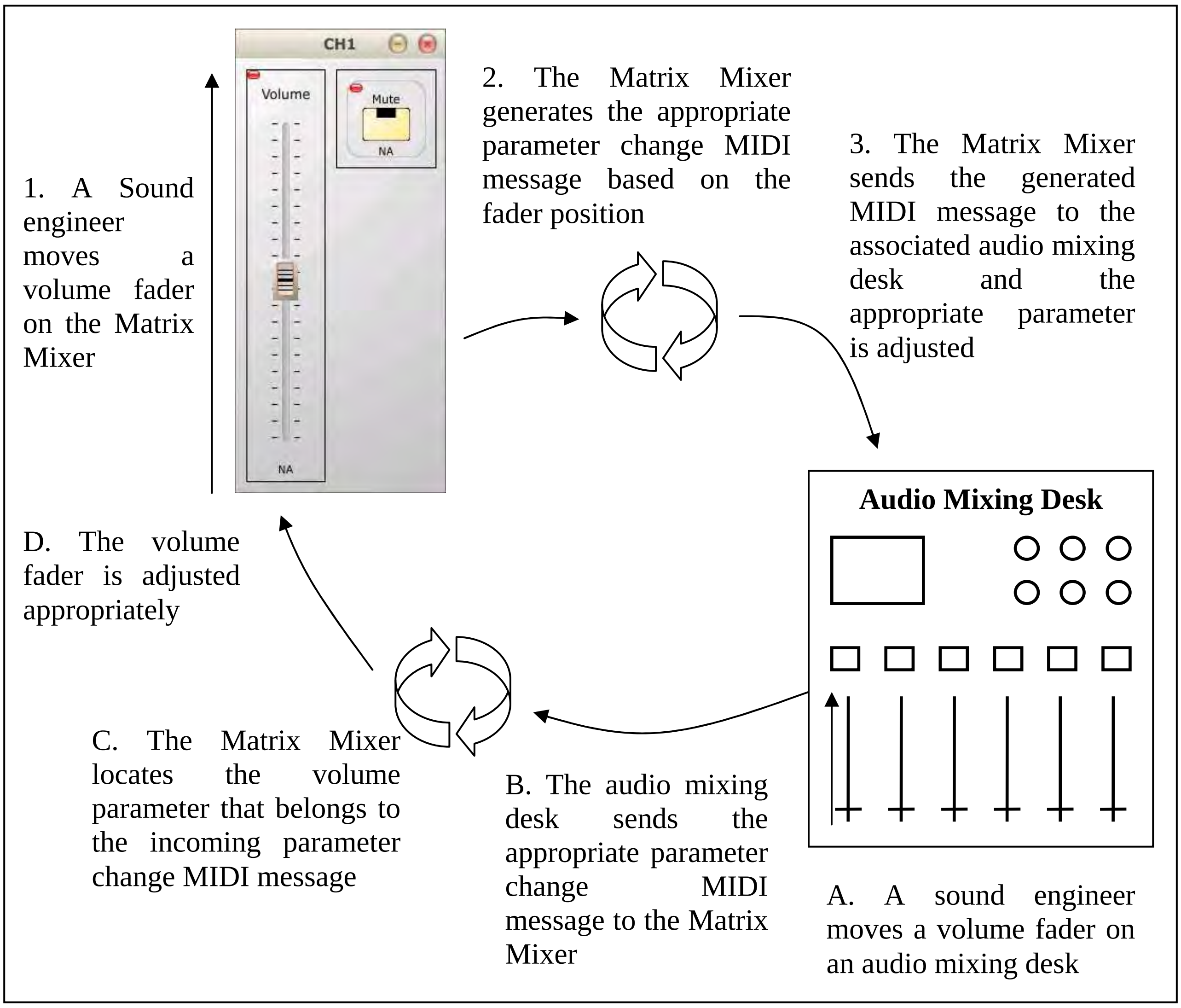

CH1
Volume
Mute
NA
NA
1. A Sound engineer moves a volume fader on the Matrix Mixer
2. The Matrix Mixer generates the appropriate parameter change MIDI message based on the fader position
3. The Matrix Mixer sends the generated MIDI message to the associated audio mixing desk and the appropriate parameter is adjusted
Audio Mixing Desk
A. A sound engineer moves a volume fader on an audio mixing desk
B. The audio mixing desk sends the appropriate parameter change MIDI message to the Matrix Mixer
C. The Matrix Mixer locates the volume parameter that belongs to the incoming parameter change MIDI message
D. The volume fader is adjusted appropriately

Figure 34: The Matrix Mixer's use of parameter change MIDI messages

4.3.3.3. Describing The Parameter Request MIDI Messages

Parameter request MIDI messages are used to request the values of parameters: Each parameter may have a set of parameter request MIDI messages associated with it. These MIDI messages are used to request the value of a specific parameter from the audio mixing desk. Sending a parameter request MIDI message to the audio mixing desk results in the audio mixing desk responding with a parameter response MIDI message. From the parameter response MIDI message, the Matrix Mixer is able to determine the value of the specific parameter and set it appropriately. Initially when the Matrix Mixer starts up all its parameters are set to their lowest values. In order to get the Matrix Mixer in same state as the audio mixing desk it is representing, it sends out the parameter request MIDI messages associated with each of its parameters. The audio mixing desk responds to each of the parameter request MIDI messages with parameter response MIDI messages which the Matrix Mixer uses in order to synchronise its state with the represented audio mixing desk.

In order to describe the parameter request MIDI messages for a particular parameter, a `parameterRequest` XML element is added as a child XML element to the parameter XML element. The child XML elements to the `parameterRequest` XML element describe the actual parameter request MIDI messages. Listing 9 shows an example `parameterRequest` XML element.

```
<volumeParameter name="Volume">
      <parameterRequest>
            <midiMessagesGroups>
                  <midiMessagesGroup>
                        <midiMessages type="single">
                              <midiMessage value="0xF0 0x43 0x30 0x3E 0x7F
0x01 0x25 0x00 0x00 0xF7"/>
                        </midiMessages>
                  </midiMessagesGroup>
            </midiMessagesGroups>
      </parameterRequest>
</volumeParameter>
```

Listing 9: An example parameterRequest XML element

In the example above, the actual MIDI message being described is a single MIDI message. It is also possible to associate a range of MIDI messages with a `parameterRequest` XML element. If, for example, a range of MIDI messages is described for a particular parameter's parameter request MIDI messages, and the Matrix Mixer wishes to send the parameter request MIDI messages associated with that particular parameter across to the associated audio mixing desk, it will generate all the possible MIDI messages in the range. Typically, there will only be a few MIDI messages in the range.

4.3.3.4. Describing A Parameter Description

Each parameter may have a textual description that is used to describe the functionality of the parameter. This description is described with the `value` XML attribute of the `parameterHelpString` XML element. In order to associate a description with a specific parameter, a `parameterHelpString` XML element is added as a child XML element to the parameter's XML element. The Matrix Mixer uses this value to display the description of the parameter when the mouse cursor is hovered over the graphical control that represents the parameter.

Figure 35 shows:

- A portion of a Matrix Mixer XML document showing how a description of a parameter is described for a particular parameter.
- A Matrix Mixer Parameter Adjust Window with the description of the parameter being displayed once the Matrix Mixer has parsed the XML document.

```
<input name="CH1">
      <parameters>

            <volumeParameter name="Volume">
                  <parameterHelpString value="Input level."/>
            </volumeParameter>

            <muteParameter name="Mute">
            </muteParameter>

      </parameters>
</input>
```

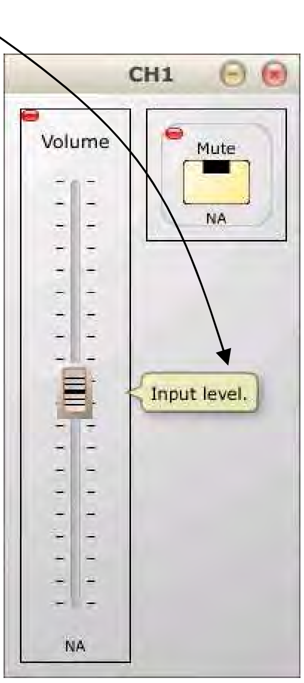


Figure 35: Representing a `parameterHelpString` XML element

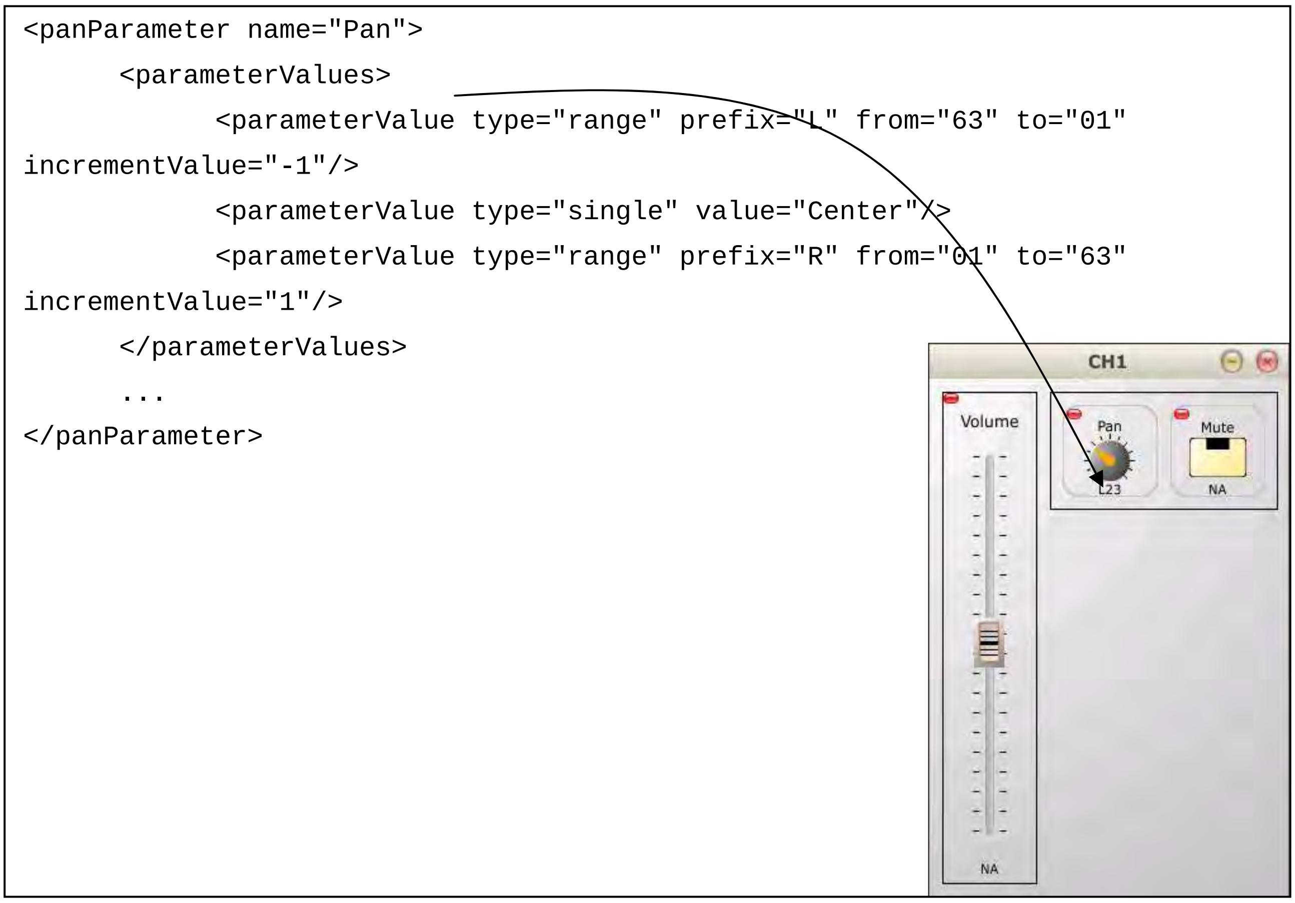
4.3.3.5. Describing Displayed Parameter Values

Each parameter has a specific value that is displayed at any one time. The displayed values of the parameters are adjusted by moving the linear faders, rotary potentiometers or by pressing the various buttons, depending on how the specific parameters are represented. Up until this point, no displayed values have been associated with the parameters that have been described in the Matrix Mixer XML document. The Matrix Mixer shows this by putting a ‘NA’ under the controls on the Parameter Adjust Window, as seen in Figure 35 above.

It is possible to describe the displayed values associated with specific continuous and switch parameters with the aid of the `parameterValues` XML element.

Figure 36 shows:

- A portion of a Matrix Mixer XML document showing a `parameterValues` XML element along with its child XML elements for a pan parameter.
- The Matrix Mixer once it has parsed the XML document.



```
<panParameter name="Pan">
      <parameterValues>
            <parameterValue type="range" prefix="L" from="63" to="01"
incrementValue="-1"/>
            <parameterValue type="single" value="Center"/>
            <parameterValue type="range" prefix="R" from="01" to="63"
incrementValue="1"/>
      </parameterValues>
      ...
</panParameter>
```

Figure 36: Representing the `parameterValues` element for a continuous parameter

If the displayed values that a specific parameter may take on need to be described, the `parameterValues` XML element describing these values is added as a child element to the XML element describing the parameter. In the above figure, the displayed parameter values of a pan parameter are being described.

The `parameterValues` XML element has one or more `parameterValue` child XML elements. These child XML elements collectively describe the range of displayed values that a specific parameter may have. For a continuous parameter, the displayed values are ordered from the value that the parameter is set to when it is at its lowest level, to the value that the parameter is set to when it is at its highest level. The number of displayed values in the range described with the `parameterValues` XML element has to correspond to the number of MIDI messages associated with the specific parameter.

There are two different types of `parameterValue` XML elements, and the specific type is specified with the `type` XML attribute. The value of the `type` XML attribute may be set to either `single` or `range`. When the value of the `type` XML attribute is set to `single`, this instructs the Matrix Mixer that the `parameterValue` element describes a single displayed value in the range of values. When the value of `type` is set to `range`, it instructs the Matrix Mixer that the `parameterValue` element describes a sub-range of displayed values within the overall range of displayed parameter values.

In the example shown in Figure 36, the first `parameterValue` XML element is used to describe the range of displayed values from ‘L63’ to ‘L01’ with an increment of -1. The second `parameterValue` XML element is used to describe a single displayed value, ‘Center’. The last `parameterValue` XML element is used to describe a range of displayed values from ‘R01’ to R63’ with an increment value of 1. Collectively these three XML elements describe the range of displayed values: L63, L62 … L02, L01, Center, R01, R02 … R62, R63. The Matrix Mixer is capable of converting the position of its faders and rotary potentiometers to parameter displayed values. This conversion is done internally by a `ParameterValues` object. If, for example, the pan potentiometer shown in Figure 36 is set ten notches up from its lowest value, the displayed value of the parameter will be set to ‘L54’.

Figure 37 shows an example of the `parameterValues` XML element that is used to describe the displayed values that a switch type parameter may have. This specific instance is used to describe the displayed values of a mute parameter. A switch type parameter may only have two displayed values, as the parameter may either be in an on state or an off state. The first value described with the `parameterValues` element is the displayed value that the parameter is set to when the parameter is in the on state, and the second value is the displayed value used when the parameter is in the off state.

```
<parameterValues>
     <parameterValue type="single" value="On"/>
     <parameterValue type="single" value="Off"/>
</parameterValues>
```

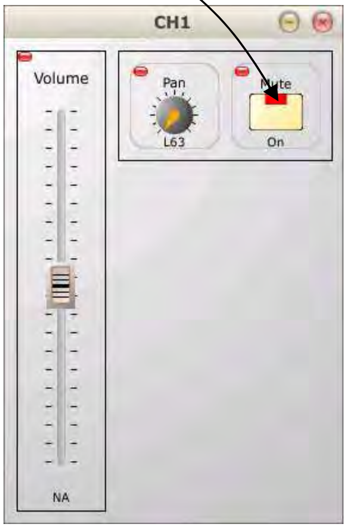
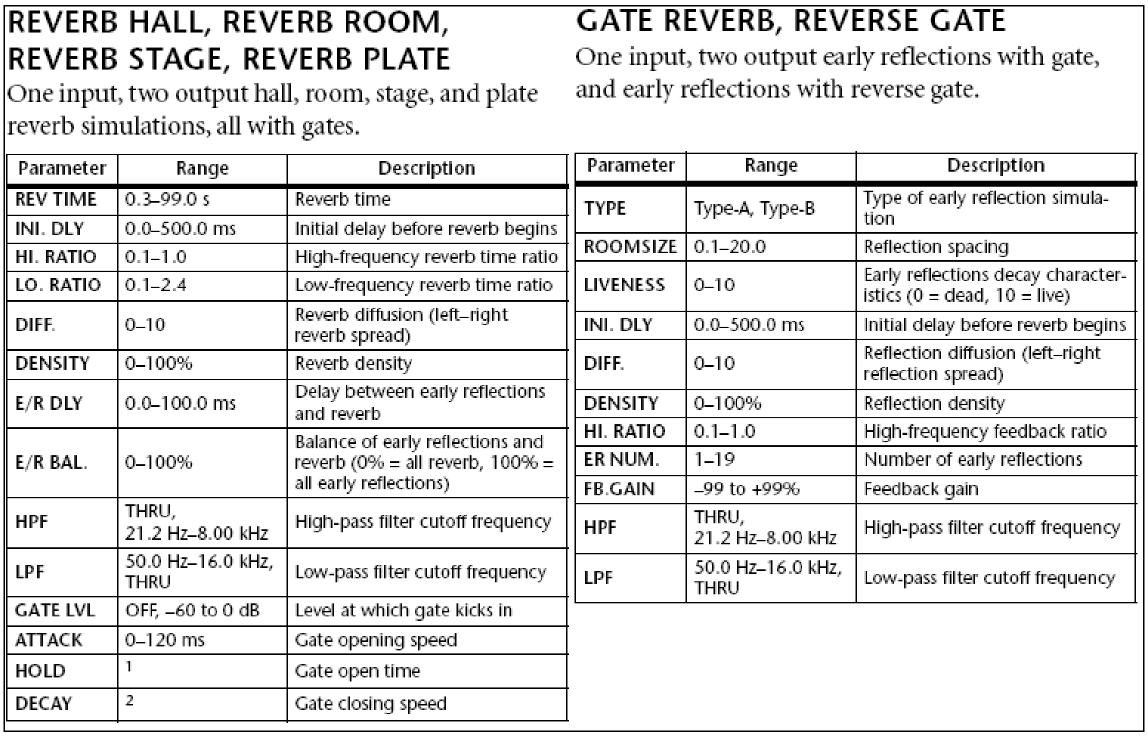


Figure 37: An example `parameterValues` XML element for a switch parameter

4.3.3.6. Describing Libraries Of Preset Values

Some signal processing components (for example, effects and dynamics processors) have libraries (consisting of a collection of library titles) of preset parameter values associated with them. Selecting one of the library titles may set the signal processing component to a specific type of that component. It also sets the parameters of the signal processing component to specific values. Each library title may be associated with a specific type of the signal processing component. For example, an effects processor may be capable of different types of effects. Some examples of different effects are ‘reverb hall’, ‘reverb room’, ‘stereo delay’ and ‘flange’. Each of these different types of effects may have a different set of parameters associated with them. Figure 38 shows examples of different types of effects found in the Yamaha 01V96 Digital Mixing Console and the various parameter types associated with each type of effect.



REVERB HALL, REVERB ROOM, REVERB STAGE, REVERB PLATE

One input, two output hall, room, stage, and plate reverb simulations, all with gates.

Parameter	Range	Description
REV TIME	0.3–99.0 s	Reverb time
INI. DLY	0.0–500.0 ms	Initial delay before reverb begins
HI. RATIO	0.1–1.0	High-frequency reverb time ratio
LO. RATIO	0.1–2.4	Low-frequency reverb time ratio
DIFF.	0–10	Reverb diffusion (left–right reverb spread)
DENSITY	0–100%	Reverb density
E/R DLY	0.0–100.0 ms	Delay between early reflections and reverb
E/R BAL.	0–100%	Balance of early reflections and reverb (0% = all reverb, 100% = all early reflections)
HPF	THRU, 21.2 Hz–8.00 kHz	High-pass filter cutoff frequency
LPF	50.0 Hz–16.0 kHz, THRU	Low-pass filter cutoff frequency
GATE LVL	OFF, –60 to 0 dB	Level at which gate kicks in
ATTACK	0–120 ms	Gate opening speed
HOLD	1	Gate open time
DECAY	2	Gate closing speed

GATE REVERB, REVERSE GATE

One input, two output early reflections with gate, and early reflections with reverse gate.

Parameter	Range	Description
TYPE	Type-A, Type-B	Type of early reflection simulation
ROOMSIZE	0.1–20.0	Reflection spacing
LIVENESS	0–10	Early reflections decay characteristics (0 = dead, 10 = live)
INI. DLY	0.0–500.0 ms	Initial delay before reverb begins
DIFF.	0–10	Reflection diffusion (left–right reflection spread)
DENSITY	0–100%	Reflection density
HI. RATIO	0.1–1.0	High-frequency feedback ratio
ER NUM.	1–19	Number of early reflections
FB.GAIN	–99 to +99%	Feedback gain
HPF	THRU, 21.2 Hz–8.00 kHz	High-pass filter cutoff frequency
LPF	50.0 Hz–16.0 kHz, THRU	Low-pass filter cutoff frequency

Figure 38: An example of the different types of effects with their parameters (Yamaha, 2004a)

The Matrix Mixer supports libraries of preset values for the dynamics and effects processors. Listing 10 shows an example of how a library is described for an effects processor with the `effectsProcessorLibrary` XML element. The `effectsProcessorLibraryTitles` element has a number of `effectsProcessorLibraryTitle` child elements. These elements collectively describe a library of preset values. Each of the `effectsProcessorLibraryTitle` child elements is used to describe a title in the library.

```
<effectsProcessorLibrary name="Library">
     <effectsProcessorLibraryTitles>
          <effectsProcessorLibraryTitle name="Reverb Hall" type="reverbhall">
               <on>
                    <parameterChange>
                         <midiMessagesGroups>
                              <midiMessagesGroup>
                                   <midiMessages type="single">
                                        <midiMessage value="0xF0 0x43 0x10
0x3E 0x7F 0x10 0x04 0x00 0x01 0x00 0x00 0xF7"/>
                                   </midiMessages>
                              </midiMessagesGroup>
                         </midiMessagesGroups>
                    </parameterChange>
               </on>
          </effectsProcessorLibraryTitle>
          ...
```

Listing 10: An example of an `effectsProcessorLibrary` XML element

The `name` XML attribute of each `effectsProcessorLibraryTitle` XML element is used to name each specific library title, and it is this value that is displayed on the Parameter Adjust Window. For each `effectsProcessorLibraryTitle` XML element representing a library title, an item is added to a combo box on the Parameter Adjust Window from which a specific title may be selected. A Parameter Adjust Window with an effects processor library displayed is shown in Figure 39. Selecting a specific effects processor library title is done by selecting the title from the combo box.

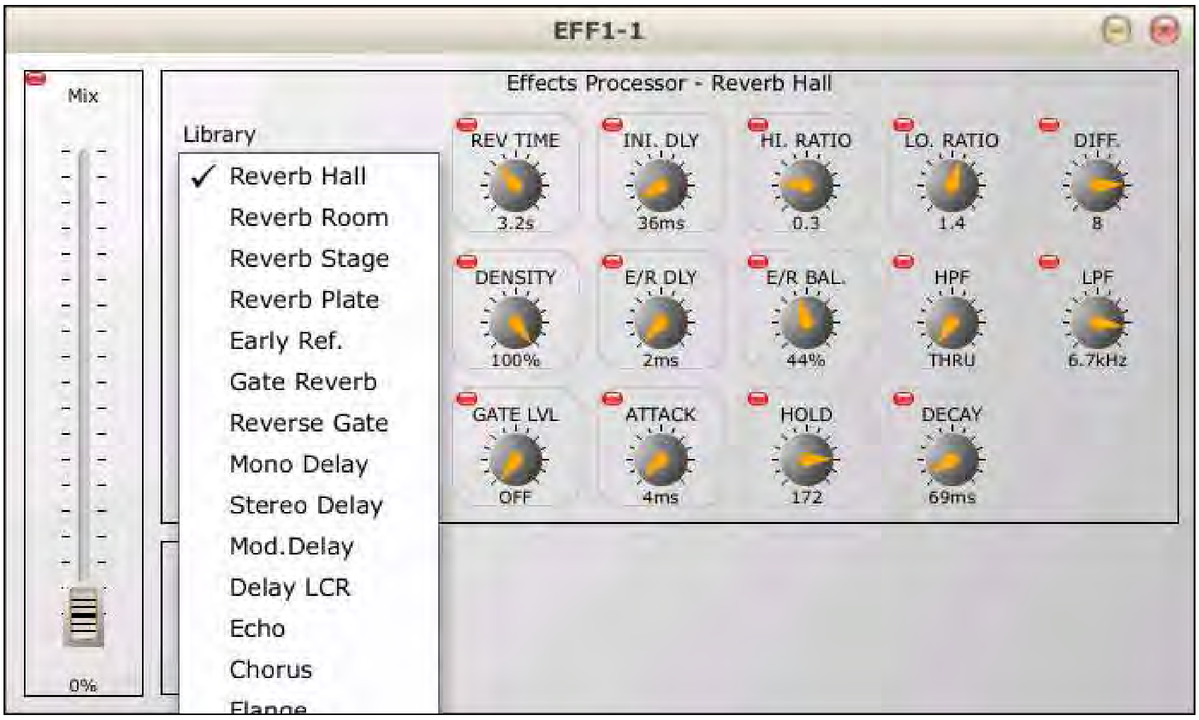


Figure 39: The Parameter Adjust Window with effects processor (reverb hall)

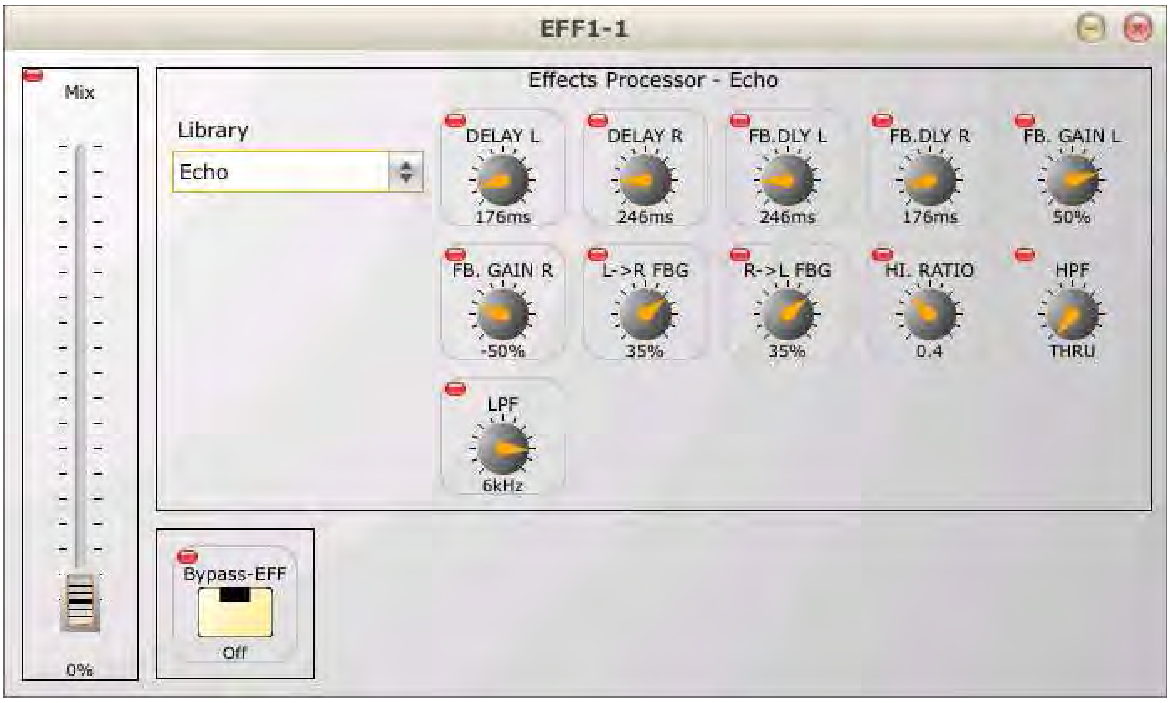
Each `effectsProcessorLibraryTitle` has an `on` child XML element that is used to describe the MIDI messages to be sent to the associated audio mixing desk when the library title is selected. This instructs the audio mixing desk to recall a specific library title.

Each `effectsProcessorLibraryTitle` XML element has a `type` XML attribute. This is used to specify the type of effect the specific library title is representing. Each type of effect is described with an `effectsProcessorType` XML element, an example of which may be seen in Listing 11.

```
<effectsProcessorType name="Reverb Hall" type="reverbhall">
      <effectsProcessorParameter name="REV TIME" type="revtime">
           …
      </effectsProcessorParameter>
     …
</effectsProcessorType>
```

Listing 11: An example of an `effectsProcessorType` element

The `effectsProcessorType name` XML attribute is used to name the specific type of effect that it is representing. This value is displayed on the Parameter Adjust Window when this specific effects processor is selected. The `type` XML attribute is used to specify the type of effect the XML element is describing. It is through the value of the `type` XML attribute that the effects processor library titles may be matched up to their corresponding effect type. When a library title is selected, the effect with the corresponding type is selected as the current effect. The specific parameters that comprise the selected effect are displayed on the display. The effects editor shown in Figure 39 above shows the ‘reverb hall’ effect as being selected along with its associated parameters. Figure 40 shows the effects editor with the ‘echo’ effects processor being shown. These two diagrams show the different parameters (along with their preset values) that are associated with each specific type of effect.


