

A CONVENIENT APPROACH TO THE
DETERMINISTIC ROUTING OF MIDI MESSAGES



RHODES UNIVERSITY
Where leaders learn

RHODES UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Submitted in fulfilment
of the requirements of the degree of

MASTER OF SCIENCE

of Rhodes University

Brent Shaw

Grahamstown, South Africa
2 May 2017

Abstract

This research investigates the design and development of a Wireless MIDI Connection Management solution in order to create a deterministic MIDI transmission system. A investigation of the MIDI protocol show it to have certain limitation that can be overcome through the use of transmission solutions. These solutions can be used to improve on the versatility of MIDI while overcoming the MIDI's notorious cable length limitation.

XMOS's deterministic XS1 microcontrollers are used to enable the design of a real-time system. The MIDINet system is investigated to identify both the strengths and weaknesses of such a connection management system, while other systems for network transmission of MIDI messages are reviewed.

These investigations lead to a design concept for a new network MIDI transmission system that allows for the remote management of connections. The design and subsequent implementation of both the transmission system and the connection management system are then detailed.

A testing methodology is then devised to allow for the newly created connection management system to be compared to the MIDINet system. The findings show the deterministic system to have lower latency than that of the MIDINet system, while utilising more compact and power efficient hardware.

Contents

1	Introduction	11
1.1	Document Structure	13
2	MIDI and Control Solutions	15
2.1	Introduction	15
2.2	An Overview of the MIDI Protocol	15
2.2.1	Hardware	16
2.2.2	Messages	18
2.3	MIDI for Control	21
2.3.1	MIDI Show control	22
2.3.2	MIDI Limitations	23
2.4	Solutions to MIDI limitations	25
2.4.1	Radio link transmission	26
2.4.2	Network transmission	26
2.5	Conclusion	35
3	XMOS Architecture	36
3.1	Introduction	36
3.2	Background	37
3.3	Difference between XMOS and Other Microcontroller Architectures	38
3.3.1	XS1 Architecture	38
3.4	XMOS-based Hardware Development Kits	40
3.4.1	Slicekit Core Board	40
3.4.2	Ethernet Slice	41
3.4.3	WiFi Slice	41
3.4.4	Audio slice	42
3.5	Conclusion	42

4	Network support	44
4.1	Introduction	44
4.2	Layer Model	45
4.2.1	Physical layer	46
4.2.2	Data Link layer	46
4.2.3	Network layer	47
4.2.4	Transport layer	48
4.2.5	Session, Presentation and Application layers	51
4.3	Support structures	51
4.3.1	Addressing	51
4.3.2	Multicast group messaging	52
4.3.3	Name resolution	53
4.3.4	Real-time transmission	54
4.4	Conclusion	55
5	Analysis of MIDINet System	57
5.1	Introduction	57
5.2	MIDINet Layer Model	59
5.3	System Design	60
5.3.1	User interface	60
5.3.2	Object Based Design	62
5.3.3	Network class	65
5.4	Operation	66
5.4.1	Start-up procedure	66
5.4.2	MIDINet Discovery	66
5.4.3	Message Routing	67
5.5	Limitations of the MIDINet system	68
5.6	Conclusion	68
6	Design	70
6.1	Introduction	70
6.2	System overview	71
6.3	Hardware	72
6.3.1	Hardware specification	72
6.3.2	Hardware configuration	72
6.4	Software	73
6.4.1	Design Methodologies	73

6.4.2	System Context	75
6.4.3	Routing MIDI Messages	77
6.4.4	Managing Connections	85
6.4.5	Providing User Controls	91
6.5	Conclusion	94
7	Implementation	95
7.1	Introduction	95
7.2	Implementation Model	95
7.2.1	Network Processor	97
7.2.2	ConMan Processor	98
7.3	MIDI Handler	98
7.3.1	Processing MIDI input	100
7.3.2	Producing MIDI output	106
7.4	ConMan Core	109
7.4.1	IDs and Routing masks	112
7.4.2	MIDI to Network routing	113
7.4.3	Network to MIDI routing	113
7.5	Networking	114
7.5.1	Non-real-time Networking	115
7.5.2	Real-time Networking	115
7.6	Web Application	120
7.6.1	Web Based Connection Management Interface	121
7.6.2	Embedded Web Server	122
7.6.3	Web Page Functions	123
7.6.4	Application Handler	124
7.7	Connection Management	125
7.7.1	MIDINet Module	126
7.7.2	Connection Handler	127
7.7.3	ID and Routing Mask Creation	129
7.8	Conclusion	131
8	Testing and Analysis	132
8.1	Introduction	132
8.2	Testing Methodology	132
8.3	USB MIDI Interface testing	134
8.3.1	Apparatus	134

8.3.2	Testing	136
8.3.3	Results	136
8.4	Apparatus testing	139
8.4.1	Testing	139
8.4.2	Results	140
8.5	Analysis of a Typical MIDI Performance	140
8.5.1	Analysis Software	141
8.5.2	Analysis	142
8.5.3	Findings	142
8.6	Stream Testing and Analysis	144
8.6.1	MIDINet test system specification	144
8.6.2	ConMan test system specification	145
8.6.3	Monitoring system specifications	146
8.6.4	System testing	146
8.6.5	Analysis	147
8.6.6	Findings	148
8.7	Interpretation	150
8.8	Conclusion	150
9	Conclusion	152
9.1	Problem statement and Hypothesis	152
9.2	Existing solutions to the problem	153
9.3	Embedded platform and reasons for its choice	153
9.4	The design and its part in lowering latency	154
9.5	Web based control over connections	155
9.6	Discoveries arising from testing	155
9.7	Confirmation of hypothesis	156
9.8	Possible future work	156
9.8.1	XMOS WiFi Slice	156
9.8.2	Compared transmission systems	157
9.8.3	MIDI message source	157
9.8.4	Testing methodology	157
9.8.5	SysEx messaging	158
9.8.6	Connection Management	158
9.8.7	User interface	158
	Appendices	165

A	xC Language Overview	166
A.1	Parallel tasks	166
A.2	Channels	166
A.2.1	Channel Transactions	168
A.3	Select Statement	168
A.4	Ports and Pins	169
A.5	C and C++ integration	169
B	ConMan System Design	172
B.1	ConMan: Essential Model	172
B.1.1	Level 0	173
B.1.2	Level 1	174
B.1.3	Level 2	176
B.2	ConMan: Implementation Model	184
B.2.1	ConMan Processor	185
B.3	MIDINet: Object-Oriented Design	187
C	MIDINet test system hardware	189

List of Tables

2.1	Table showing various types of channel voice messages. [8]	19
2.2	Table showing various types of channel mode messages.	20
2.3	Table showing System common messages.	20
2.4	Table showing System real-time messages.	21
6.1	Corruption of MIDI through merging when using running status	81
8.1	Measured Latency of tested USB MIDI interfaces (in ms)	138
8.2	Distribution of MIDI messages by Status Byte	143
8.3	Distribution of MIDI messages over MIDI channels	143
8.4	System latencies per message (ms)	148

List of Figures

2.1	Electrical layout for the In and Thru ports as described in the MIDI specification.	17
2.2	Electrical layout for the Out ports as described in the MIDI specification. .	18
2.3	Example of MIDI being used for synchronised control between a digital mixing console and a digital audio workstation.	22
2.4	Example of a radio link MIDI solution being used for synchronised control between a digital mixing console and a digital audio workstation.	26
2.5	The rackmountable Lone Wolf MidiTap unit.	27
2.6	ipMIDI user interfaces.	28
2.7	ipMIDI virtual ports in MIDI-OX.	28
2.8	Creating a new session.	29
2.9	Remote sessions are visible in Directory	30
2.10	Current connections are shown in the session panel	30
2.11	The KISS BOX MIDI solution.	30
2.12	Overview of the KissBox Editor software	31
2.13	KissBox configuration	31
2.14	Uploading information to KissBox	31
2.15	MIDINet Connection Management Dialogue.	33
3.1	The underlying XS1-L2 16 core architecture.	39
3.2	The Slicekit Core board.	40
3.3	XMOS's Ethernet Slice.	41
3.4	WiFi slice containing the TiWi-SL wireless module.	41
3.5	Slice for audio development.	42
4.1	The TCP/IP and OSI layer models.	45
4.2	IP pseudo header	47
4.3	Overview of a MAC address	48

4.4	UDP layout	49
4.5	TCP header layout	50
4.6	RTPlayout	55
5.1	Example configuration of a MIDINet system.	58
5.2	Relation of MIDINet layers to TCP/IP Layer model.	59
5.3	MIDINet Message in UDP Datagram	60
5.4	MIDINet Identification Dialogue.	61
5.5	MIDINet Connection Management Dialogue.	61
5.6	MIDINet Connections.	61
5.7	Class diagram for MIDINet system.	62
5.8	Flowchart for the ProcessMIDIdata function.	64
5.9	Discovery: Network probe from local unit.	66
5.10	Discovery: Response from remote MIDINet system.	67
6.1	Example configuration of a MIDI ConMan system.	71
6.2	SliceKit usage and configuration	73
6.3	Context diagram from MIDI ConMan system.	75
6.4	MIDI ConMan system for keyboard control of synthesiser.	75
6.5	Ward & Mellor System overview for a ConMan device.	77
6.6	Detailed view of the Route MIDI Messages task.	78
6.7	Group of tasks responsible for handling MIDI ports.	79
6.8	Task 1.1: Reading MIDI message	80
6.9	Example configuration for a MIDI merge	81
6.10	Removal of Running Status during MIDI input handling.	82
6.11	Task 1.2: Sending MIDI message	83
6.12	Group of tasks responsible for handling MIDINet messages.	84
6.13	Group of tasks responsible for routing MIDI messages.	85
6.14	Detailed view of the Manage Connections task.	86
6.15	Group of tasks responsible for handling connections.	87
6.16	Simplified class diagram for MIDINet module.	87
6.17	Group of tasks responsible for processing MIDINet commands.	88
6.18	Group of tasks responsible for handling routing information.	89
6.19	Example ConMan unit number when represented in binary.	89
6.20	Example of ConMan units compared to a routing mask.	90
6.21	Detailed view of the 'Provide User Controls' task.	91
6.22	Task and data flows associated with building a page.	92

6.23	Task and data flows associated with serving a page.	93
6.24	Task and data flows associated with handling a request page.	94
7.1	System Processor View	96
7.2	Network Processor: Parallel task structure	97
7.3	ConMan Processor: Parallel task structure	98
7.4	Handle MIDI Ports: Overview of the parallel tasks	99
7.5	Handle MIDI Input: Overview of the parallel tasks	100
7.6	Signal view of MIDI byte transmission	101
7.7	midiReadRawMore: Handling of received byte.	103
7.8	MIDI Output task: Overview of the parallel tasks	107
7.9	ConMan Core - the real-time routing of MIDI messages	110
7.10	Network Processor: Overview of the parallel task structure	114
7.11	Manage Real-Time Network: Overview of the parallel tasks	116
7.12	Manage Real-Time Network: Tasks related to MIDI communication	117
7.13	Manage Real-Time Network: Tasks related to MIDINet commands	119
7.14	Web Application: Handling Web Requests	121
7.15	Screenshot of connection management web page.	121
7.16	Connection Server: Overview of the parallel tasks	126
8.1	MIDI and Network testing apparatus	134
8.2	MIDI and Network testing apparatus	139
8.3	The MIDI Reader application	139
8.4	The MIDI Reader application	140
8.5	Bytes per second during MIDI performance	143
8.6	MIDINet Test system	145
8.7	ConMan Test System	145
8.8	Abstract overview of testing	147
8.9	MIDINet system latencies	149
8.10	ConMan system latencies	149
8.11	Density of MIDI performance in messages per second.	149
8.12	Comparison of latency: MIDINet and ConMan	150
B.1	Context diagram for MIDI ConMan	172
B.2	Level 0 - Manage ConMan System.	173
B.3	Task 1 - Route MIDI messages.	174
B.4	Task 2 - Manage connections.	175
B.5	Task 3 - Provide User Interface.	175

B.6 Task 1.1 - Read MIDI In.	176
B.7 Task 1.2 - Send MIDI Out.	176
B.8 Task 1.3 - Check Loopback flag.	177
B.9 Task 1.4 - Add Unit ID.	177
B.10 Task 1.5 - Route MIDI.	178
B.11 Task 1.6 - Send MIDINet Message.	178
B.12 Task 1.7 - Receive MIDINet Message.	179
B.13 Task 2.1 - Create Routing Mask.	179
B.14 Task 2.2 - Process MIDI Command.	180
B.15 Task 2.3 - Update MIDINet system.	180
B.16 Task 2.4 - Send MIDINet Command.	181
B.17 Task 2.5 - Create ID.	181
B.18 Task 2.6 - Process User Input.	182
B.19 Task 3.1 - Build page.	182
B.20 Task 3.2 - Parse request.	183
B.21 Task 3.3 - Serve Page.	183
B.22 System Processor View	184
B.23 Overview of the parallel task structure for Network Processor	184
B.24 Overview of the parallel task structure for ConMan Processor	185
B.25 Task 2.1 - Manage real time network	185
B.26 Task 2.2 - Manage Connection Server	186
B.27 Task 2.4 - Route MIDI Messages	186
B.28 Task 2.5 - Handle MIDI Ports	186
B.29 Task 2.5.1 - MIDI Output handling	187
B.30 Task 2.5.2 - MIDI Input handling	187
B.31 MIDINet classes within the Connection Server component.	187
B.32 Request Configuration Sequence Diagram.	188
B.33 Respond to Configuration Request Sequence Diagram.	188
B.34 Provide Configuration Sequence Diagram.	188
B.35 Transmit Local MIDINet Unit Info Sequence Diagram.	188
B.36 Add MIDINet Sequence Diagram.	188
B.37 Make Connection Sequence Diagram.	188
B.38 Add Connection Sequence Diagram.	188
B.39 Destroy Connection Sequence Diagram.	188
B.40 Transmit MIDINet Sequence Diagram.	188
B.41 Route MIDINet Message Sequence Diagram.	188
B.42 Network Disconnect Sequence Diagram.	188

Chapter 1

Introduction

Typically, audio engineers are tethered to a control center containing a mixing console and outboard units where they adjust a range of signal processing parameters. The mixing consoles tend to be large and weighty, and are often installed in a fixed location, only being moved when absolutely necessary.

Studios as well as larger venues are likely to have more than one mixing console and other signal processing devices, and these may not be positioned in close proximity to each other. Maintaining control of this equipment while remaining mobile is therefore a problem which is faced in many environments.

The Musical Instrument Digital Interface (MIDI) is an audio control protocol that was standardised in 1983. The protocol makes use of unidirectional serial messaging to transmit messages from one audio device to another. The protocol does not transmit audio, but transmits messages that are used for the production of audio or for the control of audio devices[29].

MIDI capable devices can include sound synthesisers, sound effects processors, audio recorders and even audio mixing consoles. The messaging format employed by MIDI allows for sound generators and signal processors to be controlled remotely through the use of a MIDI controller or computer[49].

Although MIDI can be used for a variety of tasks, the specification has limitations that reduce the usability of the protocol. A significant problem is that the MIDI specification limits the length of a MIDI cable to 15m due to signal attenuation [8].

MIDI has been widely used in live audio and studio environments. An extension to the

MIDI protocol has allowed it to be used for Show Control: the control of equipment used in entertainment and set management, including lights, actuators and pyrotechnics[34].

MIDI installations in all three of these environments suffer because of the cable length limitation. This has led to the development of solutions that circumvent the MIDI cable length limitation, allowing MIDI devices to be placed further than 15m apart.

Systems are available that allow for MIDI messages to be transmitted wirelessly. These systems range from radio-link point-to-point transmission systems, to network transmission systems.

Radio-link point-to-point solutions manage to overcome the cable length limitation, but remain unidirectional, resulting in a situation where the cable length is no longer an issue, but the routing of connections must still be completed manually. This form of wireless MIDI system does not allow for multiple device control or bidirectional transmission, only offering an increased distance between MIDI devices[57].

There are transmission systems that make use of wired or wireless networks to provide MIDI transmission. The features of these solutions vary, with some only providing network transmission, and others offering distributed control and connection management.

The MIDINet system is a distributed solution, utilising multicast messaging over a wired network. This allows for many-to-many transmission of MIDI messages. The system requires that all MIDI devices be attached to a computer running MIDINet [16]. Furthermore the latency of the transmission is non-deterministic, as the desktop computer and operating system which the system runs are not designed for deterministic execution.

At the outset of this project, no system deployed on small, portable embedded computing devices offered a distributed MIDI transmission system over wired or wireless networks, as well as ensuring deterministic transmission.

The hypothesis that motivated this thesis is that such a system is possible, and furthermore that deterministic processing on the communicating nodes ensures deterministic transmission. To test this hypothesis, a MIDI connection management system has been developed using XMOS-based development kits as the networked nodes.

XMOS microcontrollers make use of a multicore architecture that allows for deterministic processing. These microcontrollers have been shown to work in real-time audio applications such as Audio-Video Bridging (AVB), Controller Area Networks (CAN bus) and Digital Signal Processing (DSP). Hardware development kits are available that offer

the features and connectivity to build a MIDI transmission and connection management solution[71].

1.1 Document Structure

Chapter 2 investigates the MIDI protocol and its use for device control. The underlying protocol is examined and popular use cases reviewed so that insight can be gained into the use of the protocol in live, studio and show control environments.

The chapter also investigates the shortcomings of the MIDI protocol, and includes a survey of some existing solutions for overcoming these shortcomings, such as remote connection management and networked transmission.

Chapter 3 focuses on multicore microcontrollers. The chapter investigates the architecture of the XMOS xCore microcontrollers and lists a few prototyping and development kits that are available for this platform.

The chapter also discusses how the architecture of XMOS's microcontrollers differs from that of standard microcontrollers.

Chapter 4 provides a brief investigation into network support structures that can be utilised in the creation of an embedded system. The chapter also looks at various networking protocols and technologies that could be used in the development of a MIDI connection management system.

Chapter 5 examines the MIDINet application. The examination provides a description of the object oriented design, communication protocol and overview of the system operation.

The chapter investigates the limitations of the MIDINet implementation and how these limitations can be overcome.

The design of a MIDI connection management system, the ConMan system, is detailed in chapter 6. The ConMan system is a combination of hardware that makes use of multicore microcontrollers, real-time design techniques and an adapted design for an object oriented connection server based on the MIDINet messaging protocol.

Chapter 7 discusses the implementation of the ConMan system using XMOS development kits. The chapter covers the optimisations made during the implementation and a thor-

ough explanation of the methods used in the implementation of the parallel components within the ConMan system.

Testing and analysis of the ConMan system are described in Chapter 8. The chapter includes the development of a methodology for the testing of MIDI connection management systems. The methodology itself is tested within the chapter.

Testing included the use a MIDI file that was used to simulate the load that would be placed on a system during a live MIDI performance. This analysis of MIDI performance characteristics was used to test MIDI connection management systems to ascertain their performance under load.

Chapter 9 includes concluding remarks and an overview of the findings obtained from the research. Future work that might improve the MIDI connection management system is discussed.

Chapter 2

MIDI and Control Solutions

2.1 Introduction

The Musical Instrument Digital Interface (MIDI) is a widely used control protocol originally designed to control audio devices. It has been used for the control of synthesisers, audio processors and mixing consoles[56].

This chapter provides a brief history of the MIDI protocol, and will review the advantages and disadvantages of the protocol.

Although thought of as an audio device control protocol, MIDI has seen extensions that have allowed it to be used in environments such as Show Control. These extensions, as well as the impact they have on the usability of the protocol, are discussed.

This is followed by a summary of some alternatives and extensions to MIDI that aim to improve flexibility and control through the use of MIDI transmission systems.

2.2 An Overview of the MIDI Protocol

MIDI is a protocol originally designed to enable the transmission of note messages between keyboard controllers and synthesisers [36].

The advent of synthesizers revolutionized the music industry in the 1960s and 1970s, with musicians being capable of producing a wide range of new sounds. Analogue synthesisers were monophonic, allowing for only one note to be reproduced at a time through the use

of Control Voltages (CV) and gates/triggers. When digital synthesisers started to emerge, manufacturers were not using any standard for control, leading to many different control systems that could seldom allow for early digital synthesisers to be used together[50].

Dave Smith and Chet Wood, engineers working with Sequential Circuits, began work on a new protocol that would allow for the interoperability of synthesizers across different manufacturers. In 1981 they proposed their USI (Universal Synthesizer Interface) protocol to the Audio Engineering Society [20]. Manufacturers and members of the audio engineering community began working on a standard for synthesizer control, and in 1983 the MIDI 1.0 Specification was released [19].

The MIDI standard was widely accepted with many new technologies at the time being released with MIDI compatibility. Today MIDI is still widely used and has been incorporated into synthesizers, discrete keyboards, effects units, guitars and mixing consoles. MIDI has since become a key component of computer controlled music production[20, 19, 23].

The application scope of MIDI has been expanded. For example, MIDI is also used to send control messages to audio devices, digital audio workstations and show control devices.

2.2.1 Hardware

The connector employed by MIDI is a 5 pin DIN connector (DIN 41524). The MIDI specification states that the connectors will be employed for simplex communication. Despite the connectors featuring 5 separate pins, the MIDI specification only uses 3 of these for output connections and 2 on MIDI input connectors[23].

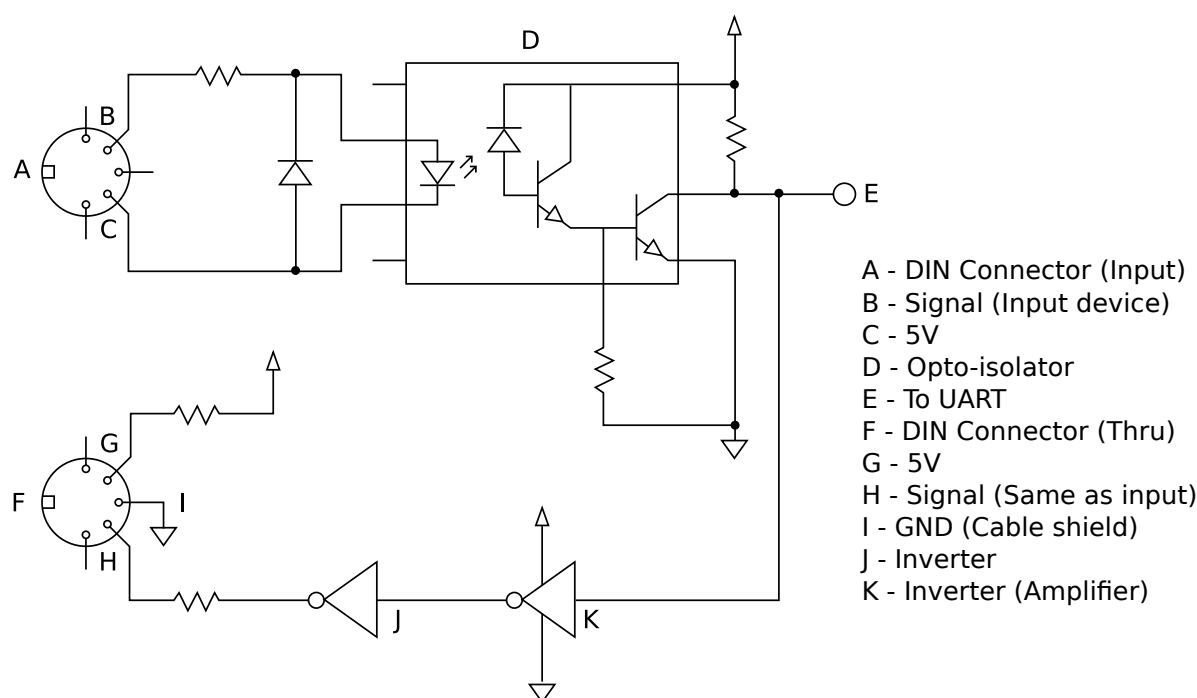


Figure 2.1: Electrical layout for the In and Thru ports as described in the MIDI specification.

Most MIDI devices offer a selection of three MIDI ports, namely input, output and through ports. The through (often written “Thru” as shown in Figure 2.1) is optional and allows for the daisy-chaining¹ of MIDI devices[23]. Figure 2.1 shows the configuration of the MIDI In and Thru connectors.

The MIDI specification requires that the interface operates at 31.25 kbaud in an asynchronous manner. The interface’s receiver is required to be opto-isolated and an output should not be allowed to feed more than one input to ensure signal strength[20].

MIDI splitting devices can be used to allow one MIDI device to send messages to multiple devices. These splitters can allow for a single keyboard controller to feed multiple synthesisers, without the need to daisy-chain the synthesisers using MIDI Thru ports.

¹Daisy-chaining is where each device provides a connection to be made to the next device in a sequence. Information received by a device is repeated out to the next device in the 'chain'. This can allow for a device with only one MIDI output to send messages to multiple receiving devices.

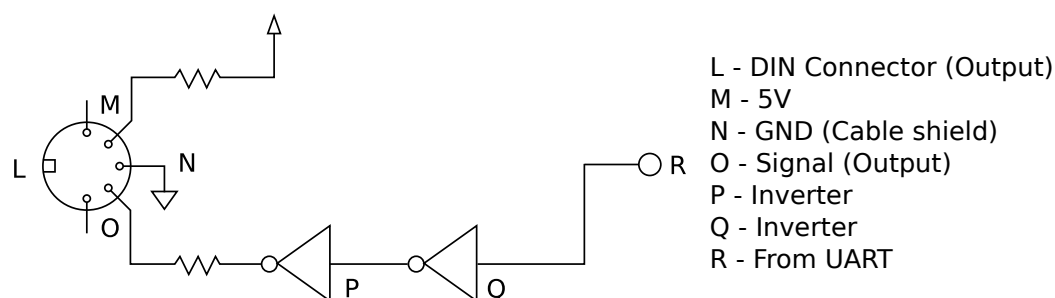


Figure 2.2: Electrical layout for the Out ports as described in the MIDI specification.

Figure 2.2 shows the electrical specification for MIDI outputs. These outputs are driven by some form of microcontroller. The output provides its own power to drive the optoisolator of the input that the MIDI out is connected to.

2.2.2 Messages

There are two main types of MIDI messages: Channel messages and System messages. A Channel message consists of a status byte, followed by two or three data bytes (depending on the message sub-type). System messages are not channel specific and are transmitted globally to all MIDI devices[35].

2.2.2.1 Channel Messages

MIDI devices can transmit control messages on a selected MIDI channel. These messages are referred to as Channel messages, which can be further categorised as Vhannel Voice messages and Channel Mode messages[8].

The former, Channel Voice messages (Table 2.1), can be used to transmit note messages, program messages and expressive messages. A note message is generated when a key on a keyboard controller is pressed; this is referred to as a “note-on” message. When the depressed key is lifted, a “note-off” message is sent[25]. The destination channel is represented by the least significant 4 bits of the Status Byte.

Note messages are separated into “note-on” and “note-off” message. The necessity for both “on” and “off” messages is owing to the fact that if the system was required to wait for the user to complete playing a note before transmission, considerable delay would be incurred[50]. When a key is pressed, the “note-on” message is sent with information such as which key was pressed (note number) and how fast (velocity)[19]. Program Change

Status Byte	Hex	Data Byte 1	Data Byte 2
Note off	80 - 8F	Note number	Release velocity
Note on	90 - 9F	Note number	Attack velocity
Poly key pressure	A0 - AF	Note number	Pressure value
Control change	B0 - BF	Controller ID	Controller value
Program change	C0 - CF	Patch number	-
Channel pressure	D0 - DF	Pressure value	-
Pitch wheel change	E0 - EF	Pitch bend (LSB)	Pitch bend (MSB)

Table 2.1: Table showing various types of channel voice messages. [8]

messages are used to set the patch or program being used on the receiving device. These are used to select instrument types on synthesisers.

Polyphonic key pressure and Channel pressure messages, also known as “Aftertouch”, are types of messages that can be produced by pressing a key down harder after the key has bottomed out[35]. Although these messages are generated in similar ways, they are two different messages. Polyphonic aftertouch provides both the note as well as the pressure associated with that note, while Channel aftertouch only provides the highest pressure level of all currently depressed keys.

Pitch Bend messages are sent using the current position of the pitch bend wheel. This message uses two data bytes with the 14 bit Pitch Bend value being sent as two 7 bit data bytes[35].

Control Change messages, often shown as “CC”, are used to control parameters such as volume, balance and modulation. Channel Mode messages are technically separate from Channel Voice messages, as they do not pertain to the modification or production of the voice or sound of a synthesiser, but these messages use the same status byte as Control Change messages[50].

Channel Mode messages are used to instruct devices on how they should respond to Channel Voice messages. Table 2.2 show the different types of Channel Mode messages, and how the second data byte should be used.

Data Byte 1	Hex	Data Byte 2
Reset all controllers	79	None; set to 0
Local control	7A	0 = off; 127 = on
All notes off	7B	None; set to 0
Omni mode off	7C	None; set to 0
Omni mode on	7D	None; set to 0
Mono mode on	7E	**
Poly mode on	7F	None; set to 0

Table 2.2: Table showing various types of channel mode messages.

2.2.2.2 System Messages

In contrast to Channel messages (which are channel specific), MIDI System messages are broadcast to all MIDI devices. MIDI System Messages are split into two sub-types: System Real-time and System Common.

Message Type	Status Hex	Data Byte 1	Data Byte 2
System Exclusive	F0	-	-
MIDI Time Code	F1	Message type and Values	-
Song Position Pointer	F2	Position pointer	Beats
Song Select	F3	Sequence	-
Tune Request	F6	-	-
End of Exclusive	F7	-	-

Table 2.3: Table showing System common messages.

System Common Messages System Common messages (shown in the Table 2.3) are mostly used for tracking the position within a song, as well as tuning analogue synthesisers. Although System Exclusive messages are part of the System Common message sub-type according to their status byte, this type of message is regarded as being its own sub-type.

MIDI Time Code (MTC) information provides synchronisation for audio and video equipment. For example, MTC can be used to synchronise an analogue multi-track recorder

with a computer running a MIDI sequencer. MTC is a MIDI based version of the SMPTE² timecode system. [52]

System Exclusive messages (SysEx) enable the sending of device-specific instructions. These messages have no predefined structure, allowing for manufacturers to create their own messaging protocols, and utilise MIDI connectivity on devices such as effects units, mixing consoles and other audio equipment. There are SysEx start and SysEx end bytes between which messages need not follow any format, but only conform to the MIDI transmission rate.

Message Type	Status Hex
Timing Clock	F8
Start Sequence	FA
Continue Sequence	FB
Stop Sequence	FC
Active Sensing	FE
System Reset	FF

Table 2.4: Table showing System real-time messages.

System Real-time Messages Table 2.4 shows system real-time messages, which are predominantly used for the control of MIDI sequences.

2.3 MIDI for Control

MIDI, although not initially designed for such use, can be implemented for use in controlling devices other than audio generators and effects units in the recording or production studio. Devices such as effects units and mixing desks can be controlled using control messages or system exclusive messages with manufacturer specific layouts. One capability gained through using MIDI-controlled mixing consoles is that of automation. In the case of mixing audio, automation can be used to change the various input and output channel levels of a digital mixing console, without the engineer adjusting the fader by hand throughout the song. The ability to program the parameter changes can not only

²Society of Motion Picture Television Engineers (SMPTE) provides a standard for labelling video (and audio) frames, allowing multiple sources to be timecoded and synchronised.

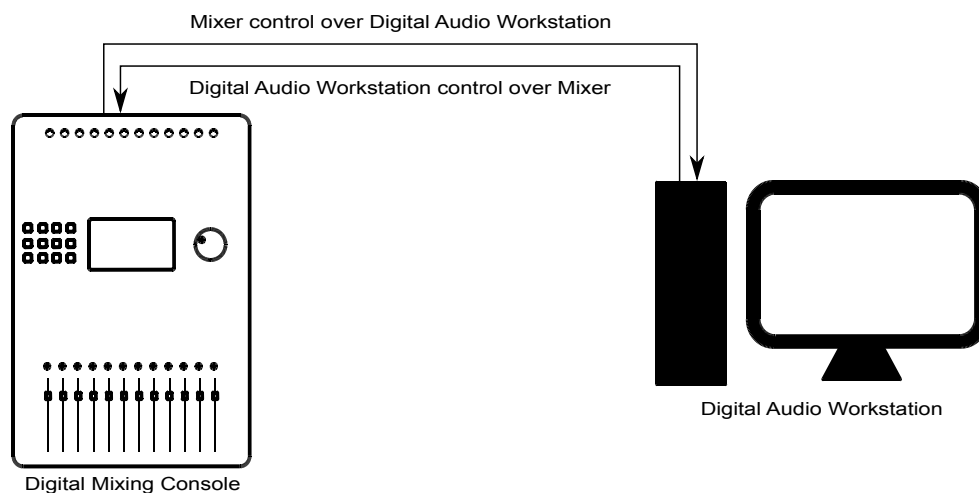


Figure 2.3: Example of MIDI being used for synchronised control between a digital mixing console and a digital audio workstation.

make mixing a multitrack recording a lot simpler, but can also allow for engineers to attempt intricate mixing techniques[20].

This can all be achieved by using a MIDI sequencer³ to store MIDI messages and have them played back during mixdown[50], although it should be noted that MIDI is not required to automate all digital mixing consoles. The incorporation of MIDI sequencing into Digital Audio Workstations (DAWs) has resulted in the creation of a simple method of providing control for mixing consoles and effects processors as shown by the example configuration in Figure 2.3. Users can change parameters via a GUI interface and have the changes reflected on the device being controlled. These changes can be recorded, modified and re-used[20].

2.3.1 MIDI Show control

MIDI Show Control (MSC) is an extension of the MIDI 1.0 specification that aimed to expand the use of the MIDI messaging format so that it could be applied to live performance systems for theatre and lighting as well as multi-media and audio-visual systems. [34]

MSC makes use of the SysEx message format to encapsulate show commands that were not originally catered for in the MIDI specification. The flexibility of the SysEx message

³MIDI sequencers are devices that are capable of recording or being loaded with a pre-recorded sequence of MIDI messages, and then replaying these stored message to recreate a MIDI performance. Modern Digital Audio Workstations incorporate MIDI sequencing capabilities, allowing them to be used to record and play back MIDI messages.

format allows for control messages to range from simple cues for audio or video equipment used in a production, to large pyrotechnics synchronisation[21].

2.3.2 MIDI Limitations

MIDI has never been fully revised⁴, and thus many of MIDI's restrictions might be due to the protocol's age. Some of the limitations that the protocol faces include the cable length being limited to 15 meters, the slow rate at which MIDI transmits data, the limited number of available channels and the protocol's unidirectional communication. Beyond the limitations that exist due to the specification, there are also improvements that would increase the flexibility and usability of the protocol[35].

2.3.2.1 Cable length

MIDI cables are limited (in the standard) to being up to 15 meters long, to avoid the risk that attenuation might result in signal loss[23]. Installations that utilise show control often span large areas, requiring that the monitoring and control are often a great distance from the devices being controlled.

Although there are systems available that can extend this distance, and some systems that even make it possible to send MIDI point to point wirelessly, this is still a restriction of standard MIDI[35].

Other wired standards such as Ethernet allow for cables carrying digital signals to have run lengths up to 100m. This shows that the need to limit cabling to such short distances is no longer necessary.

2.3.2.2 Data rate

MIDI operates serially, transferring bits one after another at 31250 baud: around 25000 bits per second of usable data⁵. This is slow when compared to other serial protocols currently in use, such as USB 2.0, USB 3.0 and Ethernet[7, 4, 46].

⁴Although MIDI has never received a full revision, it has however accepted certain extensions to its original specification. These include but are not limited to Standard MIDI File format, General MIDI and changes to transport specifications to include USB, FireWire and Bluetooth Low Energy connections.

⁵The serial implementation of MIDI makes use of start and stop bits to indicate the beginning and end of MIDI bytes. The baud rate of MIDI includes the transmission of these start and stop bits.

Certain implementations of MIDI (outlined in the extensions to the standard) allow for MIDI to run at speeds in excess of 31.25 kbaud. These speed increases are only allowed in systems where the 5 pin DIN connector is not being used, such as MIDI devices equipped directly with a USB interface[35].

USB 2.0 is capable of up to 480Mbps transmission (USB high-speed), but USB-MIDI devices are typically 12Mbps (full-speed) devices, even within the USB 2.0 specification, allowing for multiple MIDI streams to operate on a single USB connection [7]. USB devices that identify as MIDI interfaces are not bound by the MIDI speed outlined in the specification, as these are not required to communicate with other MIDI devices at a hardware level.

2.3.2.3 Number of channels

The MIDI standard provides a 4 bit address space for MIDI Channel messages. This allows for a single MIDI sequencer attached to multiple synthesisers to control up to 16 separate instruments at the same time[35].

In an interconnected studio where multiple devices offer MIDI connectivity, the ability to address only 16 devices at a time restricts the protocol's ability to exert control over all MIDI enabled devices.

Various solutions such as MIDI Thru boxes and MIDI Multiport devices attempt to address some of the limitations, but these usually result in an increase in cabling and do not increase the number of channels that can be transmitted over a MIDI cable.

2.3.2.4 Unidirectional communication

The standard specifies the use of only 2 of the 5 pins available on the DIN connector. This results in a need for 2 MIDI cables when control is required to flow in both directions. This unidirectional communication results in complicated setups, especially when devices are chained together using the MIDI Thru connector[51].

The MIDI specification considers a chain of more than 3 MIDI devices to be a long chain, stating that errors might be introduced due to the response time of the opto-isolators. This further limits⁶ the protocol's usefulness[35].

⁶It should be noted that through the use of MIDI Thru boxes the latency induced by daisy-chaining devices can be circumvented.

2.3.2.5 Limited control structure

The MIDI message format uses a single status byte to describe all possible messages, resulting in a limited number of different messages types. The byte is split into two 4 bit nibbles in the case of Channel Voice message. The first 4 bits pertaining to the message type, with the following 4 bits being used to represent the channels number.

Every status message has the most significant bit set, allowing for the easy identification of status bytes over data bytes. This results in only 8 possible main message types (including a type to represent System messages).

Although the SysEx message type allows for the transmission of messages that are not part of the MIDI standard, this has high overhead and a lack of standardisation.

2.4 Solutions to MIDI limitations

For remote control applications, two problematic limitations within the MIDI specification are the cable length limitation and the one to one transmission imposed by the electrical design.

The specification limits the length of a MIDI cable to 15 meters to ensure signal integrity. Due to the wide usage of MIDI, many solutions have been created to help lessen the effect of this limitation and allows for longer runs of MIDI to be achieved. These solutions range from the use of boosters that increase the strength of the MIDI signal, to wireless systems that remove the need for the long runs of MIDI cables entirely.

The second limitation that faces the MIDI protocol is that of routing. The unidirectional communication mechanism employs a model based on the existence of a sender and a receiver. This model does not allow for flexible routing. Transmission is intended to be done in a limited one-to-many configuration⁷, and there are no extensions that allow for simple many-to-one or one-to-many transmission, without the use of daisy chaining.

⁷Although not technically limited to a one-to-one configuration, MIDI is largely intended to be used in a one controller to many instrument configuration, but where these many instruments are limited to a total of 16, may be contained within a single synthesiser.

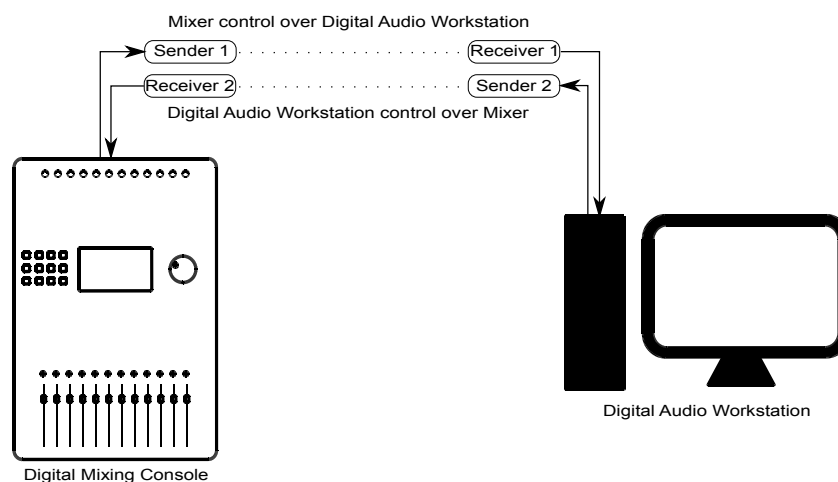


Figure 2.4: Example of a radio link MIDI solution being used for synchronised control between a digital mixing console and a digital audio workstation.

2.4.1 Radio link transmission

MIDI transmission over a radio link is a popular solution due to the large increase in the allowable distance between MIDI devices. However, these solutions only remove the cable limitation and do not improve on the original MIDI specification.

In a situation where a digital audio workstation is being used to control a digital mixing console, a single cable would be required. This requirement would need to be increased to 2 cables if changes made on the digital mixing console were to be sent to the digital audio workstation.

When a solution for radio link transmission is examined in a use case that requires control to flow in both directions as shown in Figure 2.4, the solutions suffer the same issues that the original MIDI cables suffered. Each cable would need to be replaced with both a transmitter and receiver, increasing the cost of using such solutions.

2.4.2 Network transmission

Radio link solutions are successful due to the replacement of MIDI cables and their restrictions, but they are limited in their usefulness. By using workstation computers and Ethernet networking, further functionality can be added, such as port mapping, connection management and enhanced channel numbers.

A number of MIDI network solutions have been created to overcome one or more of the



Figure 2.5: The rackmountable Lone Wolf MidiTap unit.

restrictions mentioned earlier in the chapter.

2.4.2.1 MidiTap

The Lone Wolf MidiTap system is an early example of a networked MIDI system[28]. The MidiTap uses rack mounted MIDI units (shown in Figure 2.5) that are connected via an optical network. This allows for MIDI signals to be routed over greater distances with low latency.

The system provides 4 MIDI connections per unit, with each unit being controlled via a RS-232 serial link to a computer. The system allows for flexible connection management, with ports and channels capable of being completely remapped.

The drawbacks that this solution faced were its very high cost per unit, and high network overheads. Fibre networking was needed, which together with the high cost created a barrier to entry for smaller studios and venues. The MidiTap unit is no longer produced.

2.4.2.2 mLAN

The Music Local Area Network (mLAN) protocol was designed to cater for the networked distribution of audio and music control data. The mLAN protocol made use of IEEE 1394 (Firewire) based networks, which allowed for the data rate of up to 400 Mbps. mLAN's high speeds allowed the protocol to transmit both audio and control data, including MIDI and wordclock[24].

The networked nature of mLAN might have made the protocol a decent solution for providing distributed MIDI control, but due to the lack of hardware support for IEEE 1394 at the start of this research, the decision was made to pursue more readily available network technologies.

2.4.2.3 ipMIDI

ipMIDI⁸ provides a method for transmitting MIDI over Ethernet for workstation computers running Microsoft Windows or Macintosh OSX. The software implementation is aimed at removing the need for MIDI interfaces and cabling between software-based MIDI instruments and applications[53].

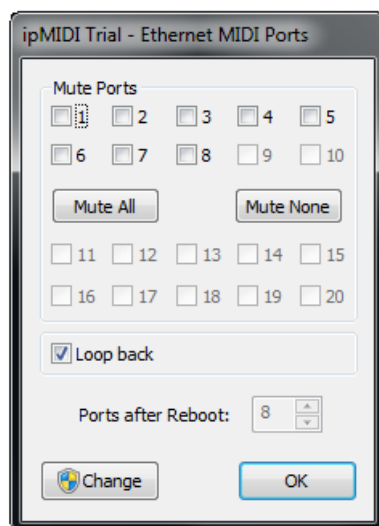


Figure 2.6: ipMIDI user interfaces.

The ipMIDI system allows for up to 20 Ethernet MIDI input and 20 Ethernet MIDI outputs as shown in Figure 2.6. Due to the software not using a physical MIDI interface, it is possible for the latency to be lower than that of physical MIDI equipment, although it should be noted that other latency may be introduced by the network or the operating system.

⁸ipMIDI is available from www.nerds.de/en/ipmidi.html.

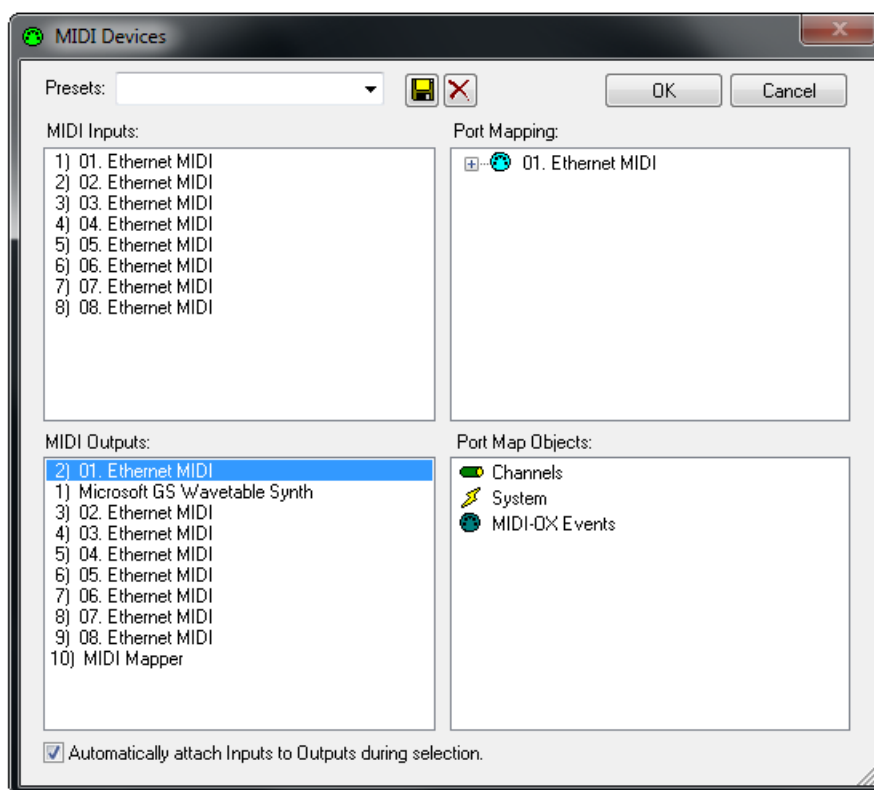


Figure 2.7: ipMIDI virtual ports in MIDI-OX.

The software will work with a limited selection of external MIDI and DMX⁹ interfaces, so that it may be used for external equipment control and show control.

The solution provides the equivalent of 40 MIDI cables worth of communication via a single Ethernet cable. Figure 2.7 shows how these virtual MIDI cables appear to MIDI applications. It allows for ports to be muted and allows for loopback on channels.

The solution provides a platform for MIDI transmission over a network. However, ipMIDI does not provide a connection management interface.

The routing system makes use of software MIDI ports on computers to allow applications to transmit MIDI over a network. The system creates 'virtual MIDI cables' to which applications can be connected.

Furthermore the solution is only deployable on computers running the Microsoft Windows or Apple Macintosh operating systems. This adds the requirement of a computer to the solution, increasing size and monetary cost.

⁹DMX512 or Digital Multiplexing is a communication standard largely used to control stage lighting.[3]

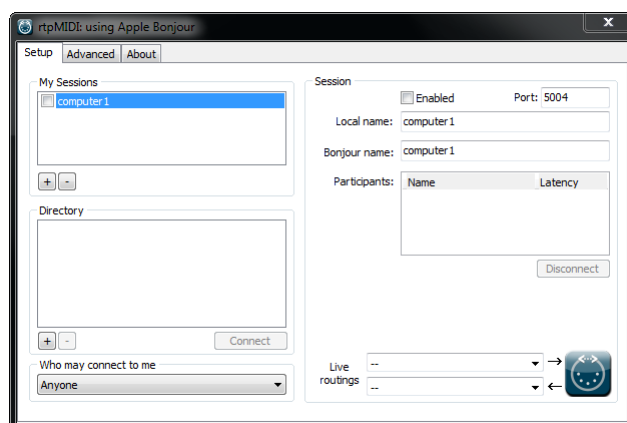


Figure 2.8: Creating a new session.

2.4.2.4 RTP MIDI

Real Time Protocol MIDI (RTP MIDI) is a virtual MIDI driver that allows for digital audio workstation software to communicate with other MIDI applications over a network [55]. The software uses the RTP protocol¹⁰ to encapsulate MIDI packets for transmission over networks[27].

The driver works in a similar manner to that of ipMIDI, allowing for applications to send and receive MIDI messages over a network. The driver works with network-MIDI¹¹ available on Macintosh OSX and iOS.

Where it differs from ipMIDI is that it does not provide a number of static virtual MIDI cables that can be used, but rather allows flexible routing of MIDI to other machines running the RTP MIDI driver.

The RTP MIDI application allows the user to add a new virtual MIDI port, via the My Sessions button denoted with a “+” that can be seen in Figure 2.8. The application allows a port to be selected and a name given to the computer the application is running on. Once the setup of the first device is complete, any other device or computer that will be joining the session needs to be set up.

The same setup process is completed for a second device as shown in Figure 2.9. Once the device setup is complete, the two ports are ready to be connected. This is accomplished by selecting the virtual MIDI port the user wishes to use on the local device (‘computer

¹⁰RTP or Real-time Transport Protocol is a protocol for network communication designed for the transmission of audio and video. The protocol is discussed in Section 4.3.4.

¹¹Network-MIDI is a protocol used by Apple. The protocol simply makes use of the RTP protocol and it’s extensions for MIDI payloads, but without the session management described in the RFCs.

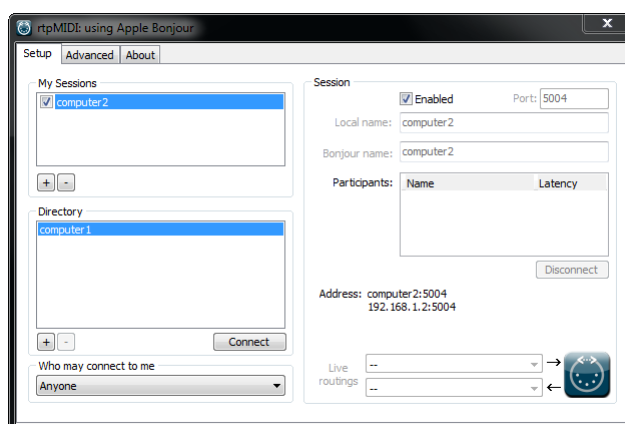


Figure 2.9: Remote sessions are visible in Directory

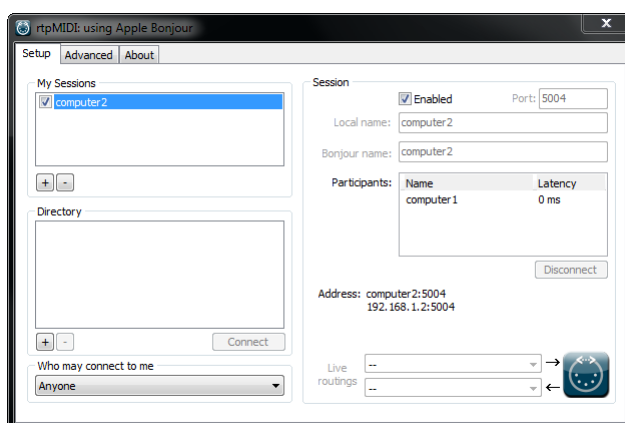


Figure 2.10: Current connections are shown in the session panel

1' in Figure 2.9) and then selecting one of the remote virtual MIDI ports that will be shown in the 'Directory' list. In Figure 2.9), 'computer 2' is selected as the remote MIDI port.

Once the two ports have been selected, they can be connected using the “Connect” button. At this point, the corresponding device will appear in the “Participants” list as shown in Figure 2.10. This can later be selected and the connection broken.

Although RTP MIDI is aimed at providing a software solution for MIDI communication for applications, it also provides a standard for encapsulating MIDI in RTP packets so that the system may be used by other devices.



Figure 2.11: The KISS BOX MIDI solution.

2.4.2.5 Kiss-Box

The Kiss-Box MIDI solution is a hardware device that allows for the RTP MIDI protocol to be interfaced with physical MIDI connections.

The device (shown in Figure 2.11) allows for 2 MIDI inputs and 2 MIDI outputs to be connected per end point, with end points being networked over a Power-Over-Ethernet (POE) network.

An application running on a desktop computer on the same network as the Kiss-Box is used to control the end-points. From the application the user can create sessions, which allow for MIDI connections to be linked together between end-points. Each Kiss-Box assigns UDP¹² ports to all inputs and outputs. Connections are created by supplying the application with a UDP port for the sender and a corresponding port for the receiver.

The Kiss-Box Editor shown in Figure 2.12 is a tool available from Kiss-Box that is used to control and configure various Kiss-Box devices. The application is available for Microsoft Windows or Apple Macintosh operating systems.

To add a Kiss-Box, the user is either required to scan the network for Kiss-Box devices (Section A), or manually add a Kiss-Box (Section B) as shown in Figure 2.12. Section C shows the output that will alert the user to the successful addition of a Kiss-Box.

Once a new Kiss-Box has been added to the system, the Kiss-Box will appear in the tree view shown in Section D of Figure 2.13. By double clicking on the device, the user is able to configure the Kiss-Box in Section E. The user is able to make changes to the IP addressing information such as subnet and gateway, as well as enter the TCP and UDP ports which the KissBoss will use.

¹²The User Datagram Protocol (UDP) is a network protocol used for the transmission of data over an IP based network. UDP and other networking nomenclature are discussed in Chapter 4

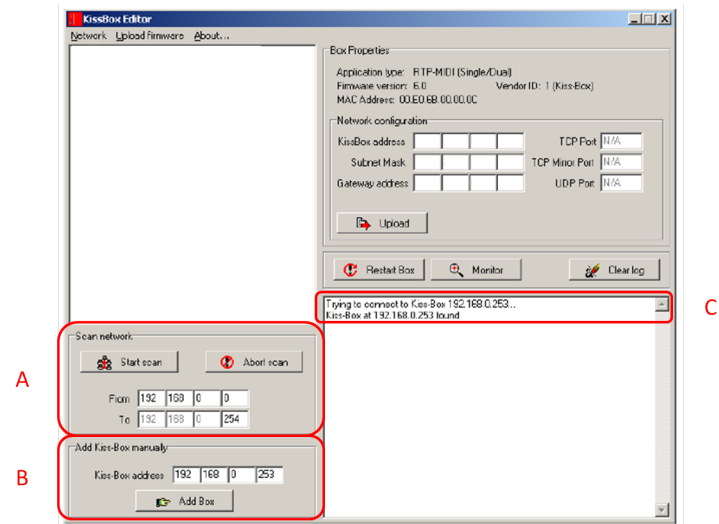


Figure 2.12: Overview of the KissBox Editor software

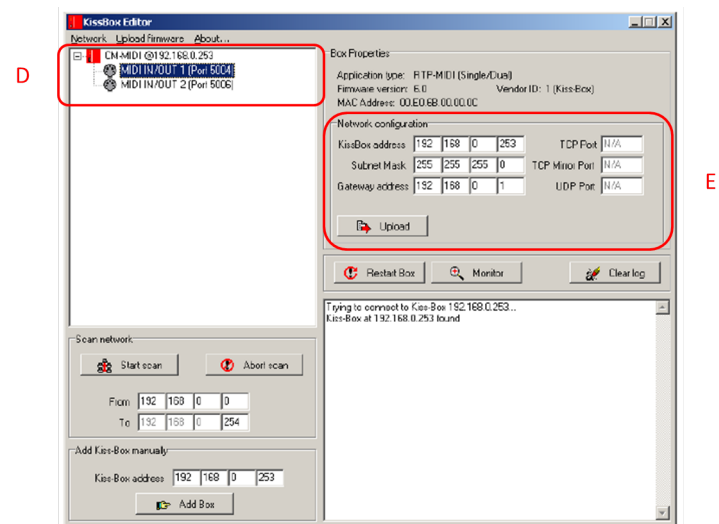


Figure 2.13: KissBox configuration

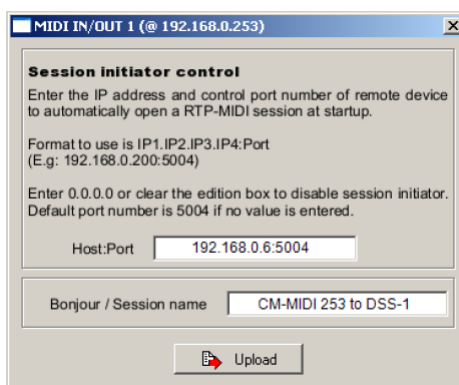


Figure 2.14: Uploading information to KissBox

Once the “Upload” button in Section E is clicked, a dialogue box will appear that will allow for a device to be added to an RTP MIDI session as shown in Figure 2.14. This dialogue box will allow the user to upload new information to the Kiss-Box, by transmitting a session name (to be used in the RTP MIDI application) to Kiss-Box.

From this point, the rest of the setup is performed as per the RTP MIDI setup described in Section 2.4.2.4. The RTP MIDI application will provide the functionality necessary to route MIDI over a network. The session name that was uploaded to the Kiss-Box can now be selected from the “Directory” list within the RTP MIDI application, allowing the user to make connections to other RTP MIDI applications.

This offers a more cost effective solution than the Lone Wolf MIDI Tap, but requires a desktop workstation to run the Kiss-Box software. It has a user interface that requires knowledge of IP addresses and ports, and is cumbersome to set up.

2.4.2.6 MIDINet

MIDINet is a desktop application that allows for MIDI to be sent between desktop computers over an Ethernet network. This differs from previously mentioned systems in that it is not designed to be used with software MIDI applications¹³, but rather to be used with USB MIDI controllers, so that external MIDI equipment can be networked.

The system allows for multiple software or hardware MIDI interfaces to be used on a desktop computer. The drivers associated with these MIDI interfaces make the MIDI

¹³Although not designed to be used with software MIDI applications, MIDINet can be used with any software MIDI application that provides a virtual MIDI port. For applications that do not provide virtual MIDI ports, 3rd party applications can be used to provide a connection to MIDINet.

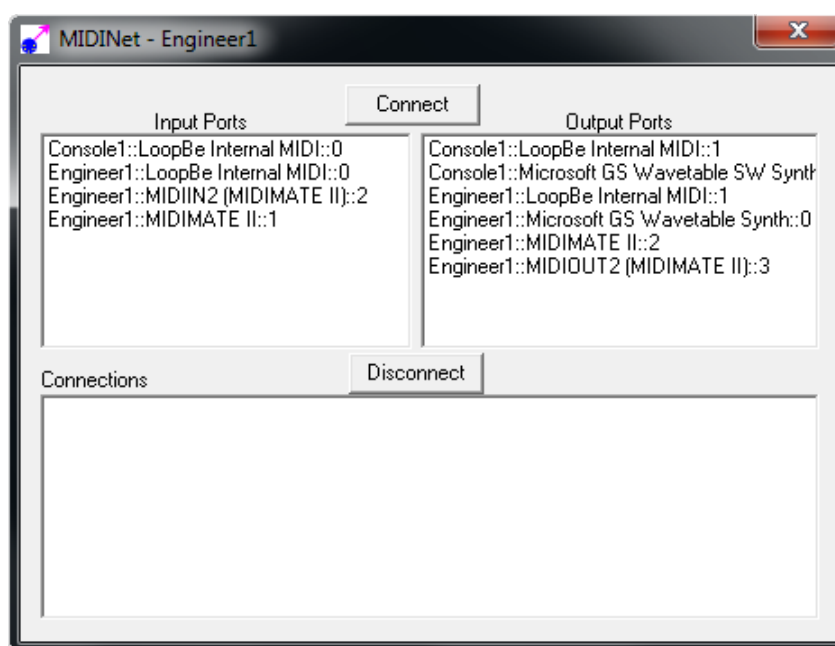


Figure 2.15: MIDINet Connection Management Dialogue.

ports¹⁴ accessible to software applications. These MIDI ports are then made available to the network so that connections can be made between MIDI ports on different computers on the network.

Figure 2.15 shows the user interface for the MIDINet system. The interface consists of two panels: the panel on the left showing all available MIDI input ports and the panel on the right showing all the available MIDI output ports. The user can then select two ports and connect these ports using the 'Connect' button centred above the input and output panels. Once a connection has been established, the connection will be listed in the 'Connections' panel. This panel displays all current connections within the system and allows the user to select a current connection and destroy it using the 'Disconnect' button (centred above the 'Connections' panel).

All computers on the network synchronise their information, resulting in the same level of access and control being available on any computer on the network running the MIDINet system.

MIDINet therefore provides a flexible connection management system, where hardware and software MIDI ports can be connected across the network, allowing for network transmission of MIDI message over Ethernet between devices and MIDI-bound applica-

¹⁴A MIDI port describes an input or output that follows the MIDI specification. The port may be a hardware input or output provided by a USB MIDI interface, or might be a software input or output for an application such as a synthesiser or MIDI sequencer.

tions.

The system overcomes the MIDI cable limitation, and manages to overcome the unidirectional communication constraints of the MIDI specification. The flexible connection management further enhances the usability of the MIDI protocol.

The system does however require the use of a desktop computer with a monitor in order to run and use the application. This increases the deployment cost, but otherwise provides a suitable improvement to the MIDI specification that could be further improved to decrease physical size and lower deployment costs.

Due to the existing features and capabilities of the MIDINet system, as well as access to the source and design documentation, MIDINet was chosen as the connection management system on which the new system would be based. To fully understand MIDINet's design and existing implementation, the system is analysed further in Chapter 5. The system's design, operation and shortcomings are examined to aid in the provision of an improved design for a MIDI Connection Management System in Chapter 6.

2.4.2.7 Open Sound Control

Open Sound Control or OSC was developed by Adrian Freed and Matthew Wright as a “protocol for communication among computers, sound synthesizers, and other multimedia devices that is optimized for modern networking technology” [68].

The OSC protocol was designed independently of its method of transport. The protocol is therefore able to be used on standard network technology such as TCP and UDP, running over Ethernet or 802.11 wireless networks[67].

The OSC protocol has been used to simply encapsulate other protocols (including MIDI messages) for transport over networks, but has also been employed for use in lighting control, DAW control as well as home Do-It-Yourself projects[67].

OSC is becoming increasingly popular in the audio engineering industry due to it being an open protocol. This allows manufacturers to widely and freely implement and use OSC in products, and encourages developers to implement OSC in their applications.

The protocol defines a data transport format that can be used to send control messages over networks. OSC does not define a set of standard messages for devices to use, rather encouraging manufacturers to create and use their own message sets known as Schemas.

Schemas refer to configurations that define the structure of devices for particular applications. Therefore the address space is application specific. There are no standard schemas currently and manufacturers need to define the structure of their devices in order to use the OSC messaging protocol[66].

The designers of OSC state that OSC is superior to MIDI in several ways. The MIDI standard was never intended for use over LANs, let alone the Internet. OSC provides a larger address space for messages, allowing for more complex and specific control messages. Modern synthesizers can sometimes exceed the MIDI specification in terms of channels and parameters, areas where OSC is not limited[12].

OSC is a software based messaging format, requiring a set of messages and function callbacks to be defined. It lends itself more easily to integration with applications than it does to physical hardware. This does not mean that it cannot be used to provide hardware control though, but rather that the hardware would require a software stack that can manage OSC before the protocol can be used.

OSC is capable of encapsulating MIDI messages for transmission over networks, and provides a message structure for MIDI. However, it does not provide a solution to connection management. The extensible nature of the message schema provides a wide range of possible messaging and control options, but lacks standardisation.

Since it is not the MIDI messaging format that we wish to enhance, there was insufficient motivation to add the additional overhead of OSC to the MIDI transmission packets.

2.5 Conclusion

While MIDI remains the most widely used control protocol, it has some disadvantages that diminishes its usability. The limitation placed on cable length by the specification requires MIDI devices to be located in close proximity to each other.

Solutions are available that mitigate the cable length restriction, but these solutions have their own disadvantages: large and bulky equipment, high cost of devices, limited usability. There are no solutions that provide a low cost standalone solution for flexible wireless MIDI transmission and connection management.

The MIDINet application offers a distributed connection management system, and system for network transmission. Were the system implemented for a low cost embedded system,

it would provide a suitable MIDI connection management system. This led to the search for a platform on which a connection management system could be developed. Chapter 3 investigates the XMOS architecture and development kits which are subsequently used in the design of a connection management system detailed in Chapter 6.