EFF1-1

Effects Processor - Echo

Library: Echo

DELAY L 176ms	DELAY R 246ms	FB.DLY L 246ms	FB.DLY R 176ms	FB. GAIN L 50%
FB. GAIN R -50%	L->R FBG 35%	R->L FBG 35%	HI. RATIO 0.4	HPF THRU
LPF 6kHz				

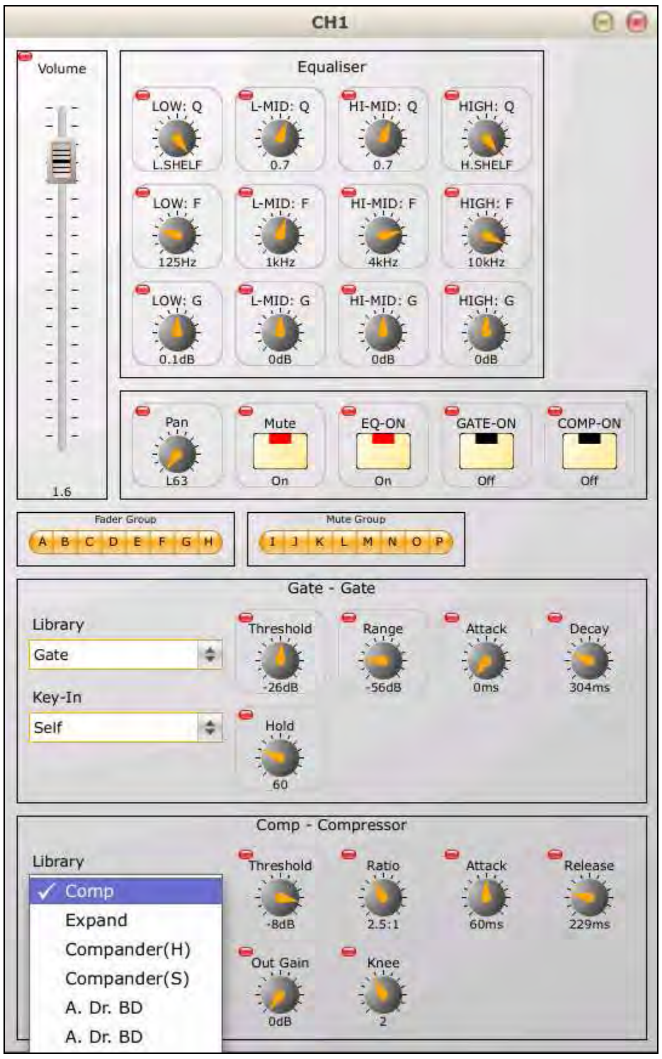
Mix 0%

Bypass-EFF Off

Figure 40: The Parameter Adjust Window with effects processor (echo)

In Matrix Mixer XML documents, the specific parameters that belong to a specific effect type are described with `effectsProcessorParameter` XML elements, as shown in Listing 11 above. These XML elements are similar to all other parameter XML elements. They may have the same child XML elements as discussed above for the `volumeParameter` and `muteParameter` XML elements.

The same mechanism exists for the dynamics processors. The dynamics processors that exist at the various signal processing points may be configured as different types. As an example, a dynamics processor may be configured as a compressor or a noise gate. Each specific type of dynamics processor has associated with it a number of parameters. Each dynamics processor library title is associated with a specific type of dynamics processor. Each dynamics processor described in the XML document has a type specified with a `type` XML attribute. Thus, as with the effects processors, when a specific dynamics processor library title is selected, the type of dynamics processor that that library title is representing is matched up to the specific dynamics processor type. The specific parameters that belong to the selected dynamics processor are then displayed on the display. Figure 41 shows a Parameter Adjust Window with two dynamics processors displayed on it. Displayed is the library associated with each dynamics processor, and the parameters associated with the specific type of dynamics processor.



Volume 1.6

LOW: Q L.SHELF
L-MID: Q 0.7
HI-MID: Q 0.7
HIGH: Q H.SHELF
LOW: F 125Hz
L-MID: F 1kHz
HI-MID: F 4kHz
HIGH: F 10kHz
LOW: G 0.1dB
L-MID: G 0dB
HI-MID: G 0dB
HIGH: G 0dB

Pan L63
Mute On
EQ-ON On
GATE-ON Off
COMP-ON Off

Library Gate
Key-In Self
Threshold -26dB
Range -56dB
Attack 0ms
Decay 304ms
Hold 60

Library ✓ Comp
Threshold -8dB
Ratio 2.5:1
Attack 60ms
Release 229ms
Out Gain 0dB
Knee 2

Figure 41: A Parameter Adjust Window showing two dynamics processors

4.4. An XML Schema For Describing Audio Mixing Desks

Document Type Definitions (DTD) [Elliotte *et al*, 2002] and XML schemas [Elliotte *et al*, 2002] are used as a means to formally specify the valid structures of specific instances of XML documents for specific applications.

A Document Type Definition (DTD) is used as a means to specify precisely which XML elements and entities are allowed to appear within an instance of an XML document. They also specify what the contents of XML elements are, and the XML attributes they may have. They can be used to specify which XML elements are child XML elements of specific XML elements and may also ensure that specific XML elements contain certain XML attributes. A DTD is used to specify what an XML document for a specific application may or may not contain. Different applications may all have different DTDs depending on their individual requirements. A validating parser may be used to compare a specific XML document against a DTD to check that it has the correct structure for a specific application.

DTDs are limited. They are unable to specify:

- What the root element of an XML document is.
- The number of times that each element may appear in an XML document.
- What the character data inside an XML element looks like.
- The semantic meaning of an element. A DTD does not specify the length, structure, meaning, allowed values, or any other aspects of the text content of an element.

An XML schema is an XML document that has a formal description of what comprises a valid XML document. Compared to DTDs, schemas provide much finer control over the format and data types that the XML elements and attribute values of an XML document may have. They also provide for much finer control over the text content of XML elements and attributes. There are a number of built in types, but schemas allow for new types to be declared, new types to be derived from old types, and the reuse of types from other schemas. Schemas also allow for more explicit restrictions to be placed on the number and sequence of child XML elements that may appear at a specific point.

XML documents have been created that describe the Yamaha 01X Digital Mixing Studio and the Yamaha 01V96 Digital Mixing Console for the Matrix Mixer. From creating these XML documents, XML elements and attributes have been defined that allow for the signal processing points, along with their signal processing components, to be described. From the structure of these two documents, an XML schema was generated and customised to provide a formal description of the valid XML documents that the Matrix Mixer is capable of accepting. This mechanism allows third parties to create XML documents that describe audio mixing desks for the Matrix Mixer. The created XML documents may be validated against the XML schema before the Matrix Mixer reads them in. The complete XML schema is shown in "Appendix – An XML Schema For Representing MIDI Controllable Audio Mixing Desks"

4.5. Summary

In this chapter, we looked at how XML has been used as a configuration tool for the Matrix Mixer. The Matrix Mixer was developed to be a generic software audio mixing desk editor which is able to represent and control common audio mixing desk parameters. In order to allow the Matrix Mixer to quickly adapt to different audio mixing desks, specific audio mixing desks are described using XML documents which the Matrix Mixer reads in when it initialises itself. From the information in the XML document, the Matrix Mixer builds itself up to represent and control a specific audio mixing desk.

The Matrix Mixer was initially developed for the Yamaha 01X Digital Mixing Studio. Once this was complete, and an initial set of elements and attributes were defined for the XML audio mixing desk descriptions, creating an XML document for the Yamaha 01V96 Digital Mixing Console was a quick process. In this time period, minor adjustments had to be made to the Matrix Mixer to accommodate the nature of both these audio mixing desks. The simple, readable nature of XML made the task of adapting the Matrix Mixer to another audio mixing desk easy to perform. A Matrix Mixer XML document is built up to describe the features of an audio mixing desk and does not concern itself with how these features are represented or controlled. The representation and control of the features of the audio mixing desks is left up to the Matrix Mixer.

From the Matrix Mixer XML documents that were created for the Yamaha 01X Digital Mixing Studio and the Yamaha 01V96 Digital Mixing Console, an XML schema was created to formally describe the valid structures that Matrix Mixer XML documents may take on. This mechanism allows third parties to describe audio mixing desks for the Matrix Mixer.

Chapter 05

Studio Connections

Steinberg, one of the world's largest audio software houses, and Yamaha, the world's largest manufacturer of professional audio equipment, have been leading a joint project known as Studio Connections [Yamaha, 2005]. The aim of this project is to offer a more convenient environment to make using audio hardware and software easier. This project came about as a result of a need for closer integration between audio software and audio hardware in music production systems.

In this chapter, we will have a look at what Studio Connections is, and how it was used to make the Matrix Mixer operate with Digital Audio Workstation (DAW) applications in order to offer the capability of recalling audio mixing desk state ('Total Recall').

5.1. Studio Connections

Traditionally, music hardware and software have been set up independently of each other. For example, a sound engineer setting up a small recording studio would first make connections between devices, possibly using a patchbay like the Yamaha Graphic Patchbay (see section 6.3.4). Then, each individual device may need to be set up. This can be done via the control surfaces of the devices, or from software device editors that are able to represent and control those devices. Then, recording software would need to be set up in order to record the signals generated from the audio devices in the studio. Each of these processes happen independently of each other. That is, they are not integrated with each other.

The first phase of the Studio Connections project has been identified as Total Recall. This phase of the Studio Connections project allows for:

- The control of the settings of hardware devices from within DAW applications via software device editors.
- The ability for audio hardware device states to be saved to DAW native song files allowing the devices to be recalled later.

This is achieved by DAW applications hosting software plug-ins. Each software plug-in allows for:

- The representation and remote control of a hardware device from within a DAW application.
- The persistence of hardware state. These plug-ins are capable of providing the state of the hardware devices they are representing to the DAW application. The DAW application may then save the state data to one of its native song files. When the DAW application loads itself from the native song file at a later stage, it extracts the state data that was originally provided by the plug-ins. This state data may then be passed back to the relevant software plug-ins. They use this state data to set their parameters and the hardware parameters to the saved state.

Currently, Total Recall may be implemented within a DAW application by hosting the Studio Manager 2 host component. Studio Manager 2 in turn hosts multiple software device editor plug-ins. These software device editor plug-ins may then be opened from within the DAW application that is hosting Studio Manager 2. Studio Manager 2 acts as a proxy between a DAW application and the software device editor plug-ins. A simplified version of this architecture is shown diagrammatically in Figure 42.

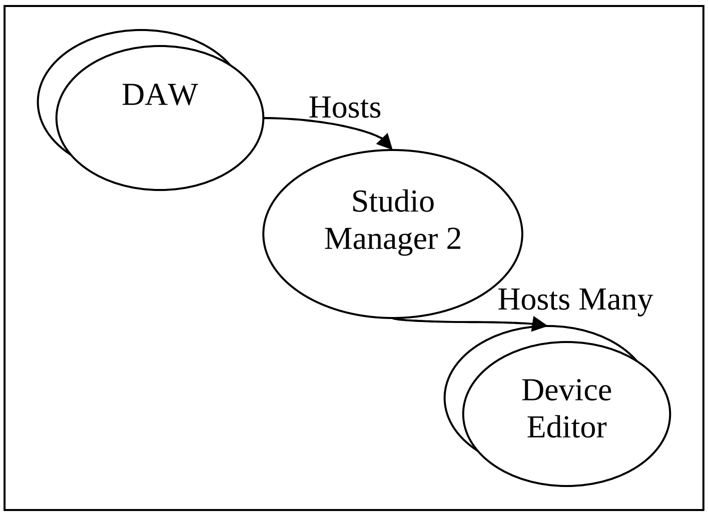
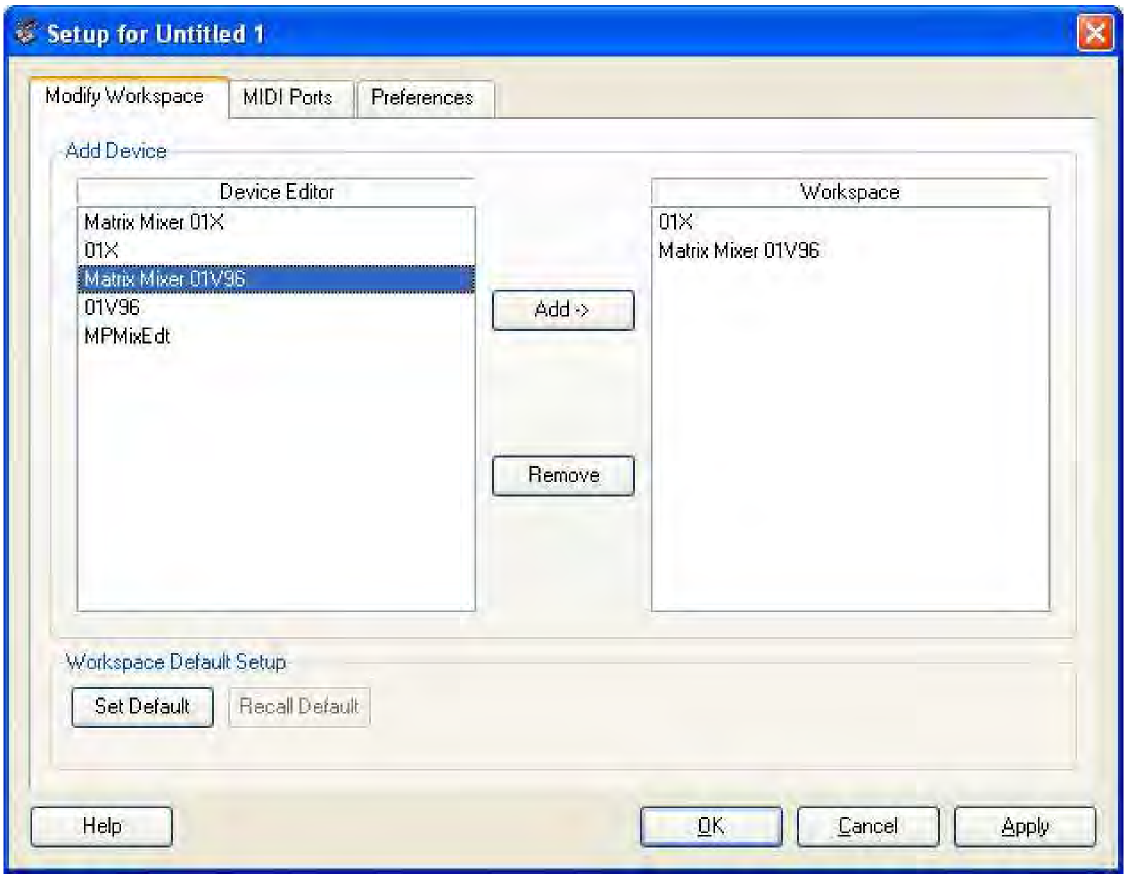


Figure 42: Studio Manager 2 integration

Typically, a software device editor provides capabilities for editing a particular device's parameters via a Graphical User Interface (GUI). Examples of these device editors are the Matrix Mixer and the Yamaha 01V96 Editor. These software device editors provide for state machine emulation of the hardware devices that they represent, and may allow for the transfer of the represented devices' states between themselves and their associated hardware devices. This can happen via protocols such as the MIDI protocol. Via Studio Manager 2, these software device editors may allow for their state to be transferred to and from themselves and a DAW application. This allows the hardware devices states to be saved to DAW native song files and later recalled.

Studio Manager 2 is a visually simple application. The main purposes of this application are to instantiate and manipulate a collection of device editors. A sound engineer is able to select which device editors (from a list of available device editors installed on the system) appear in its workspace. Total Recall operations may then be performed on the selection of devices. This approach allows sound engineers to define customised workspaces that are based on unique studio setups.

Sound engineers are able to define their own workspaces via the Studio Manager 2 Setup window, which is shown in Figure 43. This window is displayed when Studio Manager 2 starts up. The Setup window lists all the available software device editors (left-hand column) installed on the system and all the software device editors that are part of the user defined workspace (right-hand column). Software device editors are added to the workspace by selecting them from the available software device editors. If, for example, we had a recording studio with a Yamaha 01X Digital Mixing Studio and a Yamaha 01V96 Digital Mixing Console in it, and we wished to control these devices from within Studio Manager 2, software device editors that are able to represent and control these audio mixing desks would be added to the 'Workspace' list, as shown in Figure 43.



Device Editor
Matrix Mixer 01X
01X
Matrix Mixer 01V96
01V96
MPMixEdt

Workspace
01X
Matrix Mixer 01V96

Figure 43: Defining the Studio Manager 2 application's workspace

Once a sound engineer has finished setting up the custom workspace, each of the selected device editors is instantiated, and the primary window of Studio Manager 2 is populated with icons that represent each of the selected software device editors. The primary window of Studio Manager 2 is shown in Figure 44. The devices shown in the workspace are those that were defined by the user via the Setup window.

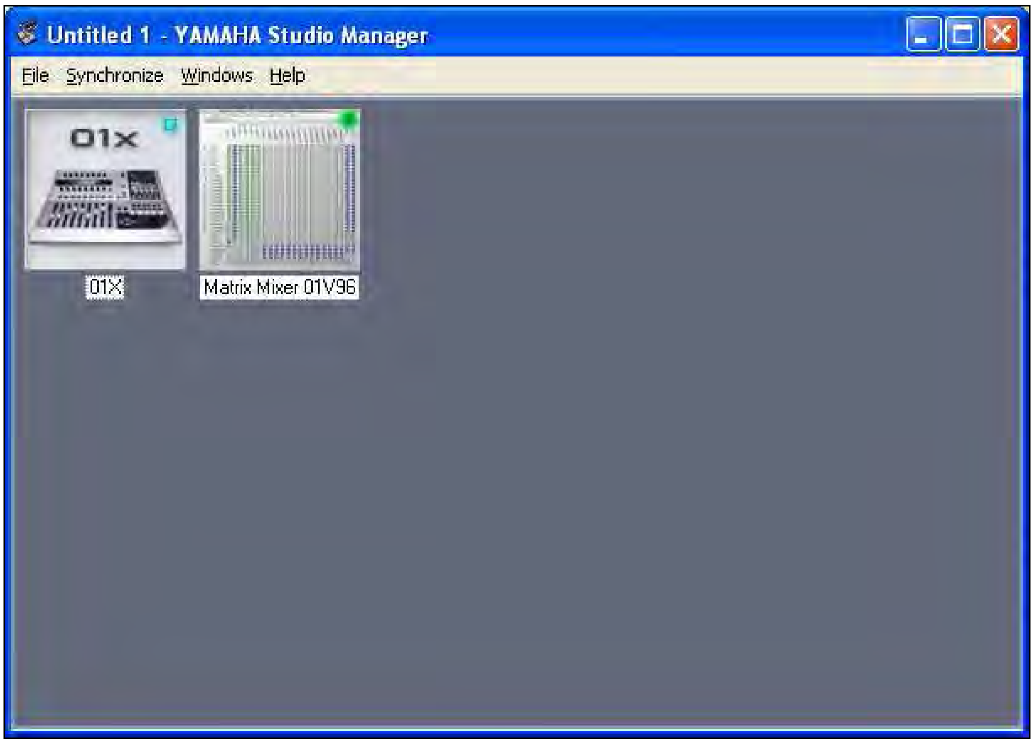

Untitled 1 - YAMAHA Studio Manager
File Synchronize Windows Help
01x
01X
Matrix Mixer 01V96

Figure 44: The Studio Manager 2 host application

From the primary window of the Studio Manager 2 application a device editor may be displayed by selecting its icon. This is shown in Figure 45 as annotation 2. Annotation 1 in the figure indicates that the Studio Manager 2 host application is being hosted and displayed by a DAW application, in this instance Cubase [Steinberg, 2007].

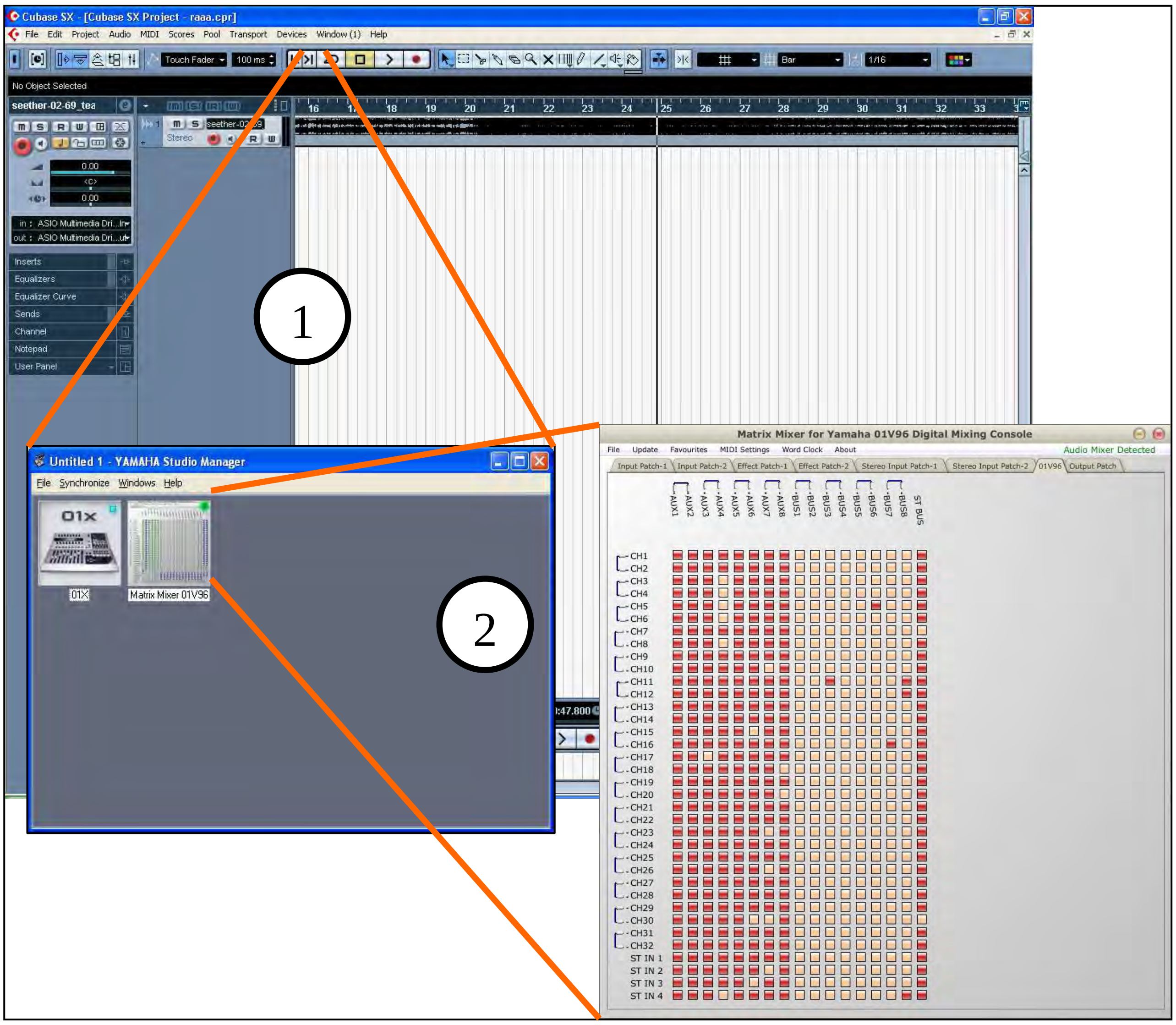

Cubase SX - [Cubase SX Project - raaa.cpr]
File Edit Project Audio MIDI Scores Pool Transport Devices Window (1) Help
Touch Fader
100 ms
Bar
1/16
No Object Selected
seether-02-69_tea
Inserts
Equalizers
Equalizer Curve
Sends
Channel
Notepad
User Panel
1
Untitled 1 - YAMAHA Studio Manager
File Synchronize Windows Help
01X
Matrix Mixer 01V96
2
Matrix Mixer for Yamaha 01V96 Digital Mixing Console
File Update Favourites MIDI Settings Word Clock About
Audio Mixer Detected
Input Patch-1 Input Patch-2 Effect Patch-1 Effect Patch-2 Stereo Input Patch-1 Stereo Input Patch-2 01V96 Output Patch

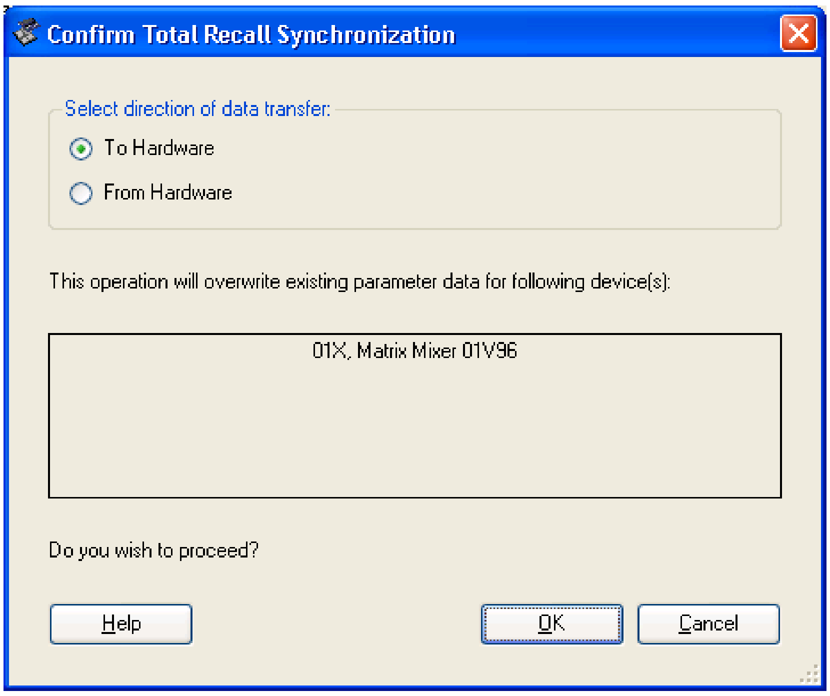
Figure 45: Studio Manager 2 host application hosting

Studio Manager 2 allows its user defined workspace to be saved to a compound document format session file in a form that suits it. Within this file, the following may be saved:

- The state of each device represented within the workspace.
- The display state of each device editor.
- The window position of each device editor.
- The device editor MIDI port selections.

Each device editor is responsible for saving data about itself. This allows this state data to be later recalled, thus allowing the hardware devices to be recalled to their previous states.

When the workspace of Studio Manager 2 is loaded from one of its saved files, it recalls its workspace to the state it was in when the file was saved. Each of the software device editor plug-ins is re-instantiated and the state of the parameters of the software devices editors are set to their saved states. A sound engineer is then able to synchronise the software device editors with their associated hardware devices. This may happen by either transferring the states of the parameters of the software device editors across to their associated parameters in the hardware devices, or the states of the hardware devices' parameters may be transferred across to their associated software device editors. Sound engineers are able to select in which direction the state transfer takes place via the Confirm Total Recall Synchronization window (which is shown in Figure 46) under the 'Select direction of data transfer:' section. Once the direction of state transfer has been selected, the state is transferred in the relevant direction. This transfer could take place in the form of MIDI messages.


Confirm Total Recall Synchronization

Select direction of data transfer:

[✓ To Hardware]

From Hardware

This operation will overwrite existing parameter data for following device(s):

01X, Matrix Mixer 01V96

Do you wish to proceed?

Figure 46: The Confirm Total Recall Synchronization window

5.1.1. Open Plug-in Technology

Studio Connections Total Recall is built on top of Open Plug-in Technology (OPT) [Yamaha, 2002]. Each device editor that is hosted by Studio Manager 2, as well as Studio Manager 2 itself, is implemented as an enhanced OPT component. OPT provides in-house and third party software developers with integrated enhancements to the feature sets of sequencer products. These enhancements are considered necessary to cope with the increasing sophistication of MIDI devices and the possibility of these devices having extensions to the MIDI specification. This architecture allows for the integration of custom plug-ins to support new hardware and MIDI extensions.

The OPT architecture is based on the Component Object Model (COM) architecture [Troelsen, 2000; Armstrong and Patton, 2000]. Each software device editor plug-in is implemented as a COM in-process server. An in-process COM server is loaded into the same memory space as the client application that loads it.

5.1.2. Component Object Model

COM is an object oriented interface-based programming architecture that may be implemented in many programming languages. COM components are used to provide some form of functionality to a software entity through an implemented interface. COM components are viewed as servers and the software entities that use the COM components are viewed as clients.

COM components can be accessed from many languages. COM provides highly scalable, reusable, and accessible binary objects. Client applications that use COM components are unaware of how the object is implemented or in which language. Interacting with a COM object takes place through an object's set of interfaces. A user of a COM object is only ever aware of the object's interfaces. The interfaces define a set of methods and properties which describe how client applications may interact with the object. The interfaces provide no state and no implementation. Classes that implement COM interfaces provide the definitions for the declared interfaces. There may be many different implementations of the same interface to provide specific functionality. COM objects are able to load other COM objects in order to perform their tasks. All COM objects are packaged together inside a "component housing".

Each COM interface, class, type library and executable (and other COM items) are uniquely identified with Globally Unique Identifiers (GUID's). These GUID's are entered into the system registry. Under the GUID entry may be found specific information pertaining to the COM component that the GUID is representing. The most important of these entries is the entry that specifies the physical path to the COM server. This removes the need for client applications to know the physical path to the COM objects that they require. They just need to be aware of the GUID of the required object. This is known as 'location transparency'.