Chapter 3

XMOS Architecture

3.1 Introduction

The survey of MIDI transmission systems covered in Chapter 2 showed that while many solutions provided adequate extensions to the MIDI protocol by overcoming the distance limitations within the MIDI specification, most did not provide connection management and were ultimately high in terms of monetary cost.

The ipMIDI solution offers improved distance and the ability to have multiple separate MIDI streams on a network connection, but the solution did not include remote connection management and is only intended for use with software applications.

The RTP MIDI application and Kiss-Box solutions overcome the distance limitation of standard MIDI and provide flexible connection management. However, the solution has a complicated user interface, requiring the user to have knowledge of UDP ports. The solution also requires that set up be performed on each device that is to be connected using RTP MIDI.

MIDINet overcomes the length limitation and provides flexible and simple connection management. The solution does require a PC running Microsoft's Windows operating system for each node, leading to non-deterministic routing. Minimal set up is required at each node, with the user having only to provide a unique name for each MIDINet node.

The raised cost in many of the solutions (including MIDINet, ipMIDI and RTP MIDI) is due to the requirement of a computer on which the solution will be run. These systems

require that computers (either desktop or notebook computers) are used in combination with USB MIDI interfaces to provide the necessary MIDI connectivity, further increasing the size and cost of the solution.

A solution that would improve upon the solutions above would include the flexibility of MIDINet's connection management, a small form factor and deterministic MIDI transmission.

This chapter will describe the XMOS architecture and development kits chosen to be the networked nodes in a MIDI connection management system, with the deterministic processing capabilities of the hardware serving as a motivation for the choice.

The chapter includes a brief history of XMOS, an explanation of the XMOS microcontroller architecture and how XMOS differs from other microcontrollers. The chapter will show how XMOS's hardware development kits can be used to rapidly prototype a hardware platform around which the system can be designed.

3.2 Background

In 1977, Tony Hoare published a paper titled 'Communicating Sequential Processes' which described a language for concurrent programming[18]. This concept evolved into a form of process calculus now widely known as CSP. CSP describes, through the use of a formal language, independent processes through their message passing attributes[1].

In 1978 David May joined INMOS, a new semiconductor company, to engineer an new microcontroller. May developed Occam, a concurrent programming language built on the process algebra from Hoare's CSP.

INMOS's prototype microcontroller, the Transputer, was completed in 1982, with the first revision of the Occam programming language being released in 1983. The Transputer was the first microcontroller designed for parallel computation, where multiple microcontrollers could be linked together to form large multi-microcontroller grids.

In 1995 David May began working at the University of Bristol as a professor. XMOS was founded in 2005 by Ali Dixon (a student at the University of Bristol at the time), James Foster, Noel Hurley, David May and Hitesh Mehta.

XMOS develops multicore microcontrollers aimed at deterministic execution for real-time tasks.

3.3 Difference between XMOS and Other Microcontroller Architectures

Since the stated hypothesis of this thesis included the transmission of MIDI messages over wireless networks, platforms were reviewed to find hardware and pre-existing libraries that would achieve these goals.

A microcontroller is an integrated circuit that has been designed so that it may execute code that is flashed onto it. The code is executed in a sequential manner until interrupted by an event scheduler. Microcontrollers can then jump to routines when an interrupt occurs. This causes the code that was executing to have to wait until the new routine has completed its execution.

Platforms other than XMOS were considered with the closest competitors being the Atmel based Arduino development kit and the ARM based Raspberry Pi. The Arduino platform provides a simple microcontroller with more than sufficient hardware expansion capabilities to provide the necessary connectivity. However, since it has single core execution it does not provide deterministic execution of simultaneous tasks¹.

The Raspberry Pi provides a hardware platform that while packaged as a small device, is simply a computer, employing a Debian Linux based operating system, and using USB for expansion to include networking and MIDI connectivity. Deterministic execution could not be guaranteed due to the scheduling of tasks by the operating system. Although the Raspberry Pi is capable of being used to create a MIDI transmission and connection management solution, the deterministic and near real-time performance of the XMOS platform was ultimately the deciding factor in the selection for a hardware development platform and the available development kits suggested this would provide a rapid path for implementation of the solution.

3.3.1 XS1 Architecture

XMOS's XS1 microcontrollers make use of tiles: scalable microcontrollers that contain multiple processing cores (known as xCores), a scheduler and 64KB of RAM. An XS1 microcontroller may contain one or more tiles, networked together through a switch. The

¹Although the Arduino platform would be capable of handling the MIDI component of the solution, it would not have been an ideal platform for the development of a web based user interface.

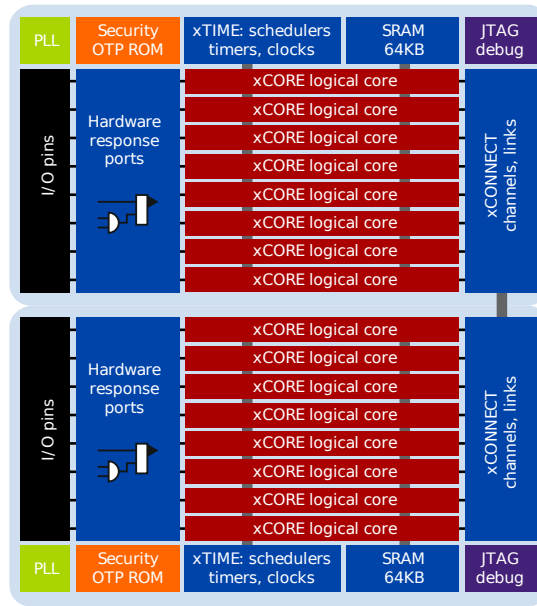


Figure 3.1: The underlying XS1-L2 16 core architecture.

16 core XS1-L2 microcontroller is comprised of a pair of 8 core tiles, provided as a single package as shown in Figure 3.1.

The XMOS microcontroller tiles have been designed to be parallel and scalable. This means that the XS1 microcontroller acts more like a grid of microcontrollers that run constantly and then pass data to each other over dedicated hardware channels.

3.3.1.1 Processing

Multiple XMOS microcontrollers can be further networked together to create larger microcontrollers with higher core counts for larger parallel workloads. The parallel design of these microcontrollers allows them to process tasks simultaneously while still being capable of managing external inputs and outputs in a deterministic manner.

3.3.1.2 Communication

As the XS1 range of microcontrollers allows for parallel process execution (either across xCores, or between tiles), these processes require the ability to communicate between themselves. This communication is done over channels: synchronous links that temporarily allow for communication between two processes.

Although standard channels are only reserved for the duration of the transmission, XMOS provides a variation of the standard channel that remains reserved at all times. These channels are known as 'streaming channels'.

3.3.1.3 Programming

XMOS microcontrollers are programmed using a language called xC. The language is best described as a set of multicore extensions to C developed to provide the functionality required to utilise the XMOS microcontrollers.

The language allows for the inclusion of C and C++ files within xC projects. An overview of the xC language and examples of its use for task creation, channel transmission and C/C++ file integration can be found in Appendix A.

3.4 XMOS-based Hardware Development Kits

XMOS while being a microcontroller chip company, provide products to aid in the development of software based solutions to run on their xCore line of microcontrollers. One such development kit is the Slicekit board.

3.4.1 Slicekit Core Board

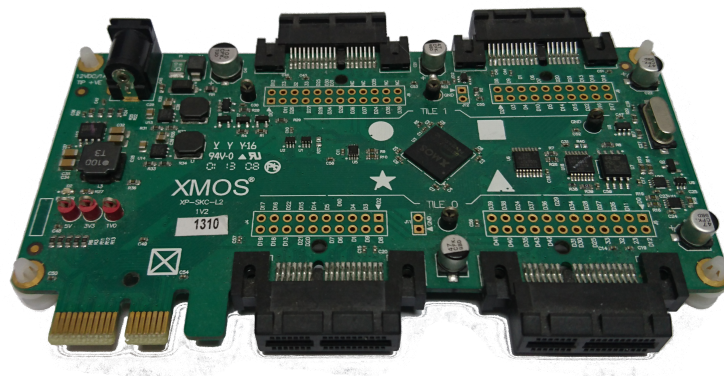


Figure 3.2: The Slicekit Core board.

The Slicekit Core board shown in Figure 3.2 is a development board that is equipped with a 16 core xCore microcontroller, an on-board power supply and a JTAG adapter for

programming. The board is also fitted with 4 expansion connectors (referred to as slots) that expose all pins available on the XS1-L2 microcontroller embedded on the board.

The board is designed to allow developers to have access to the combination of hardware that they require, without the need for custom development boards to be created. The four slots on the Slicekit Core board can be used to provide hardware connectivity such as Ethernet, Audio converters, general purpose inputs and outputs (GPIO), or to expand the device's memory using additional RAM.

These hardware slices are intended to be used with software modules provided by XMOS that give developers access to the hardware, as well as providing functions that would generally be required to use each slice.

3.4.2 Ethernet Slice

An Ethernet Slice (shown in Figure 3.3) provides a single RJ45 jack, so that projects might include network connectivity up to 100Mb/s. The slice is designed to be used with XMOS's XTCP library so that projects can implement web servers and other network related components.

The XTCP library provided by XMOS connects to the XMOS Ethernet library, allowing for TCP and UDP to be used within applications running in XMOS microcontrollers. The library uses a port of the uIP stack: a minimal stack implementation designed to minimise resource usage [11].



Figure 3.3: XMOS's Ethernet Slice.

More details relating to this slice are provided in Chapter 6, including slot choices and interactions with other components. The usage of the Ethernet Slice is also described in Chapter 6, and details about the XTCP library are documented in the Chapter 7.

3.4.3 WiFi Slice

An aspect that initially made the XMOS development kit look like a good choice for a hardware platform was the availability of a WiFi module (shown in Figure 3.4) available as a slice for the Slicekit. The WiFi slice makes use of a TiWi-SL radio chip to provide 2.4GHz 802.11b/g wireless networking.



Figure 3.4: WiFi slice containing the TiWi-SL wireless module.

The wireless module is interfaced to the XS-L2 microcontroller via SPI, which results in a maximum UDP transmission throughput of 6.9 Mbps. Since MIDI transmission over IP would only require a transmission in excess of around 31,250 bits per second, it was assumed that the 6.9Mbps limit would be sufficient for a proof of concept[30].

Unfortunately during the design phase of the project, component tests revealed that the WiFi slice software library was not fully implemented, and so only certain tasks were possible. This is detailed in the implementation chapter.

The software library that supports the WiFi Slice included a modified and simplified version of the XTCP library that was used to interface with the TiWi-SL module located on the slice. This version of the XTCP module was not fully implemented by XMOS at the time of this research, and did not make use of the existing fully XTCP library.

3.4.4 Audio slice

Another advantage of the XMOS development platform was the existence of a slice with MIDI connectivity. The Audio Slice shown in Figure 3.5 is an expansion for the Slicekit



Figure 3.5: Slice for audio development.

development board that provides 4 analogue input channels, 4 analogue output channels, S/PDIF and MIDI connectivity.

During design and component testing it became apparent that the slice was mainly aimed at audio development, as the library for the MIDI ports was a place holder for future development. The technical details about this slice are detailed in the design and implementation chapters.

3.5 Conclusion

This chapter introduced the XMOS microcontroller architecture: a multicore microcontroller capable of deterministic execution of parallel tasks. Unlike desktop computers that run operating systems, these microcontrollers are perfect for time-sensitive application where low and deterministic latency would be advantageous.

To enable developers to develop code for these microcontrollers, XMOS provide reference design boards such as the Slicekit and some peripheral slices. These allow for developers to create a hardware platform that suits their needs.

The Chapter shows that the XMOS multicore microcontrollers provide an architecture appropriate for the implementation of a MIDI connection management system. The Slicekit Core board, and associated slices fulfil the hardware requirements for such a MIDI connection management system. This all motivates the adoption of XMOS for the development of a low cost deterministic MIDI connection management solution.

Chapter 4

Network support

4.1 Introduction

In order to enable the design of a MIDI connection management system, an investigation of networking support structures was undertaken. The aim of the investigation was to gain insight into networking concepts so that these concepts could be leveraged in the design of a MIDI connection management system.

Regarding the network transmission of MIDI, the MIDINet system uses broadcast UDP messaging for transport of MIDI messages as MIDINet messages. The RTP MIDI application and Kiss-Box interface make use of Real-time Transport Protocol (RTP) packets to transmit MIDI over a network.

Due to the lack of a display and input device, an XMOS based design does not have the GUI connection management capabilities of the MIDINet system that is currently implemented on a Windows workstation. A web server was proposed as a solution, offering a graphical user interface that could be accessed from a device with a web browser.

In creating the user interface capability, it would be advantageous if the system did not require any additional overhead such as servers. For this reason, networking support structures were investigated to aid in the removal of the hardware overhead associated with web based control systems.

The XMOS SliceKit development kit described in Chapter 3 provides slices that can be used in the creation of systems for Ethernet or Wireless networking. This chapter looks at both wireless and wired networking, then investigates the underlying networking

protocols.

This chapter provides the background to understand the approach taken in designing and implementing a low cost deterministic MIDI connection management solution.

4.2 Layer Model

Device communication over computer networks can be conceptually modelled as a stack of layers, where each layer describes a set of communication standards or protocols. There are two popular models used to show network layering: the Open Systems Interconnection (OSI) model and the Transmission Control Protocol / Internet Protocol (TCP/IP) model. The models provide an abstract view of networking architectures and the encapsulation of protocols used in network communication[40].

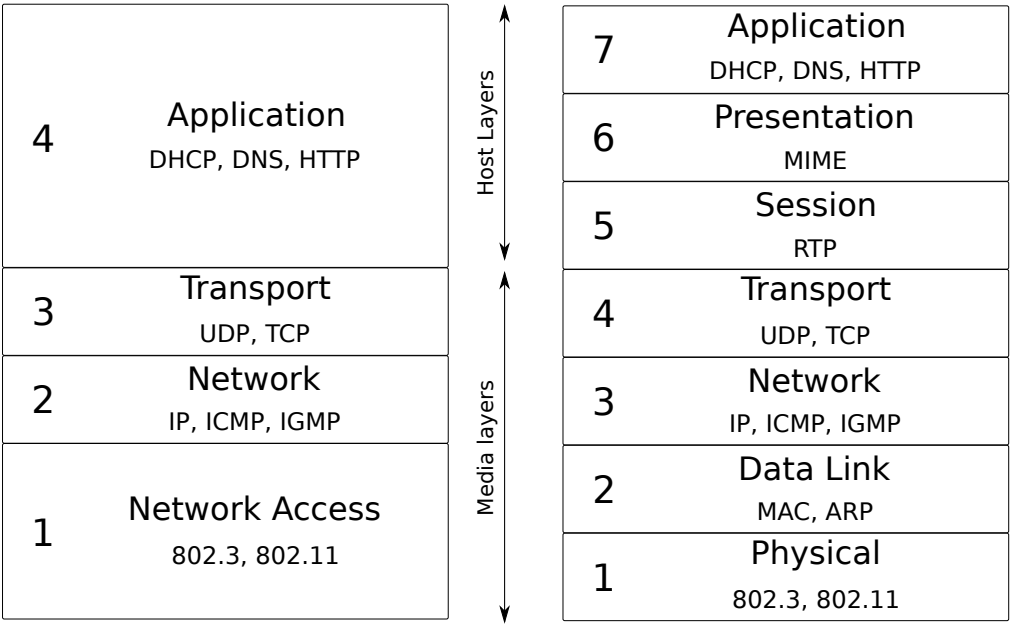


Figure 4.1: The TCP/IP and OSI layer models.

Figure 4.1 shows the arrangement of layers within the OSI and TCP/IP models. The TCP/IP model (left in Figure 4.1) consists of four layers. The OSI model (right in Figure 4.1) consists of 7 layers. The key differences between the TCP/IP model and the OSI model can be found in their separation of layers.

The OSI model represents the TCP/IP model's Application layer as 3 separate layers: Session, Presentation and Application. The TCP/IP model shows the OSI model's Physical and Data Link layers as a single Network Access Layer.

This thesis will focus on the 4 layer TCP/IP model, as this was the model used in the design of the original MIDINet system discussed in Chapter 5[17], but information and comparison to the OSI model is provided for reference.

4.2.1 Physical layer

The physical layer describes the interaction with the networking media (underlying copper cabling and radio frequency). The IEEE 802.3 standard defines Ethernet: a standard for the transmission of data using copper universal twisted pairs[46]. IEEE 802.11 standardises radio frequency transmission for wireless networking (WiFi)[22].

As seen in Chapter 3, XMOS provides both Ethernet and WiFi slices for use with its SliceKit development kit.

4.2.2 Data Link layer

The Data Link layer describes the standards and protocols used to provide communication between two or more links on a network. The IEEE 802 standard separates the Data link layer further into two layers: Logical Link Control (LLC) and Media Access Control (MAC)[26].

The MAC layer provides an interface through which the LLC can communicate with the physical layer. The MAC layer is largely used for addressing and channel control (allowing for multiple hosts on the same physical medium).

The LLC provides an interface for the MAC layer, allowing it to communicate with the Network layer. This layer allows for the use of many differing protocols to be used on the same network.

4.2.2.1 Hardware Addressing

MAC addresses are 48 bit addresses used to identify networking hardware. These addresses are issued by the manufacturer of the network interface and serve as a unique

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Source IPv4 Address																															
Destination IPv4 Address																															
Zeros								Protocol								Length															

Figure 4.2: IP pseudo header

identifier.

Since MAC addresses are unique identifiers, they are used in the identification of networking interfaces at the Data Link level. These addresses are used in the resolution of Internet Protocol addresses (described in Section 4.2.3).

4.2.3 Network layer

The network layer of the OSI model includes protocols for the transmission of data on a network. In this case, a network is a medium to which hosts are connected, and identified through the use of an address known as an Internet Protocol (IP) address.

IP addressing is separated into two predominant versions: Version 4 (IPv4) and version 6 (IPv6). This document focuses on IPv4 addressing for the design and implementation discussed in Chapters 6 and 7 respectively.

IPv4 addresses consist of a 32 bit address which is separated into 4 octets. These addresses can be assigned manually, through the use of link-local addressing, or by a Dynamic Host Configuration Server (DHCP). These address assignment mechanisms are discussed further in Section 4.3.1.

Figure 4.2 shows the header information included in all messaging. IP addresses are used to identify hosts on a network, and protocols associated with layers above the Network layer will make use of these addresses when referring to a host.

Since the Data Link layer uses MAC addresses to forward network frames, and higher level protocols make use of IP addresses, IP addresses have to be resolved to MAC addresses when routing packets.

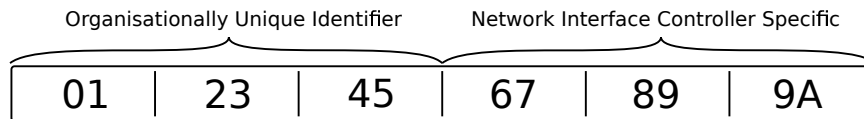


Figure 4.3: Overview of a MAC address

As shown in Figure 4.3, a MAC address is made up of 6 octets. The address comprises two 3 octet components: The Organisationally Unique Identifier and the Network Interface Controller Specific components.

The Address Resolution Protocol (ARP) is a method for associating IP and MAC addresses. ARP requests are broadcast to a network, so that the interface with the requested IP address may respond with its MAC address[41].

The Network layer also includes protocols such as the Internet Control Message Protocol (ICMP)[43] and Internet Group Management Protocol (IGMP)[5]. These protocols make use of IP addresses, but remain part of the Network layer, adding functionality to protocols at higher layers.

IGMP is a protocol that defines how hosts can be grouped, allowing for groups to be messaged rather than individual hosts, and plays a key role in IP multicast (discussed in Section 4.3.2).

4.2.4 Transport layer

4.2.4.1 UDP

The User Datagram Protocol (UDP) is the simplest transport protocol. The MIDINet system discussed in Chapter 5 makes use of UDP messaging to transmit both MIDI and connection management information[16].

The User Datagram Protocol (UDP) provides a method for messaging on a network, with minimal protocol. UDP is a transaction oriented protocol, and does not include mechanisms that allow for the reporting of successful transmission or receipt[42].

The header shown in Figure 4.4 shows the layout of the UDP datagram. The protocol makes use of ports for transmission, requiring that the source and destination ports be included in the datagram, along with the datagram length (in octets), the encapsulated

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31								
Source IPv4 Address																																							
Destination IPv4 Address																																							
Zeros								Protocol								Length																							
Source port																Destination port																							
Length																Checksum																							
Data																																							

Figure 4.4: UDP layout

data, and a checksum for the pseudo-header and datagram combined.

The UDP header that is used for datagrams is defined in the RFC768 specification of the User Datagram Protocol[45]. The RFC recommends that for reliable delivery of data streams, the Transmission Control Protocol should be used[42].

4.2.4.2 TCP

The Transmission Control Protocol (TCP) is a core protocol in the IP suite. The protocol allows for the reliable, ordered and error-checked transmission of data sent over an IP network[44].

Figure 4.5 shows the TCP header format that is to be included with the pseudo-header when transmitting data. When compared to the UDP header, the TCP header has considerably more overhead.

Although the header of a TCP packet contains more information than that of a UDP packet, it is TCP's mechanism for reliable transmission that truly introduces significant and unpredictable latency. TCP packets sent from a client are initially sent with a sequence number set. Upon receipt, the network interface of the server responds with a synchronisation request, and acknowledgement of receipt. The client then responds with the data that is intended to be sent to the server. This is known as the TCP handshake.

This procedure for transmission allows for more stable network communication, but at the cost of speed.

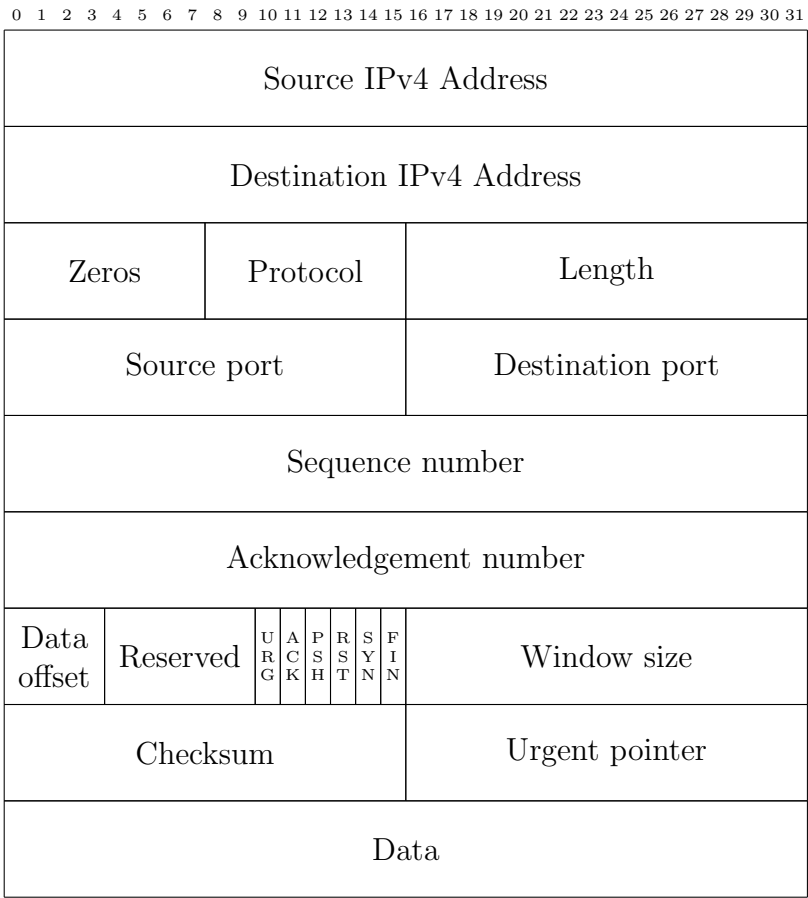


Figure 4.5: TCP header layout

4.2.5 Session, Presentation and Application layers

The three highest layers of the OSI model, the Session, Presentation, and Application layers, deal with tasks often reserved for the operating system. The Session layer is used for tasks that start and end sessions between hosts and assist in keeping these tasks separate. The Presentation layer is used to provide a form a translation for application layer protocols so that hosts on a network can communicate. The Application layer is responsible for providing received information to the user (or other applications).

The MIDINet solution runs at the application layer. Although MIDINet makes use of UDP datagrams for the transmission of messages over a network, MIDINet is an application layer system. The MIDINet protocol used to package MIDI and connection management data is used at the application layer.

4.3 Support structures

To enable the design of an embedded system that would be capable of providing transmission and connection management for MIDI messages, certain networking support structures are needed.

These include address assignment, the ability to message all relevant nodes on a network and the ability to address nodes using hostnames. This section describes these support structures. Furthermore, real-time networking extensions are investigated to determine their effectiveness in providing a platform for low latency networking.

4.3.1 Addressing

IP addresses are used to identify hosts on a network, and would need to be dynamically allocated so that the number of hosts on a network should have no effect on the operation of the system.

4.3.1.1 DHCP

The Dynamic Host Configuration Protocol (DHCP) is a service that provides IP addresses to hosts on a network. These addresses are issued by a DHCP server that is responsible for providing configuration information to hosts[10].

The server may be configured with predefined addresses that it should issue to new devices, or it may have a block of IP addresses that it is permitted to assign.

This adds overhead to a network, since a server (either hosted on a computer, router or as a discrete server) will be required for a network and will require some form of configuration.

4.3.1.2 Link local addressing

Link local addressing offers an alternative to manual IP address management, without the need for a DHCP server. This is achieved by allowing devices to determine their own IP addresses rather than using a centralised server to issue these addresses[2].

Link local addressing provides a procedure for enumerating devices within the same network link. Devices perform a transform on their MAC address in order to determine their IP address.

Once a device has performed the transform on its MAC address, the resultant IP address must not be currently in use on that network segment. The device performs a check by utilising ARP probing. The device broadcasts an ARP request for the IP address which it plans to claim. If the IP address is in use on the network, the device currently employing the IP address will respond with its MAC address, signalling that the IP is not available. This would require a different IP address to be probed[41].

Due to the transform that is performed on the MAC address, the resulting IP addresses will be predictable. A devices will retain its IP address, always resolving to the same address, unless there is a conflicting IP address on the network.

This form of IP address allocation removes the need for servers that manage addressing within a network segment. This can be used to lower the equipment overhead when designing an embedded system and so motivated the use of link local addressing for the ConMan system.

4.3.2 Multicast group messaging

Multicast describes a method of sending datagrams to a group rather than to an individual recipient. It is similar to broadcast messaging, but the broadcast is only sent to members of the multicast group.

The MIDINet system makes use of multicast for device discovery. Multicast allows MIDINet nodes to send a message to all members of the MIDINet group, so that these members can respond and assist in device configuration. This is discussed further in Chapter 5.

RFC 5771 provides the guidelines for the IP address space that is available for use with multicast groups. A device that intends to communicate with a group, will direct its IP datagram to the group IP address (an address in the range 224.0.0.0 - 239.255.255.253)[60].

Any device that is a member of that multicast group will receive that message. Devices can elect to join a multicast group, and are able to leave that group at a later stage.

This support structure will provide functionality similar to that of broadcast, but without subjecting other devices on the network to unnecessary traffic. All devices that form part of the system need to simply direct their messages to the multicast address.

4.3.3 Name resolution

IP addresses are not the most natural names for people to remember for devices. More natural names constructed from words are far simpler to remember.

Any name that is used to refer to a device, computer or server will need to be associated with an IP address that can be found on the network.

4.3.3.1 DNS

The Domain Name System (DNS) is a distributed naming system for resources on a network. DNS makes use of servers known as DNS servers that respond to requests made using a name, with the IP address associated with that name.

DNS servers, similarly to DHCP servers, require set-up and require maintenance to stay up to date. To associate a name with an IP address, the name is required to be registered with a DNS server. If the first DNS server that fields the request is not capable of resolving the name to an IP address, the request is made to the next DNS server in the hierarchy.

This once again increases the equipment overhead that would be required to resolve a name of a device on a network to an IP address.

4.3.3.2 mDNS

Multicast DNS (mDNS) is a name resolution system that removes the need for a preconfigured and centralised name server. mDNS makes use of the '.local' domain: a top-level domain associated with link-local devices.

When a request is made for a name, the request is multicast via Port 5353 to the 244.0.0.251¹ group. All devices receive this message, and if they have been configured to respond to the requested name, will respond as if the request was made to them directly.

Initially the intention was to make use of mDNS in the ConMan system to allow users to request more natural sounding addresses (such as conman.local) rather than requiring the knowledge of the device's IP address. This ultimately was excluded due to incomplete implementation of mDNS within XMOS's XTCP library.

4.3.4 Real-time transmission

The Real-time Transport Protocol (RTP) is a protocol developed for the transmission of audio and video over networks. The protocol is described in RFC 3550 as a protocol for that provides end-to-end transportation of real-time data over a network[54].

The RTP protocol is used in conjunction with the RTP Control Protocol (RTCP)[64]. The data stream will be transmitted using the RTP packet format, with RTCP being used to monitor and control the RTP data stream.

Figure 4.6 shows the RTP header format, which is used to carry information about the type of data being transmitted, the order that samples should be presented and a timestamp that is used to facilitate clocked presentation of the data. This assists in ensuring that data is presented in the correct order and that samples are presented at regular intervals, which will decrease jitter in audio and video payloads.

The protocol provides a tool-set for the transmission of data that is in the real-time domain. The protocol however does not deal with resource allocation and does not guarantee quality of service for real-time data.

The RTP protocol is not affected by underlying Network or Transport layer protocols or implementations. The protocol can assist in providing a clocked presentation of time

¹This address is used for mDNS name resolution as described in RFC 6762[6].

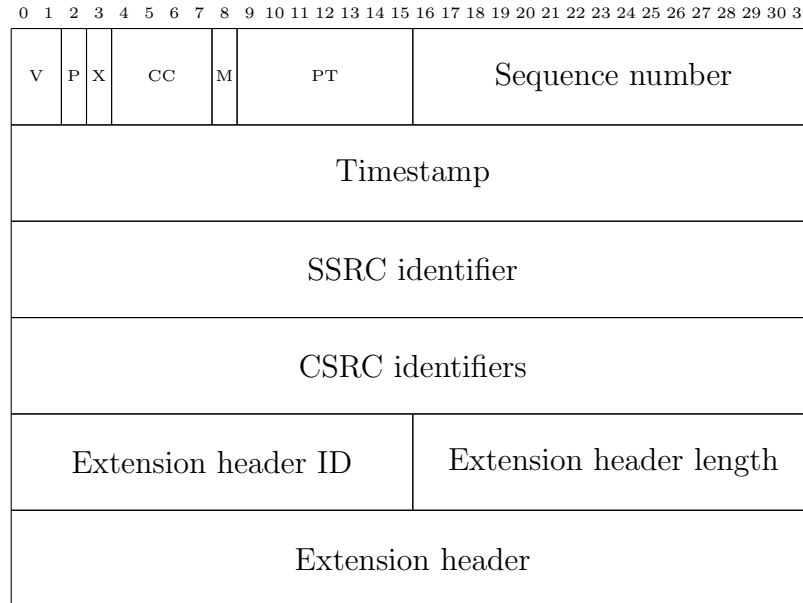


Figure 4.6: RTP layout

sensitive data transmitted over non-deterministic networks.

Although the RTP protocol could have been used in the implementation of the ConMan system, the protocol is aimed at providing better reproduction of audio or video information that is clocked. This is to avoid the advancement or retardation of samples, to ensure accurate reproduction. As MIDI messages are not sent at regular intervals, but rather on an event basis, it was decided that a focus on low latency message handling would be chosen over the use of the RTP protocol².

4.4 Conclusion

In order to gain a better understanding of the networking support structures needed, the OSI and TCP/IP model were compared and reviewed with reference to the use case requirements of a MIDI connection management system.

The physical layer, whether wireless or wired, should not have an effect on the subsequent layers, allowing for development to be done using XMOS's Ethernet slice. When a complete library is released for the WiFi Slice, the slice can be replaced without there being

²It should be noted that the research does not suggest that the RTP protocol would not be beneficial. The use of RTP would assist in providing more accurately timed/spaced messages after transmission. The primary goal of the research was lower latency, but further research into the use of is covered in

any impact on the system.

The MIDINet solution makes use of multicast messaging using UDP to transmit MIDI and connection information over the network. As discussed in Chapter 3, the ConMan system will not feature a physical display or input devices, and so networking support was reviewed to aid in the development of a web based user interface.

Typically addressing and name resolution services would require servers on the network such as DHCP or DNS. Through the use of lower level protocols and standards, sufficient functionality can be achieved without adding software overhead through application layer protocol implementations. Link local addressing can be used in place of DHCP to provide addressing to devices without the need to a centralised DHCP server. The mDNS service could be implemented to provide a name resolution system, enabling users to request device by name rather than IP address.

The data-link and network layer protocols can be used to provide lower latencies in an embedded system due to a decrease in processing required by devices that do not run full operating systems. Through the use of real-time design methodologies a deterministic system can be developed using these lower level network support structures.

Chapter 5

Analysis of MIDINet System

5.1 Introduction

MIDINet is a networked MIDI solution that allows for computers equipped with MIDI drivers to send and receive messages from other machines on the network. The system allows for connections to be created through the application's user interface, connecting the MIDI input ports of one machine to the MIDI output ports of another[16].

Computers running the MIDINet software are equipped with MIDI interfaces so that MIDI devices can be attached to the system. These interfaces allow multiple MIDI devices to be connected to each workstation.

Figure 5.1 shows a keyboard controller, a synthesiser and a digital mixing console, each attached to a desktop computer running MIDINet. MIDINet allows for connections to be made between these devices, and the MIDI messages routed through an Ethernet switch.

This results in the removal of the cable limitation imposed by the MIDI specification, and allows for more flexible routing as cables do not need to be unplugged from MIDI devices, but rather connections are added and removed in software[16].

The solution of routing MIDI over Ethernet networks does resolve many problems and limitations of the original MIDI specification, but it is limited in its use. MIDINet requires desktop computers running Microsoft's Windows operating system to be used in conjunction with adapters that provide MIDI connectivity. This makes the solution both expensive and not very space efficient. The use of desktop computers as nodes also results

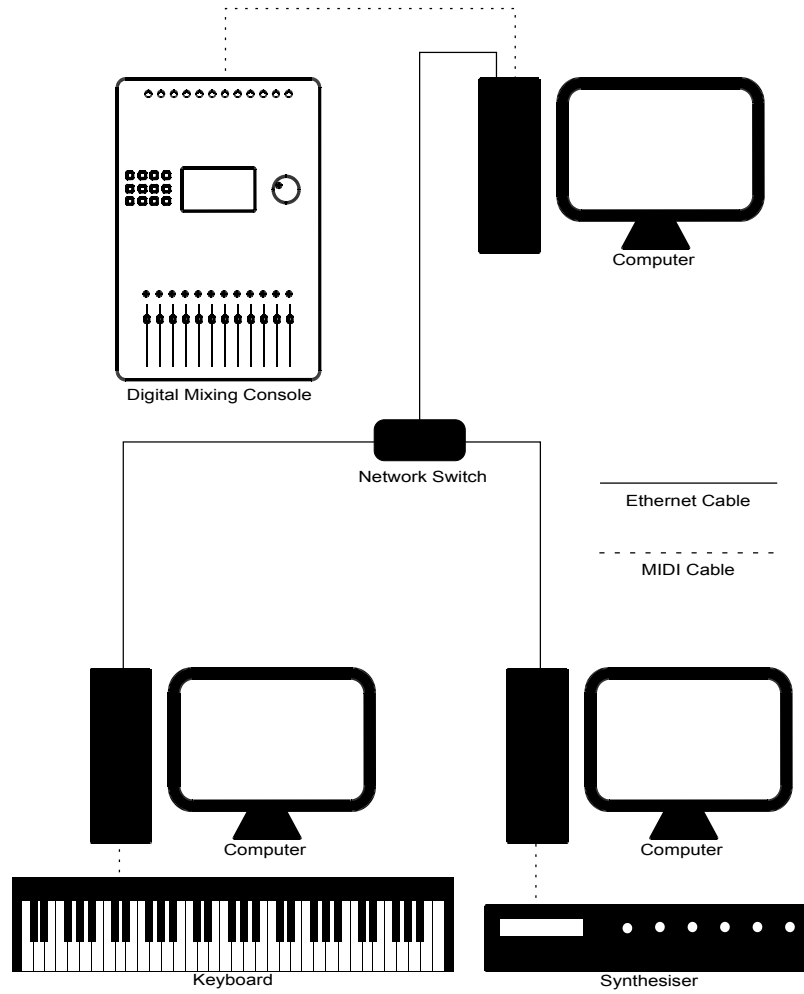


Figure 5.1: Example configuration of a MIDINet system.

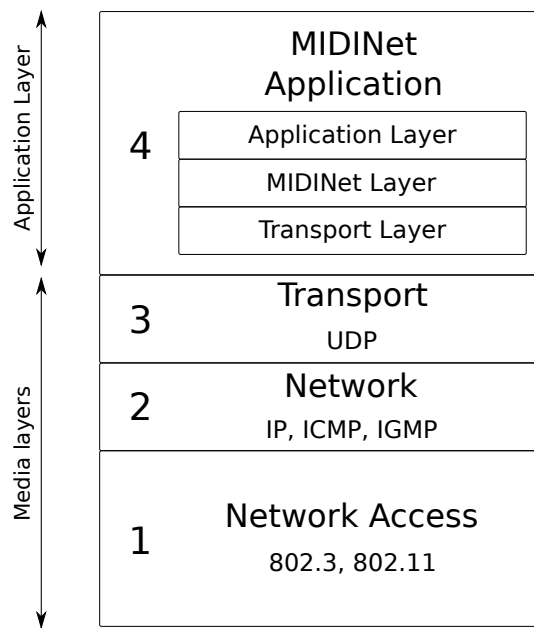


Figure 5.2: Relation of MIDINet layers to TCP/IP Layer model.

in unnecessary overhead that results in MIDI transmission being non-deterministic.

This chapter analyses the MIDINet system, and investigates its design. The messaging protocol, connection management and system design have been documented so that the system may be redesigned for use on low cost, embedded hardware. The investigation produced documentation on the operation of the system, and highlighted areas where the system could be improved.

5.2 MIDINet Layer Model

The MIDINet system was developed with its own multilayer protocol. Based on the TCP/IP model, the MIDINet protocol consists of 3 layers: The application layer, the MIDINet layer, and the Transport layer[37].

The Application layer is the top layer, and is the layer with which the user interacts. This layer provides methods for adding connections to the system and removing them again. The middle layer is the MIDINet layer, which is responsible for the routing of messages from the application layer to the transport layer. The transport layer is the lowest layer and is tasked with controlling inter MIDINet unit messaging across the network.

Figure 5.2 describes the arrangement of the MIDINet Layer Model and shows how the

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Source port																Destination port															
Length																Checksum															
Code segment																								Data segment							

Figure 5.3: MIDINet Message in UDP Datagram

MIDINet Layers relate to the TCP/IP Layer Model. The MIDINet system runs at the TCP/IP Model's Application Layer, and the layers describes by Mosala [37] in the MIDINet Layer Model all run at this layer.

MIDINet messages consist of two segments: A code segment and a data segment. The code segment of the message consists of 3 bytes while the data segment's size can vary up to 1497 as shown in Figure 5.3. The data component can be as large as the minimum length allowed by an Ethernet packet [37].

5.3 System Design

For an embedded MIDI connection management and transmission system to be designed that would make use of the MIDINet protocol, the design of the MIDINet system needed to be investigated.

The MIDINet design was investigated to ascertain which features of the MIDINet system should be adapted to be integrated with the new connection management system, and to determine how changes could be made to ensure determinism within the new system.

5.3.1 User interface

The MIDINet application consists of two windows: A window naming the MIDINet unit, and a window managing connections. When the user starts the application they are presented with a window in which a name can be provided for the MIDINet unit. Figure 5.4 shows the naming window, which is made up of a field for entering text, and a button that allows the application to proceed with the set-up.



Figure 5.4: MIDINet Identification Dialogue.

Once the name for the MIDINet unit has been confirmed, the user is then presented with a window which is used for managing connections between the various MIDINet systems on the network. Figure 5.3 shows the connection management window, which is used for selecting ports and connecting them.

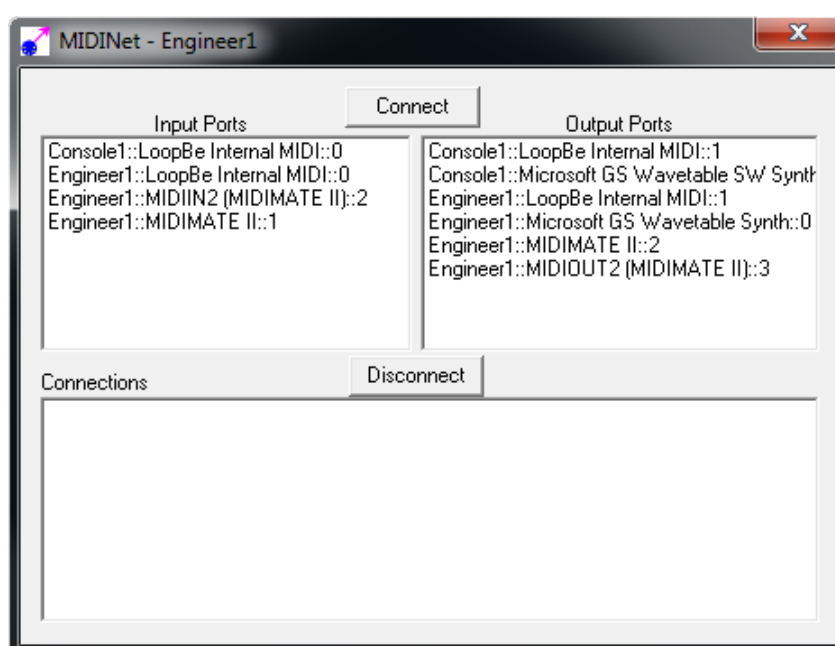


Figure 5.5: MIDINet Connection Management Dialogue.

In Figure 5.5 a list box component can be seen on the upper left hand side where the MIDI input ports are listed. The list box is populated with all MIDI input ports discovered on the MIDINet system, from both the local and remote MIDINet units.

Ports are displayed with the name of the MIDINet unit, followed by the description of

the port (MIDI interface) and then the MIDI port ID, with each name being separate by a double colon. The list box on the upper right of the window functions in much the same manner, but instead with the MIDI output ports being displayed. Each line of the list boxes is selectable, allowing for the ID of both the MIDINet unit and port to be used when creating a connection.

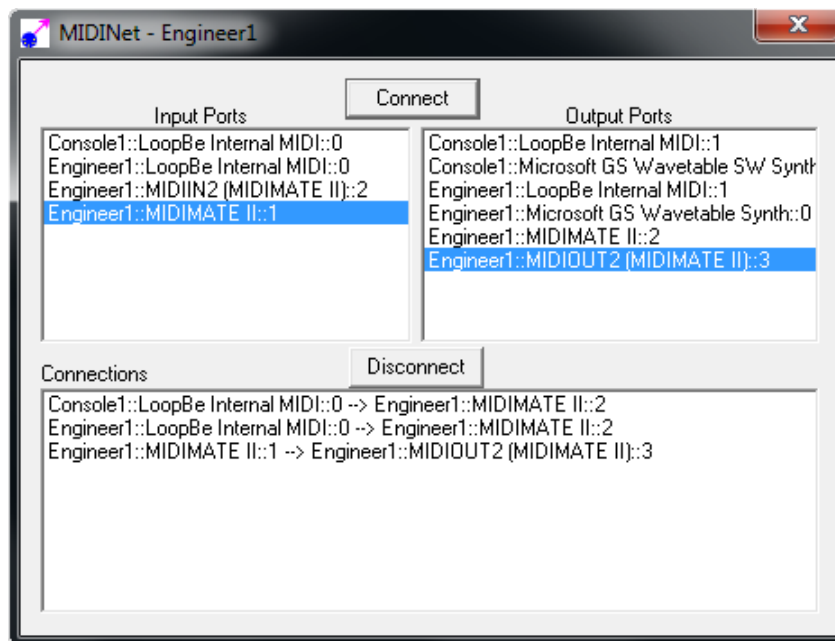


Figure 5.6: MIDINet Connections.

Lastly, the lower half of the window contains a list box where all current connections are displayed as shown in Figure 5.6. Each connection is displayed as a MIDINet unit's MIDI input port followed by a MIDINet unit's MIDI output port. Each of the connections in this list box can be selected, so that existing connections can be removed when they are no longer required.

5.3.2 Object Based Design

The MIDINet system was designed using an object-oriented approach, where inheritance was used to allow for classes with similarities in use case to share attributes with a parent class on which they are based, as shown in Figure 5.7.

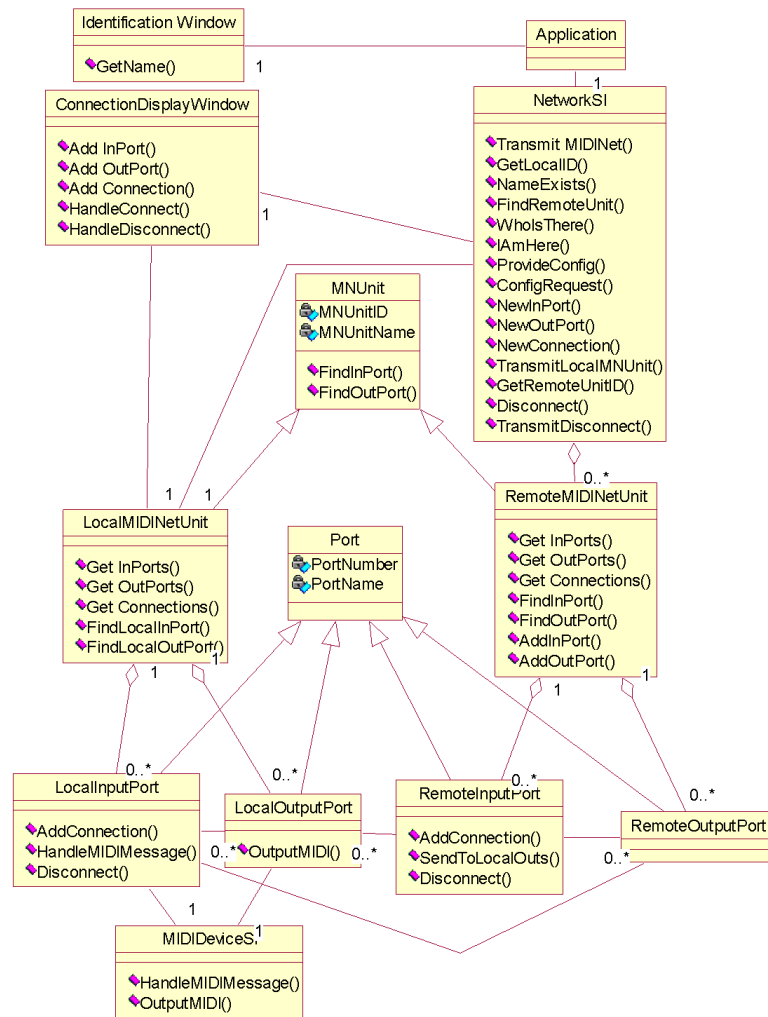


Figure 5.7: Class diagram for MIDINet system.

5.3.2.1 MIDINetUnit class

The MIDINetUnit class (MNUnit in Figure 5.7) is used to form a base class that provides storage of information pertaining to a single MIDINet unit. The class provides two private data items: A string containing the unit's name and an unsigned integer containing the unit's ID.

The class provides basic accessors for the data in the form of SetName and SetID functions as well as GetName and GetID. The class is simple and it is required to be generic so that the LocalMIDINetUnits and RemoteMIDINetUnits can inherit their base members from this class.

5.3.2.2 LocalMIDINetUnit class

The LocalMIDINetUnit class inherits from the MIDINetUnit class but also contains its own data items, namely aggregations of LocalInputPorts and LocalOutputPorts that model the input and output ports of the MIDINetUnit.

The class contains a 'CreatePorts' function which opens ports of the MIDI interface attached to the host machine and transmits information about these ports to other MIDINet units. These ports are then represented as new LocalInputPort objects, opened and added to a list of LocalInputPorts associated with the MIDINet unit. The equivalent procedure is then carried out for all MIDI output interfaces attached to the machine.

5.3.2.3 RemoteMIDINetUnit class

The RemoteMIDINetUnit class contains a number of functions that are required for storing the details of other MIDINetUnits on the network. The class includes functions for adding and attaching both RemoteInputPorts and RemoteOutputPorts as well as removing them from the MIDINetUnit.

In a similar manner to a LocalMIDINetUnit object, each Remote MIDINet object contains aggregations of input and output ports associated with the MIDINet Unit.

5.3.2.4 Port class

The Port class forms the base class of the more specialised LocalInputPort, LocalOutputPort, RemoteInputPort and RemoteOutputPort classes and thus is a more generic class providing basic functionality and identification capability.

It stores only the port's Name (as a string) and ID (as an unsigned integer). The class provides Get and Set accessors for the port's data items in much the same way the MIDINetUnit class does.

5.3.2.5 LocalInputPort

The LocalInputClass inherits port identification from the Port class but requires an external library in order to communicate with the lower level MIDI devices associated with these ports.

MaxMidiIn is a part of the Maximum MIDI library [33] which provides all the necessary functions for opening, closing, starting, stopping and handling the MIDI messages received by a MIDI input port. The class also holds a data member for identifying the MIDINet unit to which the LocalInputPort belongs.

Packaging of MIDI information into messages that will be transmitted over the network is done by the ProcessMidiData member function. Figure 5.8 shows a high level flow diagram of the ProcessMIDIData function.

The function reads the status byte of the MIDI message in order to determine whether the message is a fixed length MIDI message, or whether it is a SysEx message. As discussed in Chapter 2, the status byte of MIDI messages contains the information required for classifying MIDI messages. A status byte with a value of 0xF0 signifies a SysEx message. Any messages that follow this status byte will have their data byte added to the stored SysEx message. Once a MIDI message with a data byte value equal to 0xF7 (SysEx End) is received, the SysEx message will be packed into a MIDINet message and transmitted over the network.

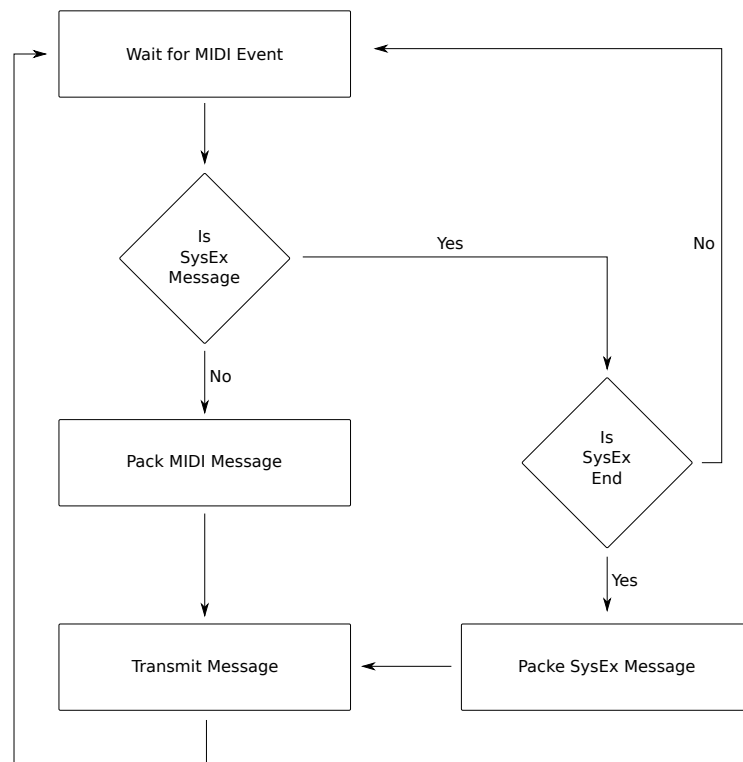


Figure 5.8: Flowchart for the ProcessMIDIData function.

5.3.2.6 LocalOutputPort

The LocalOutPort class inherits functionality from the MaxMidiOut class (a part of the Maximum MIDI library) and inherits from the Port class for storing the port name and ID. MaxMidiOut provides all the functions needed to open and close MIDI output devices, as well as output a MIDI message to a port.

The class itself has only a single data member, a pointer for storing the location of the related LocalMIDINet unit so that the port may be associated with a specific MIDINet unit. Apart from the class's constructors and destructor and excluding any inherited member functions, the class has only a single function for returning the ID of the associated MIDINet unit.

5.3.2.7 RemoteInputPort

The RemoteInputPort class inherits the standard port access methods from the Port class. The RemoteInputPort class provides functionality for requesting information about outputs (either local outputs or remote outputs) as well as connecting to and disconnecting from these outputs.

Unlike the LocalInputPort class, the RemoteInputPort class does not require any functions for processing MIDI messages, since an instance of the class represents a LocalInputPort object attached to a remote MIDINet unit hosting the specified port. MIDI message handling will be done by the LocalInputPort object on the remote MIDINet unit.

5.3.2.8 RemoteOutputPort

RemoteOutputPorts use only the Port class as the base class from which they inherit member overrides and member functions. The constructor is used to set the ID and Name of the Port as well as the RemoteMIDINetUnit to which it belongs.

Once again few functions are needed, as the RemoteOutputPort is really just an identifier for a LocalOutputPort attached to a MIDINetUnit residing elsewhere on the network.

5.3.3 Network class

The Network class provides a set of methods for initialising the network, handling messages and calling various functions from other classes.

The class makes use of a BlockingSocket library which provides the functions for creating sockets, binding to network ports, listening to network ports and other socket-related operations. The library uses Microsoft's WinSock library to communicate with the network drivers.

When a Network object is instantiated, the network is set up and the process of discovery (detailed in Section 5.4.2) is started. The network object then acquires an IP address and joins the multicast group used by the MIDINet system.

5.4 Operation

To comprehensively understand the way a system functions, its start-up and operational procedures must be examined. Gaining insight into the way a system operates will allow for the identification of any shortcomings. These observations will aid in rectifying these flaws, and thereby improve the system when it is redesigned and implemented in the connection management system described and designed in Chapter 6.

5.4.1 Start-up procedure

When a user starts a MIDINet application, the application begins by identifying any MIDI interfaces connected to the computer. Once this is complete, a connection to the network is established. The computer then joins the multicast group that is used by MIDINet so that it is able to communicate with all computers on the network running the MIDINet application. The system then starts the process of Discovery (discussed in Section 5.4.2).

Before the application can continue with discovery, the MIDINet unit requires a name through which it can be identified. The user is presented with a dialogue box where a name is entered. The name is checked against a list of existing names for uniqueness, as no two MIDINet units are permitted to have the same name. Once discovery has been completed, the system is ready to be used.

5.4.2 MIDINet Discovery

MIDINet Discovery is a process by which a computer running the MIDINet application (referred to as a MIDINet unit) probes the network for other MIDINet units. Devices that are not yet configured initiate this discovery by sending an enquiry to the multicast group as shown in Figure 5.9.



Figure 5.9: Discovery: Network probe from local unit.

The enquiry will be received by all MIDINet units in the group, and these units will respond, announcing their presence on the network as shown in Figure 5.10. The new MIDINet unit will continue the discovery by requesting configuration information from one of the MIDINet units that is already configured. The configured device will send this information to the new MIDINet unit.

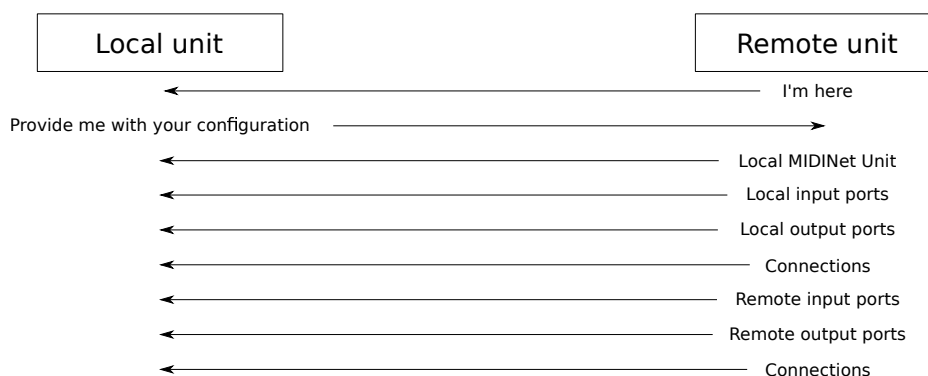


Figure 5.10: Discovery: Response from remote MIDINet system.

The information will contain the names of the units on the network, as well as the MIDI input and output ports associated with each unit. Any connection between units that already exist will be included in this information.

5.4.3 Message Routing

The control of MIDI interfaces and messaging associated with these interfaces is handled by the Maximum MIDI library. The library provides the functions necessary for sending and receiving MIDI message using the MIDI interfaces.

When MIDI messages are received from a MIDI interface to a MIDINet unit, they are packaged into MIDINet messages so that they can be transmitted to the network. The MIDI is also sent to any local MIDI output ports should the MIDI input and output ports be connected.

MIDINet messages received from the network are checked to determine whether there exists a connection between the sender and the MIDINet unit that received the message. If a connection exists, the MIDI message is then extracted from the packet and output to the connected MIDI output.

5.5 Limitations of the MIDINet system

The MIDINet system was originally implemented in Visual C++, making extensive use of Microsoft's Foundation Class (MFC) library. While this MFC implementation is not necessarily problematic, it can only be deployed on machines running Microsoft's Windows platform, which limits the application's target base[47].

The system was also designed to run on a wired LAN, with speeds of 100mb/s at the time of design. Modern networks can allow for the speeds upto 1Gb/s. Although this increase in networking speed could potentially increase system performance, the design of the system suggested that any bottlenecks in the system performance are not introduced by the network technology.

The design and implementation of the MIDINet system was more likely to introduce latency to the transmission of MIDI messages than the networking technologies being used. The design makes use of lists of units and ports that need to be searched before each MIDINet message received from the network can be output to local ports.

The time taken to route a MIDINet message would therefore be based on the number of units and ports stored at the time of routing the message. While this is not a large source of latency, it could in turn result in the latencies differing between desktop computers with different hardware and therefore different performance, based on each desktop workstation's ability to search each list¹. This non-determinism within the MIDINet system limits the use of the system within time-sensitive scenarios.

¹It should be noted that a real-time operating system can be used on a computer to mitigate these issues, but the inherent design of the MIDINet system would still technically introduce non-determinism.

5.6 Conclusion

Chapter 2 included an introduction to MIDINet alongside a few other solutions for transmission of MIDI over networks. The choice to investigate MIDINet further was based on the system's connection management capabilities.

This chapter took a closer look at MIDINet, investigating how the system was designed and implemented, and how the system can be used for flexible MIDI connection management. The chapter also investigated the limitations of the MIDINet system. As with other solutions such as RTP MIDI and ipMIDI, the system requires the use of computers on which the connection management software can be executed. This increases the cost of installing or implementing MIDINet as a connection management system.

The design of the system results in the transmission of MIDI over networking being non-deterministic. The design allows for automatic discovery and flexible routing and provides a good platform for connection management.

MIDINet's connection management would provide a versatile system that could be combined with the XMOS multicore microcontrollers discussed in Chapter 3. The design of a low cost, small form factor MIDI connection management system that can allow for the deterministic routing of MIDI messages is investigated next in Chapter 6.

Chapter 6

Design

6.1 Introduction

As discussed in Chapter 2, there are a variety of solutions that manage to overcome limitations of the MIDI specification.

Chapter 3 introduced the XMOS multicore microcontrollers. These microcontrollers are designed to enable embedded devices to communicate within time-sensitive applications. The deterministic execution of the XMOS microcontrollers makes them an ideal platform for the development of a MIDI connection management system.

As discussed in Chapter 5, the implementation of the MIDINet solution results in the system being non-deterministic. However, a deterministic MIDI transmission system would benefit from the flexible connection management capabilities of the MIDINet system. Thus, the incorporation of MIDINet's connection management required a design that allows for separation of the real-time and object-oriented components.

This chapter investigates the hardware and software aspects of the system's design, outlining how the various technologies were combined. The design details the hardware configuration and high-level design before detailing the design of the software components.

6.2 System overview

The goal of this project was to produce a system that will route MIDI messages over a wireless network while allowing for connections within the network to be remotely managed.

Although MIDI connection management can be done using typical workstation computers as nodes, the overhead in terms of both space and monetary cost would be great. Furthermore, the routing of MIDI messages within the nodes would be non-deterministic.

For this reason, the project created a small hardware device (ConMan) to remove the need for workstation computers.

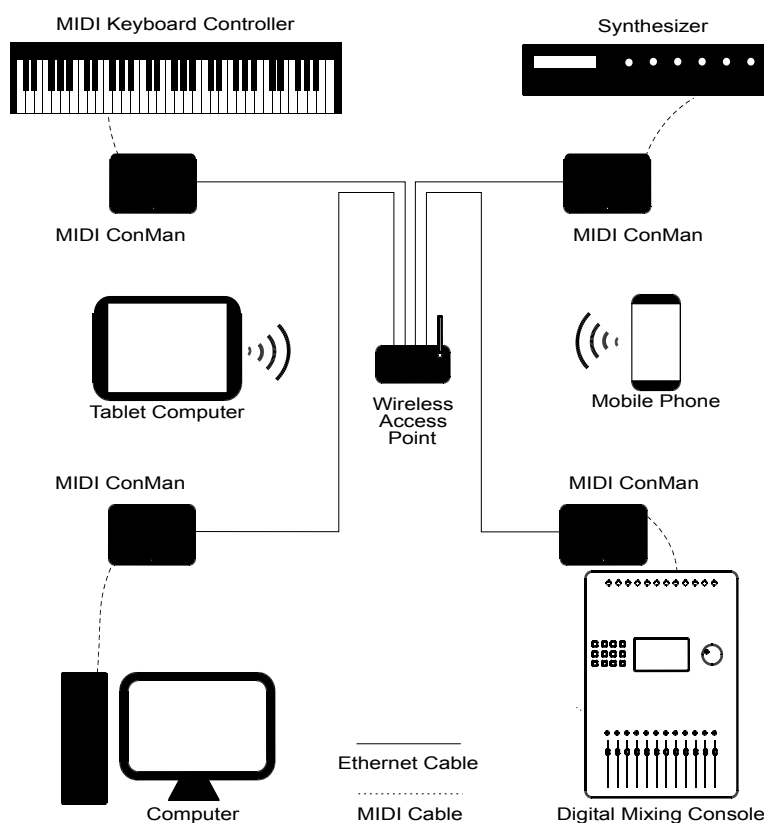


Figure 6.1: Example configuration of a MIDI ConMan system.

The configuration depicted in Figure 6.1 shows a number of devices in a small studio, with each MIDI device attached to a ConMan device. These ConMan devices are shown connecting to a centralized wireless router using Ethernet cabling, with each ConMan device then being connected to its associated MIDI device via a standard MIDI cable.

6.3 Hardware

The hardware used for this project made use of XMOS's multi-core microcontrollers discussed in Chapter 3. In particular the XMOS Slicekit was used to provide easy access to the 16 core, 2 tile XMOS microcontroller.

6.3.1 Hardware specification

The Slicekit was used in conjunction with two attached XMOS slices: the Ethernet slice which is used to provide network connectivity and the Audio slice which provides MIDI connectivity (for both MIDI input and output). The Slicekit and attached Audio and Ethernet slices comprise the hardware of the ConMan device.

The intention of the project was to make use of the MIDI connectivity provided by the Audio Slice to remove the need for an external MIDI interface. To meet the requirements for network connectivity, the Ethernet slice was attached to the Slicekit, allowing it to be connected to a network.

Originally the project planned to make use of the WiFi slice, and planned to produce a wireless connection management solution. As described in Chapter 3, the WiFi slice provided by XMOS did not meet the requirements for the project, and the choice was made to use the Ethernet slice and move to Ethernet based networking instead¹.

The design of the system planned to accommodate the WiFi slice, should a newer version of the slice be produced by XMOS. This would only be feasible should the design of the software libraries for the WiFi slice make use of the XTCP library provided by XMOS.

6.3.2 Hardware configuration

The Slicekit configuration is shown in Figure 6.2. There are allocations for the various slices, with each socket being marked with a symbol. The Ethernet Slice was placed in the socket marked with a circle, and the Audio Slice in the socket marked with a triangle.

¹Through the use of wireless access points, the system was used and tested. This shows that the MIDI connection management system can be wirelessly controlled.