Each COM object has to implement an interface known as `IUnknown`. The `IUnknown` interface defines three methods:

- `QueryInterface`: The `QueryInterface` method is used by client applications to get pointers to the interfaces implemented by a COM object. If an object implements interface `i1` and interface `i2`, and a client application has a pointer to interface `i1`, it may obtain a pointer to interface `i2` by calling `QueryInterface` on the pointer to interface `i1`. The client specifies which interface pointer it requires by passing in the GUID of that interface. If the COM object has implemented the requested interface, the requested interface pointer is returned to the client application.
- `AddRef` and `Release`: `AddRef` and `Release` are used to manage the existence of a particular object. Each time there is a new interface pointer to an object, `AddRef` should be called on that object. This increases a reference counter that the object maintains. Once an application has finished using an interface pointer, it should call `Release` on it. When `Release` is called, the internal reference counter that the object keeps is decreased by one. When the reference counter reaches a value of zero the object deletes itself.

The COM architecture was chosen for Studio Connections Total Recall as it is well suited to plug-in components since each plug-in is presented in a binary format. A common set of interfaces for each plug-in has been defined allowing Studio Manager 2 to communicate with each specific plug-in implementation (each specific software device editor). The COM architecture also avoids any problems associated with technology transfer and manufacturer-specific libraries as each COM component is presented in binary form. It allows for the expansion of the defined set of interfaces, whilst still allowing plug-ins using previous versions of the interfaces to work seamlessly with the host application, but with possibly less functionality.

5.2. Providing Total Recall Functionality To The Matrix Mixer

Initially, the Matrix Mixer was developed as a standalone application. It was implemented with the ability to save the state of its parameters to a file. In doing this, it saved the state of the audio mixing desk that it was representing. The Matrix Mixer could then read in that file at a later stage and recall itself to the saved state. The state that the parameters of the Matrix Mixer were in could then be sent across to the associated audio mixing desk in the form of MIDI messages.

The ability to save the state of an audio mixing desk is advantageous. If there is an audio mixing desk in a recording studio, and the recording studio is continuously being used by different groups of people, who all set the audio mixing desk to their own individual settings, each group could save the state of the audio mixing desk and later restore it in a different recording session. This avoids the need to have to manually set up the audio mixing desk every time that it had its parameters adjusted by other groups of people.

The standalone version of the Matrix Mixer lacks any integration with any other audio software. Consider a situation where we had a recording studio, and in that studio we were running a DAW application, and had two Yamaha 01V96 Digital Mixing Consoles. We might have two instances of the Matrix Mixer representing the two audio mixing desks. If we wanted to save our project created with the DAW application, and the state of the two audio mixing desks, these would all have to be done individually and independently of each other.

Currently there exist a number of software device editors that, with the aid of Studio Manager 2, allow for the state of the device that they are representing to be saved to native DAW song files. This functionality is provided to these programs through the use of Studio Connections Total Recall.

The next phase of this project involved implementing the Matrix Mixer to be Studio Connections Total Recall compatible. This allows the Matrix Mixer to be hosted by Studio Manager 2. This in turn allows the Matrix Mixer to be opened from within DAW applications. It also allows for saving and recalling the state that the Matrix Mixer is in (hence also the state of the associated audio mixing desk) and allows that audio mixing desk to be later recalled to its saved state.

The high level functionality that had to be implemented in order to make the Matrix Mixer Studio Connections Total Recall compatible is shown with the aid of a use case diagram in Figure 47. This functionality is provided via a collection of programming interfaces that are part of the Studio Connections - Total Recall SDK and also by Microsoft [Microsoft Corporation, 2005]. These interfaces allow Studio Manager 2 to communicate with the Matrix Mixer and vice versa. Studio Manager 2 is aware of these interfaces, and is aware of what is supposed to happen when the methods of the interfaces are called.

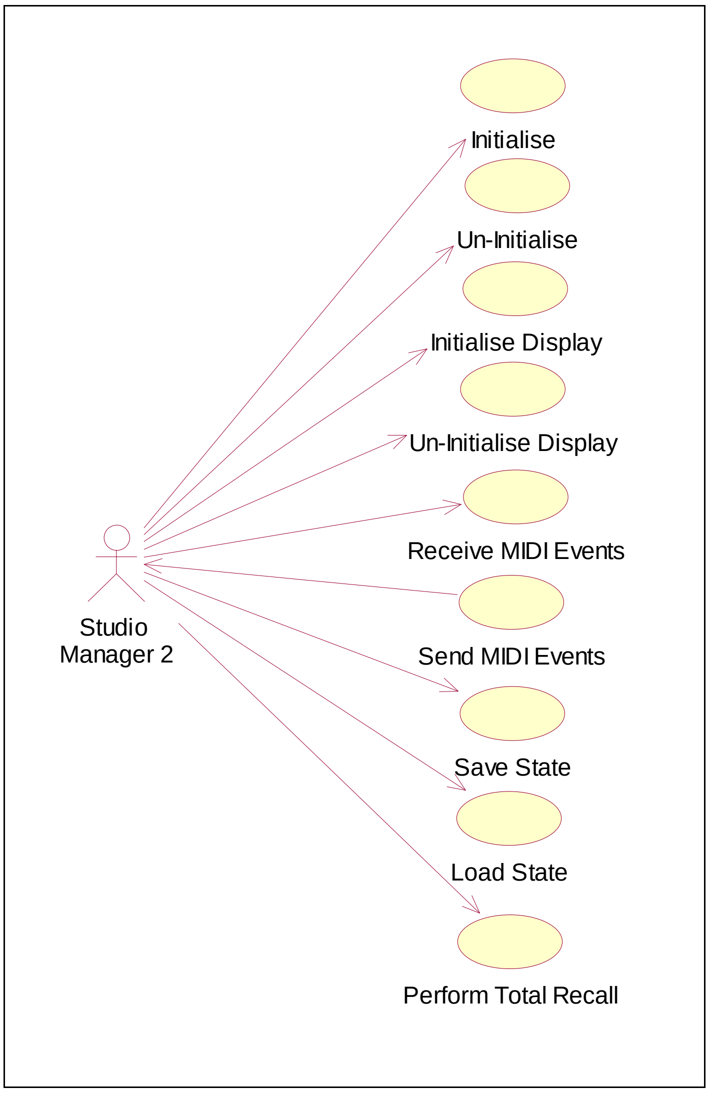


Figure 47: Matrix Mixer Studio Connections Total Recall use case diagram

Initially, the Matrix Mixer was implemented as a standalone executable. The initialisation of the Matrix Mixer was driven via the main entry point to the program. Tasks such as saving the state of the parameters of the Matrix Mixer was initiated with a sound engineer selecting the appropriate menus from the display of the Matrix Mixer. The Matrix Mixer had to be modified such that it could be dynamically loadable by a host application (for example, Studio Manager 2). It also had to be modified such that it could be driven via the methods of the interfaces that are defined in the Studio Connections – Total Recall SDK. The core interfaces that had to be implemented are listed below:

- `IMPInitialise`: Via the methods of this interface, a client application is able to initialise and un-initialise a device editor. The client application is also able to pass in pointers to its own interfaces so that the device editor may communicate with the client.

- `IMPEventFilter`: Through this interface, a client application is able to pass in its queued events to a device editor. This allows the device editor to receive incoming MIDI messages from its associated hardware device.
- `IMPServices`: The `IMPServices` interface allows a client application to control the Total Recall operations of a device editor. Total Recall is the transfer of device state from either hardware to software or from software to hardware.
- `IPropertyPage`: Through the methods of the `IPropertyPage` interface, a client application is able to control the GUI of a particular device editor. It is able to perform actions such as initialising and un-initialising the display, showing and hiding the display, and moving the display around the screen.
- `IPersistStream`: The `IPersistStream` interface allows a client application to instruct the device editor to save or load itself to and from a given stream, respectively.

5.2.1. Initialising And Un-initialising The Matrix Mixer

Once a sound engineer has chosen the devices for the Studio Manager 2 workspace from the Setup window (see Figure 43 above) each of the selected software device editors needs to be initialised. Initialisation of each device editor happens via the `MPConnect` method of the `IMPInitialise` interface. Each device editor is responsible for providing an implementation of the `MPConnect` method.

Studio Manager 2 obtains a pointer to each device editor's `IMPInitialise` interface and then calls their `MPConnect` methods (the `MPConnect` method declaration is shown in Listing 12).

```
HRESULT MPConnect(IUnknown* pUnkClient, IMPAsyncOutput* pAsync,
MPINITIALISE_MODE eMode, DWORD dwTimeContext, LPDWORD
pdwQueueFlags);
```

Listing 12: The `IMPInitialise::MPConnect` method

Through this method Studio Manager 2 is able to:

- Provide each device editor with pointers to its interfaces. Studio Manager 2 provides pointers to its `IUnknown` and `IMPAsyncOutput` interfaces to each device editor. Each device editor should retain copies of these interface pointers as these allow the device editors to communicate with Studio Manager 2. A device editor is able to obtain further interface pointers from Studio Manager 2 through the `QueryInterface` method of the `IUnknown` interface pointer passed in. The `IMPAsyncOutput` interface is used for the communication of MIDI messages. This is discussed in section 5.2.4 below.

- Initialise each device editor.

When the Matrix Mixer's implementation of the `MPConnect` method is called, it parses its XML configuration file and builds up its internal objects to represent the specific audio mixing desk described in the XML file. At this point, no user interface is displayed. In this state, and once its MIDI settings have been set appropriately, it is able to receive MIDI messages from its associated audio mixing desk (see section 5.2.3 below). From these MIDI messages it is able to extract parameter values and set them appropriately in its internal objects. Once the Matrix Mixer is initialised it waits for further instructions from Studio Manager 2 via calls to its interfaces methods.

There are times when Studio Manager 2 may wish to un-initialise any of the device editors that are part of its workspace. For example, this could happen when a sound engineer decides to change the software devices editors that are part of the workspace or when the application itself terminates. When Studio Manager 2 wishes to un-initialise any of the device editors in its workspace, it calls the `MPDisconnect` method of the appropriate device editor(s). Each device editor provides an implementation of the `MPDisconnect` method (this method is defined as part of the `IMPInitialise` interface). Each device editor is responsible for performing any un-initialisation that is required. When the Matrix Mixer's implementation of this method is called, it deletes all of the internal objects that it used to represent the associated audio mixing desk and frees up any resources it was holding. The `MPDisconnect` method declaration is shown in Listing 13.

```
HRESULT MPDisconnect(void);
```

Listing 13: The `IMPInitialise::MPDisconnect` method

5.2.2. Initialising And Un-Initialising The Matrix Mixer Display

When the Studio Manager 2 host application initialises each of its software device editors, their GUIs are not displayed. The GUIs of the devices editors are only displayed on request. A sound engineer is able to display the GUIs of the device editors in the Studio Manager 2's workspace by selecting the icons representing them. This is shown as annotation 2 in Figure 45 above. Studio Manager 2 is able to display its software device editors through the use of two Microsoft defined interfaces. These are `ISpecificPropertyPages`, and `IPropertyPage`.

A COM object that implements the `ISpecificPropertyPages` interface indicates that it supports property pages. The `ISpecificPropertyPages` interface has one method, namely `GetPages`. This method declaration is shown in Listing 14. For Studio Manager 2, this method is used to return the class identifier (CLSID) of the property page object that belongs to the specific software device editor. Through the obtained CLSID, Studio Manager 2 is able to locate the `IPropertyPage` interface of a particular device editor in its workspace.

```
HRESULT GetPages(CAUUID* pPages);
```

Listing 14: The `ISpecificPropertyPages::GetPages` method

Through the use of the `IPropertyPage` interface, Studio Manager 2 is able to control the GUI of a particular device editor. When a sound engineer selects an icon representing a software device editor from the workspace, Studio Manager 2 creates a parent window and calls the `Activate` method (shown in Listing 15) of the `IPropertyPage` interface for the selected device editor. As an argument to the `Activate` method, a handle to the parent window that was created is passed to the device editor. Here, the selected device editor creates its display window, and the states of the components on that window reflect the state that the device editor is in. It is in the parent window that the device editor places its window as a child window.

```
HRESULT Activate(HWND hWndParent, LPCRECT prc, BOOL bModal);
```

Listing 15: The `IPropertyPage::Activate` method

When the Matrix Mixer's implementation of the `Activate` method is called, it creates its routing matrices. Each of the patch buttons on the routing matrices reflect the state of the Matrix Mixer's internal objects that in turn represent the various patch points of the associated audio mixing desk.

When a sound engineer closes the parent window of a displayed device editor, Studio Manager 2 calls the `Deactivate` method (shown in Listing 16) of the `IPropertyPage` interface of the particular device editor. When this method is called, the specific device editor destroys its display, thus freeing the resources that it was utilising.

```
HRESULT Deactivate(void);
```

Listing 16: The `IPropertyPage::Deactivate` method

The `IPropertyPage` interface also declares methods that allow the device editors' displays to be moved around and hidden by Studio Manager 2.

5.2.3. Receive MIDI Events

All MIDI messages that are destined for the device editors in Studio Manager 2's workspace are received by the DAW application that is hosting Studio Manager 2. The DAW application is responsible for routing MIDI messages so that they may be received by the device editors that are part of Studio Manager 2's workspace. This process is shown diagrammatically in Figure 48.

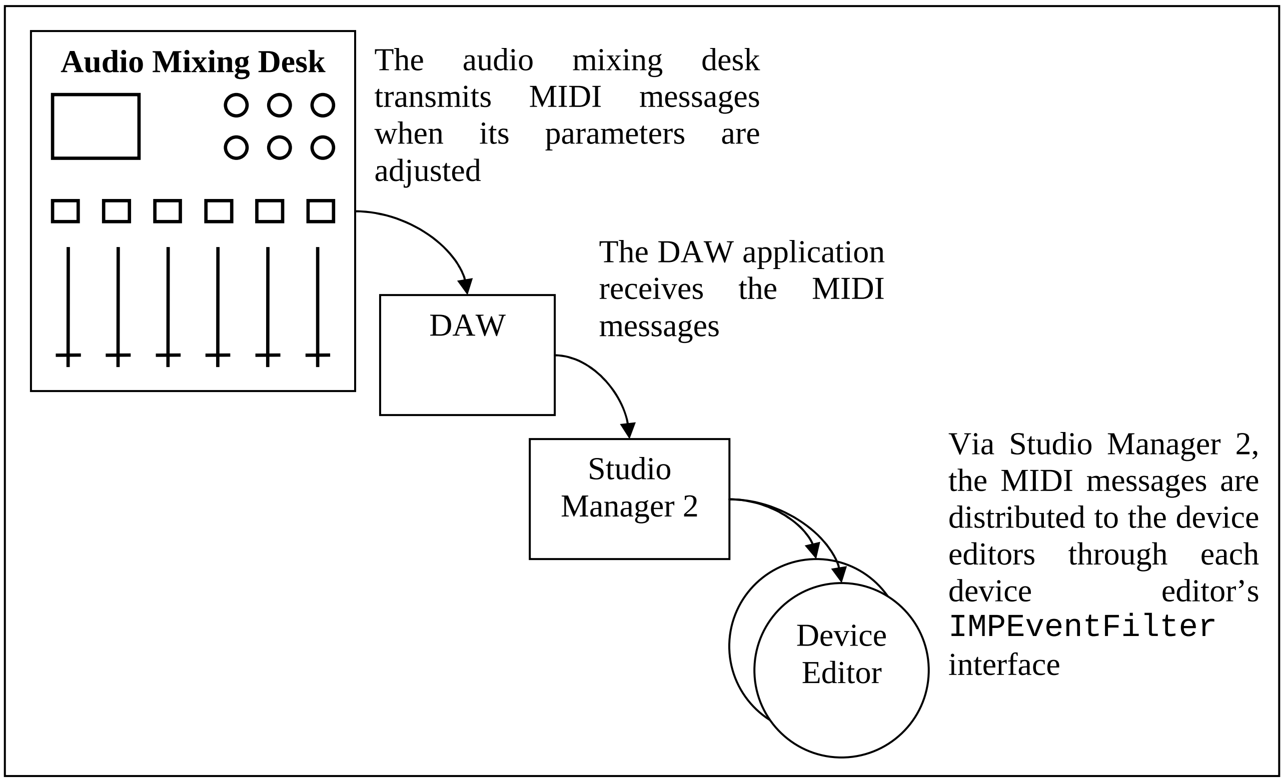

Audio Mixing Desk
The audio mixing desk transmits MIDI messages when its parameters are adjusted
DAW
The DAW application receives the MIDI messages
Studio Manager 2
Via Studio Manager 2, the MIDI messages are distributed to the device editors through each device editor's IMPEventFilter interface
Device Editor

Figure 48: Receiving MIDI messages via the `IMPEventFilter` interface

A software device editor that wishes to receive MIDI messages has to implement the IMPEventFilter interface. The IMPEventFilter interface provides methods to allow for the communication of queued events (for example, fader moves and button pushes) from a DAW application to a device editor. This interface is used to receive MIDI messages from a hosting DAW application in real-time. Each device editor receives MIDI messages through the interface's MPOnEvents method. Each device editor provides an implementation of this method which allows the device editor to process the queues of events. The MPOnEvents method declaration is shown in Listing 17.

```
HRESULT MPOnEvents(IMPEventQueue* pqInOut, REFMPQSTATUS
rqStatus);
```

Listing 17: The IMPEventFilter::MPOnEvents method

Each time the MPOnEvents method of one of the device editors is called, a pointer to an IMPEventQueue interface is passed in (through the pqInOut pointer). The IMPEventQueue interface provides a set of methods used to access a queue of time ordered MPEVENT structures. It is the MPEVENT structures that contain the MIDI messages that the software device editors use to communicate with their associated hardware devices. Each device editor is responsible for extracting the MPEVENT structures out of the queue and the processing the MIDI messages appropriately.

The IMPEventQueue interface provides methods that allow MIDI events to be filtered out by a port ID. Each device editor is set up to listen on a specific MIDI port for incoming MIDI messages and the IMPEventQueue has methods to select only those MIDI events that came from a specific port. If, for example, a sound engineer has connected the MIDI output port of a Yamaha 01V96 Digital Mixing Console to the ninth MIDI input port of a computer, and has selected that the Matrix Mixer listen on the ninth MIDI input port of the computer, the filter is set to the port ID of the ninth MIDI input port of the computer. This avoids device editors from processing MIDI messages unnecessarily and avoids any unnecessary data copying.

5.2.4. Send MIDI Events

When each device editor's `MPConnect` method of the `IMPInitialise` interface is called to initialise each device editor, Studio Manager 2 passes in a pointer to an `IMPAsyncOutput` interface (see Listing 12 above). Each software device editor retains a copy of this interface pointer, and through the `MPOutput` method of this interface each device editor is able to communicate with its associate hardware device. The declaration of this method is shown in Listing 18.

```
HRESULT MPOutput(REFMPEVENT rEvent, MPASYNC_MODE eMode);
```

Listing 18: The `IMPAsyncOutput::MPOutput` method

Each device editor is able to transmit MIDI messages through the `MPOutput` method. The specific MIDI and MIDI port data is contained in `MPEVENT` structures which the `MPOutput` method takes as an argument. The hosting DAW application receives these and sends the MIDI messages to the associated hardware device via the specified MIDI port.

5.2.5. Performing Total Recall

The `IMPAsyncOutput` and the `IMPEventFilter` interfaces allow for MIDI messages to be sent between the software device editors and their associated hardware devices. These lay the foundation for Total Recall operations to be performed on hardware devices via software device editors as they allow for the communication of state information.

The `IMPServices` interface provides methods that allow for the automation of a device editor's Total Recall operations. A device editor that wishes to support Total Recall has to implement the methods provided by the `IMPServices` interface.

Total Recall occurs when either the state of a software device editor is transferred across to its associated hardware device, or the state of a hardware device is transferred across to its associated software device editor. This state transfer takes place via MIDI messages. With Studio Manager 2, it is possible to save the state of the device editors in its workspace (see section 5.2.6 below). It is then possible to later reload that workspace, and hence the software device editors, to their previous state. It is then possible to transfer the state from the software device editors across to their associated hardware devices.

Each device editor's Total Recall functionality is controlled by Studio Manager 2 via the `MPRecall` method of the `IMPServices` interface. Each device editor provides an implementation of this method, which allows for the transfer of device state. Listing 19 shows the method declaration for the `MPRecall` method. Studio Manager 2 is able to query a device editor for a pointer to its `IMPServices` interface. Once obtained, it is able to request a device editor to perform Total Recall by calling its `MPRecall` method. Studio Manager 2 is able to control the direction of the state transfer through the use of the `bUpload` parameter. Passing in a value of `true` indicates to the device editor that it should transfer its state to its associated hardware device and a value of `false` indicates that the state of the hardware device should be transferred to the software device editor.

```
HRESULT MPRecall (const bool bUpload, IMPClientProgressHandler *
progressHandler, DWORD details = 0x0000);
```

Listing 19: The `IMPServices::MPRecall` method

Figure 49 shows an example of how Total Recall is performed. This example demonstrates how the state of a software device editor is transferred across to its associated hardware device. A DAW application may request that Total Recall be performed on a set of hardware devices. This may happen after the DAW application has been loaded from one of its native song files. Studio Manager 2 queries the user for the direction of state transfer. If the user requests that the state transfer take place to the hardware devices, Studio Manager 2 requests that each of its device editors transfer the state of their parameters across to their associated hardware devices. The state transfer takes place in the form of MIDI messages. Each device editor sends out MIDI messages that represent the state of its parameters to its associated hardware device.

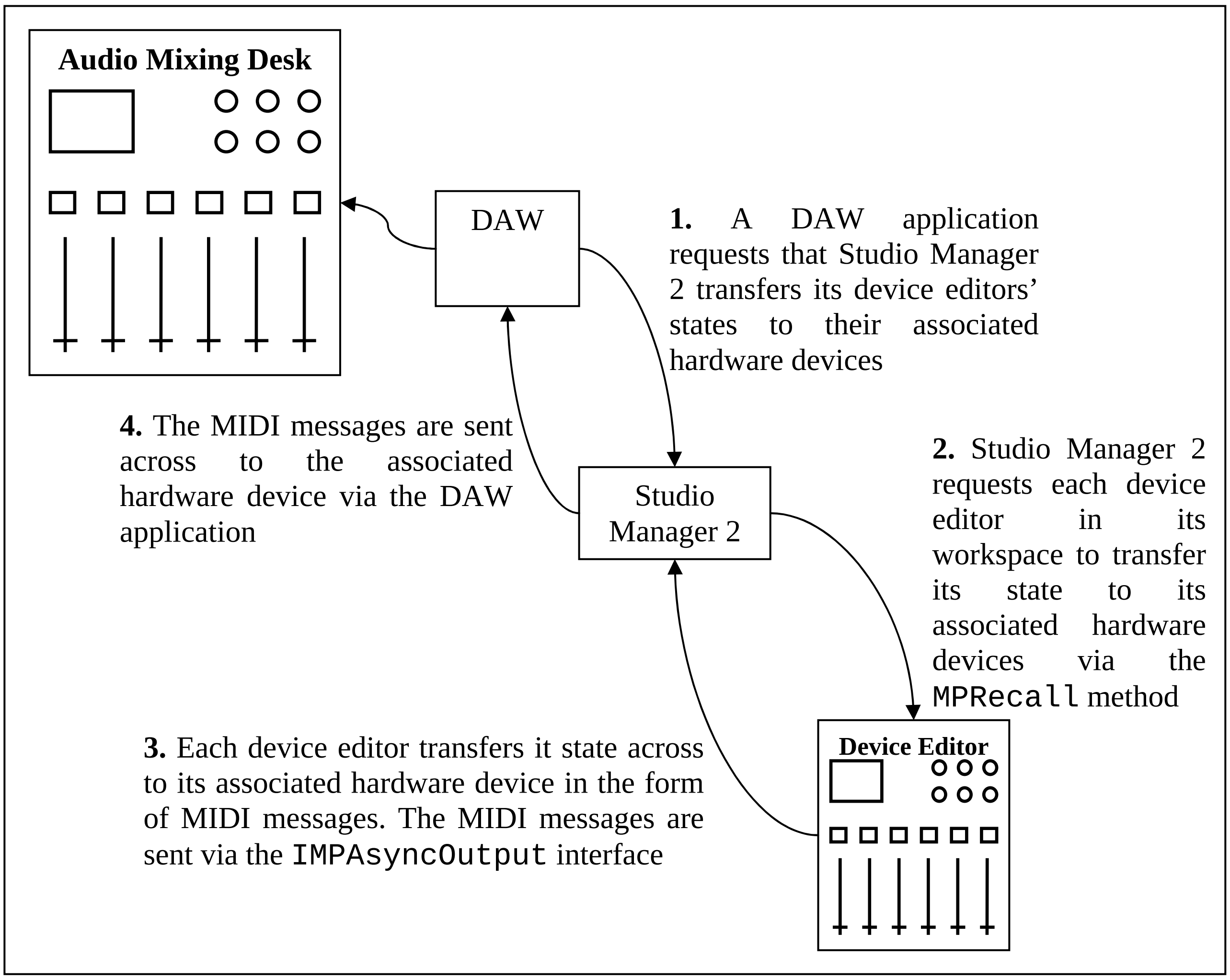

Audio Mixing Desk
DAW
1. A DAW application requests that Studio Manager 2 transfers its device editors' states to their associated hardware devices
2. Studio Manager 2 requests each device editor in its workspace to transfer its state to its associated hardware devices via the MPRecall method
3. Each device editor transfers it state across to its associated hardware device in the form of MIDI messages. The MIDI messages are sent via the IMPAsyncOutput interface
4. The MIDI messages are sent across to the associated hardware device via the DAW application
Studio Manager 2
Device Editor

Figure 49: An example of how state is transferred from a software device editor to its associated hardware device

When the Matrix Mixer's `MPRecall` method is called and it is requested to send the state of its parameters across to the associated audio mixing desk, the Matrix Mixer iterates through each of its parameter objects and sends out the MIDI messages that represent the state that the parameters are in. When the Matrix Mixer is requested to transfer the state from the hardware audio mixing desk to the software audio mixing desk editor, it iterates through each of its parameter objects and sends out the parameter request MIDI messages associated with each parameter object to the associated audio mixing desk. The hardware audio mixing desk then responds with parameter response MIDI messages, which the Matrix Mixer uses to set the values of its parameter objects.

Via the `IMPServices MPRecallFeatureSet` method, a software device editor is able to provide a list of available Total Recall feature sets that it has. It is able to define different granularities of Total Recall. A sound engineer is able to select a specific granularity when Total Recall is performed. For example, a device editor may define two Total Recall granularities. One to specify that all its parameters should have Total Recall performed on them, and another one that specifies that Total Recall should only be performed on the volume parameters of the device. When a device editor is asked to perform Total Recall, the Total Recall granularity is specified as well. The Matrix Mixer does not define any Total Recall granularity features. It performs Total Recall on all of its parameters.

5.2.6. Loading And Saving Audio Mixer State

When a sound engineer saves a project s/he is working on within a DAW application that is hosting Studio Manager 2, the state of the device editors that are part of Studio Manager 2's workspace are saved as well. The DAW application will request that Studio Manager 2 save itself to a supplied serial stream. Studio Manager 2 then in turn requests each device editor to save itself to the supplied stream.

Each device editor is capable of saving and reloading its state to a serial stream with the aid of Microsoft's `IPersistStream` interface. Studio Manager 2 may obtain a pointer to each device editor's `IPersistStream` interface and request that a device editor save or load itself to or from a stream via the `Save` and `Load` methods respectively. Each device editor provides an implementation of each of these methods and is free to save and load its required data in a format that suits it. The `Save` method declaration is shown in Listing 20.

```
HRESULT Save(IStream * pStm, BOOL fClearDirty);
```

Listing 20: The `IPersistStream::Save` method

The `Save` method is used to save an object to a specified stream. When Studio Manager 2 wishes to save the state of the device editors in its workspace, it calls this method of each device editor and passes in a pointer to an `IStream` interface to which each device editor may save itself. The `Write` method of the `IStream` interface is used to write data to the stream. This method declaration is shown in Listing 21.

```
HRESULT Write(void const* pv, ULONG cb, ULONG* pcbWritten);
```

Listing 21: The `IStream::Write` method

When the `Save` method is called, the Matrix Mixer saves its state to the provided stream in the following format:

Data	Size
`version`	`unsigned long`
`MIDI input port GUID`	`GUID`
`MIDI output port GUID`	`GUID`
`MIDI input port channel`	`int`
`MIDI output port channel`	`int`
`number of MIDI bytes`	`unsigned long`
`MIDI bytes`	

Table 5: Saved state format, version one

The `version` field is used to keep track of which format version the stream was saved in. This allows the Matrix Mixer to extract the saved data correctly. This provides backward compatibility because it allows the format of the saved data to be changed over time, whilst allowing the Matrix Mixer to work with previous formats.

Version one has the following format:

- The `MIDI input port GUID` field is used to save the GUID of the currently open MIDI input port.
- The `MIDI output port GUID` field is used to save the GUID of the currently open MIDI output port.
- The `MIDI input port channel` and `MIDI output port channel` fields are used to save the channels that the MIDI input port listens on and the MIDI output port sends data on, respectively.
- The `number of MIDI bytes` field is used to save the length of the parameter state data (the MIDI messages that represent the state of the parameters).
- The rest of the stream has the actual MIDI bytes that represent the state of the parameters of the Matrix Mixer.

Each device editor's state data may then be passed to the DAW application hosting Studio Manager 2 which may then be saved to one of its native song files, along with a sound engineer's project data.