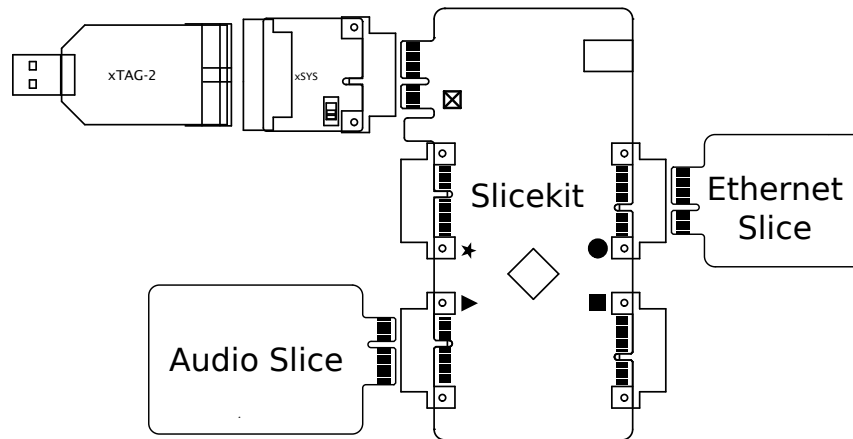


Figure 6.2: Slicekit usage and configuration

These slot allocations are due to the way in which the Slicekit has been designed and manufactured. The Audio Slice is designed to be placed in the circle socket as well as the triangle socket, but these two sockets are not identical. The most notable and important difference being that when used in the circle socket, the MIDI output lines of the Audio slice do not connect to the XS-L1 controller on the Slicekit Core board.

For this reason the Audio Slice was moved to the triangle socket, and the Ethernet slice to the circle socket. There are no adverse effects caused by moving the Ethernet Slice, which is capable of being used in any slot.

It should be noted that the xTag-2 programmer is only used during programming and debugging. It is not necessary for the operation of the ConMan system.

6.4 Software

6.4.1 Design Methodologies

The design of the ConMan system aimed to merge two designs: a flexible connection management system, and a deterministic MIDI transmission and routing system. For this reason a single design methodology is not capable of accurately representing the design of the ConMan system.

6.4.1.1 Object-oriented

To detail the design decisions and changes made to MIDINet’s connection management system, an object-oriented design methodology was used. The object-oriented design allows for a simple and concise explanation of the objects in the connection management system. Object-oriented design diagrams are used in Chapters 6 and 7 to detail the design changes and implementation of the MIDINet module.

6.4.1.2 Real-Time Systems

The design of the MIDI routing and transmission system makes use of a real-time design methodology developed by Ward and Mellor [61]. The methodology makes use of two separate models: an Essential model and an Implementation model. The choice to use this model was primarily based on its use in Dibley’s “An investigation of the XMOS XS1 architecture as a platform for development of audio control standards” [9].

The Essential model used in the structured development of real-time systems provides a set of tasks which the system will accomplish. These tasks are then grouped together to be viewed as broader tasks. The model provides an overview of all the tasks that will be handled by a system, and gives an abstract view of how these tasks might be grouped together.[61] This model is used throughout Chapter 6 to describe time-sensitive tasks for MIDI routing and functions used to interact with the object-oriented MIDINet component.

The Implementation model is used to facilitate the implementation of the system. The model uses the tasks from the Essential model and reorganises these tasks based on the hardware resources that are available during the implementation. This allows for the implementation to leverage processor architecture and language specific features. This is achieved through the regrouping of tasks and assignment of tasks to processors and processes. The Implementation model is used in Chapter 7 to detail the task assignment and implementation.

6.4.2 System Context

The software that runs on the ConMan device allows for the routing of MIDI messages over Ethernet networks, the management of the devices themselves and creation and destruction of connections within the system.

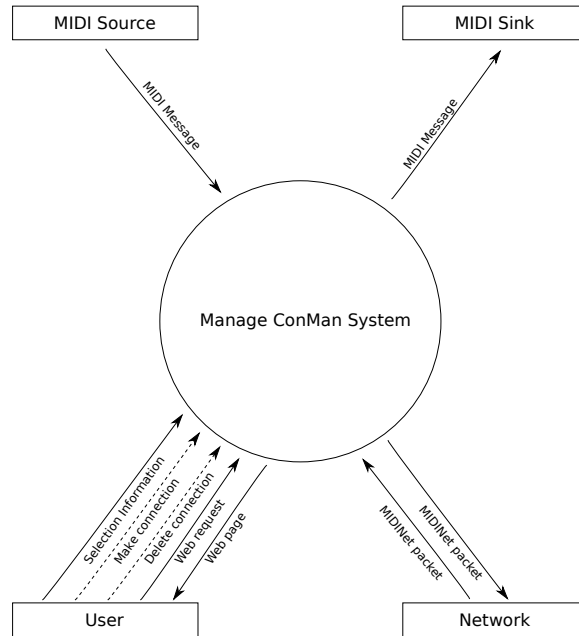


Figure 6.3: Context diagram from MIDI ConMan system.

The context diagram in Figure 6.3 shows how the system interacts with external components. The system interacts with a network through which it communicates with other identical devices.

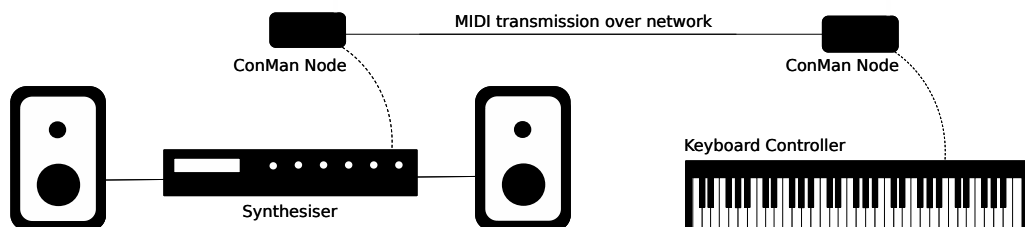


Figure 6.4: MIDI ConMan system for keyboard control of synthesiser.

The ConMan device is capable of being connected to one or more MIDI devices which would be used to either provide MIDI messages that would be transferred over the network, or to receive MIDI messages via the network as shown in Figure 6.4.

Since the MIDI ConMan hardware does not have a display, controls, or any other physical interfaces on-board for the user to control the hardware, the device makes use of an embedded web server to display a control application through standard web browsers.

The MIDI ConMan system can be broken into 3 main tasks:

- Route MIDI Messages
- Manage Connections
- Provide User Controls

The relationships between these tasks are shown in Figure 6.5. These tasks are to be carried out concurrently, allowing for MIDI message to be routed deterministically between ConMan nodes. Although the connection management is not required to be deterministic, the task of managing connections will still need to run parallel to MIDI routing, so that the addition or removal of connections does not impact the time taken to route a MIDI message.

The same applies to the task of providing user controls. These controls are required to be available at all times. Although there are no true time constraints on the user controls, it is imperative that the use of these controls does not interfere with MIDI routing activity.

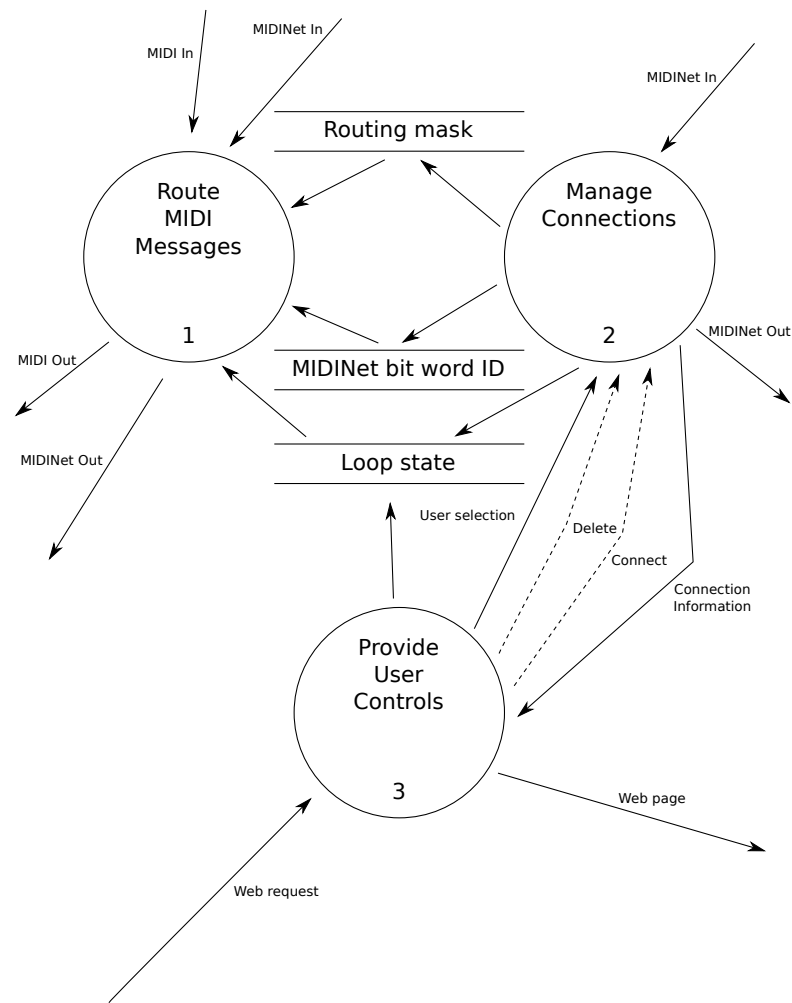


Figure 6.5: Ward & Mellor System overview for a ConMan device.

To ensure that MIDI routing is deterministic, the check for active connections needed to be deterministic. To avoid the non-determinism of the object-oriented connection management, the design makes use of a routing mask created as part of the connection management (Task 2 in Figure 6.5) so that it can be used during the MIDI routing (Task 1 in Figure 6.5).

6.4.3 Routing MIDI Messages

The task of routing MIDI messages can be further broken down into smaller tasks. Figure 6.6 shows the tasks required to route MIDI messages, and how these tasks are related.

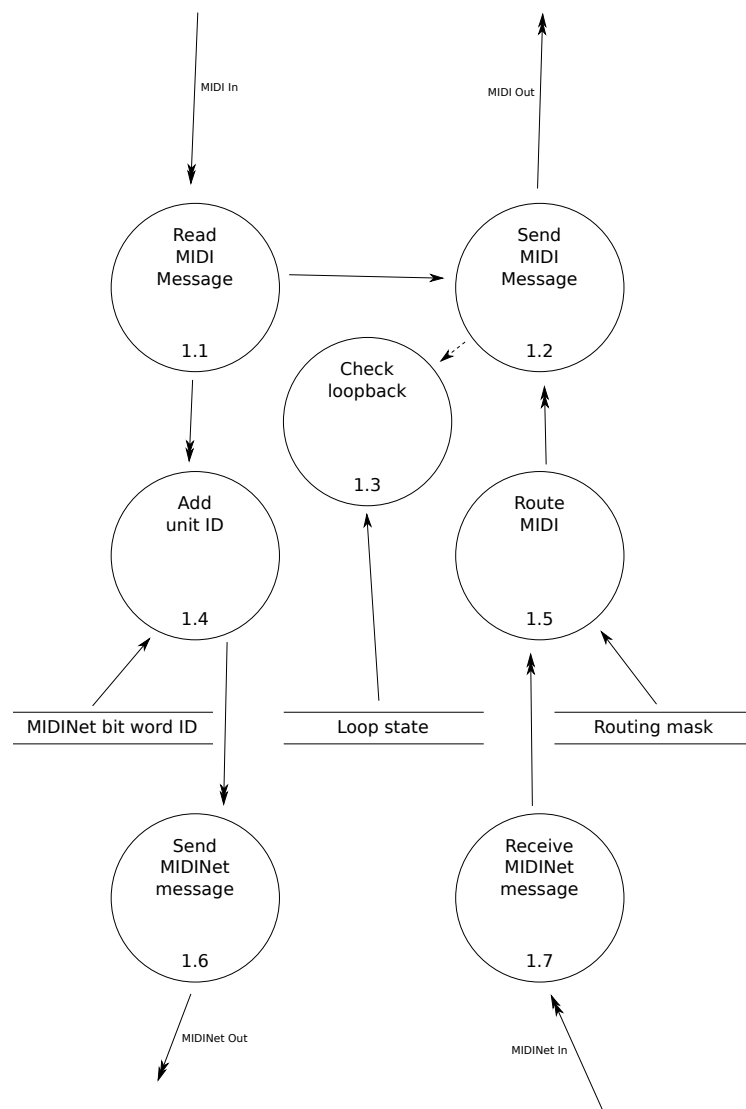


Figure 6.6: Detailed view of the Route MIDI Messages task.

These tasks can be further grouped into sets of related tasks: Handling MIDI ports, handling MIDINet MIDI messages and handling MIDI routing. These groups are now investigated and their internal tasks identified.

6.4.3.1 Handle MIDI ports

The handling of MIDI ports involves managing the sending and receiving of MIDI messages through the MIDI ports on the Audio Slice. As discussed in Chapter 3, the Audio Slice features a variety of ports, including a MIDI input port and a MIDI output port. As no MIDI handler had been provided by XMOS, the design of such a handler was investigated so that it could be implemented.

The MIDI input port on the Audio Slice receives incoming MIDI messages by reading the current status of the line and packing the received bits into a MIDI message structure when a valid message has been received.

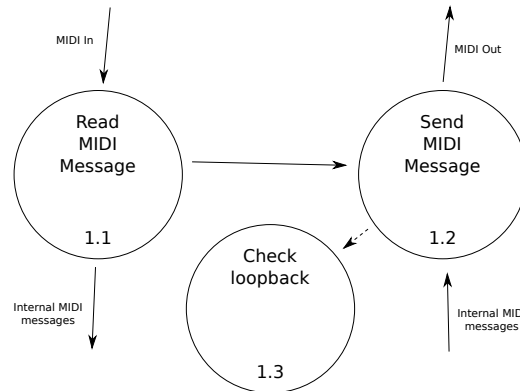


Figure 6.7: Group of tasks responsible for handling MIDI ports.

The tasks associated with sending and receiving MIDI messages using the Audio Slice can be grouped together as shown in Figure 6.7. The figure shows the sending and receiving of MIDI messages through the ports of the Audio Slice, as well as the internal loopback capability that allows for messages received on the Audio Slices input to be routed directly to the output of the same Audio Slice. The figure also shows the internal MIDI messages that are sent and received from the group of tasks that handles the routing of MIDI within the ConMan system. These tasks are described in Section 6.4.3.3.

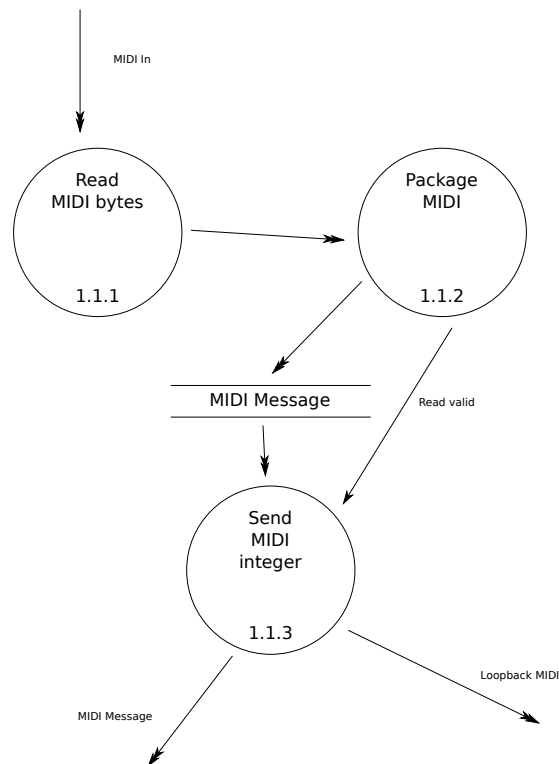


Figure 6.8: Task 1.1: Reading MIDI message

Figure 6.8 shows the task of reading in MIDI messages. Message bytes are read from the MIDI port of the Audio Slice, and packaged as integers. The choice to use integers for the packaging of MIDI was due to the largest (non SysEx) MIDI messages being 3 bytes long and an integer being 4 bytes. This allows for messages throughout the system to be dealt with as single integers.

MIDI devices that send MIDI messages do not know that MIDI streams might be merged within the ConMan system and so might implement running status messages to minimise the number of messages being sent. Running status is a mechanism implemented as part of the MIDI protocol to minimise the number of MIDI messages that are required to be sent at any time. This is achieved by lowering the number of status messages sent by only sending data bytes if the subsequent messages contain the same status. In the ConMan system, running status messages mean that MIDI streams that are merged at a receiving ConMan device might result in corrupted (incorrect status) messages.

Figure 6.9 shows an abstract view of a MIDI configuration where two keyboard controllers are controlling the same synthesiser.

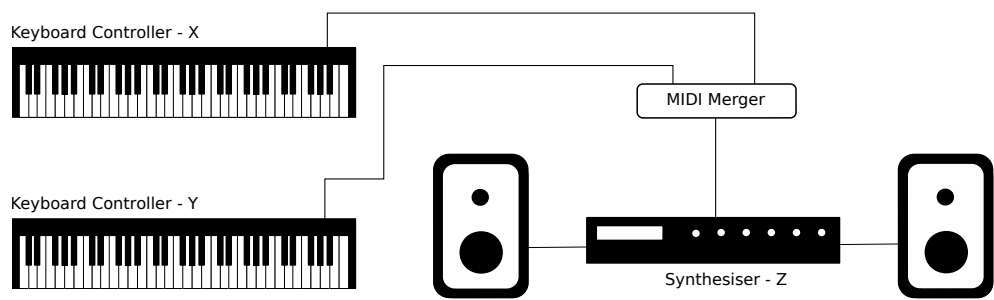


Figure 6.9: Example configuration for a MIDI merge

Table 6.1: Corruption of MIDI through merging when using running status

Timestamp	Keyboard X			Keyboard Y			Synthesiser Z		
0001	B1	00	00				B1	00	00
0002	–	00	7F				–	00	7F
0003				B2	00	00	B2	00	00
0004				–	00	7F	–	00	7F
0005	–	00	00				–	00	00

The messaging that could occur in a configuration such as the one shown in Figure 6.9 is shown in Table 6.1. The table shows Keyboard X sending a Control Change message at timestamp 1 over MIDI channel 2, setting controller 0 to 0. At timestamp 2, another message is sent, setting controller 0 to 127. Since Keyboard X is using running status, only the 2 data bytes are sent as the status byte did not change.

At timestamp 3 Keyboard Y sends a Control Change message over MIDI channel 3, setting controller 0 to 0. Keyboard Y then changes the value of controller 0 to 127 at timestamp 4. Until this point Synthesiser Z has received these messages and has made control changes correctly.

When another control change message is sent from Keyboard X at timestamp 5, changing the value of controller 0 to 0, the synthesiser does not make the correct change. Although Keyboard X intended to send the message over channel 2, the last status byte received by Synthesiser Z was for channel 3, and so the received data bytes are applied to the wrong channel, thus corrupting the MIDI messages.

To avoid the corruption of messages during the merging of streams, tasks handling MIDI ports employ their own running status system. Should the message not contain a status byte, the message will be passed with the last status that the MIDI Input received as shown in Figure 6.10. The assumption is that all MIDI messages being received at this

point are correct, and that no MIDI merging has occurred prior to the ConMan device's input port receiving the message.

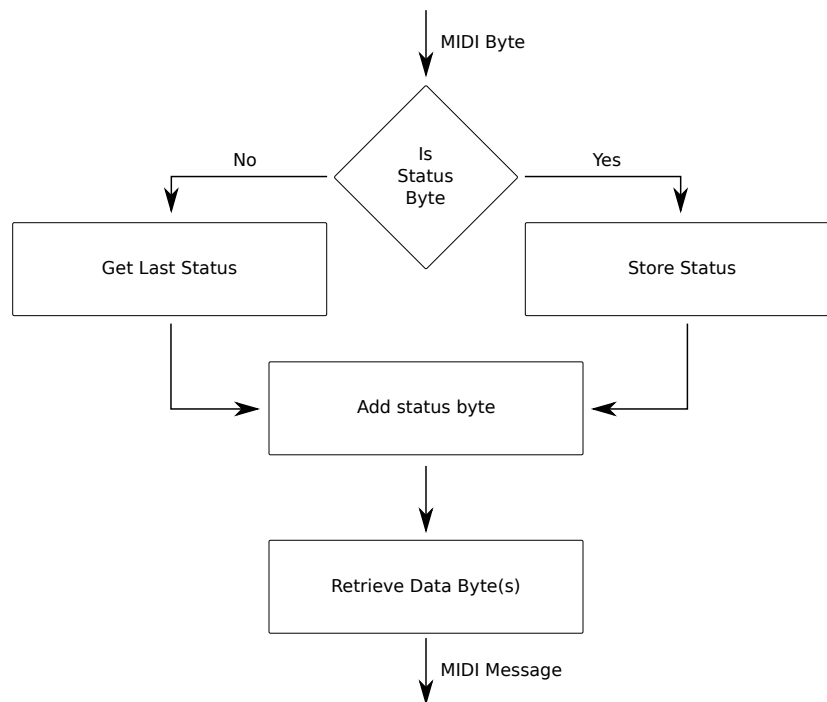


Figure 6.10: Removal of Running Status during MIDI input handling.

By doing this, all messages that are passed on from the MIDI input tasks to be sent over the network include status messages, making it impossible for data bytes in a running status system to be associated with the wrong status message.

As shown in Figure 6.6, the task that sends MIDI messages (Task 1.2) handles MIDI messages from either the MIDI input (Task 1.1), or from the network (Task 1.5). Task 1.2 is shown in more detail in Figure 6.11. On receiving a MIDI message, Task 1.2 breaks the message into bytes and sends the bytes out of the MIDI output port on the Audio Slice.

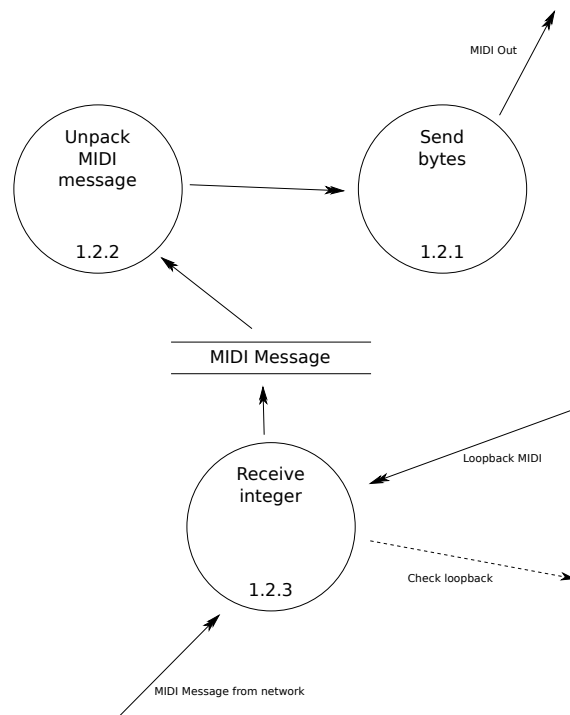


Figure 6.11: Task 1.2: Sending MIDI message

MIDI messages that are received by a MIDI input port on the Audio Slice can be routed to only two places: the local MIDI output port, or over the network.

To reduce possible latency, the MIDI handler was designed to include a loopback flag. When set, this causes all received messages to be transmitted to the output port as well as being sent to the tasks associated with routing. If a connection is created within the Connection Server between the MIDI input and output of a single ConMan device, the loopback flag is set.

This removed the need for MIDI messages to be sent onto the network and then to be routed back to the local output port, decreasing network traffic and removing the need for additional processing within the ConMan system.

All message routing, with the exception of the internal loopback, is handled by the ConMan Core. The MIDI handler does not perform any processing on the message beyond the replacement of running status bytes. The tasks associated with handling MIDI ports are further described and their implementation outlined in Section 7.3.

6.4.3.2 Handle MIDINet MIDI messages

Figure 6.12 shows the group of tasks responsible for sending and receiving MIDINet messages. MIDI messages that have been received by the MIDI input port will be packaged, along with the ID of the local ConMan unit, according to the MIDINet protocol and sent to the network. MIDINet messages received by the network that contain MIDI will be routed to the MIDI output if a connection exists.

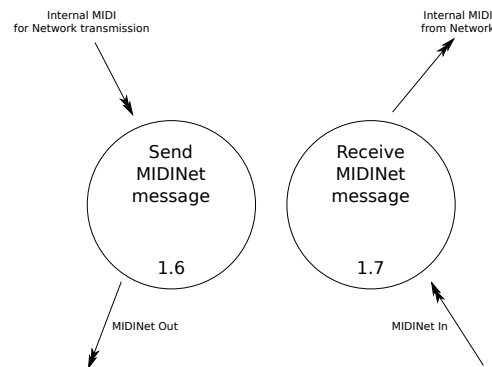


Figure 6.12: Group of tasks responsible for handling MIDINet messages.

Task 1.6 in Figure 6.12 handles the sending of MIDINet messages. This task will receive 2 integers: the first containing the ID of the local ConMan unit and the second containing the MIDI message. These messages will then be sent to the network so that they can be received by all ConMan devices.

MIDINet messages received from the network are handled in Task 1.7 of Figure 6.12. This task deals with extracting the sender's ID and MIDI message from the MIDINet packet. Once the ID and MIDI message have been read from the packet, the information can be used to facilitate the routing of the MIDI message.

6.4.3.3 Handle MIDI routing

As it is important that MIDI routing remains deterministic, the handling of MIDI messages both received from the MIDI inputs and received from the network had to require minimal processing.

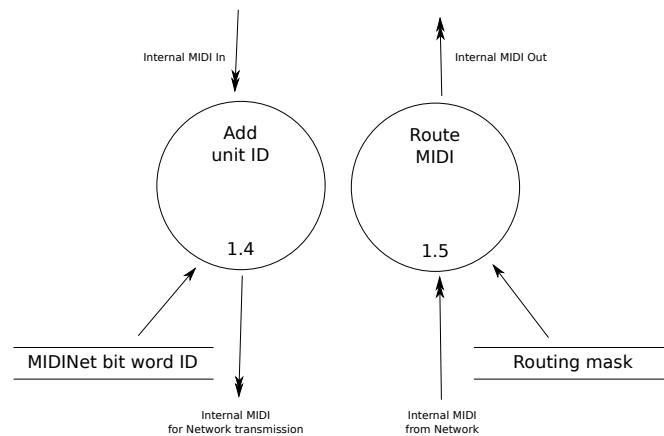


Figure 6.13: Group of tasks responsible for routing MIDI messages.

Messages received from the MIDI component will have the ID of the unit attached to them, before they are forwarded to the network for transmission. This allows all MIDI messages to have their sender identified. This is handled by Task 1.4 of Figure 6.13.

Task 1.5 (shown in Figure 6.13) handles the routing of MIDI messages received from the network. When MIDI messages are received from the network they are accompanied by the ID of the ConMan unit that sent the MIDI. This ID is compared to a routing mask to determine whether the MIDI should be output. The routing mask is provided by the Manage Connections task shown as Task 2 in Figure 6.5. The construction of the routing mask, as well as its use in routing MIDI messages is detailed in Section 7.4.

6.4.4 Managing Connections

The ConMan system provides deterministic transmission as well as flexible routing. The ConMan system makes use of MIDINet's connection management. This required that the components responsible for handling the MIDINet connection management system be transformed into a module that could be used within the ConMan system.

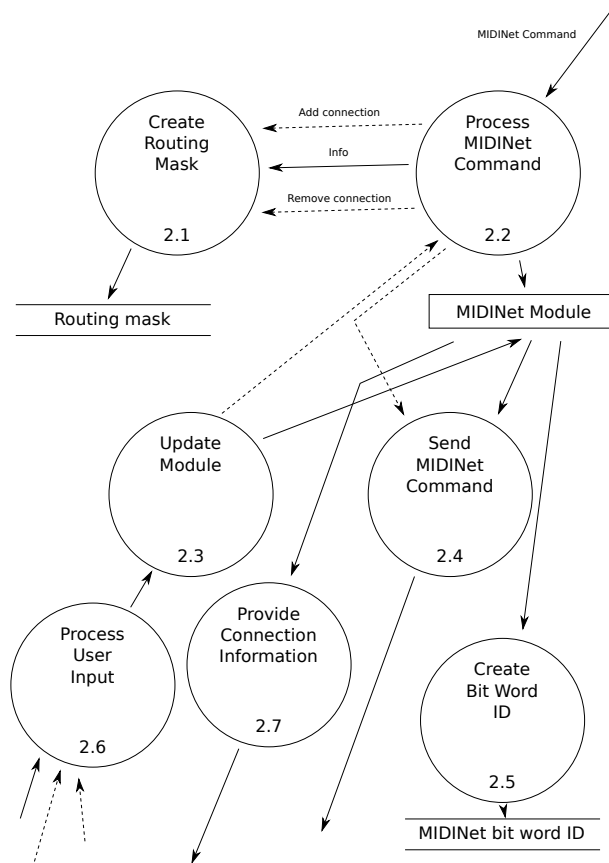


Figure 6.14: Detailed view of the Manage Connections task.

Figure 6.14 shows the tasks associated with managing connections in the ConMan system. The figure shows the interaction of tasks required to process user input, provide routing information and send MIDINet commands to the network. The figure also gives an abstract view of how the MIDINet module is used within the system. The following section will describe the components of the connection management task.

6.4.4.1 Handle Connection Server

The MIDINet module is implemented in C++, and is not considered part of the real-time system. The management of the module is handled by a set of tasks that allow the MIDINet module to communicate with real-time tasks.

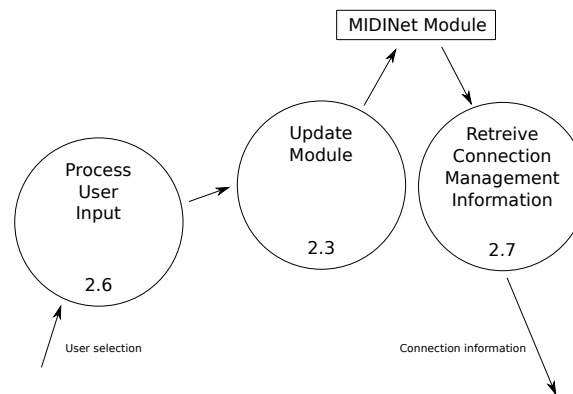


Figure 6.15: Group of tasks responsible for handling connections.

The tasks shown in Figure 6.15 provide the management necessary for the MIDINet module. The MIDINet module provides information that will be used in generating a web based user interface, and accepts commands from the user through the web interface.

Users create connections by selecting and connecting ConMan devices via the web interface. The task that handles these web requests passes the selected units to the task handling the connection server. The user selections are processed (shown in Figure 6.15 as Task 2.6) to provide the parameters that are used to update the MIDINet module (Task 2.3 in Figure 6.15).

Any information regarding ConMan units and current connections that is used within the web based user interface is provided through Task 2.7 in Figure 6.15. Task 2.7 provides this information to the tasks that handle the generation of the web application (Task 3 in Figure 6.5).

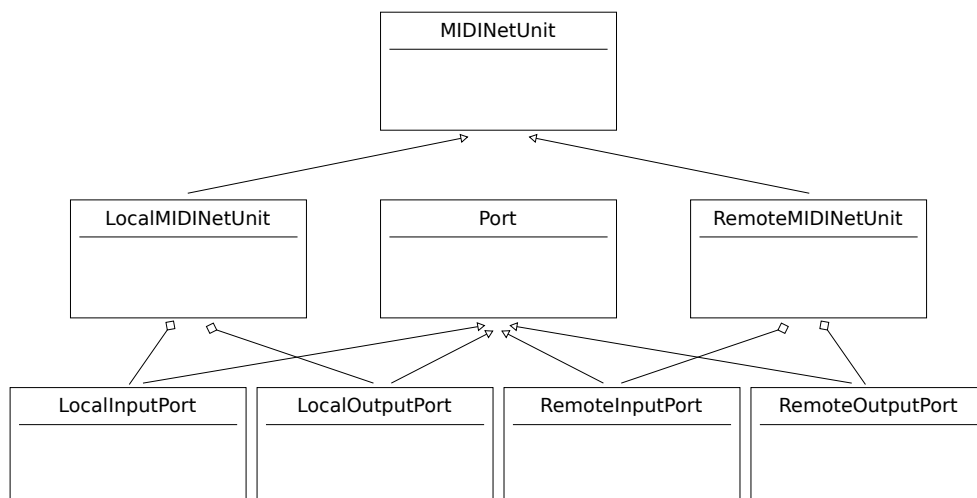


Figure 6.16: Simplified class diagram for MIDINet module.