If a DAW application reloads a native song file to which the device editors' states were written, it is able to extract that state data and pass it back to Studio Manager 2. Studio Manager 2 is then able to use the state data to restore its workspace, and the device editors that are part of the workspace. The `Load` method is used to initialise an object from a stream where it was previously saved. The caller of this method passes in a pointer to an `IStream` interface, from which the object can read data to initialise itself. This method declaration is shown in Listing 22.

```
HRESULT Load(IStream * pStm);
```

Listing 22: The `IPersistStream::Load` method

`IStream`'s `Read` method is used to read data out of the stream. The `Read` method declaration is shown in Listing 23. For the Matrix Mixer, the data is read out of the stream in the format shown in Table 5 above.

```
HRESULT Read(void* pv, ULONG cb, ULONG* pcbRead);
```

Listing 23: The `IStream::Read` method

When the Matrix Mixer's implementation of the `Load` method is called, the Matrix Mixer initialises itself from the supplied stream. The initialisation process is as follows:

- Initially, the Matrix Mixer reads the MIDI input and output port GUID's out of the stream and opens up the appropriate MIDI interfaces.
- Next, the MIDI input and output port channels are read out of the stream and set appropriately.
- Then, the stored MIDI bytes are read out of the stream and used to set the values of the specific parameter objects. The MIDI messages that are read out of the stream are sent to the Matrix Mixer via the same mechanism as the MIDI messages that are sent via a hosting DAW application.

5.3. Summary

This phase of the project set out to explore the use of the Studio Connections Total Recall SDK. This SDK allows the Matrix Mixer to be incorporated into DAW applications and thereby support Studio Connections Total Recall functionality. Studio Connections Total Recall is built on the COM architecture, which allows device editors to be implemented as software plug-ins and thus be dynamically loaded at runtime as and when required. The Studio Connections Total Recall SDK defines a set of interfaces that allows for device editors to:

- Be initialised.
- Be displayed.
- Transmit and receive MIDI messages.
- Have their state saved and loaded.
- Transfer state between themselves and their associated hardware devices.

An implementation of these interfaces to allow Studio Manager 2 to communicate with the Matrix Mixer was created. This has allowed the Matrix Mixer to provide state information to a DAW application hosting Studio Manager 2 and has allowed for the transfer of state between itself and its associated hardware audio mixing desk.

Chapter 06

Double Grid FireWire Patching

Legacy audio studios use hardware patchbays in order to patch audio signals between devices in an audio studio. The cable clutter in these studios increases rapidly as devices are added to the studio configuration. mLAN was created to, amongst other things, reduce this cable clutter by transporting audio and control signals over FireWire networks. Patching signals between devices on an mLAN network is performed via software patchbays that run on computers connected to the network.

In this chapter, we compare the configuration of a small legacy audio studio to a small mLAN studio and compare the patching techniques used in these two types of studio. We will compare the different types of software patchbays that are used to patch signals between mLAN devices and propose a FireWire patchbay to simplify the patching between mLAN devices.

6.1. Legacy Audio Systems Incorporating Hardware Patchbays

In legacy audio studios, hardware patchbays allow sound engineers to access audio signals at particular strategic points within an audio system [Robjohns, 1999]. Audio signals within an audio system are all sent to a common location (the hardware patchbay) and from there routed to their destination points. This allows for the routing configurations of an audio studio to be changed easily. There is no need to re-run cables between devices each time a routing configuration needs to be changed.

In typical legacy audio studios, the wires from the audio devices in the studio are connected to jack sockets on a patchbay. These patchbays are wired for a typical configuration. Figure 50 shows the back panel of a hardware patchbay. This panel contains two rows of jack sockets. In its typical configuration, each top jack socket is connected to the jack socket that is directly below it. In the figure, the two jack plugs plugged into the two top sockets (annotation 1 in the figure) are coming from an audio producing device (for example, a CD player). The signals entering these jack sockets are sent to the jack sockets directly below. An audio destination device (for example, an audio mixing desk) is plugged into these jack sockets (annotation 2 in the figure). The jack plugs shown as annotation 3 in the figure are also plugged into an audio destination device, but in the typical configuration there are no signals going to this device as there are no jack plugs plugged into the jack sockets directly above them.

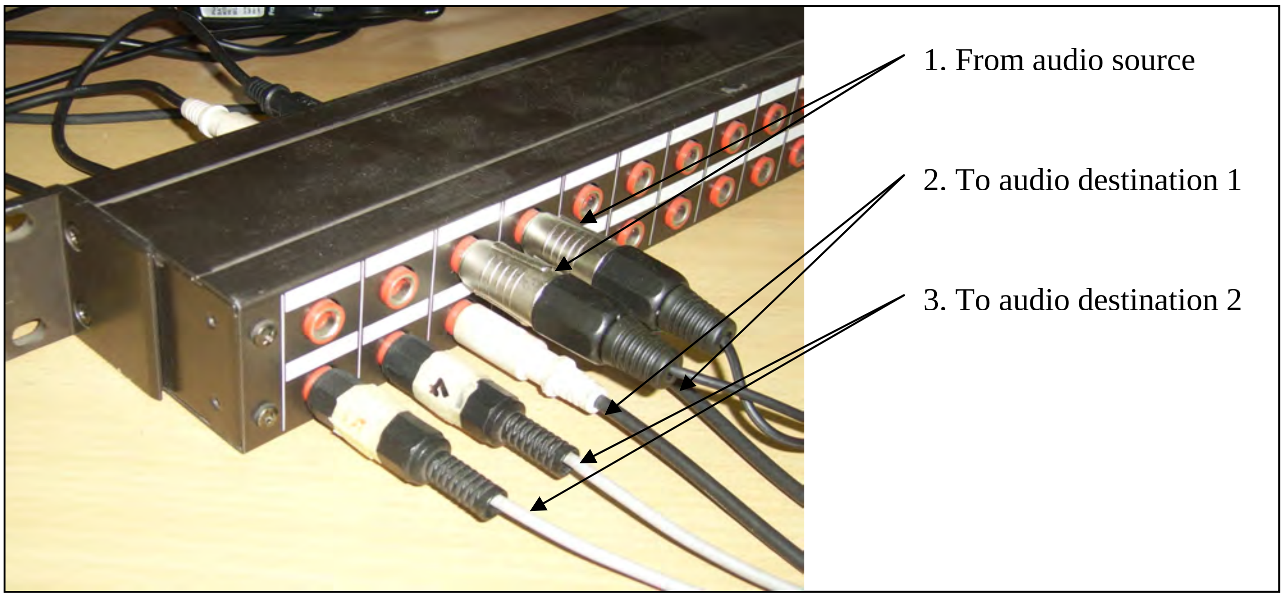


Figure 50: The back panel of a hardware patchbay

Figure 51 shows the front panel of the hardware patchbay shown in Figure 50. Via the front panel of this patchbay, a sound engineer is able to re-route the audio signals entering it. The top row of jack sockets allows a sound engineer to access the audio signals coming from audio source devices. The bottom row of jack sockets allow audio signals to be sent to the audio destination devices that are plugged into the bottom row of jack sockets on the back of the patchbay. To re-route an audio signal to a different location, a sound engineer would a plug patch cable into the socket where the signal sources exists. This may or may not interrupt the original signal flow, depending on the type of patchbay. The other end of the patch cable is inserted into the jack socket going to the destination point. This will interrupt the signal that is currently flowing to that destination point and create a new signal flow.

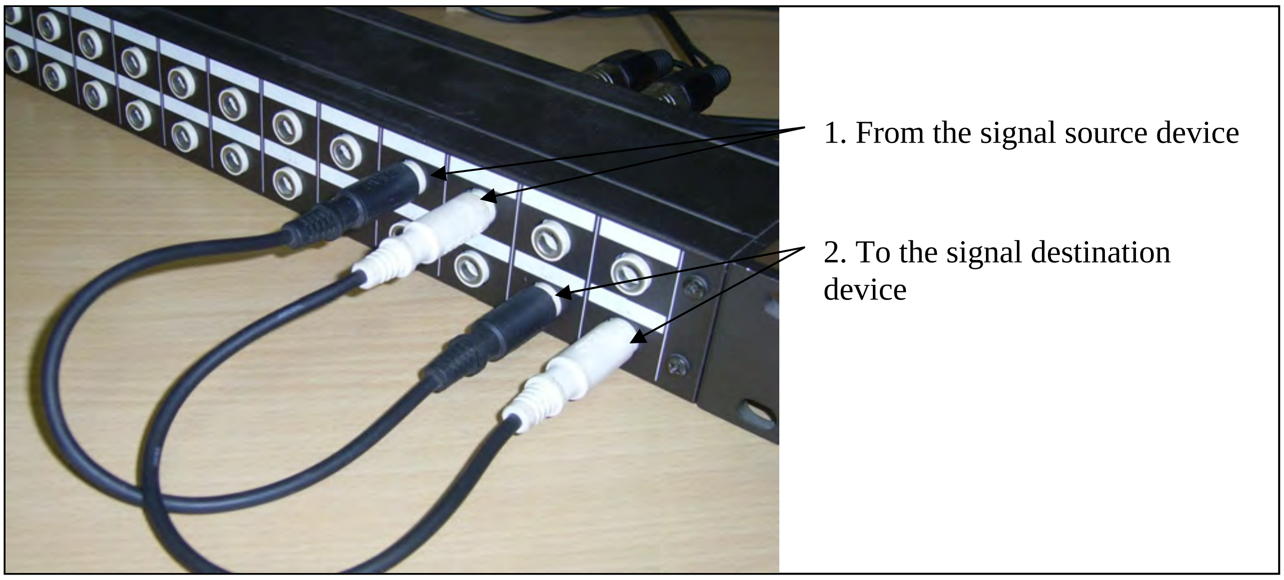

1. From the signal source device
2. To the signal destination device

Figure 51: Rerouting audio using a hardware patchbay

Figure 52 shows an example of a small studio audio system with a hardware patchbay. In this figure, it can be seen that all the devices in the studio have audio cables running between them and the patchbay. At the patchbay, the signals are routed to their destination devices. For example, the stereo audio signals from the synthesizer being sent to the patchbay may be routed through to two input channels on the computer. The output signals from the computer being sent to the patchbay may be routed through to the audio mixing desk. If a sound engineer wanted to send the signals from the synthesizer directly to the audio mixing desk, this would be configured via patch cables on the patchbay.

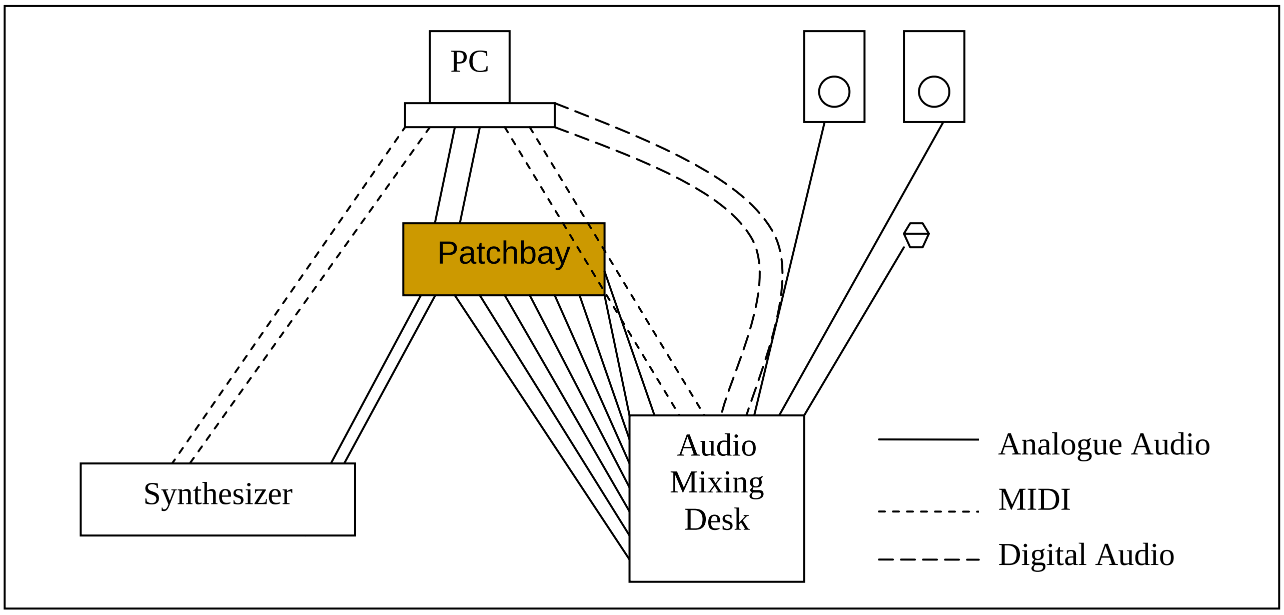


Figure 52: A small audio system with an analogue hardware patchbay

Hardware patchbays allow for routing configurations in audio studios to be changed relatively easily. All the devices in the studio are wired up to a patchbay, and semi-permanent routing configurations set up. Routing configurations are changed from one central location with patch cables. There is no need to run cables from source devices to destination devices each time a configuration needs to be changed.

6.2. Digital Audio Networking With mLAN

mLAN (music Local Area Network) [Fujimori and Foss, 2003] is a networking technology that allows for the transportation of audio and control data between audio devices present on the network, and is built on the IEEE1394 [Anderson, 1999] (FireWire) standard. FireWire is a serial bus architecture that allows nodes to transmit data between each other in a peer-to-peer fashion, without the need for the intervention of a host system.

If we consider the example legacy audio studio given in Figure 52, it is apparent, even in a small studio set up, that there are a lot of cables that are used to transport audio and control signals (MIDI) between the devices that comprise an audio studio. Each individual audio signal has its own set of cables, and own dedicated signal path from source device, to destination device. There are different types of cable that are used to transport the audio and control signals. The audio may be transmitted in analogue form or in a variety of digital forms. Each type of transmission standard may require a different type of cable and connectors.

If we replace all the cables shown in Figure 52 with one FireWire cable, we are left with a situation as depicted in Figure 53. In this figure, the devices in the audio system are daisy chained together with FireWire cabling to form a network of audio devices. Between each device exists only one piece of cable, and through this cable each device may send audio and control signals to the other devices present on the network. All the devices on the network share a cable, and routes are *not* determined by the physical path of the cable.

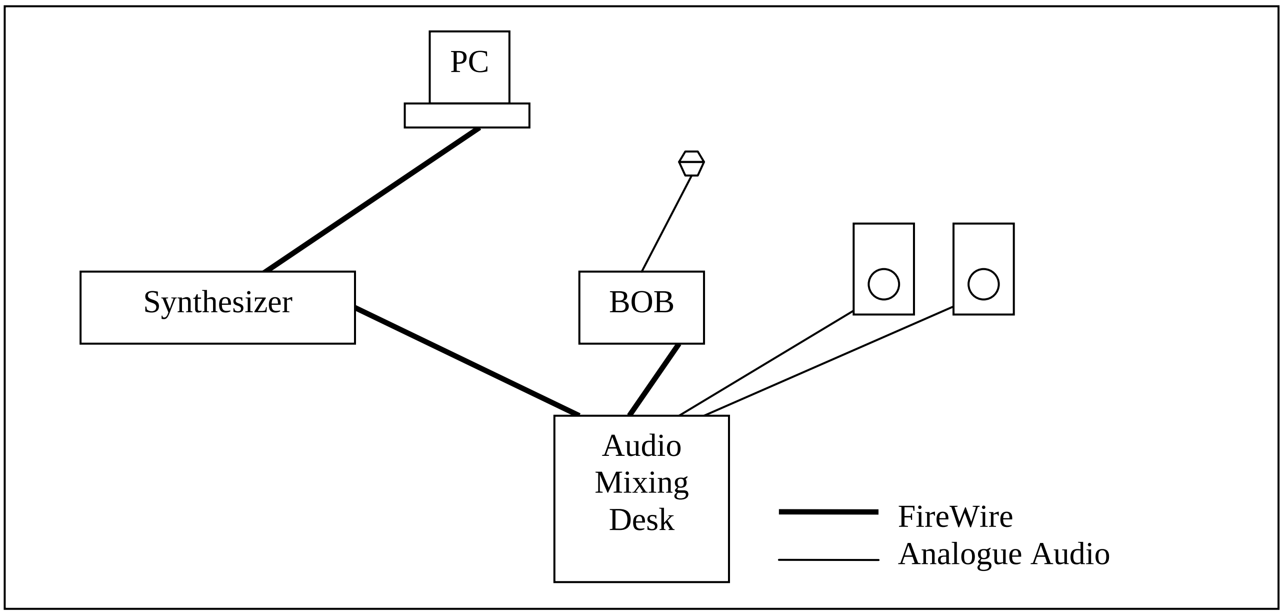


Figure 53: A small studio audio system connected via mLAN

Traditionally, each device in an audio studio may have a number of physical input and output sockets on it, through which audio signals are sent and received. For example, if a sound engineer wanted to send an audio signal to the first input channel of an audio mixing desk from the output of a synthesizer, it would require that a cable be plugged from the output of the synthesizer, to the dedicated socket on the audio mixing desk for the first input channel (this may also happen via a hardware patchbay). With mLAN, these physical plugs (‘hard plugs’) have been replaced with software plug abstractions (‘soft plugs’). The soft plugs of the devices on the FireWire network are revealed to a sound engineer via software patchbay applications that run on computers. In order to allow audio and control signals to flow between specific soft plugs on specific devices, connections in software (‘soft connections’) are established via these software patchbays.

The structure of a FireWire network is hierarchical in nature. On a small FireWire network with a few devices, the network could consist of a single FireWire bus. A single FireWire bus may contain up to a maximum of 63 devices. In order to add more than 63 devices to a FireWire network, the network needs to be broken down into a number of FireWire busses. The FireWire busses are joined together with FireWire bridges. Thus, FireWire networks form structures as depicted in Figure 54. A FireWire network consists of one or more FireWire busses. Each FireWire bus may consist of up to 63 FireWire devices. mLAN devices may each have a number of soft plugs. These soft plugs are used to transmit and receive audio and control signals across the FireWire network to and from particular mLAN devices.

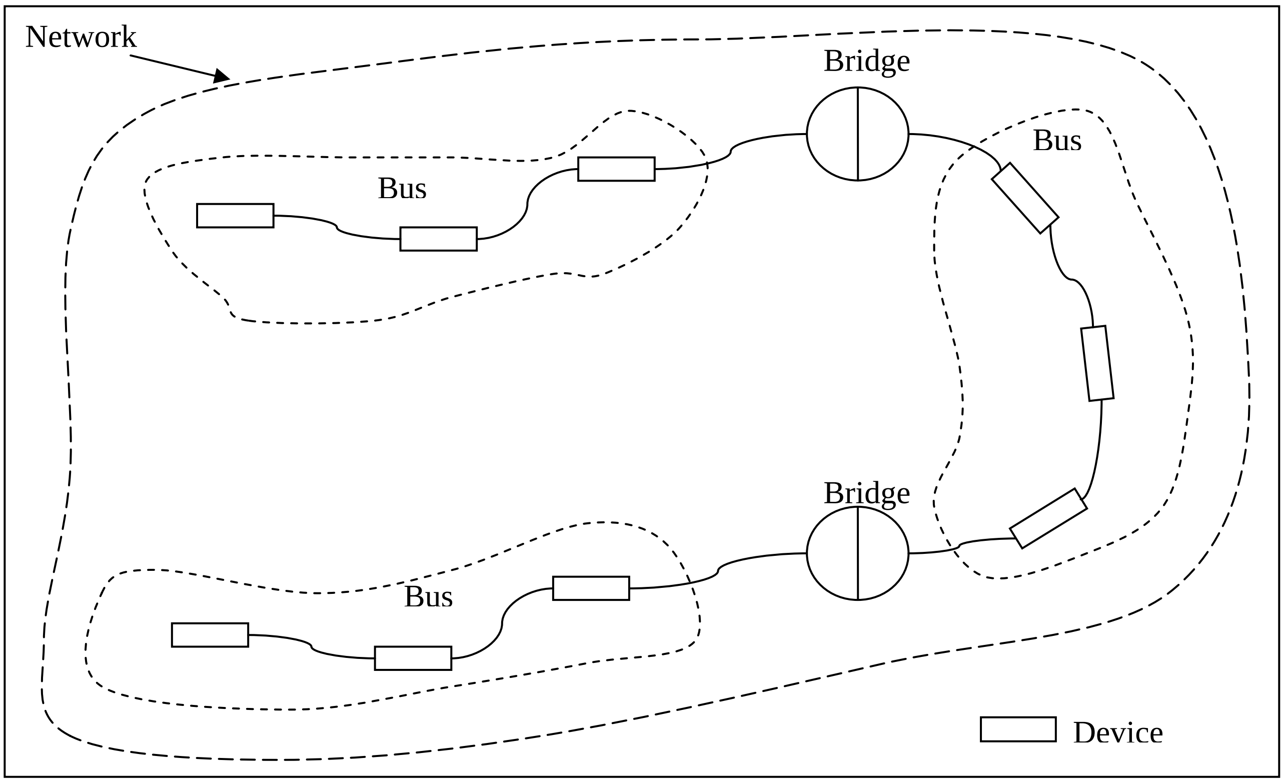


Figure 54: The hierarchical structure of a FireWire network

6.3. Software Patchbays

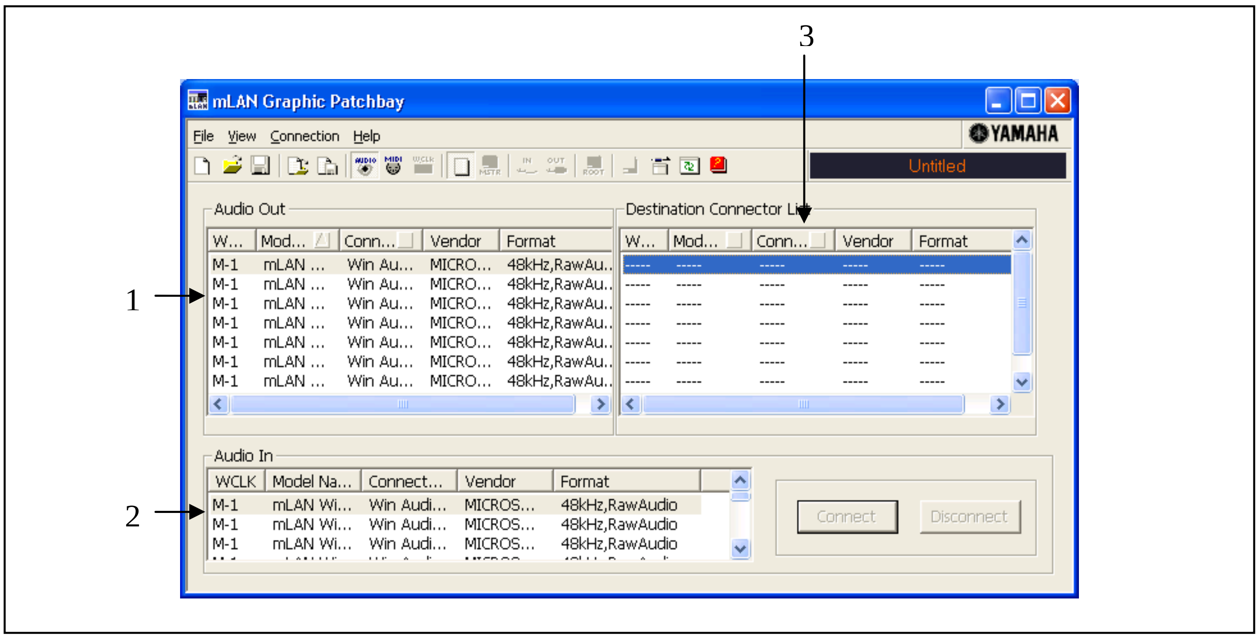
In digital audio networks, such as mLAN networks, a software patchbay is often used to configure audio routing between audio devices on the network. These patchbays display the devices on the network, along with their associated input and output soft plugs, and allow sound engineers to make and break soft connections between the soft plugs of the devices.

There are a range of software patchbay types which include list-based patchbays, tree-view-based patchbays, tree-grid-based patchbays and graphic-based patchbays.

6.3.1. List-Based Patchbays

List-based patchbays display the devices on an audio network along with their soft plugs in the form of lists. Sound engineers are able to select the source and destination soft plugs from the lists and make soft connections between the two. Figure 55 shows the mLAN Graphic Patchbay’s List View [Yamaha, 2004c]. This window shows the mLAN devices present on a FireWire network, their various audio and MIDI soft plugs, and the connection settings of the devices. The annotations in the figure are explained below:

1. This list displays the various mLAN devices on the FireWire network, and the soft output plugs present on those devices. These are the various signal sources present on the network.
2. This list displays the various mLAN devices on the FireWire network along with the soft input plugs present on those devices. These are the various signal destination points that exist on the network.
3. This list displays the soft connections between the soft plugs of the mLAN devices on the FireWire network.



Audio Out

W...	Mod...	Conn...	Vendor	Format
M-1	mLAN ...	Win Au...	MICRO...	48kHz,RawAu..
M-1	mLAN ...	Win Au...	MICRO...	48kHz,RawAu..
M-1	mLAN ...	Win Au...	MICRO...	48kHz,RawAu..
M-1	mLAN ...	Win Au...	MICRO...	48kHz,RawAu..
M-1	mLAN ...	Win Au...	MICRO...	48kHz,RawAu..
M-1	mLAN ...	Win Au...	MICRO...	48kHz,RawAu..
M-1	mLAN ...	Win Au...	MICRO...	48kHz,RawAu..

Destination Connector List

W...	Mod...	Conn...	Vendor	Format
-----	-----	-----	-----	-----
-----	-----	-----	-----	-----
-----	-----	-----	-----	-----
-----	-----	-----	-----	-----
-----	-----	-----	-----	-----
-----	-----	-----	-----	-----
-----	-----	-----	-----	-----

Audio In

WCLK	Model Na...	Connect...	Vendor	Format
M-1	mLAN Wi...	Win Audi...	MICROS...	48kHz,RawAudio
M-1	mLAN Wi...	Win Audi...	MICROS...	48kHz,RawAudio
M-1	mLAN Wi...	Win Audi...	MICROS...	48kHz,RawAudio

Figure 55: The Yamaha mLAN Graphical Patchbay's List View

Via this list-based patchbay, making soft connections between the various soft plugs of the mLAN devices is done by scrolling to the required signal source plug and selecting it (see annotation 1 in the above figure), scrolling to the required signal destination plug and selecting it (see annotation 2 in the above diagram) and then selecting the 'Connect' button.

The soft plugs of connected devices will be shown as having connections between them by displaying the destination device in the 'Destination Connector List' (see annotation 3 in the above diagram) next to the source plug.

6.3.2. Tree-View-Based Patchbays

Tree-view-based patchbays present the layout of the network in the form of tree structures. Figure 56 shows the NAS Explorer Patchbay [Chigwamba and Foss, 2007]. The tree views of this patchbay are organized in such a way as to represent the hierarchical nature of the FireWire network it is representing. This patchbay contains two tree-view structures: one to represent the signal source plugs available on the mLAN devices, and one to represent the signal destination plugs available on the mLAN devices.

The root node of each tree represents the entire FireWire network. Its child nodes represent the various FireWire busses that make up the network. Each node representing the various busses of the network has child nodes representing the mLAN devices present on the bus. Each node in the tree representing the devices on the busses has child nodes representing either the soft input plugs, or the soft output plugs of the devices, depending on the specific tree. This way, sound engineers can logically locate the require plugs by navigating their way down the tree structure.