For the MIDINet module, the MIDINet design (discussed in Chapter 5) was modified to just provide the core connection management capabilities. Most notably, the MIDINet module does not require classes for networking, a user interface, or MIDI messaging, as these tasks are handled by the ConMan system. The implementation of the MIDINet module and surrounding tasks is detailed in Section 7.7.

A simplified class diagram showing the object oriented design of the MIDINet module is shown in Figure 6.16. The full object-oriented design of the MIDINet module is detailed in Appendix B.

6.4.4.2 Process MIDINet Commands

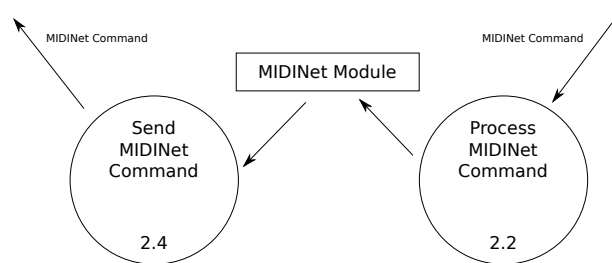


Figure 6.17: Group of tasks responsible for processing MIDINet commands.

Figure 6.17 shows the group of tasks that manage the sending and receiving of MIDINet commands. In the original MIDINet system, these tasks are handled within the Network class² which dealt with the sending and receiving of all network data, including MIDI messages. The design of the ConMan system specifies dedicated tasks that manage the routing of MIDI messages, ensuring the deterministic transmission of MIDI to and from the network.

The handling of MIDINet commands used for Discovery and Connection Management is provided by Tasks 2.4 and 2.2 of Figure 6.17. When a user creates a connection through the web based user interface, the connection is created within the MIDINet module. In order for all ConMan devices to hold synchronised connection information, Task 2.4 sends a MIDINet command to the network instructing other ConMan devices to create the new connection.

Task 2.2 handles the processing that occurs when ConMan devices on the network receive a MIDINet command for a new connection. The requested connection is created by the

²The MIDINet system is analysed in Chapter 5. The design and operation of the system and its components are investigated.

MIDINet module and the new connection is then updated in the Routing Mask if the connection affects the local ConMan device.

6.4.4.3 Manage Routing Information

Figure 6.18 shows the tasks associated with handling routing information. These tasks include the creation of the local unit's ID, and the creation and maintenance of the routing mask. An integer that is used to determine whether received messages should be output by the local ConMan device.

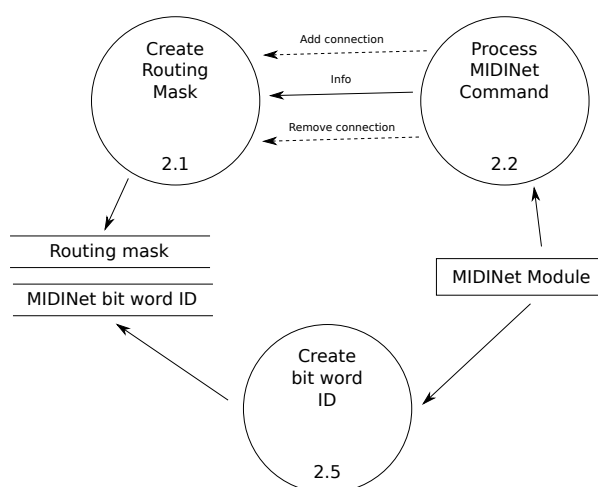


Figure 6.18: Group of tasks responsible for handling routing information.

The ID used within the ConMan system is based on the use of ID within the MIDINet system. Every ConMan unit runs a MIDINet module, and therefore has a MIDINet ID. These IDs are used to uniquely identify the units, and are used in the ConMan system to route messages.

	ConMan Unit Number							
	8	7	6	5	4	3	2	1
ConMan unit 1	0	0	0	0	0	0	0	1
ConMan unit 2	0	0	0	0	0	0	1	0
ConMan unit 5	0	0	0	1	0	0	0	0

Figure 6.19: Example ConMan unit number when represented in binary.

Task 2.5 shown in Figure 6.18 used the MIDINet ID of the local unit to produce an ID that is used when routing MIDI over a network. The ID makes use of a binary representation where a set bit represents the number of the unit. Example IDs are shown in Figure 6.19,

where a unit with a MIDINet ID of 1 (ConMan unit 1) is shown to have the first bit³ set.

The ConMan ID can be create using the formula shown below:

$$ConManID = 2^{MIDINetID-1}$$

ConMan system makes use of 32 bit unsigned integers for IDs and routing masks⁴. The examples given in this section make use of 8 bit binary representations, and are only to provide examples to demonstrate how the ID and routing mask are created and used.

The routing mask is an amalgamation of ConMan IDs which are connected to the local unit. Figure 6.20 shows an example routing mask. The mask in the example allows for messages from units 2, 3, 5 and 8 to be routed. In the figure are also the IDs for units 1, 2 and 5.

	Routing mask							
	1	0	0	1	0	1	1	0
ConMan unit 1	0	0	0	0	0	0	0	1
ConMan unit 2	0	0	0	0	0	0	1	0
ConMan unit 5	0	0	0	1	0	0	0	0

Figure 6.20: Example of ConMan units compared to a routing mask.

The creation of the routing mask is handled by Task 2.1 in Figure 6.18, which receives information on connections associated with the local unit. The number of the remote unit, as well as an instruction to either add or remove the connection is provided by the MIDINet module. This information is used to update the current routing mask so that it contains all current connections.

The ID and routing mask are used to determine if a MIDI message should be routed. This is done by comparing the binary representation of the ID and the routing mask. If the bit that is set in the ID is also set in the routing mask, a connection exists between the sender and the local unit and the message is output.

In Figure 6.20 the routing mask contains set bits for ConMan units 2 and 5, resulting in messages from these units being routed, and message from ConMan unit 1 being ignored. This routing is handled by Task 1 in Figure 6.5.

³The first bit in this context represents the least significant bit.

⁴Although the ConMan system currently makes use of 32 bit integers for IDs and routing masks, this could be modified to accommodate a larger number of ConMan devices and connections.

The routing mask-based approach is an enhancement to the MIDINet system and ensures fast routing.

6.4.5 Providing User Controls

As the ConMan hardware has no physical user interfaces through which the user can access the device's settings or the functions of the ConMan system that will run on it, a web based user interface is used to provide the user with a graphical control system. The web application provides all the functionality that the user will require to create or destroy connections within the ConMan system.

Tasks involved in managing connections make use of the MIDINet protocol, allowing for the creation of MIDINet units, inputs, outputs and connections. This allows the core functionality of the MIDINet system to interact with the real-time aspects of the ConMan system.

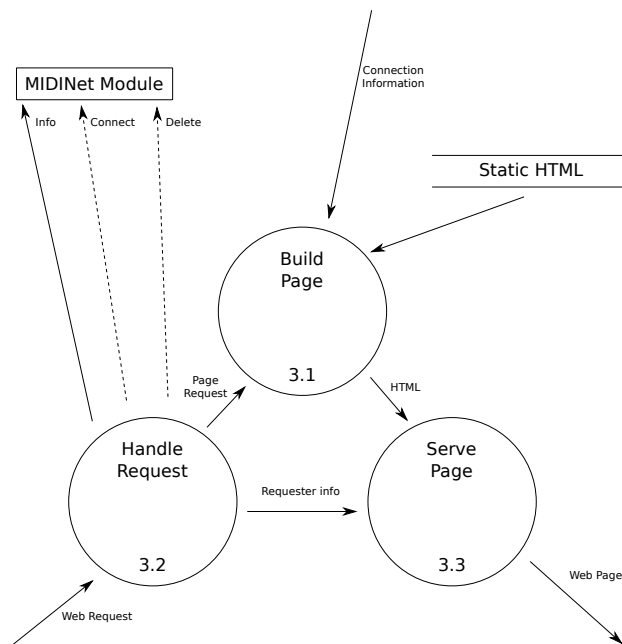


Figure 6.21: Detailed view of the 'Provide User Controls' task.

Task 2 in Figure 6.5 manages connections and the MIDINet module. All interaction with the user is handled by Task 3: Provide User Controls. Task 3 (detailed in Figure 6.21) is responsible for providing the user with a web page, handling requests from the user, and retrieving information from the MIDINet module.

6.4.5.1 Build page

The task of building a web page within the ConMan system is shown in Figure 6.22. Pages built by this task consist of two main elements: The page template and the connection information.

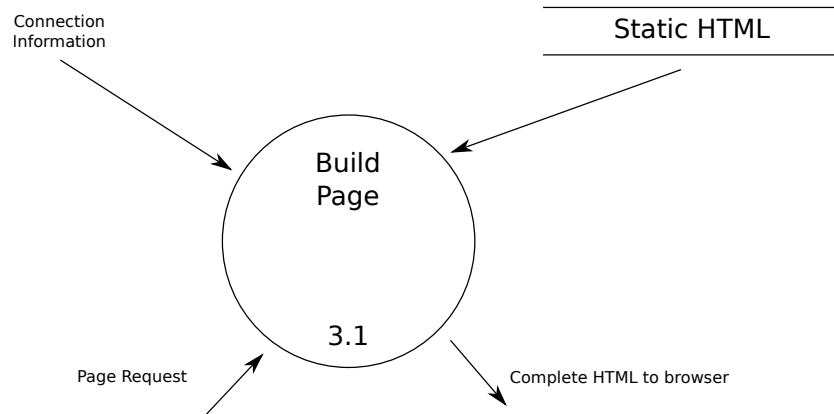


Figure 6.22: Task and data flows associated with building a page.

The page template is the HTML required for the application. It provides sections to show input ports, output ports and current connections, as well as buttons that will provide the requests necessary to either create or destroy connections.

The connection information is provided by the task that manages connections (Task 2 in Figure 6.5). This is the information from the MIDINet module, which in the original MIDINet application would have been available via the graphical user interface.

Building a page requires that the MIDI inputs, outputs and connections are wrapped in HTML so that they may be embedded within the template, and a complete web page served to users. This is handled by Task 3.1 in Figure 6.22, with the completed HTML containing all connection information and the functionality required to manage connections within the ConMan system.

6.4.5.2 Serve Page

Web servers operate by sending information to a client that has requested it. Task 3.3 shown in Figure 6.23 handles this aspect of the web server. The diagram shows the task that serves the HTML of the complete web application to the client as a web page.

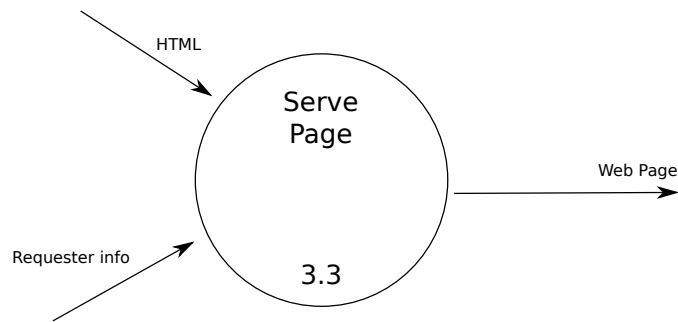


Figure 6.23: Task and data flows associated with serving a page.

When a web page is requested, the requester's information needs to be supplied to the task that serves the page so that the complete HTML can be sent to the correct client. The task also requires the HTML that has been built by Task 3.1, so that the complete lists of inputs, out and connections are included in the application.

The task provides the user with the controls necessary for interacting with the ConMan system by sending the web page to the user. The user's web browser translates the page into a graphical user interface.

6.4.5.3 Handle Request

Building and serving pages handle the aspects of the web application that involve the user interface. Once the user has the interface they are able to select input and output ports and create connections.

The selection of ports generates a request that needs to be handled so that the MIDINet module can create (or destroy) a connection and forward these changes to the rest of the ConMan units in the system.

Figure 6.24 shows the task of handling a web request from a client. These web requests can either be requests for the web application, or requests generated by interaction with the web application.

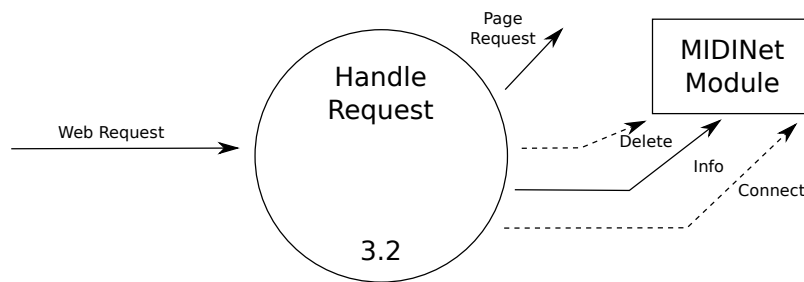


Figure 6.24: Task and data flows associated with handling a request page.

When a user requests the web application, Task 3.2 provides a page request, which results in the web page being built and then served to the client. These are classified as page requests and involve providing the user with the current state of the MIDINet Module by serving a web page.

Connection Management requests are used to inform the ConMan system that the user has selected ports to connect or has selected a connection to be destroyed. These requests are generated by the web application and sent to the ConMan device based on choices made by the user.

6.5 Conclusion

The ConMan system offers a low cost, small form factor design for a MIDI connection management system. The systems design allows for the creation of a connection management system that does not require a computer to function, but makes use of a web application for configuration and management.

Real-time design methodologies were used to describe a parallel software design that enables the implementation of a deterministic system for the transmission of MIDI messages over a network.

The design described in this chapter provides a template for the implementation of the ConMan system. Chapter 7 describes the implementation phase of the ConMan system.

Chapter 7

Implementation

7.1 Introduction

The XMOS architecture was introduced in Chapter 3, where key aspects of the architecture were discussed. The multiple cores of the XMOS microcontroller are made available through the SliceKit development board, and allow for the design and implementation of parallel software components.

The design of the ConMan system was discussed in Chapter 6, where the Essential Model was used to identify the tasks that are required for the system to operate. The design was accompanied by a hardware configuration for the XMOS SliceKit when used with the Audio Slice and Ethernet Slice. This information was used in the creation of an Implementation Model (introduced in Chapter 6).

This chapter describes the implementation of the ConMan system according to the Implementation model. The Implementation model uses the Essential model to assign tasks to processors and group similar functions so that features of the XMOS architecture and xC language can be leveraged. The chapter will then detail the implementations of the various concurrent tasks that run within the ConMan system.

7.2 Implementation Model

The XC language provides a set of extensions to C that allow for the utilisation of the multicore parallel architecture of XMOS's xCore microcontrollers. The design outlined in

the Chapter 6 aimed to make use of the features provided by XMOS through the use of real-time design methodologies.

The Slicekit development board discussed in Chapter 3 makes use of a microcontroller that comprises of two tiles, where each tile has 8 cores and its own set of input and outputs.

Section 6.3 of the Design chapter showed the intended hardware configuration of the Slicekit when used with the Audio Slice and Ethernet Slice. The Audio Slice is attached to the Triangle slot, which connects it with Tile 0. The Ethernet Slice is attached via the Circle slot, connecting it to Tile 1.

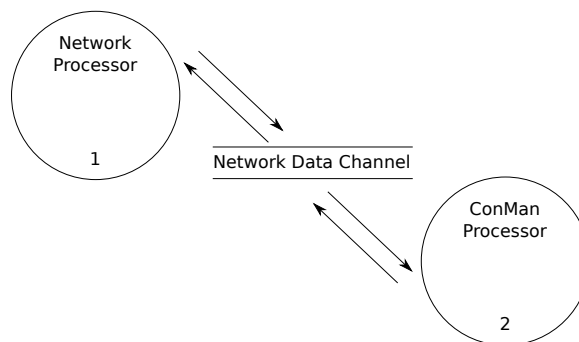


Figure 7.1: System Processor View

The Processor View of the Implementation Model identifies which processors the tasks described in the Essential Model should be implemented on. The configuration of the XMOS Slicekit results in the processor view shown in Figure 7.1, where one tile of the microcontroller handles network related tasks, and the other tile manages the ConMan system.

The figure shows the system having two processors: A Network processor and a ConMan Processor. These processors communicate over a channel, shown as the Network Data Channel in Figure 7.1. As the two XMOS tiles do not share the same input and output pins or memory, all communication between them is done over the channel.

It should be noted that the distribution of tasks to tiles was primarily based on grouping of related tasks such as MIDI or networking, but beyond that largely based on even distribution of resources on the XMOS tiles. This was done to aid future modification or development, but was not strictly necessary.

7.2.1 Network Processor

The network processor runs the parallel components required to handle network related tasks. Figure 7.2 shows the high-level overview of the tasks that run on the network processor.

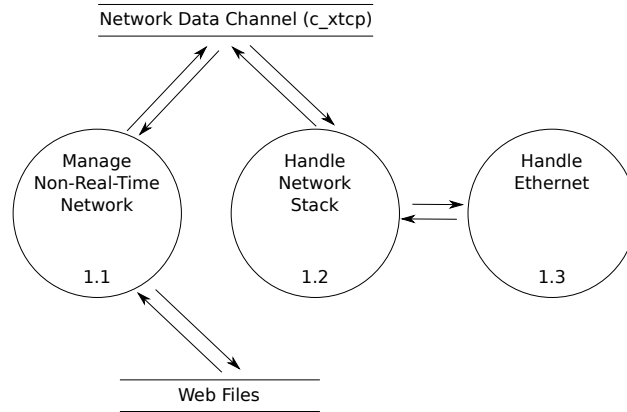


Figure 7.2: Network Processor: Parallel task structure

Tasks 1.2 and 1.3 in Figure 7.2 show the handling of the Network stack and Ethernet respectively. These tasks are implemented using the XTCP library provided by XMOS[70]. The implementation of these tasks is discussed in Section 7.5.

Task 1.1 in Figure 7.2 handles the non-real-time aspects of the network. These tasks are primarily related to the handling of network traffic for the web server and their implementation is discussed in Section 7.5.1.

The Embedded Webserver library provided by XMOS is used to create the web application which manages connections in the ConMan system. The design of the web application was introduced in Section 6.4.5 where the required operations of the application were discussed. The implementation of the web application and related features is explained in Section 7.6.

The task of handling the real-time network traffic is implemented on the ConMan processor and is discussed in Section 7.2.2. Both the real-time and non-real-time network handlers make use of the Network Data channel to communicate with the network stack.

7.2.2 ConMan Processor

The ConMan processor runs a set of separate parallel tasks as seen in Figure 7.3. The figure shows a high level overview of the logical task layout of the system, and shows the channels over which these tasks communicate.

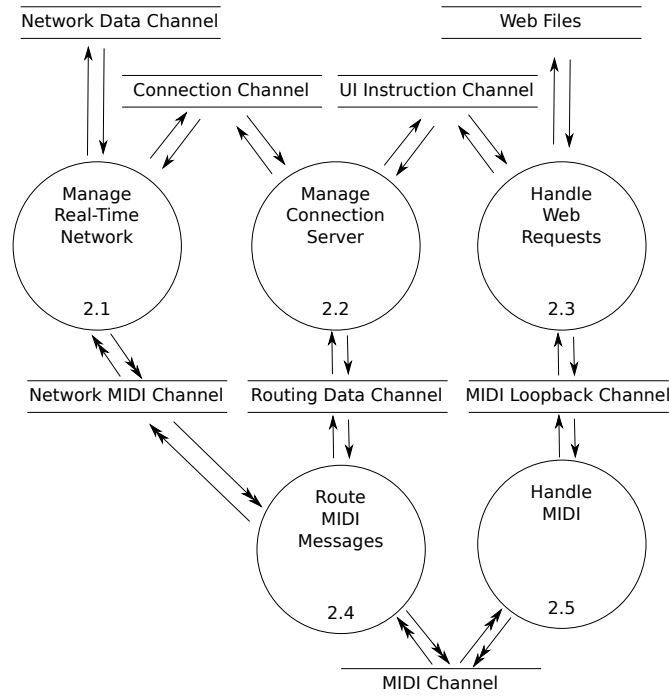


Figure 7.3: ConMan Processor: Parallel task structure

The channels that are shown receiving communication with double arrowheads represent fixed channels: channels that have been declared as streaming channels so that they will be permanently available for use by the connected tasks. Channels that are shown as receiving data flows with a single arrowhead are declared as regular channels.

The ConMan processor can be viewed as having two separate but related systems: a real-time routing system that uses streaming channels and a non-real-time connection management system.

7.3 MIDI Handler

At the commencement of the project, there was no implementation of a MIDI handler for the Audio Slice provided by XMOS. A small work-in-progress MIDI application[14]

by Barna Faragó was found on the xCore Community forums[38]. This application was used as a base for the MIDI handler created for this project.

The MIDI handler was designed to ensure that messages being read in or sent out experience the lowest possible latency. To achieve this it was decided that all channels that are used for the transmission of MIDI messages were to be implemented using streaming channels so that the channels are set up permanently and are not renegotiated when needed.

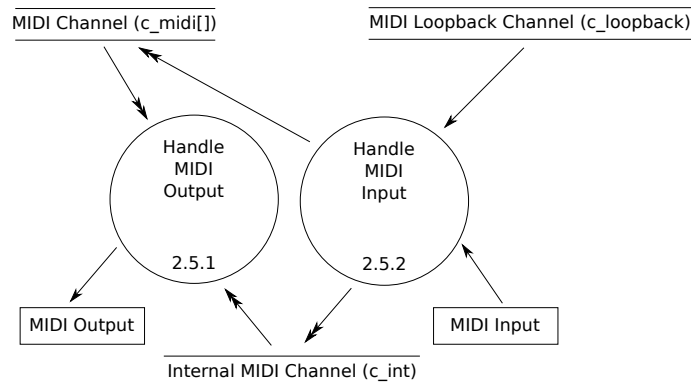


Figure 7.4: Handle MIDI Ports: Overview of the parallel tasks

Figure 7.4 shows the parallel nature of the tasks that comprise the MIDI handler. These tasks include handling MIDI messages coming from the input MIDI port, and outputting MIDI to the output MIDI port.

The handler communicates with the ConMan Core (discussed in Section 7.4) through the MIDI Channel, sending MIDI so that it can be routed to the network, and outputting MIDI messages received from the ConMan core.

Commands that enable or disable loopback are received via the MIDI Loopback Channel, allowing the the MIDI input handler to use a separate Internal MIDI Channel to route MIDI messages directly to the output handler.

Listing 7.1, shows the implementation of the MIDI handler (task referred to as `midi_handler` within the code). The MIDI handler runs two parallel tasks: `MIDIIn` (Task 2.5.2 in Figure 7.4) and `MIDIOut` (Task 2.5.1 in Figure 7.4).

Listing 7.1: MIDI Handler

```

void midi_handler(chanend c_loop, streaming chanend c_midi[])
{
    streaming chan c_int;    //Channel for loopback of MIDI messages
    par
    {
        MIDIIn(c_int, c_loop, c_midi[1]); //Handles MIDI input port
    }
}

```

```

        MIDIOut(c_int , c_midi[0]); //Handles MIDI output port
    }
}

```

As shown in Listing 7.1, the tasks make use of an array of streaming channels called `c_midi` (MIDI Channel in Figure 7.4), over which MIDI messages are sent and received from the ConMan Core. The use of two streaming channels allows for a continuous stream of MIDI to be processed by the MIDIIn task while not interrupting the operation of the MIDIOut task that is handling the output of MIDI received from the network.

The streaming channel called `c_int` (Internal MIDI Channel in Figure 7.4) allows MIDIIn to send MIDI messages directly to MIDIOut. This is explained further in Section 7.3.1, where the implementation of the MIDIIn task is detailed.

Commands for enabling and disabling loopback are sent to MIDIIn using the `c_loop` channel (MIDI Loopback Channel in Figure 7.4). As described in Section 6.4.3.1, users can create connections between MIDI inputs and outputs on the same ConMan device. When this occurs the loopback is enabled rather than making use of the routing mask, ensuring the fastest possible MIDI loopback.

7.3.1 Processing MIDI input

As described in Section 6.4.3.1, the handling of the MIDI input within the ConMan system deals with two primary tasks: Reading the port and producing MIDI messages to be used by ConMan. Figure 7.5 shows the Implementation model for handling MIDI input.

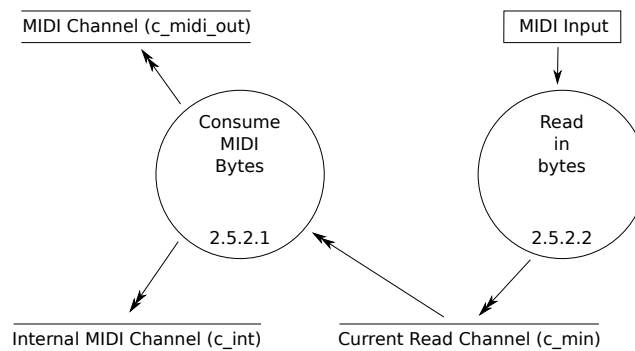


Figure 7.5: Handle MIDI Input: Overview of the parallel tasks

The implementation of the MIDIIn task is shown in Listing 7.2. The task contains a set of parallel tasks responsible for reading in MIDI messages. As discussed in Chapter 3, the Audio Slice has an opto-isolated MIDI input, which is connected to the XS1

microcontroller via a 1 bit wide port (1J). This port is set up and accessed through the `g_MidiUartInPin` pin definition that is taken as an argument by the `sk_midi_port_rx` function.

Listing 7.2: MIDI Input Handler

```
void MIDIIn(streaming chanend c_int, streaming chanend c_midi_out){
    streaming chan c_min; //Channel for MIDI bytes
    sk_midi_port_init_rx(g_MidiUartInPin); //Set up MIDI input pins
    par{
        sk_midi_port_rx(g_MidiUartInPin, c_min); //Read MIDI input port
        consume(c_min, c_int, c_midi_out); //Package MIDI data
    };
}
```

The `sk_midi_port_rx` function is tasked with reading incoming messages received at the MIDI In port of the Audio Slice, and passing them to the Consumer. The Consumer classifies these messages and packages them into MIDI integers to be sent to the ConMan Core. As shown in Listing 7.2, these two functions run in parallel and make use of the `c_min` (Current Read Channel in Figure 7.5) to communicate.

MIDI messages are sent serially as bytes (bit level little-endian). Figure 7.6 shows an example of the serial transmission of a byte. The line remains high (1) when idle, dropping to 0 when signalling the start of a new byte. Once the byte has been received, the line goes high again.

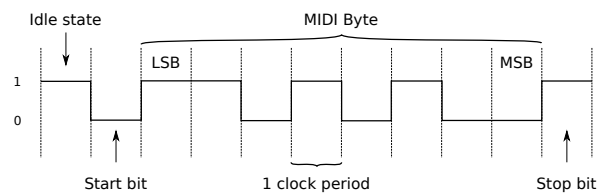


Figure 7.6: Signal view of MIDI byte transmission

The `sk_midi_port_rx` function (Task 2.5.2.2 in Figure 7.5) shown in Listing 7.3 reads the level of the signal that is carried over the wire constantly, waiting for the level to be pulled low, signalling the start of a MIDI byte.

Listing 7.3: Receiver: MIDI Input Port handler

```
void sk_midi_port_rx(SkMidiUartInPinDef.T &dIn, streaming chanend cOut){
    int clocks= dIn.mClkDiv;
    int dt2 = (clocks * 3)>>1;
    int dt = clocks;
    int t;
    unsigned int data = 0;
    while (1)
    {
        if (dIn.mEnabledIn){
            dIn.mPortIn when pinseq(0) :> int - @ t; //Wait for line to go low: Start bit
            t += dt2; //Clock
        }
    }
}
```

```

    for(int i = 0; i < 8; i++) //Read bits
    {
        dIn.mPortIn @ t := >> data; //Package byte
        t += dt; //Clock
    }

    data >>= 24;
    cOut <: (unsigned char) data; //Send byte to Consumer
    dIn.mPortIn @ t := int -; //Line goes high: Stop bit
    data = 0;
}
}
}

```

Once a new byte has been detected, 8 bits are read from the line and stored within an unsigned integer. Once all 8 bits have been read, the unsigned integer is sent via the `c_midi` channel end to be processed by the consumer. The line then goes high signifying the end of the message, the unsigned integer is cleared, and the function is ready to start reading the next byte as soon as the line goes low again.

Listing 7.4: MIDI Consume function

```

void consume(streaming chanend c_min, streaming chanend c_int, streaming chanend c_midi)
{
    MidiRawPacket_T packet;
    int loopback = 1;
    unsigned char d=0;
    while(1)
    {
        select
        {
            case c_loop := loopback : //Store loopback flag
                break;

            default :
                do {c_min := d;} while (!midiReadRawMore(c_min,d,packet)); //Read bytes
                c_midi <: packet.data.midint; //Send integer to ConMan Core
                if (loopback == ENABLED) c_int <: packet.data.midint; //Send integer for loopback
                break;
        }
    }
}

```

The unsigned integer that is passed out of the `sk_midi_port_rx` function is sent over the `c_min` streaming channel, and passed into the consume function shown in Listing 7.4. The consume function (Task 2.5.2.1 in Figure 7.5) allows for multiple MIDI bytes to be read in, processed and classified via the `midiReadRawMore` function before they are sent internally for network transmission or returned to the MIDIOut task for loopback.

To ensure that there is the lowest possible MIDI latency, a loopback flag in the MIDI consume function can be set. The loopback flag enables MIDI messages to be copied directly to the producer over an internal streaming channel in the MIDI handler when a connection is created between an input and an output on the same device.

As shown in Listing 7.4, the Consume function reads from the Current read channel (`c_min`) and checks each byte that is read using the `midiReadRawMore` function. Figure

7.7 gives an abstract overview of the flow of the `midiReadRawMore` function.

When a byte is received from the `sk_midi_port_rx` function, the byte is first examined to determine whether it is a status byte. If the byte received is a status byte, the handler then determines which type of MIDI message the status represents. If the status is for a SysEx message, the function makes use of the SysEx Start handler shown in Listing 7.7. If the status does not indicate a SysEx message, the handler for Channel messages determines how many data bytes are to follow, as shown in Listing 7.5.

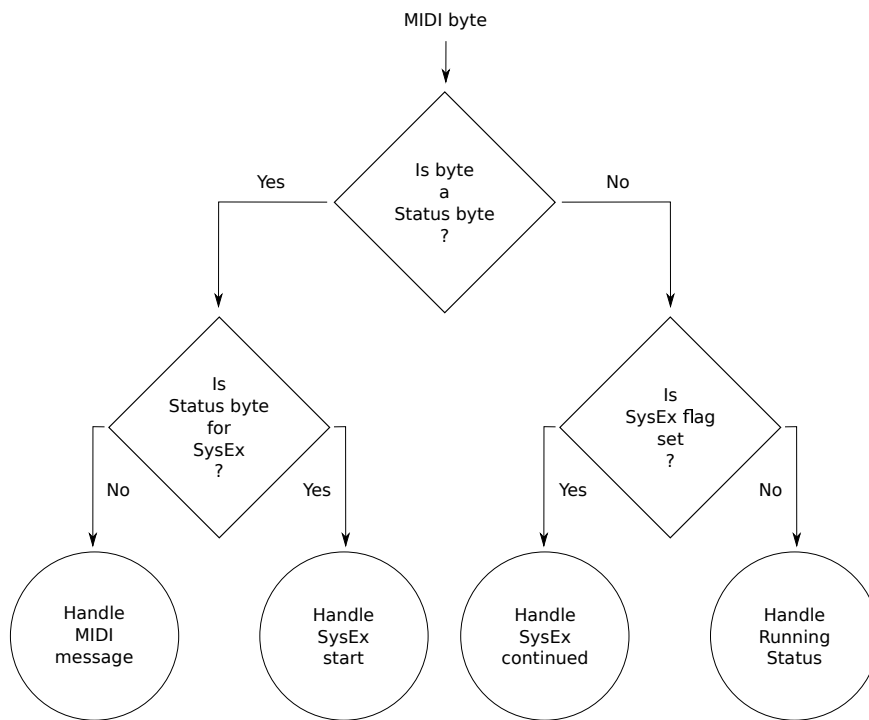


Figure 7.7: `midiReadRawMore`: Handling of received byte.