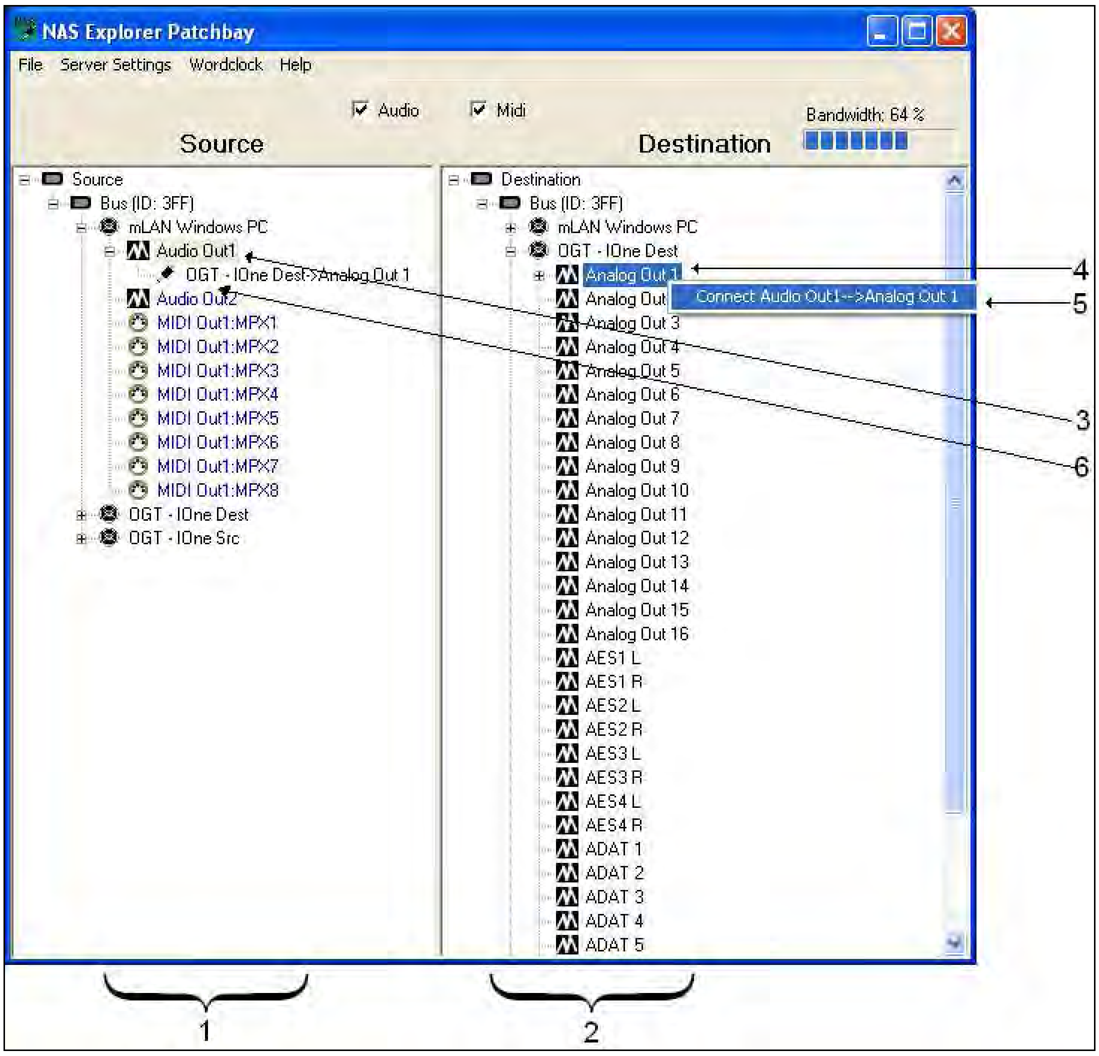


Figure 56: The NAS Explorer Patchbay

The annotations in Figure 56 are explained below:

1. This tree-view represents the structure of the FireWire network along with the mLAN devices and their soft signal source plugs, hierarchically.
2. This tree-view represents the structure of the FireWire network along with the mLAN devices and their soft signal destination plugs, hierarchically.
3. Making a soft connection between a soft signal source plug and a soft signal destination plug requires that a sound engineer first select the soft signal source plug.
4. The second stage for a soft connection requires selecting the soft signal destination plug.
5. Once the soft signal source and destination plugs have been selected, a soft connection between the two plugs is made by right clicking on one of the selected plugs, and selecting the ‘Connect’ menu item.

6. The NAS Explorer Patchbay displays a soft connection by making the connected soft plug a child node of the soft plugs involved in the connection. Disconnecting the soft plugs is performed by right-clicking this child node, and selecting the ‘Disconnect’ menu item.

6.3.3. Tree-Grid-Based Patchbays

Tree-grid-based patchbays display the devices on a network, along with their associated soft plugs, along the axes of grids. The cross points on the grids allow for soft connections between the soft plugs of devices to be made or broken by selecting or deselecting the points on the grids, respectively. Figure 57 shows the Routing Matrix of the Otari ND 20B mLAN Control Software [Otari, 2005]. This tree-grid-based patchbay displays Otari ND 20B units on an mLAN network, their associated soft plugs and the soft connections between them. The Otari ND 20B units, along with their soft plugs, are shown hierarchically on the axes of the grid with tree views. Making and breaking soft connections is performed by selecting and deselecting the cross points on the grid. The annotations in this figure are explained below:

1. The left hand column shows the names of the devices present on the network and the input channels available on each device, hierarchically.
2. The top row shows the names of the devices on the network and the output channels associated with each device, hierarchically.
3. The checkered section of the grid shows which output channels are routed through to which input channels. Making and breaking soft connections is performed by selecting and de-selecting the cross points on the grid where the required output channels intersect the required input channels.

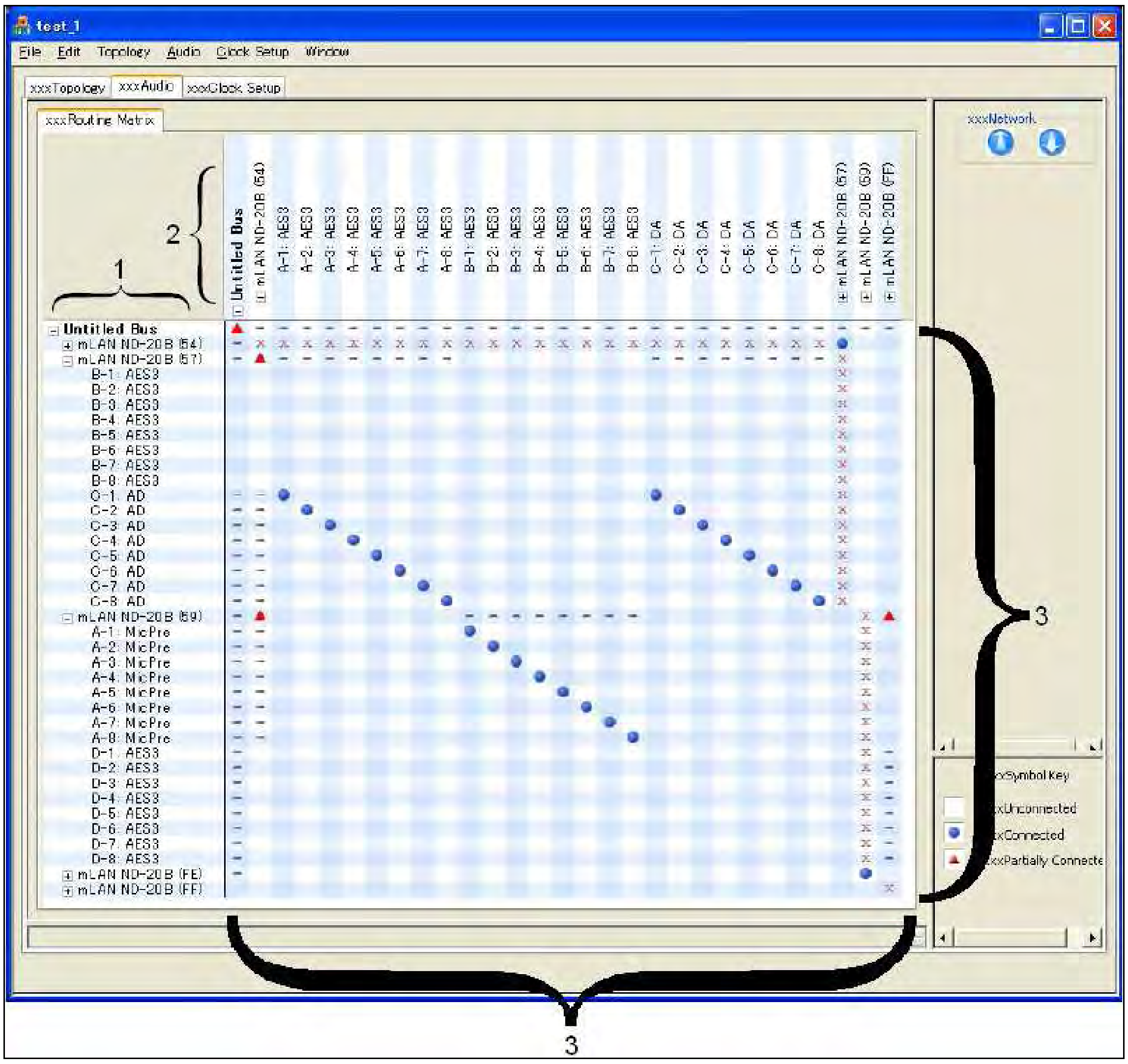


Figure 57: The Otari ND 20B mLAN Control Software Routing Matrix

6.3.4. Graphic-Based Patchbays

A graphic-based patchbay represents devices on an audio network with the aid of icons. Each icon represents a device and the soft plugs associated with it. The soft connections between the devices are represented with cable-like lines drawn between the plugs. The Yamaha Graphic Patchbay [Yamaha, 2004c] is an example of a graphic-based patchbay. This patchbay is used to make connections between mLAN devices on a FireWire network. The primary window of this patchbay is shown in Figure 58. The annotations in this figure are explained below:

1. The mLAN devices on the FireWire network are shown on the workspace with the aid of different coloured icons, and each icon may contain detail pertaining to the particular device.
2. The soft input and output plugs on each device are shown on the side of the icons that represent the different mLAN devices.

3. The soft connections between the devices are shown with graphic cable-like connectors. Different colours are used to distinguish the different virtual cables.
4. In order to make a soft connection between the soft plugs on two different devices, the ‘out’ section of one device, and the ‘in’ section of another device is selected. This displays two Connector windows that display the relevant plugs available on the selected devices. A connection is then made by selecting the required soft output plug, and then selecting the required soft input plug. Soft connections are then visually shown with graphical cables between the soft output plugs, and the soft input plugs.

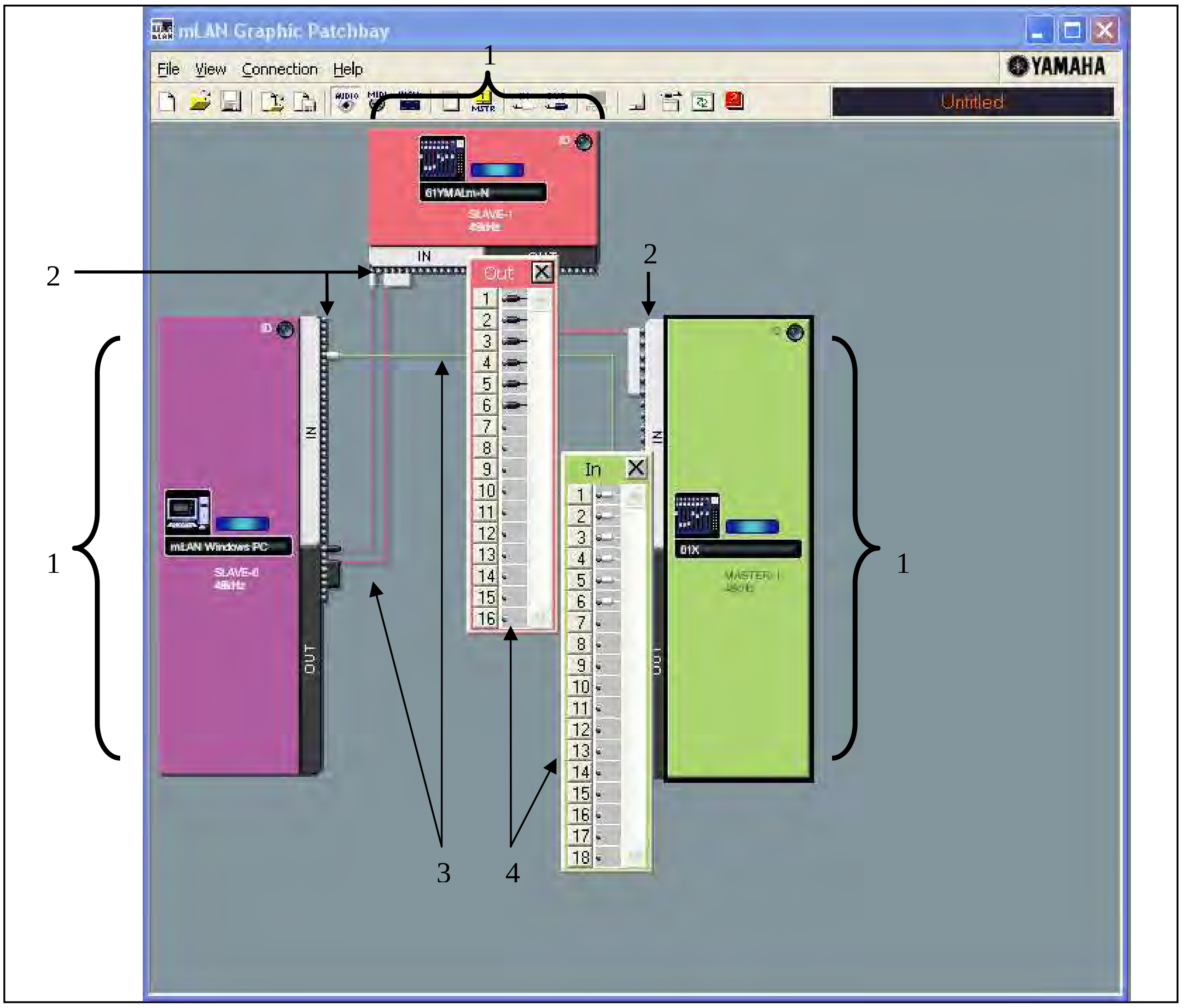


Figure 58: The Yamaha mLAN Graphic Patchbay

6.3.5. A Comparison Of Patchbays

As the number of devices on an audio network increases, the complexity of the patchbay representing the devices on the network may increase as well. Besides the list-based patchbay discussed above, all of the patchbays represent the network hierarchically. This hierarchical representation reflects the structure of the network itself. This makes navigating to required devices and soft plugs take place in a logical way. It also reduces the amount of clutter on the displays of the patchbays, as sound engineers have the option of only displaying the information that they are interested in.

Table 6 shows the number of mouse button clicks it would take a sound engineer to make a connection between two mLAN devices on a single bus FireWire network, using the patchbays discussed above. The table distinguishes between the number of mouse button clicks it takes to navigate to the required plugs and to make the connection.

	List-based	Tree-view-based	Tree-grid-based	Graphic-based
Navigate to plugs	0 (Scroll)	6	4	2
Make connection	3	4	1	2

Table 6: The number of mouse clicks to make a soft connection

With all the patchbays discussed above, a sound engineer is required to individually navigate to the required source and destinations soft plugs to make a soft connection.

The patchbays discussed above, except the tree-grid-based patchbay, require that sound engineers individually select the source and destination soft plugs before a connection can be made. Even though the list-based patchbay has the least number of mouse clicks to make a connection, the process of selecting the plugs could become tedious. The lists in which the plugs are listed may grow to the point where each list contains literally hundreds of individual items even for a small studio.

When making a connection with the tree-grid-based patchbay, there is no need to explicitly select the soft plugs required for a soft connection. The connection between the two soft plugs is performed by selecting the cross point on the grid that is used to represent the two plugs. The soft plugs are implicitly selected.

6.4. A Double Grid-based Patchbay

The abovementioned patchbays, apart from the list-based patchbay, represent the associated FireWire network in a hierarchical form. This hierarchical structure reflects the hierarchical structure of the actual FireWire network. Since it is a reflection of the network's structure, finding required soft plugs occurs in a logical manner. Sound engineers would first locate the FireWire bus that the required device is on, then locate the required mLAN device on that FireWire bus, and then locate the required soft plugs. The hierarchical approach also reduces the amount of clutter presented on the display. Sound engineers have the option of only displaying the soft plugs that are relevant to their needs.

It is also apparent that actually making soft connections between the required soft plugs of devices may involve many mouse button clicks. Besides the tree-grid-based patchbay, making soft connections between the soft plugs of devices involves selecting the source soft plug and selecting the destination soft plug, which then makes the connection or a connection between the selected plugs has to be requested.

This section describes a FireWire patchbay which is an extension of the grid-based patchbay. It uses two grids in order to perform connection management and has the following goals:

- Represent a FireWire network hierarchically.
- Ease the navigation to the required source and destination soft plugs on mLAN devices in order to make soft connections between them.
- Ease the process of making connections between the required source and destination soft plugs.

6.4.1. Representing Devices On A FireWire Network

The double grid-based patchbay represents the devices on the associated FireWire network hierarchically in the form of grids. Figure 59 shows the primary window of the double grid-based patchbay. The devices shown on this grid are the computer connected to the network, an evaluation board, and two breakout boxes. The annotations in this figure are explained below:

1. The tabs along the left-hand-side of the grid represent the source FireWire busses available on the associated FireWire network.

2. The tabs along the top of the grid represent the destination FireWire busses that are available on the associated FireWire network.
3. The labels along the left-hand-side of the grid represent the source mLAN devices that exist on the associated FireWire bus selected by the source bus tab.
4. The labels along the top of the grid represent the destination mLAN devices that exist on the associated FireWire bus selected by the destination bus tab.
5. The grid is used to display the soft plugs of the required source mLAN device and destination mLAN device. In order to display the soft plugs, the cross point on the grid where the required source mLAN device row intersects the required destination mLAN device column is selected. The plugs associated with these devices are displayed (see 6.4.2).

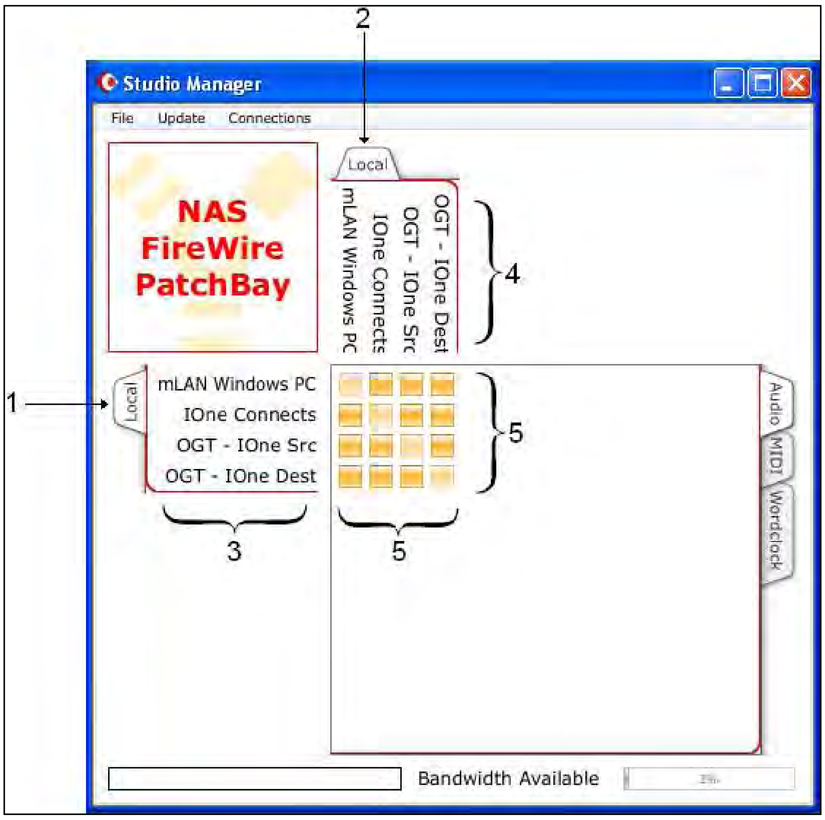


Figure 59: The double grid-based patchbay primary window

If a sound engineer is working in an environment where all the devices on the FireWire network are on a single FireWire bus, navigating to the required soft source and destination plugs of mLAN devices will always require a single mouse button click. This is because navigating to the required source and destination soft plugs is done by clicking on the cross point on the grid where the source device row intersects the destination device column. The grid approach alleviates the need to individually navigate to the required source and destination devices.

6.4.2. Representing The Soft Plugs Of mLAN Devices

When one of the cross points on the grid of the primary window is selected (see Figure 59 above), the source and destination soft plugs associated with the selected devices are shown. An example of a window that shows the source and destination soft plugs is shown in Figure 60. The particular window shown is displayed when selecting the cross point between the ‘mLAN Windows PC’ and ‘OGT – IOne Source’ labels. The annotations in this figure are explained below:

1. The label along the left-hand-side of the grid shows the source mLAN device that was selected from the primary window.
2. The label along the top of the grid shows the destination mLAN device that was selected from the primary window.
3. The labels along the left-hand-side of the grid show the source soft plugs that are associated with the selected source mLAN device.
4. The labels along the top of the grid show the destination soft plugs that are associated with the selected destination mLAN device.
5. Making connections between the soft plugs of the selected devices involves selecting the cross points on the grid where the rows of the source soft plugs intersect the columns of the destination soft plugs. Similarly, breaking a connection involves de-selecting the cross point on the grid where the source and destination soft plug labels intersect on the grid. If, for example, a sound engineer wanted to route the audio signal from the ‘Audio Out8’ output of the ‘mLAN Windows PC’ device to the ‘AES1 L’ input of the ‘OGT – IOne Source’ device, the cross point between these two soft plug labels would have to be selected, as shown with the circle in the diagram.

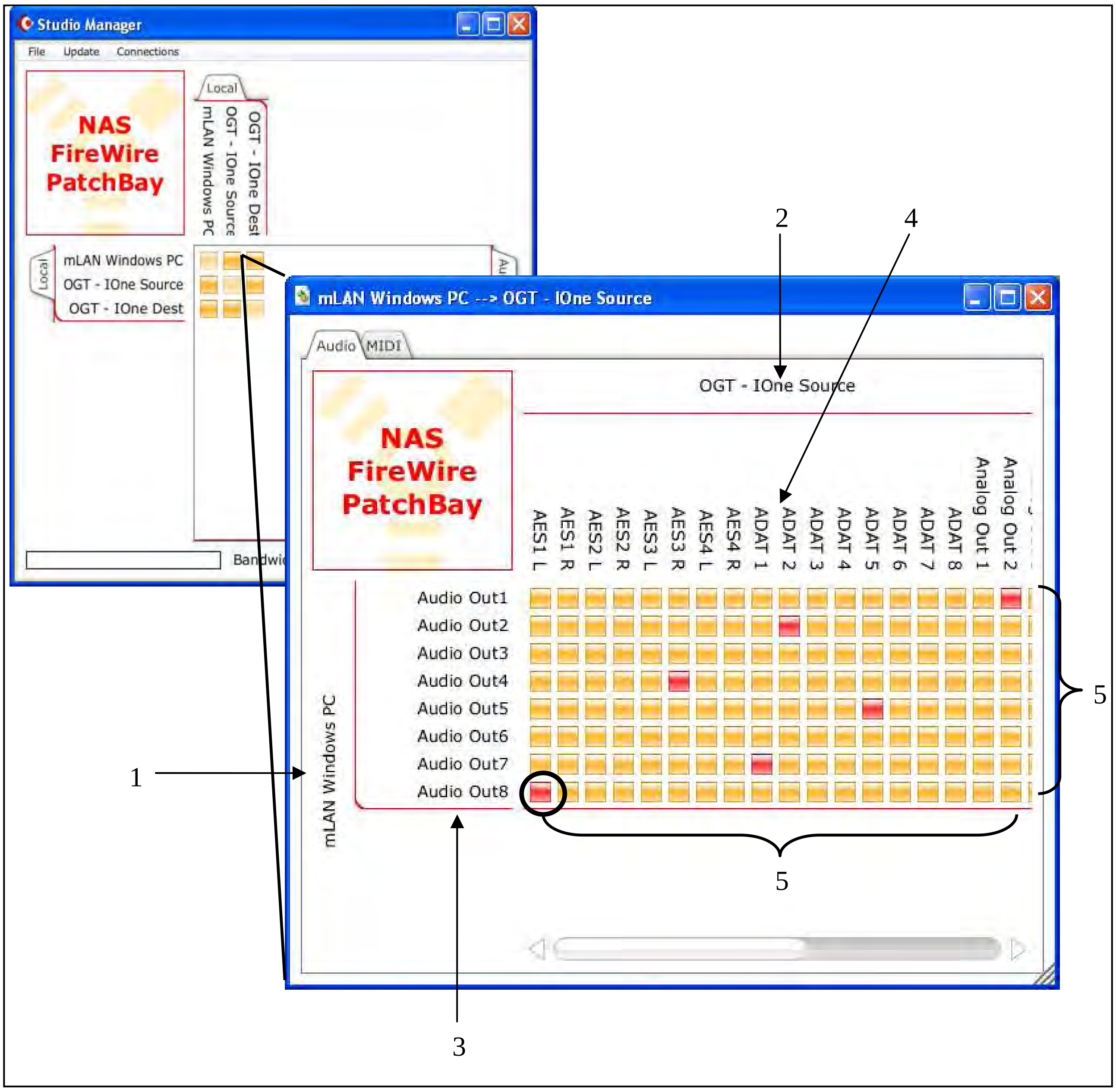


Figure 60: The double grid-based patchbay selected source and destination soft plugs window

On this patchbay, an active soft connection is shown with a red button, and an inactive soft connection is shown with an orange button.

6.4.3. Summary Of The Double Grid-Based Patchbay

The double grid-based patchbay approach to representing a FireWire network is hierarchical in nature. This allows for mLAN devices and their associated soft plugs to be located logically. This approach avoids clutter as the patchbay only displays the devices, and their associated soft plugs, that the sound engineer is interested in.

With the patchbays mentioned in section 6.3, locating source and destination soft plugs happen separately: First, the source device's source soft plugs are located, and then the destination device's destination soft plugs. With the double grid-based approach to patching, once the correct busses have been selected via the tabs, locating the required source and destination soft plugs is performed by selecting the cross point on the grid where the source device label intersects the destination device label. This action requires a single mouse button click.

Apart from the tree-grid-based graphic patchbay, making connections on the patchbays mentioned in section 6.3 happened by first selecting the source soft plug and selecting the destination soft plug. Making the connection either happens after the plugs have been selected, or a connection between the two selected soft plugs has to be explicitly requested. With the double grid-based patchbay shown in Figure 60, making a connection between the required source and destination soft plugs requires that the cross point between source soft plug label and the destination soft plug label be selected. Patching with a grid-based patchbay requires one mouse button click per connection.

In total, making a single connection (on a signal bus FireWire network) with the double grid-based patchbay requires two mouse button clicks. When compared to the other approaches listed in Table 6, the number of mouse button clicks is reduced, yet it retains its hierarchical nature.

6.4.4. The Double Grid-Based Patchbay Architecture

The double grid-based patchbay is a client to the mLAN Connection Management Server (mCMS) [Fujimori, Foss, Klinkrant and Bangay, 2003b and Chigwamba, *et al*, 2007]. The mCMS has been developed to allow soft connections between the soft plugs of mLAN devices on a FireWire network to take place. The server runs on a computer that is connected to the FireWire network and client applications are able to connect to the server using the Internet Protocol. This architecture is shown in Figure 61.

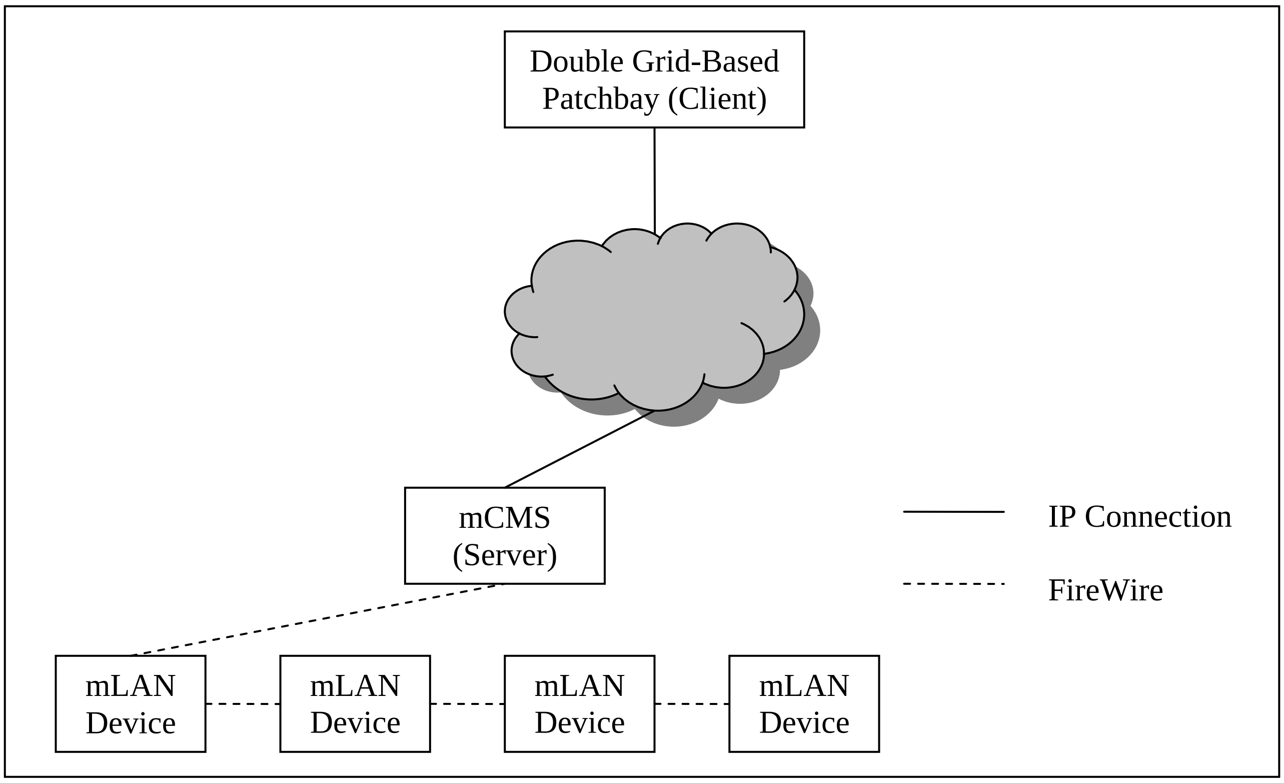


Figure 61: The mCMS architecture

Once a client application is connected to the server, it may request that the server make soft connections between the mLAN devices on the FireWire network connected to the server computer. The server may then perform the requested actions, or respond with error messages. Communication between a client and the server application happens via a defined collection of XML documents [Networked Audio Solutions, 2004]. A client application may send specific request XML documents to the server, and the server may respond with response XML documents.