If the byte received was not determined to be a status byte, the `midiReadRawMore` function handles it as a data byte. The system can receive two types of data byte that are not preceded by a status byte: a continued SysEx message, or a running status message.

If the SysEx flag is set, the function handles the data bytes using SysEx Continued handler shown in Listing 7.8. If the SysEx flag is not set, the byte must be part of a running status message, and is handled by the Running Status handler as shown in Listing 7.6.

The function uses the status byte to determine how many data bytes are expected to follow (based on the MIDI 1.0 Specification [35]). Once a complete and valid MIDI message has been received, it is packaged within an unsigned integer so that it can be easily passed to

functions within the ConMan system.

Listing 7.5: MIDI Handling: Channel messages

```
switch (GETCMD(packet)){ //Determine status
case CM_NoteOff:
case CM_NoteOn:
case CM_KeyAfter:
case CM_CC:
case CM_Pitch:
    cIn:> packet.data.byte[1];
    cIn:> packet.data.byte[2];
    packet.data.byte[3]=3;
    ret= 2; //Message has 2 data bytes
    break;
case CM_PC:
case CM_ChanAfter:
    cIn:> packet.data.byte[1];
    packet.data.byte[2]=0;
    packet.data.byte[3]=2;
    ret= 1; //Message has 1 data byte
    break;
```

The GETCMD macro expression seen in Listing 7.5, takes a packet (MIDI packet) as an argument, and using the first byte within the packet determines the type of MIDI messages being processed. This allows for the handler to receive the correct number of MIDI data bytes associated with each status.

The ConMan system allows for flexible connection management making it possible to merge MIDI streams from two different senders. This avoids situations where the running status of MIDI messages can result in MIDI messages being corrupted as described in Section 6.4.3.1.

Listing 7.6: MIDI Handling: Running Status

```
packet.data.byte[0] = lastStatus; //From earlier message
packet.data.byte[3]=1;
switch (GETCMD(packet)) //Determine status
{
    case CM_NoteOff:
    case CM_NoteOn:
    case CM_KeyAfter:
    case CM_CC:
    case CM_Pitch:
        cIn:> packet.data.byte[1];
        cIn:> packet.data.byte[2];
        packet.data.byte[3]=3;
        ret= 2; //Message has 2 data bytes
        break;
    case CM_PC:
    case CM_ChanAfter:
        cIn:> packet.data.byte[1];
        packet.data.byte[2]=0;
        packet.data.byte[3]=2;
        ret= 1; //Message has 1 data byte
        break;
    case CM_Other:
        //Handle SysEx
}
```

To deal with this issue, MIDI messages received without a status byte have the last received status set as their status as shown in Figure 6.10 in Section 6.4.3.1. The imple-

mentation of the code that handles running status messages (forming part of the `midiReadRawMore` function) is shown in the first two lines of Listing 7.6. All messages coming out of the `MIDIIn` thread are complete messages with correct status messages so that no corruption due to the loss of current status is possible.

Due to the packaging of MIDI messages and the encapsulation in network packets, the inclusion of the status messages has no impact on the latency of MIDI packets. This allows for MIDI messages to all carry status bytes within the ConMan system, removing the possibility of corrupted status messages when merging streams.

The original project on which the MIDI handler was based did not contain an implementation of SysEx messages. In MIDI, there have been few extensions to the specification, with the core messaging structure being oriented toward synthesiser and musical instrument control. The SysEx message type is partly the reason the specification has not required further expansion, as it provides a method for any data to be encapsulated within an open format MIDI message.

As the prevalence of MIDI as an interface for a large number of audio devices can partly be attributed to the flexibility of the SysEx message format, it was seen as an important feature to be implemented in the MIDI Handler. Although not necessary for the proof of concept, testing of the system with SysEx messages is a valuable source of information on the usability of such a system and lends itself to the verification of the hypothesis.

The `midiReadRawMore` function also handles incoming SysEx messages. As SysEx messages can have varying length, the handling of these messages has been implemented so that other MIDI messages take preference. This is discussed further in Section 7.5, where the real-time network transmission component's implementation is detailed.

The code shown in Listing 7.7 shows what occurs when the start of a SysEx message is received. On receiving a SysEx status, the `midiReadRawMore` function sets a SysEx flag to ensure that the function handles the bytes that follow as part of a SysEx message and not running status bytes.

Listing 7.7: MIDI Handling: SysEx Start

```

case CM.Other: //SysEx
    isSysEx = 1; //Is still SysEx
    cIn:> packet.data.byte[1]; //Read next byte
    if (packet.data.byte[1] == 0xFF) //Check for SysEx end
    {
        ret= 1;
        packet.data.byte[2]=0;
        packet.data.byte[3]=2;
        isSysEx = 0; //No longer SysEx
        break;
    }
    cIn:> packet.data.byte[2]; //Read next byte

```

```

    if (packet.data.byte[2] == 0xFF) //Check for SysEx end
    {
        isSysEx = 0; //No longer SysEx
    }
    packet.data.byte[3]=3;
    ret= 2;
    break;

```

All bytes that follow the start of a SysEx message are packed into integers in the same manner as other MIDI messages. Since a SysEx message is most likely longer than 3 bytes, it cannot be contained within a single integer. Subsequent SysEx bytes are stored in integers, where the status byte is set to represent a SysEx message.

Listing 7.8 shows subsequent data bytes of a SysEx messages being received and packaged into a MIDI integer. The union of the integer with an array allows for each byte of the integer to be address individually, simplifying the packaging of MIDI bytes.

Listing 7.8: MIDI Handling: SysEx Continued

```

packet.data.byte[0] = 0xF0; //Is still SysEx
cIn:> packet.data.byte[1]; //Read next byte
if (packet.data.byte[1] == 0xFF) //Check for SysEx end
{
    ret= 1;
    packet.data.byte[2]=0;
    packet.data.byte[3]=2;
    isSysEx = 0; //No longer SysEx
    break;
}
cIn:> packet.data.byte[2]; //Read next byte
if (packet.data.byte[2] == 0xFF) //Check for SysEx end
{
    isSysEx = 0; //No longer SysEx
}
packet.data.byte[3]=3;
ret= 2;
break;

```

A SysEx status byte is included in the integer until a SysEx end byte is received. At this point the function disables the SysEx flag. It should be noted that the systems SysEx handing was tested programmatically during testing, but due to the nature of SysEx being a container for manufacturer or device specific messages, it was not tested with hardware¹.

7.3.2 Producing MIDI output

The task of outputting MIDI messages, the MIDIOut task, comprises of two sub tasks: producing MIDI bytes from the MIDI integer, and serially sending these bytes out of the MIDI output of the Audio Slice. Figure 7.8 shows the implementation model for the MIDI Output task, which is implemented as the MIDIOut function shown in Listing 7.9.

¹SysEx testing is addressed further in Chapter 9.

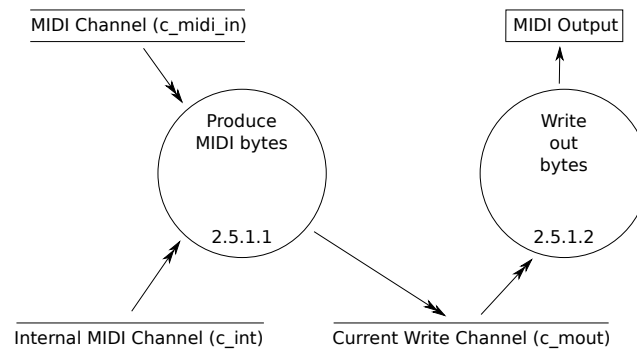


Figure 7.8: MIDI Output task: Overview of the parallel tasks

The sub task that produces the MIDI bytes (Task 2.5.1.1 in Figure 7.8), receives integers from either the MIDI Channel or from the Internal MIDI Channel. The bytes are then passed via the Current Write Channel to the task handling the writing out of bytes (Task 2.5.1.2 in Figure 7.8). The MIDIOut task runs in parallel to the MIDIIn task within the MIDI Handler, allowing for MIDI messages to be sent and received simultaneously.

The MIDIOut task produces MIDI messages that are either received from the ConMan Core or the MIDIIn (when loopback is enabled), sending them out the MIDI output port of the Audio Slice. This is handled by two parallel functions within the MIDIOut task shown in Listing 7.9.

Listing 7.9: MIDI Output Handler

```

void MIDIOut(streaming_chanend c_int, streaming_chanend c_midi_in)
{
    streaming_chan c_mout; //Channel for MIDI bytes
    sk_midi_port_init_tx(g_MidiUartOutPin); //Set up MIDI output pins
    par{
        sk_midi_port_tx(g_MidiUartOutPin, c_mout); //Handle MIDI output
        produce(c_mout, c_int, c_midi_in); //Break message into bytes
    };
}

```

The MIDIOut task first initialises the MIDI output port, which is attached to pin 7 of the 8 bit port 8D of the XS1 microcontroller. As the port is 8 bits wide, only the most significant bit of the port needs to be addressed in order to access the MIDI output port. Thus, the port has been set up using the `g_MidiUartOutPin` pin definition which directs data to the most significant bit of the port.

Listing 7.10 shows the produce function, which is tasked with the production of MIDI messages. The function is the implementation of Task 2.5.1.1 in Figure 7.8. The produce function calls the `toChannel` function to facilitate the channel transmission based on either network or internal MIDI messages.

Listing 7.10: MIDI Produce function

```

void produce(streaming_chanend c_mout, streaming_chanend c_int, streaming_chanend c_midi)
{
    MidiRawPacket_T packet;
    unsigned char len=0;
    char volume7=127;
    unsigned timeout;
    timer_tmr;
    while(1)
    {
        select
        {
            case c_midi :> packet.data.midint: //Message routed from ConMan
            {
                toChannel(c_mout, c_midi, packet);
            }
            break;
            case c_int :> packet.data.midint: //Internal loopback
            {
                toChannel(c_mout, c_int, packet);
            }
            break;
        }
    }
}

```

The `toChannel` function shown in Listing 7.11 unpacks the MIDI bytes received from either the internal MIDI channel used for loopback, or the MIDI channel for network transmission. The function then sends the bytes to the current write channel in order to be output to the MIDI output port by the `sk_midi_port_tx` function.

Listing 7.11: Channel producer for MIDI

```

void toChannel(streaming_chanend c_midi_out, streaming_chanend c_midi_in, MidiRawPacket_T &current, )
{
    if (current.data.byte[3]<2) return;
    if (current.data.byte[3]>3) return;
    if (current.data.byte[0] != 0xF0) //If not SysEx
    {
        c_midi_out <: current.data.byte[0]; //Send status
        c_midi_out <: current.data.byte[1]; //Send data
        if (current.data.byte[3] == 3) c_midi_out <: current.data.byte[2]; //Send data
    }
    else //If SysEx
    {
        c_midi_out <: current.data.byte[0]; //Send SysEx Start
        while(1) //Until SysEx end
        {
            c_midi_out <: current.data.byte[1];
            if (current.data.byte[1] == 0xFF) break; //Break at end
            if (current.data.byte[3] == 3)
            {
                c_midi_out <: current.data.byte[2];
                if (current.data.byte[2] == 0xFF) break; //Break at end
            }
            else c_midi_in :> current.data.midint; //Get more bytes
        }
    }
}

```

Listing 7.12 provides the implementation of the Task 2.5.1.2 in Figure 7.8 of the Implementation Model. The `sk_midi_port_tx` function shown in Listing 7.12 outputs each MIDI byte separately to the MIDI output port of the Audio Slice.

Before transmitting a byte, the `sk_midi_port_tx` function pulls the MIDI output line low, signalling the start of a new MIDI byte. The byte is then sent one bit at a time from, from least significant, through to most significant, before setting the line high again to signal the end of the byte.

Listing 7.12: Transmitter: MIDI Output Port handling

```
void sk_midi_port_tx(SkMidiUartOutPinDef.T &dOut, streaming_chanend cIn){
    int t;
    unsigned char b;
    unsigned char vLo=dOut.mLo;
    unsigned char vHi=dOut.mHi;

    unsigned char v;

    int clocks=dOut.mClkDiv;
    while (1)
    {
        cIn :> b; //Get byte
        if (dOut.mEnabledOut){
            dOut.mPortOut <: vLo @ t; //Set line low: Start bit
            t += clocks; //Clock
            for(int i = 0; i < 8; i++) { //Send MIDI byte
                v=(b&1)?vHi:vLo; //If 1, set high, if 0 set low
                dOut.mPortOut @ t <: v;
                b>>=1;
                t += clocks; //Clock
            }
            dOut.mPortOut @ t <: vHi; //Set line high: Stop bit
            t += clocks; //Clock
        }
    }
}
```

7.4 ConMan Core

Although the XMOs microcontrollers provide sufficient speed and allow for deterministic execution of functions, the Connection Server that houses the MIDINet module poses some concerns for the real-time nature of the system.

The original design of the MIDINet system used lists of objects (MIDINet Units) to store information regarding the units in the network. When MIDI messages were required to be routed, the receiving system would search the list of running units, checking whether a connection had been created between it and the sender. This search meant that there was no method by which the length of the search could be predicted, other than by an estimated worst case scenario based on the number of MIDINet units within the list.

This meant that the system was non-deterministic and latency could not be guaranteed. The XMOs microcontrollers do allow for deterministic execution, and therefore the loss of this determinism was seen to be suboptimal and a more time sensitive method of routing was devised though the use of integer comparison of sender IDs and routing masks.

The primary task of routing MIDI messages, as discussed in the Chapter 6, involves routing MIDI messages from MIDI ports to the network and routing MIDI messages received from the network to local MIDI ports. This routing is based on connections created by the Connection Server using the MIDINet protocol.

The tasks involved in routing MIDI messages have different requirements from an efficiency standpoint, and also present different needs in terms of their access to the rest of the ConMan system.

The task of routing MIDI messages requires deterministic execution with minimal processing occurring during the routing of messages from the MIDI input to the network. When receiving a MIDI message from the network, the check for whether a MIDI message should be output is also required to be deterministic, to ensure low latencies.

Although updating a routing mask is a critical task, it is not necessarily time-sensitive. The management of connections using the MIDINet module requires the creation of connections within the object-oriented system. Once a connection is created, a routing mask can be created and sent to the routing task. These tasks are not time-sensitive, but should not interfere with the deterministic operation of the routing tasks.

For this reason the routing of MIDI has been implemented in two stages. The first stage looked at the real time aspect of the MIDI message routings, and methods of improving the latency for networked MIDI transmissions. The second stage involved the integration of a system that could create efficient routing masks. This allows for the real-time MIDI routing side of the system to be decoupled from the non real-time object oriented components of the system.

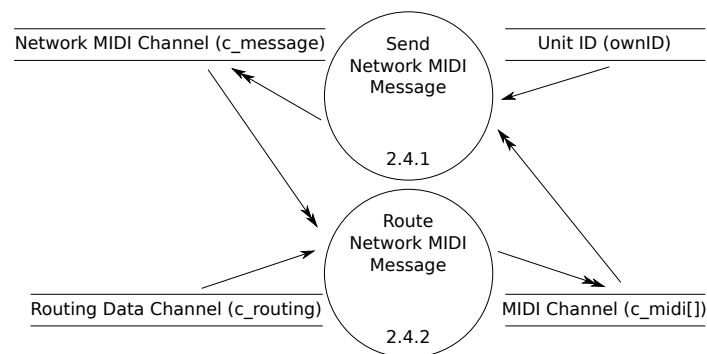


Figure 7.9: ConMan Core - the real-time routing of MIDI messages

Figure 7.9 shows the Implementation model for the real-time routing of MIDI messages termed the 'ConMan Core'. The diagram shows the two parallel tasks that deal with the

routing of messages between the network and the MIDI handler.

Task 2.4.1 shown in Figure 7.9 receives MIDI messages from the MIDI Handler via the MIDI Channel. Each of these messages is received as an integer containing the MIDI bytes. Once received, a bit position representation of the Unit's ID is sent to the Network Handler via the Network MIDI Channel, followed by the MIDI integer. The implementation of this task is detailed in Section 7.4.2.

The task of routing MIDI messages that have been received from the network is handled by Task 2.4.2 (shown in Figure 7.9). The task receives two integers from the Network handler via the Network MIDI Channel. The first integer contains the sending unit's bit position ID, which is used to route the MIDI integer that follows. The sender's ID is then compared to the routing mask. If sender's ID is present within the routing mask, the MIDI integer is sent via the MIDI Channel to be output by the MIDI handler. The implementation of Task 2.4.2 is detailed in Section 7.4.3.

Tasks 2.4.1 and 2.4.2 are implemented as shown in Listing 7.13 as the ConMan Core. The ConMan Core uses 2 sets of streaming channels for MIDI messaging, and uses another channel for updating routing information. The `c_routing` channel is used for connection management, and allows the Connection Server to maintain the ConMan Core's routing table by providing an updated routing mask whenever connections are created or destroyed. The implementation of this is described in Section 7.4.1.

MIDI messages sent to or received from the MIDI Handler are communicated via the `c_midi` array of channels. The array contains two streaming channels, which allow for simultaneous sending and receiving of MIDI messages to and from the ConMan Core. The `c_message` array of channels provides the Network Handler with similar connectivity, enabling the simultaneous sending and receiving of MIDI to and from the network handler.

Listing 7.13: ConMan Core function

```
void conman_core(chanend c_routing, streaming chanend c_midi[], streaming chanend c_message[])
{
    unsigned int RouteMask = 0;
    unsigned int SenderID = 0;
    unsigned int tmp=0;

    c_routing := unsigned int ownID; //Local ID from MIDINet

    while(1)
    {
        select
        {
            //Task 2.4.1
            case c_midi[1] :=> tmp: //When MIDI integer received from MIDI handler
                c_message[1] <: ownID //Send local ID to network
                c_message[1] <: tmp; //Send MIDI integer to network
                break;
        }
    }
}
```

```

//Task 2.4.2
case c.message[0] :> SenderID: //When ID received from network
    c.message[0] :> tmp; //Store MIDI integer
    if ((RouteMask & SenderID) == SenderID) //If ID set in routing mask
        c.midi[0] <: tmp; //Send MIDI integer to MIDI Handler
    break;
case c.routing[1] :> tmp: //If MIDINet sends update
    RouteMask = tmp; //Store routing mask
    break;
default:
    break;
}
}
}

```

7.4.1 IDs and Routing masks

The design introduced in Chapter 6 makes use of routing masks to ensure low latency routing of MIDI messages. These routing masks are compared to a bit position representation of the unit's ID when used for routing, as described in Section 7.4.3.

On start-up of a MIDINet system, the MIDINet module performs discovery to establish a record of the other MIDINet systems on the network. Once this discovery is complete, the unit's MIDINet ID is transformed into a bit position representation by the MIDINet module.

This bit position representation of the ID is then passed to the ConMan Core via the `c.routing` channel. This is the ID that will precede all MIDI messages that are sent from the ConMan Core to the network. This ID does not change, and is therefore only set once, on start-up. The ID is stored in an unsigned 32 bit integer, the same size as the routing mask. The details of this design are discussed in Section 6.4.3.1, and the implementation within the connection management task explained within Section 7.7.2.

To reduce the amount of processing done within the ConMan Core, the creation of the routing mask is handled by the MIDINet module. When a connection is established within the MIDINet system, a routing mask that represents the current connections to the local unit is created, as described in Chapter 6.

The routing mask is then sent to the ConMan Core where the new mask replaces the existing mask. This ensures that the state of the MIDINet system will match that of the routing mask. The routing mask is passed over the `c.routing` channel.

7.4.2 MIDI to Network routing

The implementation model shown in Figure 7.9 shows the tasks that comprise the ConMan Core. Task 2.4.1 in the diagram handles the routing of MIDI messages to the Network Handler.

When the ConMan Core receives a MIDI message from the MIDI handler via the `c_midi` channel array, the message is already packed into a 32 bit integer. The case statement shown in Listing 7.14 shows the events that the ConMan Core handles when routing a MIDI message to the network.

When transmission is received from the `c_midi[1]` channel, the ConMan Core stores this message to be routed. For other units to be capable of routing the message, the ID of the unit that sends the MIDI must be included within the MIDINet packet.

Listing 7.14: Routing: MIDI Input port to Network

```
case c_midi[1] :> tmp:
  c_message[1] <: ownID //Send local ID to network
  c_message[1] <: tmp; //Send MIDI integer to network
  break;
```

It would be time-consuming to create a bit position representation of the unit's ID whenever a MIDI message needs to be sent, so instead this is done during set up. After the MIDINet module has finished discovery, the ID is received by the ConMan core as a bit shifted ID in the form of an integer. When a MIDI message is received, the bit shifted ID is sent over channel `c_message[1]` to the real-time networking tasks, followed by the integer containing the MIDI message. The creation of the bit position ID is described in Section 7.7.3 where the implementation of the connection management components is described.

7.4.3 Network to MIDI routing

The task of routing MIDI messages that have been received from the Network to the MIDI Handler is shown in Figure 7.9 as Task 2.4.2.

After a packet has been handled by the networking component and the MIDINet payload removed, the routing and MIDI information is sent to the ConMan Core via the `c_message[0]` channel as two integers to be routed. As shown in Listing 7.15, the case performs a bitwise comparison between the received ID and the routing mask.

Listing 7.15: Routing: Network to MIDI Output port

```

case c.message[0] :> SenderID: //When ID received from network
  c.message[0] :> tmp; //Store MIDI integer
  if ((RouteMask & SenderID) == SenderID) //If ID set in routing mask
    c_midi[0] <: tmp; //Send MIDI integer to MIDI Handler
  break;
\label{listing-Network to MIDI}

```

If the routing mask has a set bit matching that of the received ID, then the integer containing the MIDI message is passed to the MIDI Handler over channel `c_midi[0]` to be output.

The comparison and bitwise operations that are performed as part of the routing process aid in ensuring that the ConMan Core inflicts the lowest possible latency on the MIDI messages being routed.

7.5 Networking

As discussed in Section 7.2.1, the networking tasks handled by the ConMan device have been separated into two separate sets of tasks: Real-time and non-real-time networking tasks.

As shown in Figure 7.1, the ConMan system makes use of 2 processors: A network processor and a ConMan processor. The network stack is provided through XMOS's XTCP library which runs on the Network Processor along with the tasks associated with managing the non-real time network.

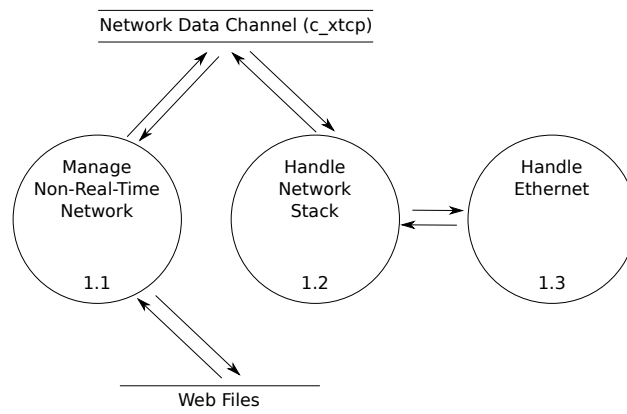


Figure 7.10: Network Processor: Overview of the parallel task structure

Figure 7.10 shows the implementation model for the Network Processor. The network stack (Task 1.2) and Ethernet hardware (Task 1.3) handlers are provided by XMOS as part

of the XTCP library, and required only configuration, with no additional implementation necessary.

7.5.1 Non-real-time Networking

The implementation model shown in Figure 7.10 depicts the handler for the non-real-time networking tasks. The Embedded Webserver Function library was introduced in Chapter 3. The library allows for a web server to be run on the XMOS microcontrollers, making use of the XTCP library to provide the networking connectivity required for the web server to function.

Listing 7.16: TCP Handler function

```
void tcp_handler(chanend c_xtcp, chanend c_control)
{
    xtcp_connection_t web_conn; //current web connection

    web_server_init(c_xtcp, null, null); //start embedded webserver
    init_web_state(c_control); //set up instruction channel

    //Task 1.1 - Non-real-time networking
    while (1) {
        select {
            //Handle TCP events
            case xtcp_event(c_xtcp, web_conn):
                web_server_handle_event(c_xtcp, null, null, web_conn); //Handle web request
                break;
        }
    }
}
```

The non-real-time networking task (Task 1.1 in Figure 7.10) was implemented as the TCP Handler shown in Listing 7.16. The TCP Handler starts the web server and then forwards all web requests received from the network to the Embedded Webserver to be handled. The handling of web requests is discussed further in Section 7.6.

7.5.2 Real-time Networking

The task of handling real-time networking does not run on the Networking Processor, but runs on the ConMan Processor as shown in Figure 7.3. The primary goal of this task is to transfer data between the network and the ConMan Core as fast as possible.

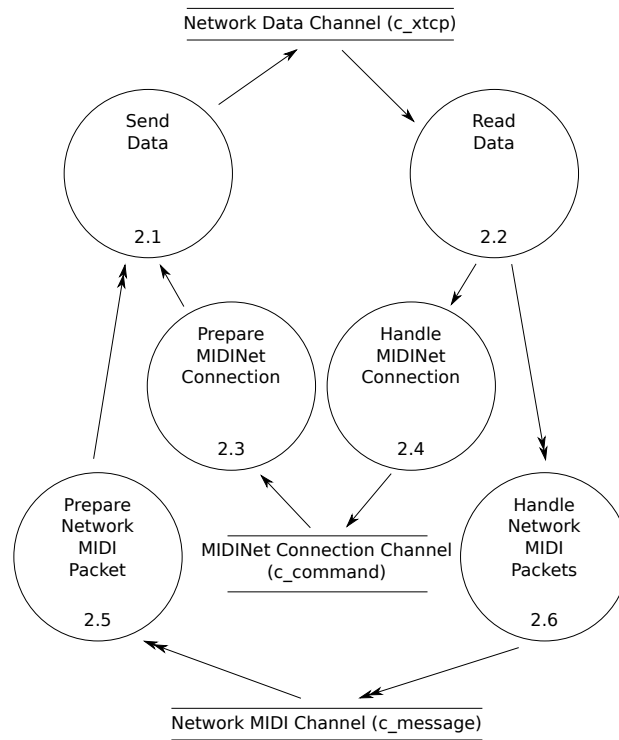


Figure 7.11: Manage Real-Time Network: Overview of the parallel tasks

Figure 7.11 shows the tasks within the Implementation Model of the Real-Time Networking task. As shown in the figure there are only 3 channels that are used for communication, with the common channel being the Network Data Channel.

This allows the tasks associated with handling the real-time network to be bisected and grouped into MIDI tasks and MIDINet tasks. The MIDI tasks require the lowest possible latency, with the MIDINet messages being recognised as less time-sensitive.

The Implementation Model for these real-time networking tasks led to the implementation of the UDP Handler. The event switching design for the UDP handler was taken from XMOS's UDP reflector demo project [73]. Some modifications were needed to make the switcher suit the use case.

7.5.2.1 MIDI Communication

The group of tasks that are associated with MIDI messages are concerned with the communication that occurs between the ConMan Core and the network. Figure 7.12 shows the Implementation model for the group of tasks that are used for MIDI communication. Tasks 2.1 and 2.5 handle the transmission of MIDI messages to the network, while Task

2.2 and 2.6 receive messages from the network and pass them to the ConMan Core.

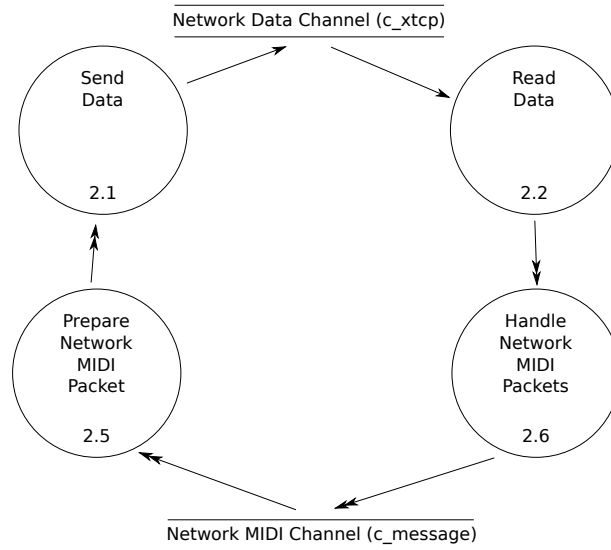


Figure 7.12: Manage Real-Time Network: Tasks related to MIDI communication

Sending MIDI to the network requires packing the local unit's ID and the MIDI message into a MIDINet packet. To identify that the packet contains a MIDI message, the MIDI code is stored in the transmission buffer. Listing 7.17 shows the ID and MIDI messages being copied into the transmission buffer after they have been received from the ConMan Core.

Listing 7.17: UDP Handler: Transmit MIDI

```

tx_buffer[1] = MIDIMSG; //MIDINet code: MIDI

memcpy(tx_buffer+2, uROUTE.array, sizeof(uROUTE.array)); //Local ID
memcpy(tx_buffer+6, uMIDI.array, sizeof(uMIDI.array)); //MIDI integer

if (broadcast_connection.id != INIT.VAL && !broadcast_send_flag)
{
    broadcast_len = 10;
    xtcp_init_send(c_xtcp, broadcast_connection); //Send UDP
    broadcast_send_flag = TRUE;
}

```

The ID and the MIDI message are stored in unions that allow for their data to easily be copied into the transmission buffer. The `memcpy`² function is used to populate the transmission buffer, which will contain the MIDINet code, sender ID and MIDI message. Once the buffer is ready, it is sent to the network.

As SysEx messages are received by the real-time networking components as a set of individual integers each containing a SysEx status, they are handled separately to other

²The `memcpy` function copies a block of memory. It is used to copy a source to a destination, based on the block size (in bytes) requested.

MIDI messages. As shown in Listing 7.18, each integer is unpacked and stored within a SysEx message buffer. Once the whole SysEx message has been received, the buffer is copied to the transmission buffer to be sent to the network.

Listing 7.18: UDP Handler: Transmit SysEx

```

if(uMIDI.array[0] == SYSEX) //MIDINet code: SysEx
{
    if(tx_sysexcount == 0)
    {
        tx_sysex[tx_sysexcount++] = uMIDI.array[0]; //MIDINet code: SysEx
    }
    for(int i=1; i<=(uMIDI.array[3]); i++)
    {
        if (uMIDI.array[i] != 0xFF) tx_sysex[sysexcount++] = uMIDI.array[i]; //Package
            SysEx data
        else
        {
            tx_sysex[sysexcount] = uMIDI.array[i]; //Package SysEx end
            tx_buffer[1] = SYSEX; //MIDINet code: SysEx
            memcpy(tx_buffer+2, uROUTE.array, sizeof(uROUTE.array)); //Local ID
            memcpy(tx_buffer+6, tx_sysex, tx_sysexcount); //SysEx message

            if (broadcast_connection.id != INIT_VAL && !broadcast_send_flag)
            {
                broadcast_len = 10;
                xtcp_init_send(c_xtcp, broadcast_connection); //Send UDP
                broadcast_send_flag = TRUE;
            }

            sysexcount = 0;
            break;
        }
    }
}

```

Receiving from the network is handled by the XTCP server. When the XTCP server states that there has been an XTCP_RECV_DATA event, the UDP handler reads the packet so that it can be unpacked and sent to the ConMan Core for routing.

Listing 7.19 shows the actions taken when a UDP packet is received. If the MIDINet code categorises the packet as a MIDI message, the sender ID and the MIDI message are sent to the ConMan Core to be routed.

Listing 7.19: UDP Handler: Handler received MIDI

```

case MIDIMSG: //MIDINet code: MIDI

    memcpy(iuROUTE.array, rx_buffer+2, sizeof(iuROUTE.array)); //Remote ID
    memcpy(iuMIDI.array, rx_buffer+6, sizeof(iuMIDI.array)); //MIDI integer

    //Send to ConMan Core
    c.message[0] <: iuROUTE.route;
    c.message[0] <: iuMIDI.midi;

    break;

```

SysEx messages received from the network are sent to be routed as a set of integers with SysEx status, as described in Section 7.3.1. To ensure that the entire SysEx message is correctly routed, each MIDI integer is sent with a preceding sender ID as shown in Listing

7.20. This allows the ConMan Core to operate using the same efficient routing method for both MIDI and SysEx messages.

Listing 7.20: UDP Handler: Handler received SysEx

```

case SYSEXMSG: //MIDINet code: SysEx

    memcpy(iuROUTE.array , rx_buffer+2, sizeof(iuROUTE.array)); //Remote ID

    for (int i = 6; i < len; i+= 4) //Break SysEx into bytes
    {
        memcpy(iuMIDI.array , rx_buffer+i, sizeof(iuMIDI.array)); //SysEx message

        //Send to ConMan Core
        c.message[0] <: iuROUTE.route;
        c.message[0] <: iuMIDI.midi;
    }

    break;

```

7.5.2.2 MIDINet communication

In the original MIDINet application, all network tasks were handled together within the Network class. The ConMan system differs from this approach, as the routing and connection management tasks are separate.