6.4.5. Learning About The Network Configuration And Representing It

When the double grid-based patchbay application is started up, it is unaware of the configuration of the network that it is meant to represent. In order for it to be able to represent the network, the configuration has to be sent from the server at the request of the client application. Once the patchbay is connected to the server, a ‘refresh request’ XML document is sent to the server. This XML document is shown in Listing 24. This XML document requests the server to send the configuration of the network to the client application.

```
<mLANServerCommand version="1.0">
  <object name="patch" namespace="">
    <method name="refresh"/>
  </object>
</mLANServerCommand>
```

Listing 24: The mCMS refresh request XML document

The server responds with a ‘refresh response’ XML document which details the configuration of the network. This XML document describes:

- The FireWire network.
- The FireWire busses that comprise the network.
- The mLAN devices on each FireWire bus.
- The input and output audio and MIDI soft plugs present on each mLAN device.
- The various plug-layouts available to an mLAN device.
- The various wordclock synchronisation sources present on each mLAN device.
- The wordclock outputs that are present on each mLAN device.

Listing 25 shows a portion of the ‘refresh response’ XML document that is sent from the server to the client patchbay application detailing the configuration of the FireWire network. The actual configuration specifics are described with XML elements that are child elements to the `mLANConfiguration` XML element.

```
<mLANClientCommand>
      <object name="patch">
            <method name="refresh">
                  <parameter name="configuration" value="Tue Feb 27 11:10:43
                  2007">
                        <mLANConfiguration>

                              ...

                        </mLANConfiguration>
                  </parameter>
            </method>
      </object>
</mLANClientCommand>
```

Listing 25: A portion of a 'refresh response' XML document

Listing 26 shows the `IEEE1394Network` XML element. This XML element is a child to the `mLANConfiguration` XML element (shown in Listing 25 above). This XML element is used to describe the FireWire network that the server computer is attached to.

```
<IEEE1394Network>
      <IEEE1394Bus bandwidthAvailable="2756" busName="3FF">
            <IEEE1394Device GUID="0013f00400400011" firmware="DICE II OGT
            0.1" model="DICE II Evaluation Board" nickname="IOne Connects-
            left" nicknameIsWriteable="yes" numPossibleDeviceConnections="4"
            vendor="WaveFront">
                  <mLANDevice>
                        ...
                  </mLANDevice>
            </IEEE1394Device>
      </IEEE1394Bus>
</IEEE1394Network>
```

Listing 26: A portion of a 'refresh response' XML document used to describe a FireWire network

- The `IEEE1394Network` XML element is used to describe the FireWire network (attached to the server computer) as a whole.

- Each `IEEE1394Bus` XML element is used to describe a FireWire bus. It has XML attributes that allow the server to inform the client application how much bandwidth is available on the specific bus (`bandwidthAvailable`), and to provide it with a name (`busName`).
- Each of the `IEEE1394Device` XML elements is used to describe a device present on the relevant FireWire bus. The XML attributes associated with this XML element allow a Globally Unique Identifier (`GUID`), a name (`nickname`) and other information to be associated with the device.
- The `mLANDevice` XML element is used to describe information that is relevant to a particular mLAN device.

Listing 27 shows how the soft plugs that are associated with a particular mLAN device are described. Each soft plug (regardless of its type) is described with a `plug` XML element. The XML attributes associated with each `plug` XML element are used to describe properties associated with each soft plug. Included in these are XML attributes used to specify the direction of the soft plug (whether or not it is used to send signals, or receive them) (`direction`), the type of soft plug that it is (whether it is an audio plug, or a MIDI plug) (`plugType`) and a unique ID (`id`), amongst other properties.

```
<mLANDevice>
      <mLANDevicePlugs>
            <plug direction="out" id="0" isDangling="yes"
            nameIsWriteable="no" plugName="AES1 L" plugType="audio"/>
            <plug direction="out" id="1" isDangling="yes"
            nameIsWriteable="no" plugName="AES1 R" plugType="audio"/>
            ...
      </mLANDevicePlugs>
      ...
</mLANDevice>
```

Listing 27: A portion of a 'refresh response' XML document used to describe the soft plugs associated with an mLAN device

Each mLAN device may have a number of plug layouts associated with it. Plug layouts represent mutually exclusive configurations for a particular device. Each plug layout that is available to a device contains a set of different types of soft plugs and wordclock sources. mLAN devices may be capable of transmitting signals at various sample rates. As this sample rate increases, the number of soft plugs of the device may be reduced. In this type of situation, it could be useful to define two plug layouts; one for low sample rates, and another for high sample rates. The plug layout for low sample rates will contain the plugs that are available to the device when it is using a low sample rate (for example, the device may transmit on thirty two soft plugs), and the plug layout for high sample rates will contain the soft plugs that are available to a device when it is using a higher sample rate (for example, the device may only transmit on sixteen plugs).

Listing 28 shows how the plug layouts for an mLAN device are described in a ‘refresh response’ XML document. Each `plugLayout` XML element is used to describe a particular plug layout. The XML attributes associated with this XML element are used to associate a unique ID (`id`) and a name to each plug layout (`plugLayoutName`), amongst other properties.

```
<mLANDevice>
      ...
      <mLANDevicePlugLayouts currentPlugLayoutID="1" numPlugLayouts="2">
            <plugLayout id="0" nameIsWriteable="no"
            plugLayoutName="Low Sample Rate Pluglayout"/>
            <plugLayout id="1" nameIsWriteable="no"
            plugLayoutName="High Sample Rate Pluglayout"/>
      </mLANDevicePlugLayouts>
      ...
</mLANDevice>
```

Listing 28: A portion of a 'refresh response' XML document used to describe the plug layouts associated with an mLAN device

It is possible that each mLAN device on the FireWire network may have a number of wordclock outputs associated with it. A common wordclock is used to ensure that the devices on a FireWire network all use exactly the same sampling rate. If a sending and receiving device do not sample an audio signal at the same rate, the resulting audio signal could contain glitches as the receiving device may sample incoming audio too slowly or too quickly. The wordclock output of a particular device (master) is used to synchronise the wordclock outputs of other mLAN devices (slaves). Each device’s wordclock outputs are described using the `wordClockOutput` XML element, as seen in Listing 29. It has XML attributes that are used to:

- Describe the ID of the wordclock output (`id`).
- Associate it with a specific synchronisation source (`currentSyncSourceID`).

- Specify the master wordclock output ID (`masterWordClockOutputID`) and device GUID (`masterGUID`), if the wordclock output is acting as a wordclock slave.

```
<mLANDevice>
      ...
      <mLANDeviceWordClockOutputs numWordClockOutputs="1">
            <wordClockOutput currentSyncSourceID="1" id="0"
            masterGUID="0013f00400000011" masterWordClockOutputID="0"/>
      </mLANDeviceWordClockOutputs>
      ...
</mLANDevice>
```

Listing 29: A portion of a 'refresh response' XML document used to describe the wordclock outputs associated with an mLAN device

Each wordclock output may have various synchronisation sources. The wordclock may be provided internally by the device, or it may be generated by external devices. Each of these wordclock sources is described with a `syncSource` XML element, as seen in Listing 30. Each `syncSource` XML element has a number of XML attributes associated with it that allow for properties to be described. Amongst these are XML attributes to describe:

- A descriptive name (`syncSourceName`).
- The sample rate at which the synchronisation source is currently set to (`currentSampleRate`).
- An ID that is used to identify the synchronisation source (`id`).
- The sample rates that the synchronisation source is capable of supporting (`supportedSampleRates`).

```
<mLANDevice>
      ...
      <mLANDeviceSyncSources numSyncSources="2">
            <syncSource currentSampleRate="0000bb80" id="0"
            nameIsWriteable="no"
            supportedSampleRates="00007d00&#124;0000ac44&#124;0000bb80&#124;
            " syncMode="3" syncSourceName="Internal Clock"/>

            <syncSource currentSampleRate="0000bb80" id="1"
            nameIsWriteable="no"
            supportedSampleRates="00007d00&#124;0000ac44&#124;0000bb80&#124;
            " syncMode="1" syncSourceName="SYT Clock"/>
      </mLANDeviceSyncSources>
      ...
</mLANDevice>
```

Listing 30: A portion of a 'refresh response' XML document used to describe the synchronisation sources that may be associated with wordclock outputs

The `mLANConfiguration` XML element also has child elements to describe the connections that exist between the soft plugs of the mLAN devices on a FireWire network. These soft connections are described using `Patch` XML elements. Each `Patch` XML element gives the GUID (`NODE_GUID`) and the plug ID (`MLAN_PLUG_ID`) of the destination (`destEndPointLocator`) and source (`srcEndPointLocator`) device involved in the soft connection. Listing 31 shows a portion of a ‘refresh response’ XML document that describes the connections that exist between devices.

```
<Connections>
      <Patch destEndPointLocator=
      "NODE_GUID='0090270001b37283',MLAN_PLUG_ID='96'" srcEndPointLocator=
      "NODE_GUID='0013f00400000014',MLAN_PLUG_ID='0'"/>

      <Patch destEndPointLocator=
      "NODE_GUID='0090270001b37283',MLAN_PLUG_ID='97'"
      srcEndPointLocator= "NODE_GUID='0013f00400000014',MLAN_PLUG_ID='1'"/>
</Connections>
```

Listing 31: A portion of a 'refresh response' XML document used to describe the soft connections between mLAN devices

From the information contained within the 'refresh response' XML document, the double grid-based patchbay is able to learn about the configuration of the FireWire network that the server computer is connected to. The double grid-patchbay application is able to build up its internal object structure, as depicted in Figure 62, to reflect the FireWire network described in the 'refresh response' XML document. There exist objects that allow for the representation of the various components that make up a FireWire network. At the bottom of the figure, the associations between the specific plug objects represent the connections that may exist between them.

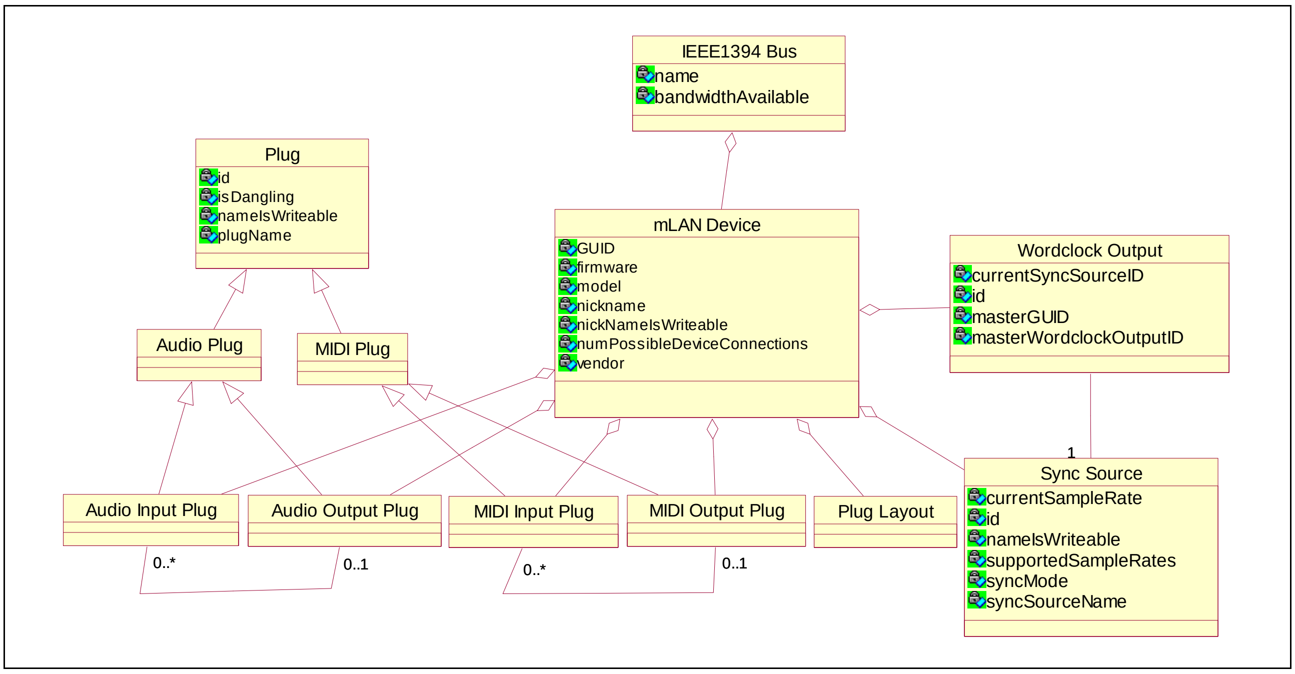


Figure 62: A portion of the double grid-based patchbay's object model

From the information contained within the internal objects of the system, the application is able to build up the patchbay displays to represent the FireWire network. This is shown via the following annotations in Figure 63:

1. The `IEEE1394 Bus` objects are used to represent FireWire buses. For each `IEEE1394 Bus` object that the `IEEE1394 Network` object has as a child object, a tab is added to the devices window to represent these busses.
2. Each `mLAN Device` object is used to represent an mLAN device on a FireWire bus. For each mLAN device that is part of a particular FireWire bus, a label is added to the source devices list, and the destination devices list. In the figure below, the 'Local' FireWire bus has three different devices on it.
3. The various soft plugs that each device has are represented with the specific instances of the `Plug` class. On the window that allows for connections to be made between the soft plugs of

devices, the source soft plugs of the selected device are shown as labels along the left-hand-side of the routing matrix.

4. The destination soft plugs are shown as labels along the top of the routing matrix.
5. In the object model, a connection between two soft plugs is represented as an association between the two specific plug objects. Visually, these connections are represented by activating the relevant buttons on the routing matrix.

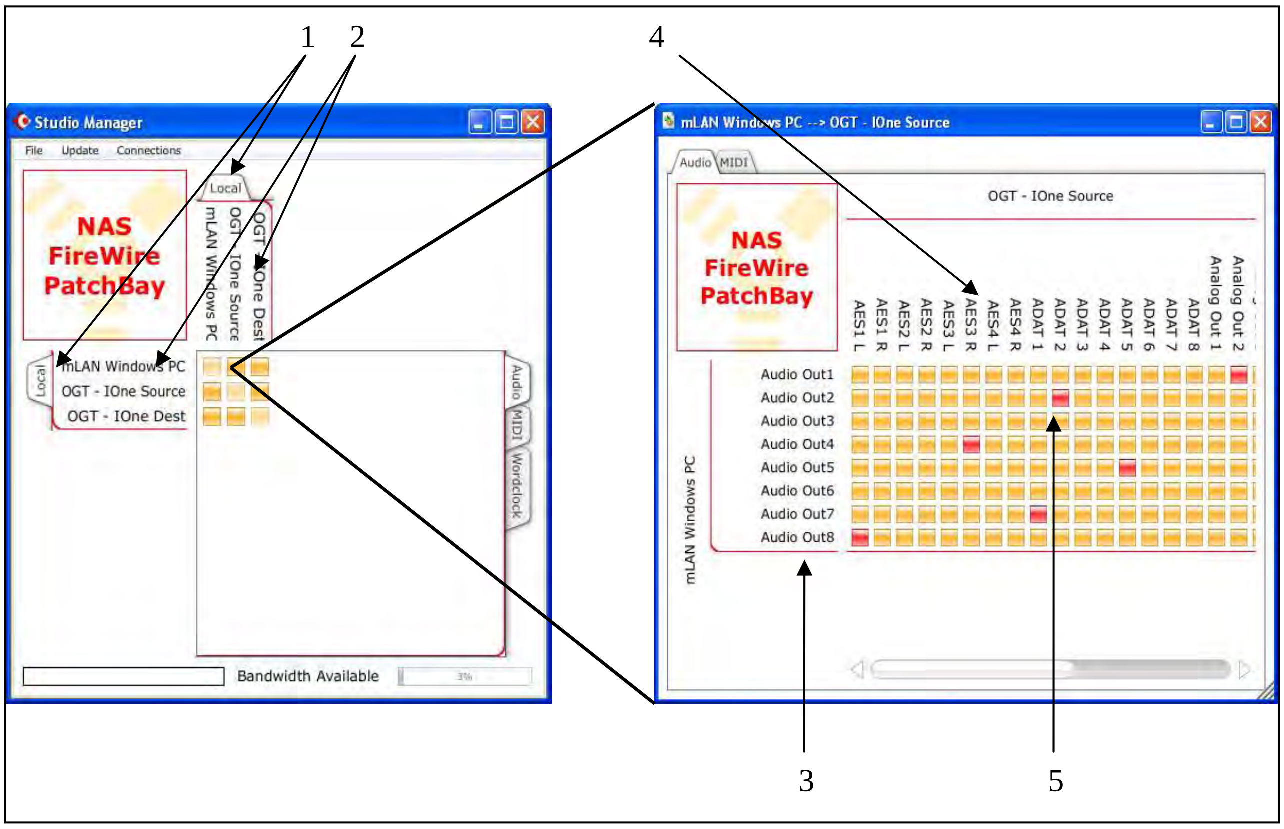


Figure 63: The visual representation of the double grid-based patchbay's objects

6.4.6. Making A Connection

When a sound engineer selects one of the cross points on the routing matrix (in order to make a connection between the corresponding soft plugs), a ‘connect request’ XML document is generated by the double grid-based patchbay and sent to the mCMS. An example of a connection request document is shown in Listing 32. This XML document contains:

- A request to make a connection. This is specified through the use of the connect method (see the `method` XML element) of the patch object (see the `object` XML element).

- The soft plug information of the plugs that are to be connected. This information is described with the use of the `parameter` XML elements. Specifically, the GUID's of the source and destination devices, the type of plugs being connected and the ID's of the plugs being connected are described with the `parameter` XML elements. This information was previously obtained from the server via the 'refresh response' XML document. The server is able to use the XML document to make a connection between the two soft plugs of the devices.

```
<mLANServerCommand version="1.0">
  <object name="patch">
    <method name="connect">
      <parameter name="sourceGUID" value="0013f00400000014"/>
      <parameter name="sourcePlugType" value="audio"/>
      <parameter name="sourcePlugID" value="6"/>
      <parameter name="destinationGUID" value="0090270001b37283"/>
      <parameter name="destinationPlugType" value="audio"/>
      <parameter name="destinationPlugID" value="102"/>
    </method>
  </object>
</mLANServerCommand>
```

Listing 32: An example of a 'connect request' XML document

6.4.7. Breaking A Connection

As with making a connection between two soft plugs, when a sound engineer wishes to break a connection between two soft plugs, the double grid-based patchbay application requests the server to perform this via a 'disconnect request' XML document. An example of a 'disconnect request' XML document is shown in Listing 33. This XML document contains:

- A request to the server to break a connection. This is specified through the use of the disconnect method (see the `method` XML element) of the patch object (see the `object` XML element).
- Information about the destination device involved in the connection. The required information is described with the use of `parameter` XML elements. Specifically, the GUID of the destination device, the type of plug involved in the connection, and the ID of the destination plug.

```
<mLANServerCommand version="1.0">
  <object name="patch">
    <method name="disconnect">
      <parameter name="destinationGUID" value="0090270001b37283"/>
      <parameter name="destinationPlugType" value="audio"/>
      <parameter name="destinationPlugID" value="102"/>
    </method>
  </object>
</mLANServerCommand>
```

Listing 33: An example of a 'disconnect request' XML document

6.4.8. Making Other Requests

It is possible to request the mCMS to perform other actions as well, such as adjusting the current plug layout of a device, setting a particular device to act as a global wordclock master, and setting up individual master/slave wordclock relationships. These requests all happen via XML documents that are similar in nature to the 'connect request' and 'disconnect request' XML documents discussed above.

6.4.9. Request Failures

When a 'connect request' or a 'disconnect request' (or any other request) XML document is sent to the server, it is possible that the requested action could fail. In this case, the server sends an 'error notify' XML document to the double grid-based patchbay specifying the reason the requested action could not be performed. An example of such a document can be seen in Listing 34. When the patchbay receives such error notifications, the error message is displayed to the user.

```
<mLANClientCommand>
  <object name="error">
    <method name="notify">
      <parameter name="description" value="Failed to create the requested
      connection"/>
    </method>
  </object>
</mLANClientCommand>
```

Listing 34: An example of an 'error notify' XML document

6.5. Summary

The patchbays discussed in this chapter (except the list-based patchbay) graphically present the audio network that they are representing hierarchically. The hierarchical representation allows for logical navigation to required soft plugs, and avoids the displays of these patchbays becoming cluttered. The hierarchical nature of these patchbays does however increase the number of mouse button clicks a sound engineer has to perform in order to locate required soft plugs. The tree-grid-based patchbay was identified as requiring less mouse button clicks in order to actually make a connection between two soft plugs. A single click on the grid implicitly selects the required source and destination soft plugs. The newly developed double grid-based patchbay uses a grid for navigating to required soft plugs and for actually making the connections between the soft plugs. This approach has reduced the number of mouse button clicks required to make connections, but still represents the network hierarchically.

The double grid-based patchbay application was successfully developed to be a client application to the mCMS. The server computer is connected to a FireWire network and has the task of actually making the connections between the soft plugs of the mLAN devices. The client and the server applications communicate using XML documents. The client application sends request XML documents to the server, and the server application optionally responds with response XML documents. Via the double grid-based patchbay, a sound engineer is able to make soft connections between the soft plugs of mLAN devices.

Chapter 07

Connection And Device Parameter Recall

The software applications discussed in the previous chapters provide certain useful capabilities:

- The routing matrices of the Matrix Mixer allow for the remote control over the routing and manipulation of audio signals *within* audio mixing desks.
- The Studio Manager 2 application allows for:
    - The representation and remote control of audio devices from within DAW applications via software device editors.
    - The saving and restoring of the parameters of the hardware devices that the software devices editors are representing.
- The double grid-based FireWire patchbay allows for audio signals to be routed *between* mLAN devices on FireWire networks.

This chapter proposes a single connection management application that integrates a number of studio control capabilities, namely:

- The representation and remote control of audio devices via software device editors, with the ability to represent and control audio mixing desks with software routing matrices.
- The representation and remote control over the audio routing between the audio devices.
- The ability to save the state of the parameters of each software device editor, and hence the hardware devices that they represent.
- The ability to save the state of the soft connections between the audio devices.
- The ability to restore the state of each software device editor, and hence the hardware devices that they represent.
- The ability to restore the soft connections that existed between the audio devices.
- The ability for this representation, control and parameter saving and recalling to happen from within DAW applications.

7.1. Introduction

In order to allow for the representation, control, and state recall of IEEE1394 audio devices from within DAW applications, the double grid-based patchbay was implemented in such a way that it may be hosted by Studio Connections compatible DAW applications. The patchbay was also implemented such that it is capable of hosting the various device editors that are available on the host system. The role of the double grid patchbay is such that it:

- Performs the same tasks as Studio Manager 2 would. Specifically, it should:
    - Host software device editors that allow for the representation and remote control of hardware audio devices.
    - Allow for the state of the parameters of the hardware devices to be saved and restored.
- Allow for connections between the audio devices that are represented on the patchbay to be managed.
- Allow the connections between devices to be saved and restored.

The role of the double grid-based patchbay as a device editor host is shown in Figure 64.

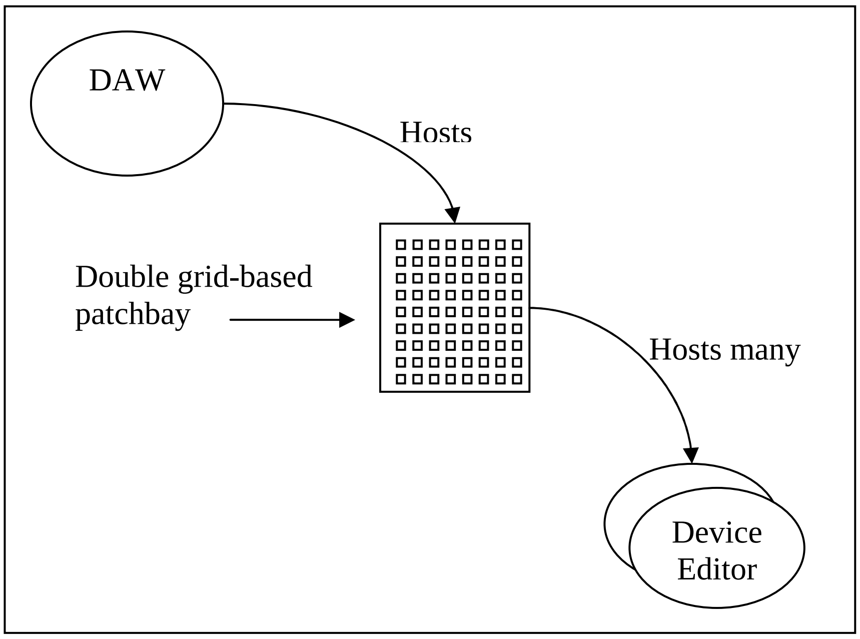


Figure 64: The double grid-based patchbay hosting

The integration of the double grid-based patchbay, the Matrix Mixer and a DAW application is shown in Figure 65.

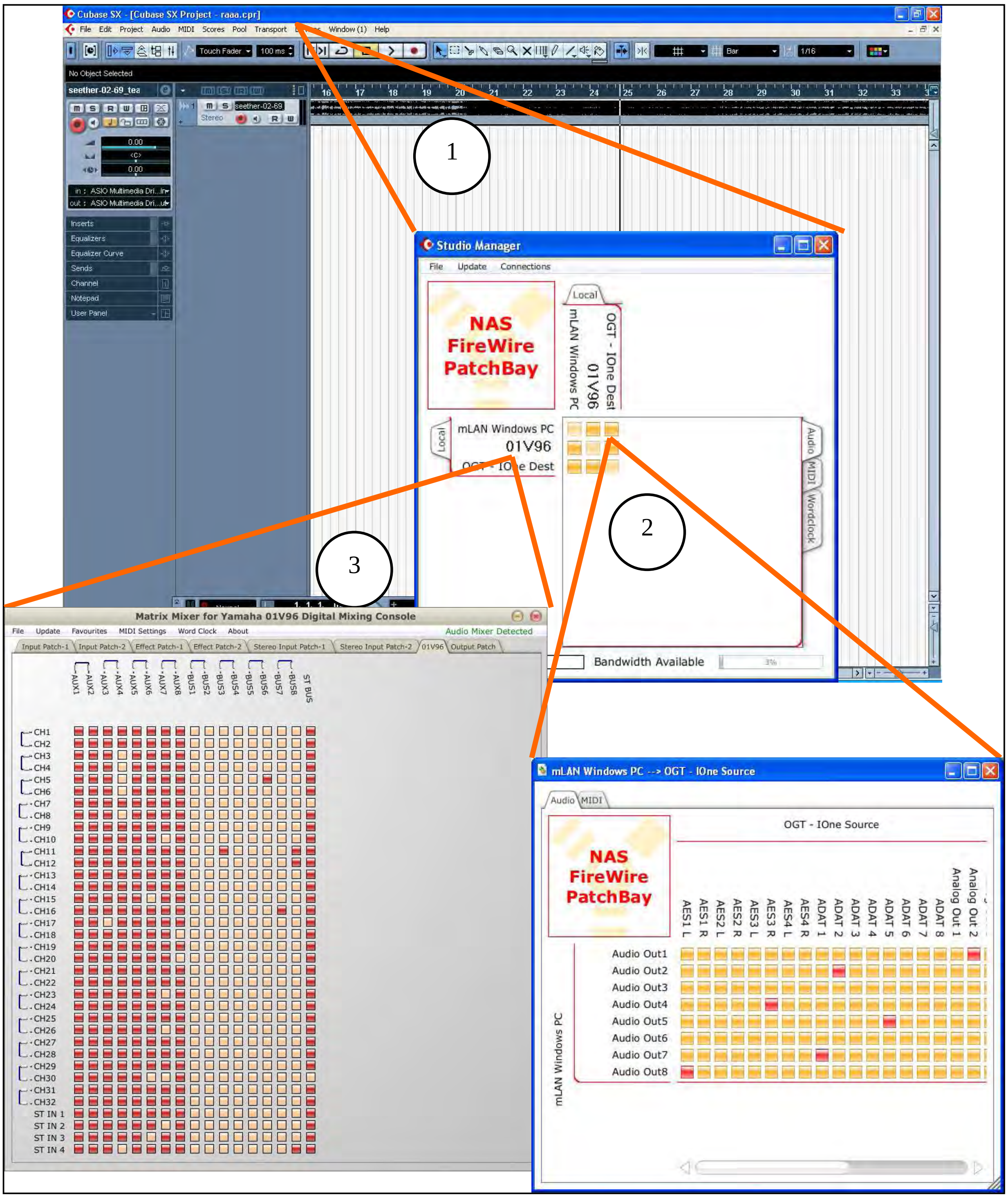


Figure 65: The double grid-based patchbay hosting from within Cubase

The annotations in Figure 65 are explained below:

1. When the DAW application is initialised, it hosts and initialises the double grid-based patchbay. This patchbay may then be displayed from within the DAW application by selecting it from the appropriate menu.

2. Once the patchbay is displayed, it is possible to make connections between the soft plugs of the mLAN devices being represented on the patchbay. This is done by selecting a cross point between the required source and destination devices in order to display the source soft plugs of the source mLAN device, and the destination soft plugs of the destination mLAN device. A sound engineer is then able to make the required connections by selecting the cross points on the displayed routing matrix.
3. It is possible to associate software device editors with devices that are represented on the double grid-based patchbay. From the primary window of the double grid-based patchbay, a sound engineer is able to display a device editor that is associated with a particular mLAN device by selecting the appropriate label. Shown here is the Matrix Mixer being associated with the '01V96' device. Via the device editor, a sound engineer is able to manipulate the device that it represents. Associations between devices and device editors are set up manually.

With the aid of the double grid-based patchbay, a DAW application is able to save the state of the soft connections between the devices its representing, as well as save the state of each one of the devices. These states are saved to the hosting DAW's native song file. When the DAW application later reloads one of its saved song files, it is able to extract the state data from the song file. The state data is passed back to the double grid-based patchbay. The patchbay is then able to use this data to restore the connection and device states of the audio devices.

7.2. Providing Studio Wide Total Recall

This section will provide a description of how the capabilities shown in section 7.1 were designed and implemented. The functionality that was implemented to:

- Allow the patchbay to be hosted by a DAW application.
- Allow the patchbay to host software devices editors.
- Allow for studio wide Total Recall.

is shown in Figure 66 in the form of a use case diagram.

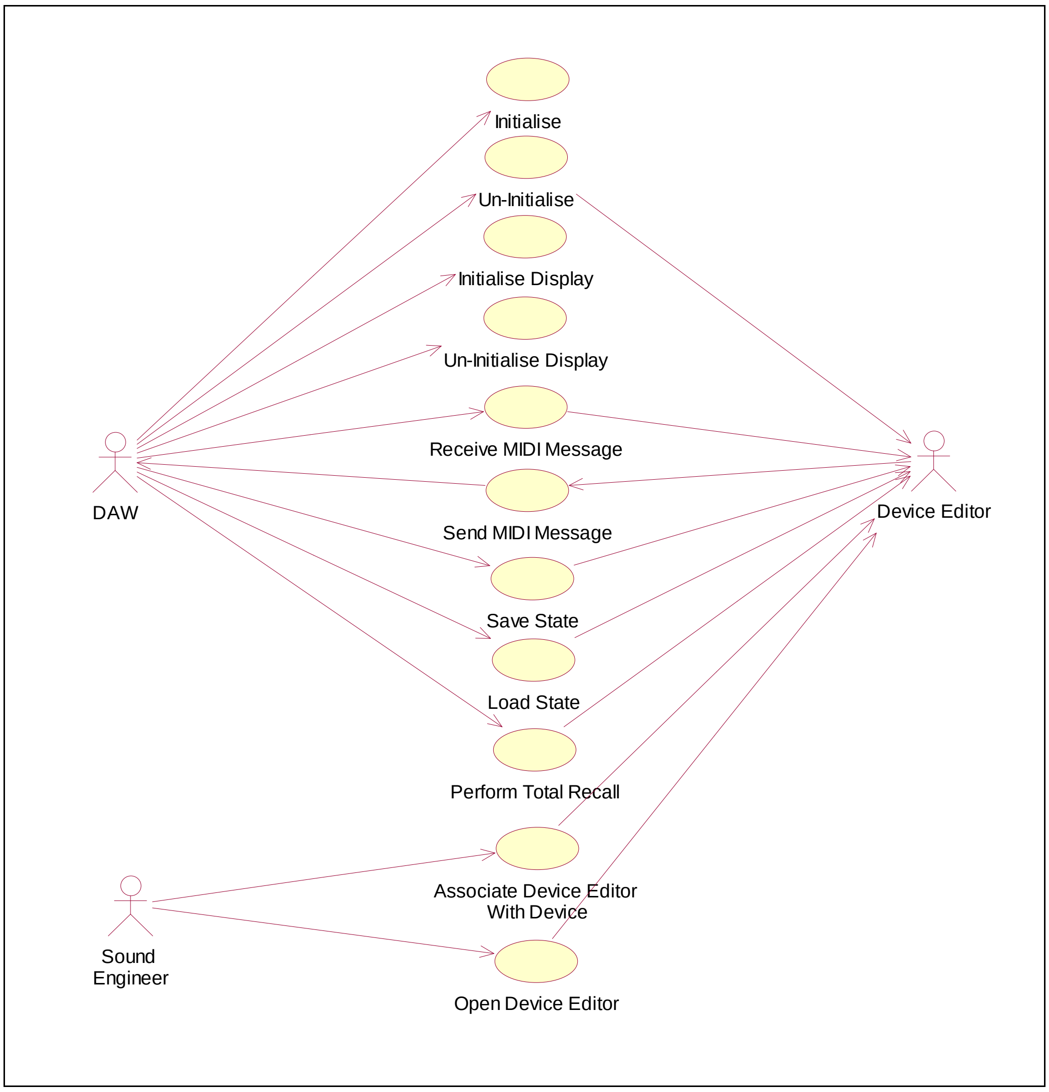


Figure 66: Double grid-based patchbay hosting use case diagram

Each device editor that Studio Manager 2 is able to host is implemented as an enhanced OPT component. Studio Manager 2 itself is also implemented as an enhanced OPT component. These components are built using the COM architecture and there is a set of defined programming interfaces through which DAW applications may communicate with Studio Manager 2.

In order to allow compatible DAW applications to communicate with the double grid-based patchbay (rather than Studio Manager 2) an implementation of these interfaces was created. The nature of the interfaces implemented for the double grid-based patchbay is the same as the interfaces that are implemented by the device editors. These programming interfaces were discussed in "Chapter 05 Studio Connections".

The core interfaces that had to be implemented by the double grid-based patchbay are listed below:

- `IMPInitialise`: The methods of the `IMPInitialise` interface are used by a DAW application to:
    - Pass in its interface pointers to the double grid-based patchbay. This enables the patchbay to communicate with the DAW application.
    - Initialise and un-initialise the patchbay application.
- `IPropertyPage`: The methods of the `IPropertyPage` interface allow for the display of the double grid-based patchbay to be manipulated. Most importantly, it allows a DAW application to initialised, un-initialised and show the display of the patchbay.
- `IMPEventFilter`: The `IMPEventFilter` interface allows for the communication of queued events from the DAW application to the patchbay application.
- `IPersistStream`: Through the methods of the `IPersistStream` interface, the double grid-based patchbay is able to save and restore its state to and from a supplied serial stream.
- `ISM2Comp`: The methods of the `ISM2Comp` interface allows a hosting DAW application to display the device editors that are hosted by the patchbay application, and it provides a way to control the Total Recall operations of the patchbay.

7.2.1. Initialising And Un-Initialising The Double Grid-Based Patchbay

The initialisation and un-initialisation of the double grid-based patchbay is controlled through two methods that are defined as part of the `IMPInitialise` interface. These two methods are `MPConnect` and `MPDisconnect`.

When a DAW application wishes to initialise the double grid-based patchbay (for example, this may happen when the DAW application is started up), it will query the patchbay application for a pointer to its `IMPInitialise` interface. The DAW application may then call the `MPConnect` method on that interface pointer. When the double grid-based patchbay's implementation of the `MPConnect` method is called, it connects to the mLAN Connection Management Server (mCMS). Once a connection has been established, it sends a 'refresh request' XML document to the server requesting the configuration of the FireWire network that is attached to the server computer. The patchbay application is initialised and it waits for a response from the server. At this point as well, the display of the patchbay application is not displayed and no interaction between the patchbay and the devices editors has taken place. Once the server has responded with 'refresh response' XML document to the patchbay it builds up its internal objects, as shown in Figure 62 in the previous chapter. This initialisation process is shown diagrammatically in Figure 67.

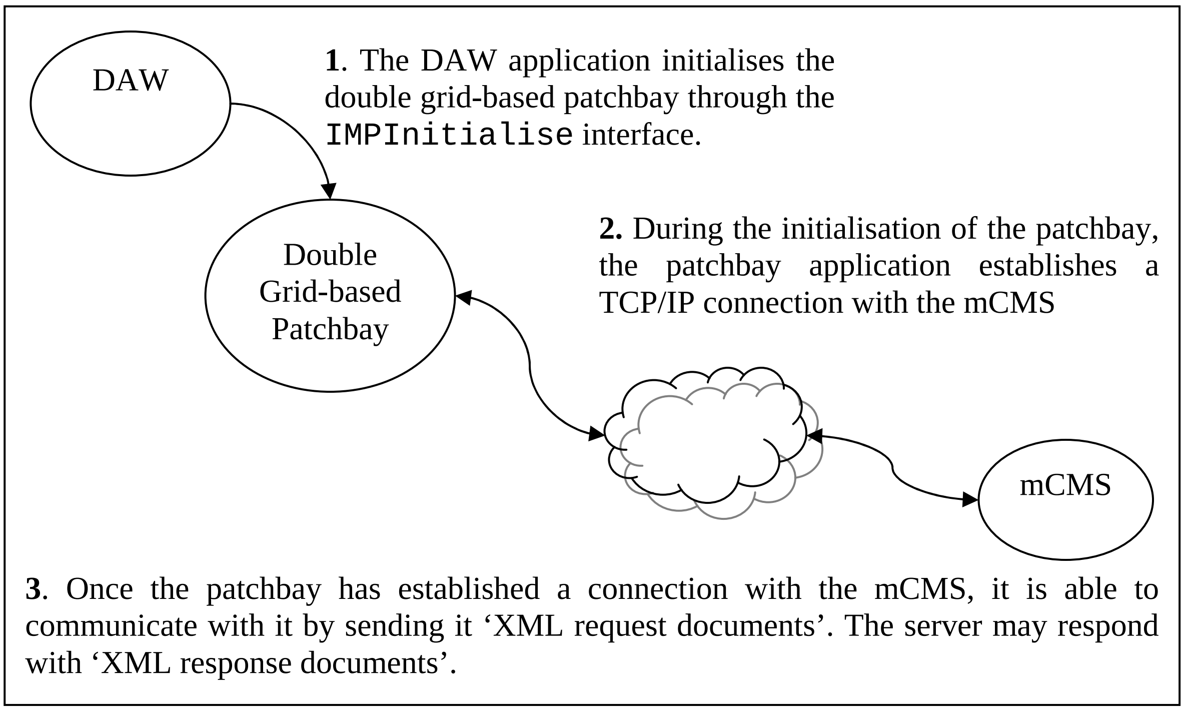

DAW
1. The DAW application initialises the double grid-based patchbay through the `IMPInitialise` interface.
Double Grid-based Patchbay
2. During the initialisation of the patchbay, the patchbay application establishes a TCP/IP connection with the mCMS
mCMS
3. Once the patchbay has established a connection with the mCMS, it is able to communicate with it by sending it 'XML request documents'. The server may respond with 'XML response documents'.

Figure 67: The initialisation of the double grid-based patchbay by a DAW application

When the DAW application that is hosting the patchbay wishes to un-initialise it (for example, this could happen when the DAW application itself shuts down), it calls the patchbay's `MPDisconnect` method of the `IMPInitialise` interface. When the patchbay's implementation of this method is called, it disconnects itself from the mCMS and frees the resources that it was utilising.

7.2.2. Initialising And Un-Initialising The Double Grid-Based Patchbay's Display

When a DAW application initialises the double grid-based patchbay, its display is not shown. The display of the patchbay is only shown on request from the hosting DAW application. This may happen once a user has requested the DAW application to display the patchbay (for example, this may happen via one of the DAW application's menus). The initialisation and un-initialisation of the display happens via the `Activate` and `Deactivate` methods of the `IPropertyPage` interface, respectively, of which the patchbay has an implementation.

When the double grid-based patchbay's implementation of the `Activate` method is called, the primary window of the application is built up to reflect the internal objects of the system, as seen in Figure 63 of the previous chapter. It provides a representation of the FireWire network, its busses, and the mLAN devices that are on the FireWire busses. Once the display is initialised, the DAW application may then visually display it by calling the `Show` method of the `IPropertyPage` interface.

When a sound engineer closes the patchbay, the patchbay's implementation of the `Deactivate` method is called. It is here that the patchbay frees up any resources that the display was utilising.

7.2.3. Associating Device Editors With Devices

When the patchbay application is initialised, no software device editors are associated with the mLAN devices that are represented on the patchbay. These associations have to be set up by a sound engineer. A sound engineer is able to set the associations by selecting the device and selecting the 'Set device editor…' menu item. Via the 'Device Editor Setup' window that is displayed, a sound engineer is able to select which software device editor is associated with which device. This window can be seen in Figure 68. This window displays the available devices on the mLAN network, and the device editors that are installed on the computer system. From this window, a sound engineer selects the required mLAN device (from the 'Devices' list) and the device editor it is to be associated with (from the 'Select Device Editor' list).

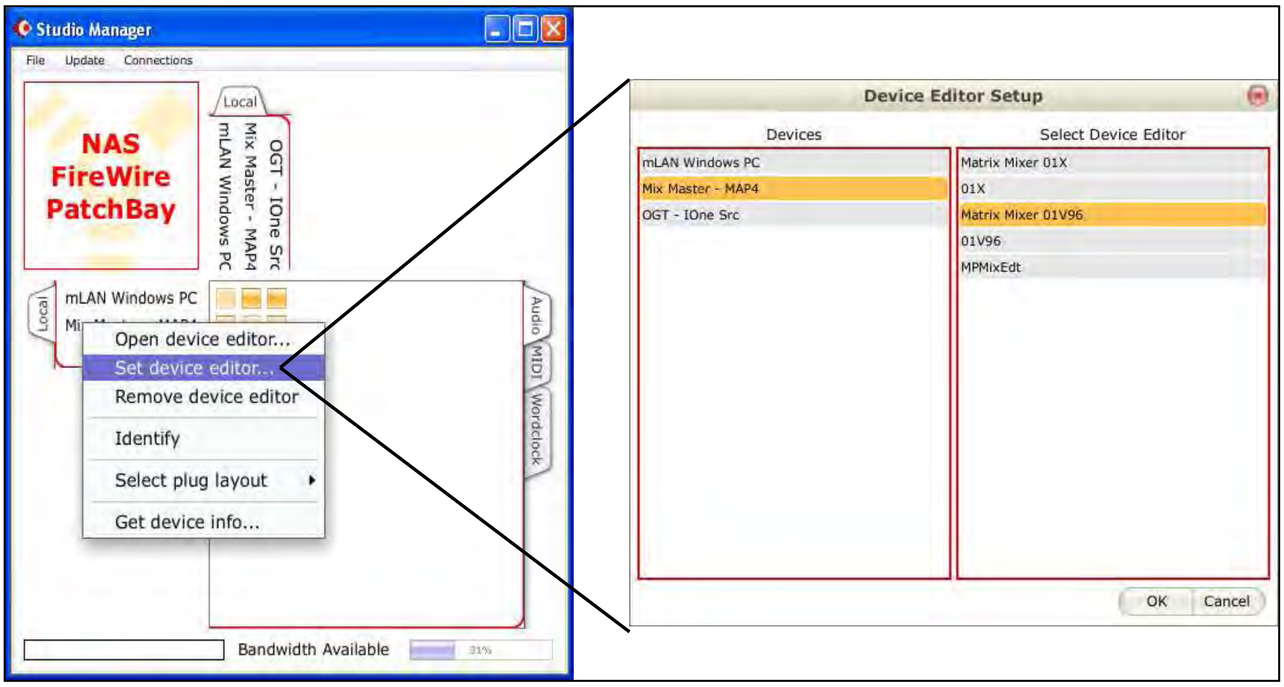


Figure 68: Associating software device editors with hardware devices

Once device editors have been associated with devices, each device editor is initialised. The patchbay is able to initialise each device editor by querying it for its `IMPInitialise` interface pointer. Once a pointer to this interface has been obtained, the double grid-based application will call the `MPConnect` method on that interface pointer. This instructs the particular device editor to initialise itself. After each device editor has been initialised, the patchbay awaits further instructions from the patchbay application.

7.2.4. Displaying Device Editors

Once the mLAN devices that are represented on the patchbay have device editors associated with them, it is possible to display them and use them to manipulate their corresponding hardware devices. In order to display a device editor that is associated with an mLAN device, a sound engineer would select the required device and then select ‘Open device editor…’ from the popup menu. This then displays the device editor, as shown in Figure 69.

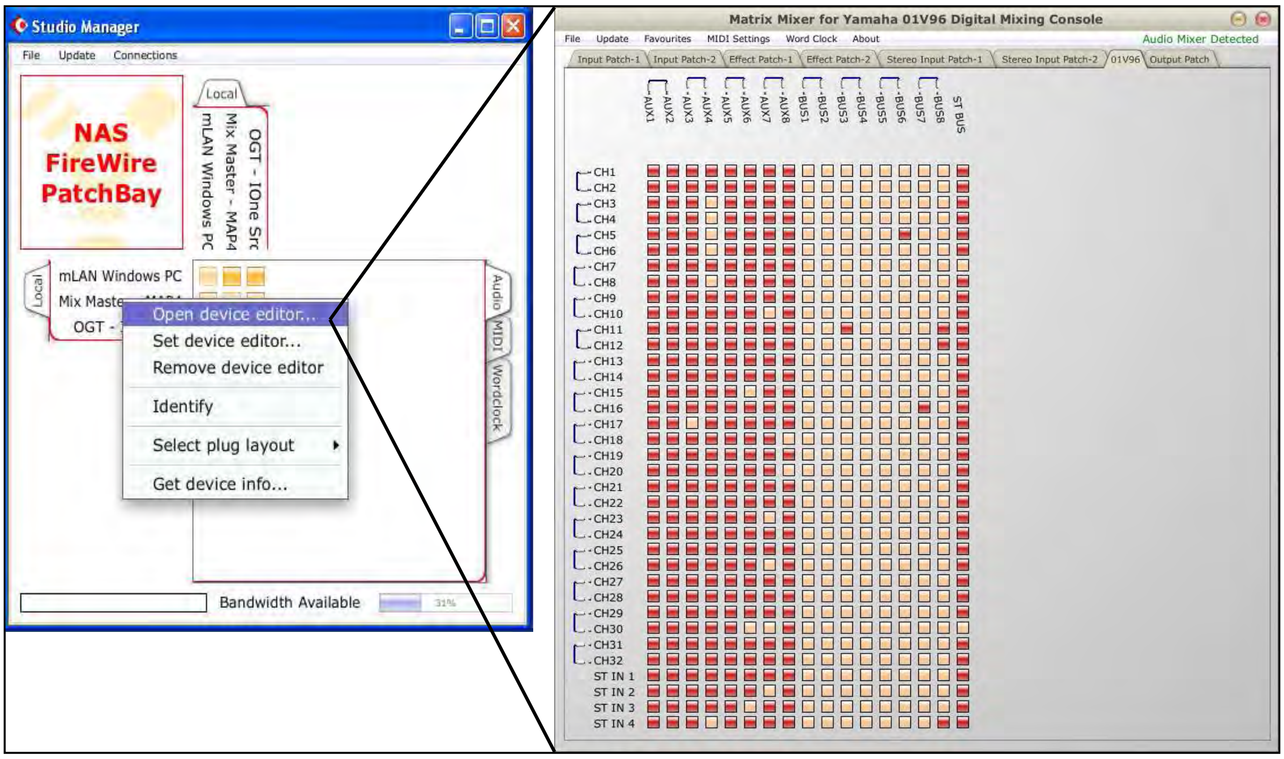


Figure 69: Displaying device editors from the double grid-base patchbay

When a sound engineer requests the patchbay to display a device editor, the patchbay queries the specific device editor for its `IPropertyPage` interface. Once the patchbay has obtained a pointer to the particular device editor's `IPropertyPage` interface, it calls the specific device editor's implementation of the `Activate` method. The device editor will then create its window to reflect the state that the device editor is in. The patchbay then displays the device editor's window by calling the `IPropertyPage`'s `Show` method.

7.2.5. Receiving MIDI Messages

Any MIDI messages sent by hardware audio devices destined for their corresponding software device editors are received by the DAW application that is hosting the double grid-based patchbay. The DAW application routes the incoming MIDI messages to the double grid-based patchbay. The patchbay is then responsible for routing the incoming MIDI messages to each individual device editor that it has initialised. Each device editor is responsible for processing the incoming MIDI message appropriately.

The patchbay provides an implementation of the `IMPEventFilter`'s `MPOnEvents` method. It is through this method that the MIDI messages are passed from the DAW application to the patchbay. The patchbay application queries each device editor for its `IMPEventFilter` interface and is able to pass the incoming MIDI messages to the device editors through each device editor's implementation of the `MPOnEvents` method. Each device editor processes the incoming MIDI messages appropriately.

7.2.6. Sending MIDI Messages

When the double grid-based patchbay application is initialised via its `MPConnect` method, the hosting DAW application passes in a pointer to its `IMPAsyncOutput` interface. The patchbay application retains a pointer to this interface. The `MPOutput` method of this interface is used to output MIDI messages. When each device editor is initialised (via its `MPConnect` method), a pointer to the DAW application's `IMPAsyncOutput` interface is passed to it. Each device editor may then send MIDI messages to their corresponding hardware device via this interface's `MPOutput` method.

7.2.7. Save State

A DAW application may request the patchbay application to save itself to a given stream. For example, this could happen when a sound engineer requests the DAW application to save the project he is working on. The state data provided by the patchbay application may be saved to the DAW application's native song file. When the DAW application wants to save the state of the connections between the devices represented on the patchbay and the state of the device editors, it calls the patchbay's implementation of the `IPersistStream`'s `Save` method. The `Save` method requests the patchbay to save its state to a supplied serial stream.

When the patchbay's implementation of this method is called, for each device that is represented on the patchbay the following data is saved to the supplied stream (the numbers below correspond to the annotations in Figure 70):

1. The position in the stream after the current device's state data. Once the state of a device, and its associated device editor, has been written to the stream, the patchbay application saves the position in the stream where the next device's state data is written to.
2. The GUID of the device.

3. The model ID of the device.
4. The vendor ID of the device.
5. The ‘connect request’ XML documents that represent the connections from the soft plugs of the device.
6. If there is a device editor associated with the device, the GUID of the device editor.
7. The stream position after the device editor’s state data.
8. The device editor’s state data, as saved by the device editor itself.

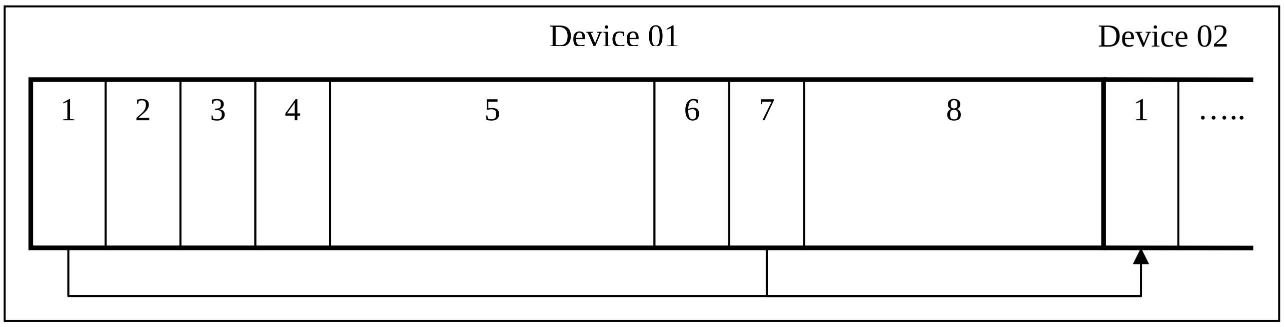


Figure 70: The double grid-based patchbay's state data format

Each device editor is responsible for writing its own state data to the stream (annotation 8 in Figure 70 above). The patchbay application cycles through each of the instantiated device editors and calls their implementations of the `IPersistStream`’s `Save` method. Via this method, the patchbay passes in a pointer to the stream and each device editor saves its state data to the stream.

7.2.8. Load State

A DAW application may request the double grid-based patchbay application to restore its state from a supplied serial stream. The DAW application does this by obtaining a pointer to the patchbay’s `IPersistStream` interface and calling the patchbay’s implementation of the `Load` method. This method requests the patchbay application to restore itself to a previous state from the data which may be obtained from the supplied serial stream. A pointer to the stream object is supplied to the patchbay via the `Load` method.

The data is read out of the stream in the format that it was written to the stream (see section 7.2.7 above). Initially, for each mLAN device, the following data is read out of the stream:

- The position in the stream after the mLAN device’s state data (see annotation 1 in Figure 70 above).
- The GUID of the mLAN device.
- The model ID of the mLAN device.
- The vendor ID of the mLAN device.

The patchbay application will check to see if the mLAN device still exists on the network. This is done by checking to see if the GUID read out of the stream matches the GUID of any of the devices currently on the FireWire network. If the device is not found on the network, the patchbay application checks to see if the device has been replaced with a device of the same type. For example, a sound engineer may save the state of a Yamaha 01V96 Digital Mixing Console, remove it from the network and replace it with another Yamaha 01V96 Digital Mixing Console. The patchbay application is able to find a replacement device from the model and vendor ID’s that were saved to the stream. The patchbay application cycles through each of its mLAN device objects to check if devices with the same vendor and model ID’s have been placed on the network. If the original mLAN device has been replaced with another mLAN device that has the same vendor and model ID, the state data that applied to the original device will apply to the replacement device. If more than one replacement device has been found, the application requests the sound engineer to select which device should be the replacement device. If it is found that the mLAN device has been removed, and no replacement device is available, the patchbay application skips over the data in the stream that is related to the removed device, and reads out the data for the next device. The patchbay application is able to use the initial value read out of the stream to skip over the state data (see annotation 1 in Figure 70 above).

Once an appropriate device is found on the network, the ‘connect request’ XML documents associated with the mLAN device are read out of the stream, and temporarily stored. The XML documents need to be stored as it may be necessary to alter the device GUID’s that are contained within the documents, as the devices may have been replaced with other devices. The application is only aware of the replacement devices once the entire stream has been read.

Next, if a device editor was originally associated with a particular device, the GUID of the device editor is read out of the stream. The GUID is used to locate the appropriate device editor from the system registry and to instantiate it. Once the device editor is instantiated, it is associated with the appropriate mLAN device. If the device editor fails to initialise, the patchbay application skips past the data relevant to the device editor and reads out the data for the next mLAN device.

If a device editor is successfully instantiated, the patchbay application obtains a pointer to its `IPersistStream` interface and calls its implementation of the `Load` method. Through the `Load` method, the patchbay passes in the pointer to the stream object. The device editor is then able to read out its state data from the stream and use that data to restore its state to the state that was written to the stream.

Once all of the relevant 'connect request' XML documents have been read out of the stream and all of the device editors have been initialised, the GUID's in the 'connect request' XML documents are updated appropriately, if needed. Once all of the documents have been updated, the information contained in them is used to set up the associations between the relevant `Plug` objects of the `mLAN Device` objects (the associations denote connections between the plugs).

7.2.9. Perform Total Recall

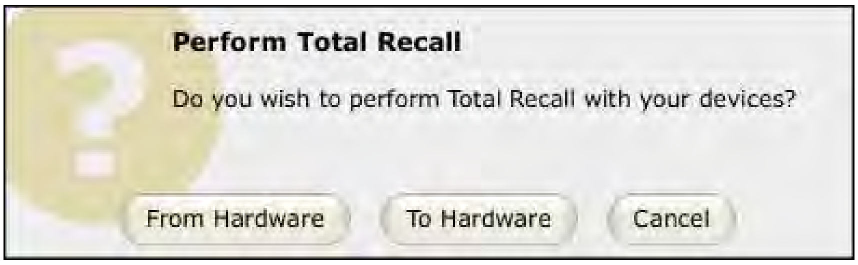
Total Recall refers to the transfer of state from network to software, or from software to network. In the context of the double grid-based patchbay, Total Recall takes place when the state of the patchbay and its device editors is transferred across to the relative hardware devices, or the state of the soft connections between devices and the state of the hardware devices is transferred across to the patchbay application and its device editors. For example, if a DAW application has loaded a particular song file, and it has restored the state of the double grid-based patchbay application and its device editors, Total Recall would refer to the subsequent transfer of software state to hardware, or to the transfer of state from hardware to software. This enables the software and hardware to synchronise their states.

Total Recall is controlled by the DAW application hosting the patchbay through the `ReSync` method of the `ISM2Comp` interface. The method declaration is shown in Listing 35. When this method is called by a hosting DAW application, it is the responsibility of the patchbay to find out which direction the state transfer should take place and then to perform the transfer of state.

```
HRESULT ReSync(HWND hWndParent);
```

Listing 35: The `ISM2Comp::ReSync` method

When the method is called, the patchbay application displays a dialog window from which a sound engineer may select the direction in which the state transfer should take place. The dialog window is shown in Figure 71. From this dialog window, a sound engineer may select that the state transfer takes place from the hardware devices to the software by selecting the ‘From Hardware’ button, or a sound engineer my select the ‘To Hardware’ button to specify that the state transfer should take place from the software to the hardware devices.


Perform Total Recall

Do you wish to perform Total Recall with your devices?

Figure 71: Total Recall direction dialog window

If a sound engineer requests that the state transfer takes place from the hardware devices to the software, the patchbay:

- Requests the state of the soft connections between the hardware devices. This is done by sending a ‘refresh request’ XML document to the mCMS. The mCMS responds with a ‘refresh response’ XML document and the patchbay is able to find out the topology of the FireWire network that the server computer is attached to, and what soft connections exist between the mLAN devices that are on the FireWire network.
- Requests each device editor to transfer state from their hardware devices to themselves. For each device editor that is associated with a device, its implementation of the `MPRecall` method of the `IMPServices` interface is called with a request to transfer state from its corresponding hardware device to itself.

If a sound engineer requests that the state of the software patchbay and its device editors to be sent across to the hardware devices, the patchbay:

- Cycles through each of its objects used to represent the output soft plugs (`Audio Output Plug`, `MIDI Output Plug`) of the mLAN devices. For each specific input plug object (`Audio Input Plug`, `MIDI Input Plug`) that is associated with a specific output plug object (denoting a soft connection), a ‘connect request’ XML document is generated containing the connection information. Each of the generated ‘connect request’ XML documents is sent across to the server to request the server to make the connection.

- Cycles through each of the device editors that are associated with the mLAN devices being represented on the patchbay. Each device editor’s implementation of the `MPRecall` method of the `IMPServices` interface is called requesting the device editor to send its current state to its associated hardware device.

7.3. Summary

Through the use of the interfaces of the Studio Connection – Total Recall SDK, it was possible to implement the double grid-based patchbay application in such a way that it could:

- Be dynamically loadable, so that it can be hosted by a compatible DAW application.
- Dynamically load compatible software device editors.
- Provide the state data representing the connections between the mLAN devices it is representing, and provide state data representing the devices to the hosting DAW application.
- Be able to restore itself from the state data provided by a DAW application.
- Transfer connection and device state from itself to the hardware devices, and be able to transfer connection and device state from the hardware devices to itself.

Through the integration of compatible DAW applications, the double grid-based FireWire patchbay and compatible software device editors, and with the aid of the Studio Connections – Total Recall SDK, the goal of providing remote control and recall of the properties of IEEE1394 devices was achieved.

Chapter 08

Conclusion

The aim of this project was to provide remote control and recall over the various properties of IEEE1394 audio devices via a series of graphical routing matrices. In order to reach this goal, the project was broken down into a number of different phases, and the outputs of each phase were integrated.

An audio mixing desk is central to most audio studios:

- It receives audio signals from the various devices that are in an audio studio (for example, the output audio signals from microphones and effects units).
- It processes and mixes them (for example, each incoming input signal may be shaped by equalisers and dynamics processors, and each of these signals mixed together to form a new signal).
- It routes audio signals to external devices for further processing (for example, once the incoming signals have been mixed together, the mixed signals may then be routed to an external effects processor to have an effect applied to it).

Essentially, the task of an audio mixing desk is to route audio signals from its various analogue and digital audio inputs, through its various internal signal processing components (for example, equalisers and dynamics processors), onto various audio busses (where the audio signals are mixed together) and eventually to the outputs of the same audio mixing desk.

The Matrix Mixer was developed in order to represent the various signal processing points and signal flows that exist within audio mixing desks. The layout of the routing matrices mimics a typical audio mixing desk block diagram. It allows for remote control over remotely controllable audio mixing desks. It emulates the audio mixing desk it is representing. The Matrix Mixer consists of a series of routing matrices that allow audio signals to be routed between the various signal source and destination points that exist within an audio mixing desk. This is performed by selecting the cross points on the routing matrices to enable audio patches, and de-selecting the cross points to disable audio patches. The matrix design also allows for parameters that are available at the signal processing points to be adjusted. Required signal processing points may be selected in order to graphically expose the parameters available at the specific points. A graphical control may be adjusted in order to instruct the audio mixing desk to adjust the parameter associated with the control.

In order to allow the Matrix Mixer to quickly adapt to different audio mixing desks, it was designed and developed to be a generic software audio mixing desk editor. A number of parameters were identified as being common to audio mixing desks, and the Matrix Mixer was developed to be able to represent and control these identified parameters. Specific audio mixing desks are described using XML documents, for which XML elements and attributes have been defined. Each Matrix Mixer XML document has XML elements and attributes that are used to describe the various signal processing points that exist within an audio mixing desk, the various parameters that exist at the signal processing points, and the control messages that are used to remotely control the parameters of the audio mixing desk. When the Matrix Mixer starts up, it will load an XML configuration file for a specific audio mixing desk, and from that configuration file it is able to build up its internal objects and routing matrices to represent the audio mixing desk.

The Matrix Mixer was implemented to be Studio Connections compatible. The Studio Connections – Total Recall SDK provides a number of programming interfaces that may be implemented in order to provide compatibility to software sequencers. The Matrix Mixer was converted to be dynamically loadable and an implementation of these programming interfaces was provided. This enabled the Matrix Mixer to be dynamically loadable by other software applications, such as the Studio Manager 2 application. The Studio Manager 2 application may in turn be hosted by a compatible Digital Audio Workstation (DAW) application. Through the Studio Manager 2 application, the Matrix Mixer is able to be displayed and manipulated from within a compatible DAW application. It also allows the Matrix Mixer to supply the state of its associated hardware audio mixing desk to the hosting DAW application. The DAW application may save the state data to its native song files. When the DAW application reloads a native song file again, it is able to pass the state data back to the Matrix Mixer and the Matrix Mixer is able to restore it parameters to a previous state. The state of the Matrix Mixer’s parameters may then be transferred to the associated hardware audio mixing desk in order to restore it to a previous state.

In digital audio networking, such as mLAN, a software patchbay is used to patch audio signals between audio devices on a network. Traditionally, audio devices may have a number of hardware sockets on them through which individual audio and control signals are sent and received by the device. With mLAN, these physical hardware plugs have been replaced with software plug abstractions (soft plugs). The soft plugs that are present on a device are revealed to a sound engineer through a software patchbay with runs on a computer. Through the use of these patchbays, a sound engineer may patch audio signals between the soft plugs of the devices on the network. There are various types of software patchbays available, and it was decided to implement a variation on the grid-based patchbay, know as a double grid-based patchbay. The double grid-based patchbay approach to patching was chosen due to the hierarchical nature of the grids (which reflects the structure of the network it represents, thus allowing for ease of navigation to the required soft plug of devices), and due to the fact that this type of patchbay requires less mouse button clicks by a sound engineer to patch audio signals between devices. The buttons on the grids of these patchbays each represent a signal source and destination pair, thereby alleviating the need to individually select the soft plugs required for a soft connection.

The double grid-based patchbay presents a sound engineer with a graphical grid. Selecting one of the cross points on the grid where the source device label intersects a destination device label displays a second graphical grid. On the second grid, audio signals may be routed between the displayed signal source and destination plugs by selecting the cross points on the grid between the required soft plugs. The double grid-based patchbay application was successfully implemented as a client application to the mLAN Connection Management Server (mCMS). The mCMS runs on a computer attached to a FireWire network. The client application communicates with the mCMS through a set of predefined request and response XML documents. Through the use of these XML documents, the client application is able to discover the topology of the FireWire network attached to the server computer, and request the mCMS to perform actions such as routing audio between the soft plugs of the devices on the network.

In the final phase of the project, the previous phases were integrated in order to achieve the goal of complete control and recall of an audio studio. At the outset, it was known that Studio Manager 2 can be hosted by a DAW application and that a Studio Connections compatible DAW application is able to communicate with Studio Manager 2 through a set of defined programming interfaces. These interfaces are provided as part of the Studio Connections – Total Recall SDK. An implementation of these interfaces was provided within the double grid-based patchbay in order to allow it to be hosted by a compatible DAW application, and to allow for communication between the two entities. The double grid-based patchbay application was implemented in such a way that it could host compatible software device editors, such as the Matrix Mixer, and display them (in the same way that Studio Manager 2 does). A DAW application hosting the double grid-based application is able to request the patchbay to perform tasks via the implemented interface methods. The patchbay application in turn is able to request its software device editors to perform tasks by calling the methods of their implemented interfaces.

By virtue of the patchbay hosting and displaying software device editors, it is possible to provide remote control over the soft connections between devices, and remote control over the parameters of the devices. It is possible to associate available device editors with the devices that are available on the associated FireWire network, and to display them by selecting the devices.

A hosting DAW application can request the double grid-based patchbay application to save its state. When requested to do so, it passes the state of the soft connections between the associated mLAN devices to the DAW application. Also, via each device editor that the patchbay is hosting, it passes the state of the audio devices on the FireWire network to the DAW application. The DAW application may then save the supplied state data to its native song files. When the DAW application reloads its native song files, it can extract the state data and pass it back to the patchbay application. The patchbay application may then use the state data to restore itself, and its device editors, to a previous state. The DAW application may also request the patchbay application to transfer state between itself and its associated hardware devices. For example, this could happen once the DAW application has requested the patchbay to restore itself from the supplied state data. The patchbay may either transfer its state from itself to its associated hardware devices or from the hardware devices to itself. The transference of state from the patchbay to the hardware devices allows for the audio studio to be recalled to a previous state.

Previous software implementations that allow for the control over the various parameters of audio mixing desks have displays that are varying, cluttered and tedious to navigate. The graphical interfaces of these software entities usually reflect the front panel of the device they represent. Inter device connection management may be performed by software patchbays, each with varying user interfaces. These software applications usually work independently of each other. This project has demonstrated a common grid approach to both intra device and inter device routing. This has happened through the integration of various software entities into a single application that may be hosted by a DAW application. All routing and device parameter settings may be saved to the hosting DAW application’s native song files. This allows the routing and parameter settings of the audio devices to be restored at a later stage. It is the intention that this paradigm of grid-based patchbays will demystify routing within and between devices, thereby allowing for easier sound system set up.

Appendix – An XML Schema For Representing MIDI Controllable Audio Mixing Desks

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- edited with XMLSpy v2006 sp1 U (http://www.altova.com) by Philip Foulkes
(Rhodes University) -->
<!--W3C Schema generated by XMLSpy v2006 sp1 U (http://www.altova.com)-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
      <xs:element name="mixer">
            <xs:complexType>
                  <xs:choice maxOccurs="unbounded">
                        <xs:element name="parameterGroups"
type="parameterGroupsType" minOccurs="0"/>
                        <xs:element name="activeSense" type="activeSenseType"
minOccurs="0"/>
                        <xs:element name="wordclock" type="wordclockType"
minOccurs="0"/>
                        <xs:element name="select" type="selectType"
maxOccurs="unbounded"/>
                  </xs:choice>
                  <xs:attribute name="name" type="xs:string" use="required"/>
            </xs:complexType>
      </xs:element>
      <xs:complexType name="parameterGroupsType">
            <xs:sequence>
                  <xs:element name="parameterGroup" type="parameterGroupType"
maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="parameterGroupType">
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="type" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="activeSenseType">
            <xs:sequence>
                  <xs:element name="on" type="onType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="timeOut" type="xs:nonNegativeInteger"
use="required"/>
      </xs:complexType>
      <xs:complexType name="wordclockType">
            <xs:sequence>
                  <xs:element name="samplingFrequencies"
type="samplingFrequenciesType"/>
                  <xs:element name="mLANAutoWordclockParameter"
type="mLANAutoWordclockParameterType" minOccurs="0"/>
                  <xs:element name="wordclockSelectParameters"
type="wordclockSelectParametersType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="samplingFrequenciesType">
            <xs:sequence>
                  <xs:element name="fourtyFourPointOne"
type="fourtyFourPointOneType"/>
                  <xs:element name="fourtyEight" type="fourtyEightType"/>
                  <xs:element name="eightyEightPointTwo"
type="eightyEightPointTwoType"/>
                  <xs:element name="ninetySix" type="ninetySixType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="fourtyFourPointOneType">
            <xs:sequence>
                  <xs:element name="on" type="onType"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="fourtyEightType">
            <xs:sequence>
                  <xs:element name="on" type="onType"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="eightyEightPointTwoType">
            <xs:sequence>
                  <xs:element name="on" type="onType"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="ninetySixType">
```

```
            <xs:sequence>
                  <xs:element name="on" type="onType"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="mLANAutoWordclockParameterType">
            <xs:sequence>
                  <xs:element name="on" type="onType"/>
                  <xs:element name="off" type="offType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="wordclockSelectParametersType">
            <xs:sequence>
                  <xs:element name="wordclockSelectParameter"
type="wordclockSelectParameterType" maxOccurs="unbounded"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="wordclockSelectParameterType">
            <xs:sequence>
                  <xs:element name="on" type="onType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="selectType">
            <xs:sequence>
                  <xs:element name="inputs" type="inputsType"/>
                  <xs:element name="outputs" type="outputsType"/>
                  <xs:element name="patches" type="patchesType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="default" type="xs:boolean"/>
            <xs:attribute name="type" type="xs:string"/>
      </xs:complexType>
      <xs:complexType name="inputsType">
            <xs:choice maxOccurs="unbounded">
                  <xs:element name="input" type="inputType"
maxOccurs="unbounded"/>
```

```
                  <xs:element name="channelPair" type="channelPairType"
minOccurs="0" maxOccurs="unbounded"/>
            </xs:choice>
      </xs:complexType>
      <xs:complexType name="outputsType">
            <xs:choice maxOccurs="unbounded">
                  <xs:element name="output" type="outputType"
maxOccurs="unbounded"/>
                  <xs:element name="channelPair" type="channelPairType"
minOccurs="0" maxOccurs="unbounded"/>
            </xs:choice>
      </xs:complexType>
      <xs:complexType name="patchesType">
            <xs:choice maxOccurs="unbounded">
                  <xs:element name="patch" type="patchType"
maxOccurs="unbounded"/>
                  <xs:element name="patchProcessorGroup"
type="patchProcessorGroupType" minOccurs="0" maxOccurs="unbounded"/>
            </xs:choice>
      </xs:complexType>
      <xs:complexType name="patchProcessorGroupType">
            <xs:sequence>
                  <xs:element name="patch" type="patchType"
maxOccurs="unbounded"/>
                  <xs:element name="off" type="offType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:sequence>
            <xs:attribute name="alwaysHasOneSelected" use="required">
                  <xs:simpleType>
                        <xs:restriction base="xs:boolean"/>
                  </xs:simpleType>
            </xs:attribute>
      </xs:complexType>
      <xs:complexType name="inputType">
            <xs:choice minOccurs="0">
                  <xs:element name="parameters" type="parametersType"/>
            </xs:choice>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="isBus" type="xs:boolean" use="required"/>
            <xs:attribute name="type" type="xs:string"/>
      </xs:complexType>
      <xs:complexType name="outputType">
            <xs:choice minOccurs="0">
                  <xs:element name="parameters" type="parametersType"/>
            </xs:choice>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="isBus" type="xs:boolean" use="required"/>
            <xs:attribute name="alwaysHasASource" type="xs:boolean"/>
            <xs:attribute name="type" type="xs:string"/>
      </xs:complexType>
      <xs:complexType name="patchType">
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                  <xs:element name="parameters" type="parametersType"/>
            </xs:choice>
            <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="channelPairType">
            <xs:all>
                  <xs:element name="channelPairOnOptions"
type="channelPairOnOptionsType"/>
                  <xs:element name="off" type="offType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
                  <xs:element name="channelsToPair" type="channelsToPairType"/>
            </xs:all>
            <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="channelsToPairType">
            <xs:choice>
                  <xs:element name="input" type="inputType" minOccurs="2"
maxOccurs="2"/>
                  <xs:element name="output" type="outputType" minOccurs="2"
maxOccurs="2"/>
            </xs:choice>
      </xs:complexType>
      <xs:complexType name="channelPairOnOptionsType">
            <xs:sequence>
                  <xs:element name="channelPairOnOption"
type="channelPairOnOptionType" maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="channelPairOnOptionType">
            <xs:sequence>
```

```
                  <xs:element name="on" type="onType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="parametersType">
            <xs:all>
                  <xs:element name="patchParameter" type="patchParameterType"
minOccurs="0"/>
                  <xs:element name="dynamicsProcessorGroups"
type="dynamicsProcessorGroupsType" minOccurs="0"/>
                  <xs:element name="effectsProcessor"
type="effectsProcessorType" minOccurs="0"/>
                  <xs:element name="volumeParameter" type="volumeParameterType"
minOccurs="0"/>
                  <xs:element name="muteParameter" type="muteParameterType"
minOccurs="0"/>
                  <xs:element name="panParameters" type="panParametersType"
minOccurs="0"/>
                  <xs:element name="eq" type="eqType" minOccurs="0"/>
            </xs:all>
      </xs:complexType>
      <xs:complexType name="patchParameterType">
            <xs:sequence>
                  <xs:element name="on" type="onType"/>
                  <xs:element name="off" type="offType" minOccurs="0"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType" minOccurs="0"/>
            </xs:sequence>
            <xs:attribute name="worksWithPairedChannel" type="xs:boolean"
default="false"/>
      </xs:complexType>
      <xs:complexType name="dynamicsProcessorGroupsType">
            <xs:sequence>
                  <xs:element name="dynamicsProcessorGroup"
type="dynamicsProcessorGroupType" maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="dynamicsProcessorGroupType">
            <xs:sequence>
                  <xs:element name="dynamicsProcessorOnParameter"
type="dynamicsProcessorOnParameterType"/>
```

```
                  <xs:element name="dynamicsProcessorLibrary"
type="dynamicsProcessorLibraryType"/>
                  <xs:element name="dynamicsProcessors"
type="dynamicsProcessorsType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="worksWithPairedChannel" type="xs:boolean"
default="false"/>
      </xs:complexType>
      <xs:complexType name="dynamicsProcessorOnParameterType">
            <xs:sequence>
                  <xs:element name="parameterHelpString"
type="parameterHelpStringType"/>
                  <xs:element name="parameterValues"
type="parameterValuesType"/>
                  <xs:element name="on" type="onType"/>
                  <xs:element name="off" type="offType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="worksWithPairedChannel" type="xs:boolean"
default="false"/>
      </xs:complexType>
      <xs:complexType name="dynamicsProcessorLibraryType">
            <xs:sequence>
                  <xs:element name="dynamicsProcessorLibraryTitle"
type="dynamicsProcessorLibraryTitleType" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="dynamicsProcessorLibraryTitleType">
            <xs:sequence>
                  <xs:element name="on" type="onType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="type" type="xs:string" use="required"/>
            <xs:attribute name="worksWithPairedChannel" type="xs:boolean"
use="optional" default="false"/>
      </xs:complexType>
      <xs:complexType name="dynamicsProcessorsType">
            <xs:sequence>
```

```
                  <xs:element name="dynamicsProcessor"
type="dynamicsProcessorType" maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="dynamicsProcessorType">
            <xs:sequence>
                  <xs:element name="on" type="onType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
                  <xs:element name="dynamicsProcessorParameter"
type="dynamicsProcessorParameterType" maxOccurs="unbounded"/>
                  <xs:element name="keyInSources" type="keyInSourcesType"
minOccurs="0"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="type" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="dynamicsProcessorParameterType">
            <xs:all>
                  <xs:element name="parameterValues" type="parameterValuesType"
minOccurs="0"/>
                  <xs:element name="parameterHelpString"
type="parameterHelpStringType" minOccurs="0"/>
                  <xs:element name="parameterChange"
type="parameterChangeType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:all>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="type" type="xs:string" use="required"/>
            <xs:attribute name="worksWithPairedChannel" type="xs:boolean"
use="optional" default="false"/>
      </xs:complexType>
      <xs:complexType name="keyInSourcesType">
            <xs:sequence>
                  <xs:element name="keyInSourceGroup"
type="keyInSourceGroupType" maxOccurs="unbounded"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType" minOccurs="0"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="keyInSourceGroupType">
```

```
            <xs:sequence>
                  <xs:element name="on" type="onType" minOccurs="0"/>
                  <xs:element name="keyInSourceParameter"
type="keyInSourceParameterType" maxOccurs="unbounded"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="keyInSourceParameterType">
            <xs:sequence>
                  <xs:element name="parameterChange"
type="parameterChangeType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="effectsProcessorType">
            <xs:sequence>
                  <xs:element name="effectsProcessorOnParameter"
type="effectsProcessorOnParameterType"/>
                  <xs:element name="effectsProcessorLibrary"
type="effectsProcessorLibraryType"/>
                  <xs:element name="effectsProcessorTypes"
type="effectsProcessorTypesType"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="effectsProcessorTypesType">
            <xs:sequence>
                  <xs:element name="effectsProcessorType"
type="effectsProcessorTypeType" maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="effectsProcessorTypeType">
            <xs:sequence>
                  <xs:element name="effectsProcessorParameter"
type="effectsProcessorParameterType" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="type" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="effectsProcessorParameterType">
            <xs:all>
```

```
                  <xs:element name="parameterHelpString"
type="parameterHelpStringType"/>
                  <xs:element name="parameterValues" type="parameterValuesType"
minOccurs="0"/>
                  <xs:element name="parameterChange"
type="parameterChangeType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:all>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="type" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="effectsProcessorOnParameterType">
            <xs:all>
                  <xs:element name="parameterHelpString"
type="parameterHelpStringType"/>
                  <xs:element name="parameterValues"
type="parameterValuesType"/>
                  <xs:element name="on" type="onType"/>
                  <xs:element name="off" type="offType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:all>
            <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="effectsProcessorLibraryType">
            <xs:sequence>
                  <xs:element name="effectsProcessorLibraryTitles"
type="effectsProcessorLibraryTitlesType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="effectsProcessorLibraryTitlesType">
            <xs:sequence>
                  <xs:element name="effectsProcessorLibraryTitle"
type="effectsProcessorLibraryTitleType" maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="effectsProcessorLibraryTitleType">
            <xs:sequence>
                  <xs:element name="on" type="onType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="type" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="volumeParameterType">
            <xs:all>
                  <xs:element name="addToParameterGroupParameters"
type="addToParameterGroupParametersType" minOccurs="0"/>
                  <xs:element name="parameterHelpString"
type="parameterHelpStringType"/>
                  <xs:element name="parameterValues"
type="parameterValuesType"/>
                  <xs:element name="parameterChange"
type="parameterChangeType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:all>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="worksWithPairedChannel" type="xs:boolean"
default="false"/>
      </xs:complexType>
      <xs:complexType name="muteParameterType">
            <xs:all>
                  <xs:element name="addToParameterGroupParameters"
type="addToParameterGroupParametersType" minOccurs="0"/>
                  <xs:element name="parameterHelpString"
type="parameterHelpStringType"/>
                  <xs:element name="parameterValues"
type="parameterValuesType"/>
                  <xs:element name="on" type="onType"/>
                  <xs:element name="off" type="offType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:all>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="worksWithPairedChannel" type="xs:boolean"
use="optional" default="false"/>
      </xs:complexType>
      <xs:complexType name="panParametersType">
            <xs:sequence>
                  <xs:element name="panParameter" type="panParameterType"
maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="panParameterType">
            <xs:sequence>
                  <xs:element name="parameterValues"
type="parameterValuesType"/>
                  <xs:element name="parameterHelpString"
type="parameterHelpStringType"/>
                  <xs:element name="parameterChange"
type="parameterChangeType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="eqType">
            <xs:sequence>
                  <xs:element name="equaliserOnParameter"
type="equaliserOnParameterType"/>
                  <xs:element name="bands" type="bandsType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="worksWithPairedChannel" type="xs:boolean"
default="false"/>
      </xs:complexType>
      <xs:complexType name="equaliserOnParameterType">
            <xs:all>
                  <xs:element name="parameterHelpString"
type="parameterHelpStringType"/>
                  <xs:element name="parameterValues"
type="parameterValuesType"/>
                  <xs:element name="on" type="onType"/>
                  <xs:element name="off" type="offType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:all>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="worksWithPairedChannel" type="xs:boolean"
default="false"/>
      </xs:complexType>
      <xs:complexType name="bandsType">
            <xs:sequence>
```

```
                  <xs:element name="band" type="bandType"
maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="bandType">
            <xs:sequence>
                  <xs:element name="eqParameter" type="eqParameterType"
maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="eqParameterType">
            <xs:all>
                  <xs:element name="parameterHelpString"
type="parameterHelpStringType"/>
                  <xs:element name="parameterValues"
type="parameterValuesType"/>
                  <xs:element name="parameterChange"
type="parameterChangeType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:all>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="worksWithPairedChannel" type="xs:boolean"
default="false"/>
      </xs:complexType>
      <xs:complexType name="addToParameterGroupParametersType">
            <xs:sequence>
                  <xs:element name="addToParameterGroupParameter"
type="addToParameterGroupParameterType" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="worksWithPairedChannel" type="xs:boolean"
use="optional" default="false"/>
      </xs:complexType>
      <xs:complexType name="addToParameterGroupParameterType">
            <xs:sequence>
                  <xs:element name="parameterHelpString"
type="parameterHelpStringType" minOccurs="0"/>
                  <xs:element name="on" type="onType"/>
                  <xs:element name="off" type="offType"/>
                  <xs:element name="parameterRequest"
type="parameterRequestType"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="type" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="parameterHelpStringType">
            <xs:attribute name="value" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="parameterValuesType">
            <xs:sequence>
                  <xs:element name="parameterValue" type="parameterValueType"
maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="parameterValueType">
            <xs:attribute name="type" use="required">
                  <xs:simpleType>
                        <xs:restriction base="xs:string">
                              <xs:pattern value="range|single"/>
                        </xs:restriction>
                  </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="value" type="xs:string"/>
            <xs:attribute name="prefix" type="xs:string"/>
            <xs:attribute name="from" type="xs:decimal"/>
            <xs:attribute name="to" type="xs:decimal"/>
            <xs:attribute name="incrementValue" type="xs:decimal"/>
            <xs:attribute name="postfix" type="xs:string"/>
            <!--single-->
            <!--range-->
      </xs:complexType>
      <xs:complexType name="onType">
            <xs:sequence>
                  <xs:element name="parameterChange"
type="parameterChangeType"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="offType">
            <xs:sequence>
                  <xs:element name="parameterChange"
type="parameterChangeType"/>
```

```
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="parameterChangeType">
            <xs:sequence>
                  <xs:element name="midiMessagesGroups"
type="midiMessagesGroupsType"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="parameterRequestType">
            <xs:sequence>
                  <xs:element name="midiMessagesGroups"
type="midiMessagesGroupsType"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="midiMessagesGroupsType">
            <xs:sequence>
                  <xs:element name="midiMessagesGroup"
type="midiMessagesGroupType" maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="midiMessagesGroupType">
            <xs:sequence>
                  <xs:element name="midiMessages" type="midiMessagesType"
maxOccurs="unbounded"/>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="midiMessagesType">
            <xs:sequence>
                  <!--single-->
                  <xs:element name="midiMessage" type="midiMessageType"
minOccurs="0" maxOccurs="unbounded"/>
                  <!--range-->
                  <xs:element name="midiMessagesStartPart"
type="midiMessagesStartPartType" minOccurs="0"/>
                  <xs:element name="midiMessagesVariablePart"
type="midiMessagesVariablePartType" minOccurs="0" maxOccurs="unbounded"/>
                  <xs:element name="midiMessagesEndPart"
type="midiMessagesEndPartType" minOccurs="0"/>
            </xs:sequence>
            <xs:attribute name="type" use="required">
                  <xs:simpleType>
                        <xs:restriction base="xs:string">
```

```
                              <xs:pattern value="range|single"/>
                        </xs:restriction>
                  </xs:simpleType>
            </xs:attribute>
      </xs:complexType>
      <xs:complexType name="midiMessageType">
            <xs:attribute name="value" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="midiMessagesStartPartType">
            <xs:attribute name="value" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="midiMessagesVariablePartType">
            <xs:attribute name="from" type="xs:string" use="required"/>
            <xs:attribute name="to" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="midiMessagesEndPartType">
            <xs:attribute name="value" type="xs:string" use="required"/>
      </xs:complexType>
</xs:schema>
```

Bibliography

Anderson, D. *FireWire System Architecture* (2nd Edition). Addison-Wesley. 1999.

Armstrong, T and Patton, R. *ALT Developer's Guide* (2nd Edition). Hungry Minds, Incorporated. 2000.

Chigwamba, N. and Foss, R. *Enhanced End-User Capabilities in High Speed Audio Networks.* Audio Engineering Society. 2007.

C-Mexx Software. *The Visualizer For The Yamaha 03D Mixing Console.* 1998.

Eargle, J. *Handbook of Recording Engineering* (4th Edition). Kluwer Academic Publishers Group: Boston. 2003.

Elliotte, R. H. and Means, W. S. *XML In A Nutshell* (2nd Edition). O'Reilly & Associates, Inc: Sebastopol, CA. 2002.

EtherSound. *EtherSound.* 2007. Available: http://www.ethersound.com/ [Accessed 27/10/07].

Foss, R. *Audio Engineering – Computer Science Honours Level Course Notes.* 2006.

Foss, R. and Foulkes, P. *The Representation of, and Control over Mixing Desks via a Software-Based Matrix.* Audio Engineering Society. 2006.

Fujimori, J. and Foss, R. *A New Connection Management Architecture for the Next Generation of mLAN.* Audio Engineering Society. 2003a.

Fujimori, J., Foss, R., Klinkrant, B. and Bangay, S. *An mLAN Connection Management Server for Web-Based, Multi-User, Audio Device Patching.* Audio Engineering Society. 2003b.

Mackie, *Mackie.* 2007. Available: http://www.mackie.com/ [Accessed: 14/09/07].

Mawzer, *Mawzer*. 2007. Available: http://www.mawzer.com/ [Accessed: 11/09/07].

MIDI Manufacturers Association. *MIDI Manufacturers Association.* 2007. Available: http://www.midi.org/ [Accessed: 17/09/07].

Microsoft Corporation. *MSDN Library for Visual Studio 2005.* 2005.

Networked Audio Solutions. *mLAN Connection Management Server. Client-Server Communication 0.0.4*. 2004.

Otari, *mLAN Control Software Operation Manual.* 2005.

Robjohns, H. *Patchbays. Frequently asked questions*. 1999. Available: http://www.soundonsound.com/ [Accessed: 15/10/07].

Steinberg. *Steinberg*. 2007. Available: http://www.steinberg.net/ [Accessed: 31/10/07].

Troelsen, A. W. *Developer's Workshop to COM and ATL 3.0.* Wordware Publishing Inc. 2000.

Yamaha. *01V96 Digital Mixing Console Version 2 Owner's Manual.* 2004a.

Yamaha. *01V96 Editor Owner's Manual*. 2004b.

Yamaha. *01X Digital Mixing Studio Owner's Manual.* 2003a.

Yamaha. *03D Digital Mixing Console Owner's Manual.* 1997.

Yamaha. *mLAN Graphic Patchbay Owner's Manual*. 2004c.

Yamaha. *Open Plug-in Technology Specification.* 2002.

Yamaha. *Studio Connections – Total Recall SDK.* 2005.

Yamaha. *Studio Manager for 01X Owner's Manual*. 2003b.