The Implementation model in Figure 7.13 shows the tasks associated with sending and receiving MIDINet's connection management data. The tasks pass data received from the network to the tasks handling the MIDINet Module (discussed in Chapter 6). When new connections are created through the web application, they are sent to the network using the MIDINet message format.

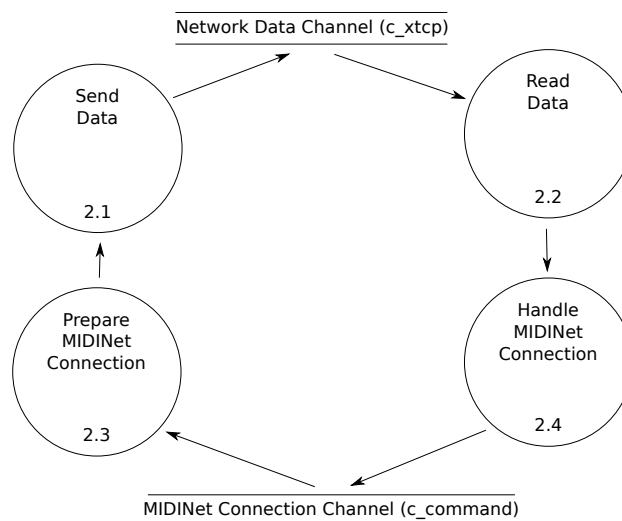


Figure 7.13: Manage Real-Time Network: Tasks related to MIDINet commands

Tasks 2.1 and 2.3 in Figure 7.13 send MIDINet connection messages to the network so that other ConMan devices can update their connection server information. Task 2.2 and 2.4 are responsible for taking packets received from the network, and forwarding the data to the Connection server as they contain MIDINet information.

Task 2.3 of the Implementation model shown in Figure 7.13 is implemented as shown in Listing 7.21. MIDINet commands are received via the `c_command` channel. The data is written into the transmission buffer and then sent to the network.

Listing 7.21: UDP Handler: Transmit MIDINet command

```
case c_command :> response_len: //Number of bytes inbound from MIDINet module
    tx_buffer[0] = response_len;
    for (int i=1; i <= response_len; i++)
    {
        c_command :> tx_buffer[i]; //Populate transmission buffer
    }

    if (broadcast_connection.id != INIT.VAL && !broadcast_send_flag) {
        broadcast_len = response_len+1;
        xtcp_init_send(c_xtcp, broadcast_connection); //Send UDP
        broadcast_send_flag = TRUE;
    }
    break;
```

The networking handler is not required to know the type of MIDINet message that is being sent. The MIDINet Code, and relevant data are sent from the connection management task and packed into the transmission buffer directly.

Listing 7.22: UDP Handler: Handle received MIDINet command

```
default: //If not MIDI or SysEx
    for (int i=1; i <= len; i++)
    {
        c_command <: rx_buffer[i]; //Send to MIDINet module
    }
    break;
```

Listing 7.22 shows the default handler for UDP messages. This handler is used as the implementation of Task 2.4 of the Implementation model shown in Figure 7.13. If the received message is not identified as a MIDI message or SysEx message, the buffer of received data is sent over channel `c_command` to the connection management task.

7.6 Web Application

Figure 7.14 shows a portion of the Implementation model, where the channel interaction of the tasks that handles web requests is shown. The task receives web requests, and responds with the appropriate web page. In Figure 7.14, handling web requests is shown as a single task, but the implementation of the web application actually consists of four

main components: the web server, web page functions, application handler and web interface.

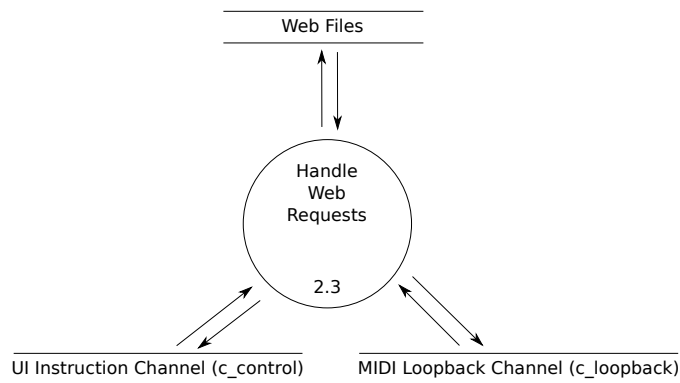


Figure 7.14: Web Application: Handling Web Requests

XMOS provides an Embedded Webserver module that is used to serve users web pages. This enabled the development of a graphical user interface for the ConMan system. An Application handler was implemented to allow the web interface to communicate with the connection management tasks through the UI Instruction Channel shown in Figure 7.14.

Each of the four components that comprise the Web Application are discussed in the following subsections.

7.6.1 Web Based Connection Management Interface

The interface allows the user to interact with the ConMan system from a web browser, removing the need for a separate ConMan application. Within this section, the use of the words 'ports' and 'connections' refer to the MIDI input and MIDI output ports of ConMan units and current connections as provided by the ConMan system.

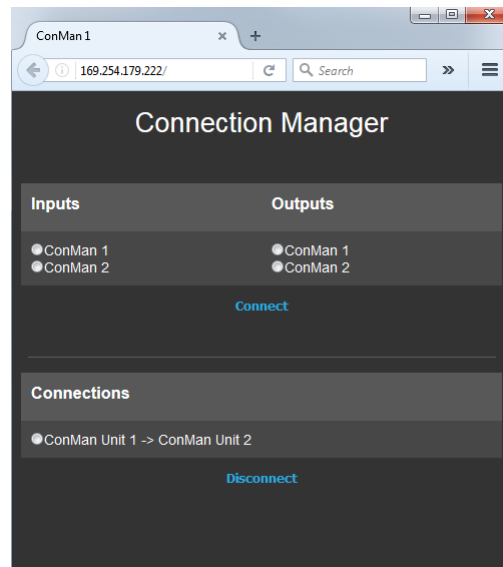


Figure 7.15: Screenshot of connection management web page.

The interface, as shown in Figure 7.15, provides a list of all the MIDI input ports as well as a list of all the output ports. There is a “Connect” button that allows the user to create a connection between a selected input and selected output. The interface also displays a list of current connections. A “Disconnect” button allows the user to disconnect the selected connection from the list.

7.6.2 Embedded Web Server

XMOS provides an embedded web server module that allows for their microcontrollers to host fully functioning websites [69]. The embedded web server module allows for the device to respond to page requests by returning the requested web page. This can be done by building the page line by line, or by passing a web page that has already been created.

The embedded web server module allows for HTML web pages to be stored within the source code prior to compilation. By doing this developers are able to create web pages and have them served by the Embedded Web Server at runtime.

7.6.2.1 Dynamically built pages

HTML stored within the source will only provide a static web page, having no connection to the running system or its current state. These static pages cannot be used to monitor

or control the system, only containing information that was stored at compilation.

To increase the usability of the web server, XMOS makes use of a system that allows for pages to include information about the current state of the system. This is achieved through the use of an XMOS specific notation that is included in the web page's HTML prior to compilation.

Listing 7.23: Example of static HTML in source code

```
<p>
    {% createHTML( buffer ) %}
</p>
```

In Listing 7.23, the `{%...%}` syntax, hereafter referred to as dynamic content tags, marks the code to be evaluated before the rendering will continue. The function placed within the dynamic content tags will be called, and executed.

The XMOS Embedded Webserver Library Programming Guide lists 3 different variables that can be used to return data to the web page. These include a character buffer that will replace the tags, an integer representing the connection state and an integer representing the application state.

Listing 7.24: Example of function within web server component

```
int createHTML(char buffer[])
{
    char msg[] = "<p>Example_HTML_for_a_paragraph.</p>"; //HTML to replace tag
    strcpy(buffer, msg);
    return sizeof(msg)-1;
}
```

An example function is shown in Listing 7.24. The function creates a string of HTML that is then returned to the web server, and inserted in place of the original dynamic content tags. In this function, this is done by writing to an array of characters and returning the length of the string to be written.

7.6.3 Web Page Functions

The web page 'functions' component gathers and wraps the information from the system in HTML so that it can be utilised in the population of web pages. The component allows for web pages that contain dynamic content tags to call associated functions, which enable web-based control of the ConMan system.

The implementation of the web page functions includes routines used to retrieve the local unit's ID, local unit name, number of remote units, remote unit IDs, remote unit names

and number of active connections. Functions were also implemented to create the HTML for the radio groups used to list the units and connections. These functions are used in the construction of the web based connection management application.

Listing 7.25: Example of dynamic content tag for HTML title

```
<head>
    <title>ConMan {% get_local_unit_ID(buffer) %}</title>
</head>
```

The code in Listing 7.26 demonstrates how a buffer will be used to replace the dynamic content tags containing the `get_local_unit_ID` function shown in Listing 7.25. The function retrieves the local unit's ID from the MIDINet module, and stores it within the Application Handler (discussed further in Section 7.6.4). The ID is then written into the buffer so that when returned the web server will replace the dynamic content tags with the buffer.

Listing 7.26: Example of function within the web server component

```
int get_local_unit_ID(char buf[], int app_state, int connection_state)
{
    chanend c_control = (chanend) ((app_state_t *) app_state)->c_control;
    get_ID(c_control, &((app_state_t *) app_state)->connection_data);
    unsigned int i = ((app_state_t *) app_state)->connection_data.ID;
    char selstr[2] = {i+OFFSET, '\0'}; //ID for Web Page
    strcpy(buf, selstr);
    return strlen(selstr);
}
```

When a user requests the connection management web page from a ConMan device, the Embedded Webserver will process the static HTML, replacing dynamic content tags with the output from the web page functions. A web page requested from the ConMan device with an ID of '2', will produce the HTML shown in Listing 7.27. The result of the `get_local_unit_ID` function shown in Listing 7.26 has replaced the dynamic content tags and provided the web browser with a web page specific to that ConMan device.

Listing 7.27: Example of dynamic content tag for HTML title

```
<head>
    <title>ConMan 2</title>
</head>
```

This and other web page functions operate to provide HTML that replaces the dynamic content tags within the static HTML web template. These functions communicate with the Application Handler to retrieve information from the ConMan system, so that it may be displayed via web pages.

The web page functions are written in C and do not natively make use of channel communication. The Application Handler is implemented in xC and makes use of channels to communicate with other tasks in the ConMan system. Although channels can be passed

into C functions through the inclusion of an xC compatibility header (xccompat.h), the channel communication still occurs within the xC functions.

This allows for the information to be retrieved using channels, and passed to the web page functions to be used in the creation of the HTML web page.

7.6.4 Application Handler

The Embedded Web Server Function library makes use of an Application Handler that is used to track the state of the web application and allows for the web application to hold information that will be used within the web pages. The Application Handler is a component implemented in xC, and handles the channel communications necessary to interact with other components in the system.

As an example, the `get_local_unit_ID` function described in Section 7.6.3 is responsible for getting the ID of the local unit, and returning the ID to be included in the HTML. The function makes use of the `get_ID` function shown in Listing 7.28. The `get_ID` function sends a function notification to the application event handler³ via the `c_control` channel. The application event handler will return the ID via the `c_control` channel for it to be stored.

Listing 7.28: Web Application: Get ID

```
void get_ID(chanend c_control, connection_state_t &data)
{
    c_control <: APP_HANDLER_GET_ID; //Request ID from App Handler
    c_control :> data.ID; //ID from App Handler
}
```

When the application event handler receives a notification from the `get_ID` function, a request is made to the connection server via the `c_midinet` channel. Once the connection server has returned the ID, the ID is passed to the `get_ID` function to be stored.

Listing 7.29: Application Event Handler: Request ID from MIDINet module

```
case APP_HANDLER_GET_ID: //ID Request
{
    unsigned int tmp;
    c_midinet <: LOCALID; //Request ID from MIDINet Module
    c_midinet :> tmp; //ID from MIDINet Module
    c_control <: tmp; //Send ID to Web Page functions
}
```

The connection server (Discussed in Section 7.7.2) implements a request handler that returns the requested information from the MIDINet Module.

³Application event handler is a case based handler that reacts to channel events. The handler is part of the Application Handler allows for information to be easily sent and received from the MIDINet Module.

Listing 7.30: MIDINet Module: Provide ID

```

case c_midinet :> request: //Requests from App Handler
  switch (request)
  {
    case LOCALID: //Request for ID
      c_midinet <: GetLocalMNUID(); //Send ID to App Handler
      break;
    // Other requests
  }

```

Thus the Application Handler enables requests for information, and provides an event handler that can redirect requests and information to the correct components, for example the Connection Server.

7.7 Connection Management

The implementation model for the tasks responsible for connection management is shown in Figure 7.16. The model shows the interaction of the tasks with the communication channels and the MIDINet module.

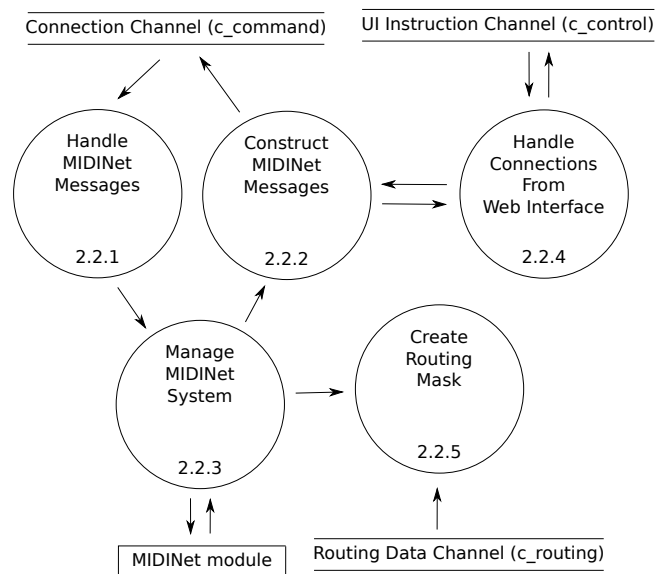


Figure 7.16: Connection Server: Overview of the parallel tasks

The Implementation model shows the various tasks required for connection management. While the whole component is concerned with handling connection management, the MIDINet module ultimately maintains the list of units and current connections.

Thus the connection management component is implemented as two parts: The MIDINet module and the Connection Handler. The MIDINet module features the flexible object-

oriented design of the original MIDINet system, while the Connection Handler provides the functionality necessary for the the components of the ConMan system to communicate with the MIDINet module.

7.7.1 MIDINet Module

The MIDINet system needed to be separated into its various components. The system had been designed to be run on computers equipped with displays, network interface cards and MIDI interfaces. This meant that although the core connection management was required in the ConMan system, other components were not.

The original MIDINet system had been implemented using Microsoft's Foundation classes for C++. The issue with this was that many of the objects used in the system were based on data structures that do not exist in the standard C++ language. This entailed rewriting the classes to work with all C++ compilers, using only standard data types.

The MIDINet system was thus reimplemented as a module (introduced in Section 6) that could be used within the real-time XMOS system to provide flexible connection management.

The MIDINet Module no longer contains a network class, as the ConMan system already contains suitable networking capabilities. For this reason, tasks originally implemented within the networking class were moved and reimplemented in the Connection Handler.

7.7.2 Connection Handler

The Connection Handler issues commands to the MIDINet module, and also relays commands from MIDINet to other parts of the system. This component handles the role of Task 2.2.3 in Figure 7.16.

The Connection Handler also performs initial MIDINet module set-up, discovery and event handling. Since the MIDINet module is implemented in C++, it is not capable of channel communication⁴. To enable the module to send information such as unit names over channels, a function called `SendData` was implemented in the Connection Handler.

⁴More detail on the use of C and C++ on XMOS devices can be found in Appendix A.

The SendData function is shown in Listing 7.31. The function has as parameters:

- a channel over which the information will be sent
- an integer for length of the information being sent
- an integer for the memory location

The function allows the MIDINet module to store data at a location in memory, and then request that the Connection Handler's SendData function performs the channel communication. This allows for C and C++ functions to use the SendData function to send information over channels, using a known channel communication format.

Listing 7.31: SendData function

```
void SendData(chanend output, int Length, unsigned message)
{
    unsigned char len = Length;
    output <: len;
    char buffer[32];
    unsafe
    {
        char * unsafe prcv = (void*) message; //Memory start
        memcpy(buffer, prcv, Length*sizeof(char)); //Buffer block
    }
    for (int i = 0; i < Length; i++)
    {
        output <: (unsigned char) buffer[i]; //Send over channel
    }
}
```

To make use of the SendData function the MIDINet module requires a 'compatibility header' so that channels can be declared. The SendData function is a generic function that is used for sending blocks of data, and is used throughout the MIDINet module. Functions within the MIDINet module are then able to populate arrays with any information that needs to be sent to the network during discovery or to the web server to produce the user interface. The original MIDINet system placed such data in a datagram to be sent to the network. The excerpt of code in Listing 7.32 shows an array that contains the local unit information required for discovery being transmitted via the SendData function.

Listing 7.32: Memory block transmission

```
int NameLength = Name.length();
unsigned char * pmsg = &MNUnitMsg[0]; //Start of array
unsigned x;
asm("mov_%0,%1" : "=r"(x) : "r"(pmsg)); //Array start in memory
SendData(c, NameLength+2, x); //Send block over channel
```

Task 2.2.2 shown in Figure 7.16 involves sending MIDINet messages to the network handler, and makes use of the SendData function to accomplish this.

Task 2.2.1 of Figure 7.16 handles the MIDINet messages that have been received from the network. An event handler investigates the MIDINet code received in the MIDINet

message, and processes the code accordingly. When a MIDINet code is received by the MIDINet code handler from the Connection Channel (`c_command`), the relevant functions within the MIDINet module are called. If the MIDINet message requires a response, the appropriate response is transmitted via the Connection Channel to the network.

Listing 7.33 shows an outline of the event handler for the MIDINet module that deals with MIDINet messages. The handler deals with the tasks of discovery and connection management that formed part of the MIDINet system's networking class described in Chapter 5. The listing shows in detail the case for handling the 'Who is there' portion of discovery. In this case the event handler receives a message from a new unit on the network requesting other units and responds with an "I am here" message and the Local Unit ID.

Listing 7.33: MIDINet Code handler: Who Is There

```
switch (byte)
{
    case WHO_IS_THERE:
        c_command := unsigned char newID; //ID of new unit
        c_command <: (unsigned char) 2; //Number of bytes
        c_command <: (unsigned char) I_AM_HERE; //MIDINet code
        c_command <: (unsigned char) GetLocalMNUID(); //Local ID
        break;

    case I_AM_HERE:
        // Respond with a request for configuration.
        // ...

    case CONFIG_REQ:
        // Respond with local unit information
        // and information about other remote units
        // ...

    case MNUNIT:
        // Add: MIDINet unit information
        // ...

    case DELETEMNUNIT:
        // Remove: MIDINet unit information
        // ...

    case INPUT:
        // Add: Input port information
        // ...

    case DELETEINPUT:
        // Remove: Input port information
        // ...

    case OUTPUT:
        // Add: Output port information
        // ...

    case DELETEOUTPUT:
        // Remove: Output port information
        // ...

    case CONNECT:
        // Add connection between two ports
        // ...

    case DISCONNECT:
        // Remove connection between two port
        // ...
}
```

```
}

```

The original MIDINet system's networking class also includes handling for MIDI and SysEx messaging, but the implementation of the ConMan Core (Section 7.4) and MIDI Handler (Section 7.3) has removed the need for these MIDINet messages to be handled within the MIDINet module.

7.7.3 ID and Routing Mask Creation

When the MIDINet Module has completed discovery, the system can prepare to route MIDI messages. To do so, the ConMan Core requires the local ID of the unit in its bit shifted form as shown in Listing 7.34. The MIDINet module created the bitwise ID and sends it to the ConMan Core, allowing the system to begin routing MIDI.

Listing 7.34: MIDINet Module: Production of bitshifted ID

```
unsigned integer bID = 1 << GetLocalMNUID();
c_routing <: bID;
```

The ConMan Core makes use of a routing mask to determine if received messages should be output from the MIDI port. The mask is created by the MIDINet Module and updated when new connections are created.

The MIDINet message handler for the 'CONNECT' message is shown in Listing 7.35. When the MIDINet message handler receives a 'CONNECT' code from the network, the handler receives from the Connection Channel (c_command) the units and ports which the MIDINet module is to connect. The connection information is then passed to the NewConnection function.

Listing 7.35: MIDINet Code handler: New connection

```
case CONNECT:
    unsigned char Len;
    c_command :> Len;

    unsigned char InPortMNUNo;
    unsigned char InPortNo;
    unsigned char OutPortMNUNo;
    unsigned char OutPortNo;

    // Store connection parameters
    c_command :> InPortMNUNo;
    c_command :> InPortNo;
    c_command :> OutPortMNUNo;
    c_command :> OutPortNo;

    // Call for new connection in MIDINet module
    NewConnection(InPortMNUNo, InPortNo, OutPortMNUNo, OutPortNo);

    break;
```

When the `NewConnection` function is called, the MIDINet module adds the connection as per the MIDINet protocol (described in Chapter 5). If the connection involved a local output port, the input port in the connection message is added to the routing mask. Listing 7.36 shows the code that updates the routing mask. A bitshifted representation of the ID of the input port in the connection manager is created, and added to the routing mask through the use of a bitwise 'OR' operation. The updated routing mask is then sent to the ConMan Core.

Listing 7.36: MIDINet Module: New connection

```
void NewConnection(unsigned int InPortMNUNo, unsigned int InPortNo, unsigned int OutPortMNUNo, unsigned int OutPortNo)
{
    // Create connection
    ...
    //

    if (OutPortMNUNo == pLocalMNUnit->GetID()) //If local output
    {
        unsigned integer UnitID = 1 << InPortMNUNo; //Get bitwise ID
        routingMask |= UnitID; //Add ID to Mask
        c.routing <: routingMask; //Send Mask to ConMan Core
    }
}
```

When a connection is removed, a similar process takes place, but removing the ID of the input from the routing mask. As shown in Listing 7.37, a bitshifted ID is created, and removed from the routing mask. This is done by inverting the ID and then applying a bitwise 'AND' operation, ensuring the ID is removed from the mask while leaving all other connections unaffected.

Listing 7.37: Modification of routing mask for removal of connection

```
void Disconnect(unsigned int InPortMNUNo, unsigned int InPortNo, unsigned int OutPortMNUNo, unsigned int OutPortNo)
{
    // Delete connection
    ...
    //

    if (OutPortMNUNo == pLocalMNUnit->GetID()) //If local output
    {
        unsigned integer UnitID = 1 << ID; //Get bitwise ID
        routingMask &= ~UnitID; //Remove ID from mask
        c.routing <: routingMask; //Send Mask to ConMan Core
    }
}
```

7.8 Conclusion

The implementation of the MIDINet system included the creation of a MIDI handler to process MIDI input and output, a network handler that allows for communication with

other ConMan devices, and the ConMan Core that provides deterministic routing of MIDI messages.

The routing masks used in the ConMan Core component are created by the Connection Server: an implementation of MIDINet designed to run on the XMOS architecture. The Connection Server coupled with the Web Server provides a web application that enables the flexible connection management of ConMan devices on a network.

The successful implementation of the ConMan system alone does not guarantee that the system is capable of providing connection management and MIDI transmission. The system required testing to ascertain whether the transmission had sufficiently low latencies for time-sensitive use cases. Chapter 8 provides a methodology for testing MIDI connection management systems and compares the ConMan system to the MIDINet system.

Chapter 8

Testing and Analysis

8.1 Introduction

Solutions for MIDI connection management were introduced in Chapter 2, with a more in-depth investigation into MIDINet being conducted in Chapter 5. The ConMan system whose design is described in Chapter 6 and implementation in Chapter 7 makes use of concepts from MIDINet for its connection management, but uses optimisations made to leverage the features of the XMOS development kits described in Chapter 3.

This chapter proposes a methodology for the testing and comparison of MIDI connection management systems. The chapter describes testing of the methodology as well as the application of the methodology through a comparison of the ConMan and MIDINet systems.

8.2 Testing Methodology

In order to test the ConMan system against the MIDINet system, a methodology for testing was required. Research into the testing of MIDI based systems did not reveal suitable¹ testing methodologies or apparatus. This led to the development of a method

¹Testing methodologies such as the Midi-Wave Method introduced by Brandt and Wright [65] and has since been used in subsequent work by Thom and Nelson. Other testing methods including oscilloscopes would also have provided more accurate sampling. Such testing methodologies made use of equipment that was not available at the time of testing and was financially out of scope. More adequate testing is addressed in Section 9.8.

that would provide a set of standardised and repeatable tests to aid in the analysis of the two MIDI connection management systems.

The testing approach was to produce MIDI messages, have these propagate through the system being tested, and receive them back again. The assumption was that the difference in time between sending and receiving messages would be the latency of the system being tested.

The testing comprised the transmission of a MIDI file through the system thereby emulating a performance². The same test was conducted using the MIDINet system, and the ConMan system.

MIDI is largely used as a performance tool and the time constraints that live use impose upon a system have a substantial impact on the usability and relevance of the system in question. For this reason testing focused on core MIDI functionality, and SysEx messaging was not tested. The possible future testing of SysEx messaging is addressed in Section 9.8.

The way in which MIDI ports are implemented on a computer means that if one application is using a MIDI port, another application is not able to make use of that same port. Thus, a MIDI player that is producing MIDI messages using the output port of a MIDI interface renders that port inaccessible to other applications. This results in a situation where MIDI that is being produced (by a MIDI player) cannot be monitored internally, but must be physically output and then routed back to the input of a second MIDI interface, so that an application can capture the incoming MIDI messages.

The testing apparatus that was constructed includes USB MIDI interfaces for sending and receiving MIDI messages. Two interfaces are needed so that two separate MIDI streams can be monitored: one that loops back directly and one that has passed through the system being tested. The system uses a MIDI splitting device that allows for a single set of MIDI messages to be sent to two separate destinations, the loopback's stream and the testing system's stream. The apparatus also includes a desktop computer that is used to send MIDI messages as well as monitor the inputs of the USB MIDI interfaces.

The system depicted by the figure includes a network switch. This design was chosen as the systems being tested were Ethernet based. To remove any variances from differing

²The term "performance" is used throughout this chapter, but is context specific. When in the context of MIDI, the term describes the production of MIDI messages from a MIDI file: emulating a live MIDI performance. In the context of a system, the term describes the measurement of a system's ability to complete a task.

network speeds the same switch was used for the entirety of the testing process.

To ensure that this apparatus would provide a suitable platform for testing, the various parts of the apparatus were tested and these tests are described in sections 8.3, 8.4, and 8.5.

It should be noted that MIDI-OX is not an ideal application for testing MIDI latency. The resolution of the application does not offer any more than 1ms accuracy. The creation of a better monitoring system would have provided better insight, but was seen to be outside the scope of the project. The creation of a more suitable monitoring application is discussed in Section 9.8.

8.3 USB MIDI Interface testing

This section investigates the USB MIDI interface component of the testing apparatus. A suite of system tests were planned, all based on the production and monitoring of MIDI messages. The apparatus shown in Figure 8.1 makes use of USB MIDI interfaces to send and receive MIDI message from the computer being used to do the testing.

It was predicted that there would be latency experienced within the test systems, and so it was imperative that testing was performed to ascertain the latency a standard MIDI interface would introduce.

8.3.1 Apparatus

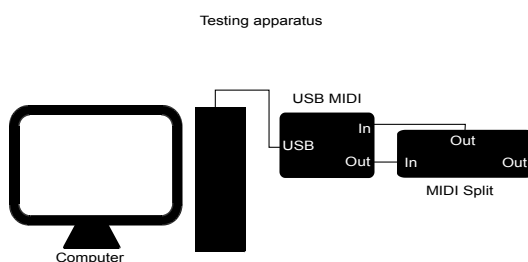


Figure 8.1: MIDI and Network testing apparatus

The production of MIDI messages and the monitoring of these was done using MIDI-OX³. MIDI-OX is a MIDI tool that allows for MIDI that is being sent out of a MIDI

³MIDI-OX is freeware application, although commercial licences are available and can be downloaded from <http://www.midiox.com>

interface to be captured. It also allows the recording of the MIDI being received on a separate interface. The tool enables the computer's keyboard to be used to generate MIDI messages that are sent out of the selected MIDI output.

Figure 8.1 shows the USB MIDI interface that sends and receives MIDI messages to and from the MIDI split device. A range of USB MIDI interfaces were tested:

- Roland UM-ONE [48]
- ESI MIDIMate II [13]
- Focusrite Scarlett 2i4 [15]
- M-Audio Axiom 25 [31]
- Steinberg UR22 [58]

The testing of a wider range of interfaces would have been useful in providing a perspective on the variance in latency of different MIDI interfaces, but for this research was not necessary.

The USB MIDI interfaces that were included in the testing did not all have the same format MIDI connectors, with the ESI and Roland interfaces making use of male connectors which cannot be plugged into each other. For this reason the MIDI split device that was to be used later in testing was used so that the output of an interface could be connected to the input.

USB MIDI interfaces made by Focusrite, M-Audio and Steinberg had female MIDI ports, requiring the use of a cable to connect the MIDI output to the MIDI input. This meant that these interfaces did not necessarily require the MIDI splitter⁴ to be used, but to keep the testing set-up as similar as possible, the MIDI interfaces were attached to the splitter using standard MIDI cables, each of which was 2m in length. Each USB MIDI interface to be tested was attached to the MIDI Splitter, so that messages sent from an output port could be immediately routed back to the computer via the input port to be recorded.

⁴MIDI splitter here is used to refer to a form of MIDI Thru box. As no off the shelf MIDI Thru box has been used, the term splitter was used to differentiate it.

8.3.2 Testing

Each USB MIDI interface was connected to the apparatus, and a set of 100 messages⁵ was sent (performed as 50 key presses). This allowed for Note-On and Note-Off messages to be sent in quick succession with slight variance in the gaps between key presses.

The variation in key press times was not problematic since the latency between the key being pressed and the MIDI being received after loopback is what was to be measured.

The latency of the interfaces being employed for testing was important, as any latency experienced during MIDI transmission through the MIDI connection management systems needed to be separated from any latency attributed to the USB interfaces themselves.

The ESI MIDIMate interfaces provide the option for the MIDI connectors to either be used as inputs or as outputs. To ensure that the choice of connectors for inputs or outputs had no influence on the latency of the interface, ESI MIDIMate interfaces were tested in each configuration.

The first configuration of the ESI MIDIMate interfaces, Config A, has MIDI connector 1 as the input and MIDI connector 2 as the output. The second configuration, Config B, is the opposite of Config A with MIDI connector 2 as the input and MIDI connector 1 as the output.

8.3.3 Results

The output produced by MIDI-OX during monitoring is in the form of a plain text monitor file, and thus required pre-processing before analysis. MIDI-OX allows a user to record the MIDI activity of specified MIDI ports, and writes the activity out to a file. The file contains the following information:

- Timestamp
- Input Source
- MIDI Port
- Status Byte

⁵Messages in this context refers to the MIDI messages being sent, either Note On or Note Off message. Each of these contains 3 bytes.

- Data Byte 1
- Data Byte 2
- MIDI Channel
- MIDI Note
- Event Type

An application called MIDI-Ox Reader was written that could read in a MIDI-OX monitor file and process it to produce a grid containing only the required information. Since the testing apparatus allows for two MIDI streams to be monitored, MIDI-Ox captures both these streams within the same file, as interleaved lines of text.

The input component of the MIDI-OX Reader application is used for reading a plain text monitor file from MIDI-OX so that the file can be used by the MIDI-Ox Reader Application. The MIDI-OX monitor file contains a capture taken during the testing of the ConMan system. The capture was performed using two ESI USB MIDI interfaces [13], which appear as ports 2 and 7 in the MIDI-OX monitor file.

The application has two output components that allow the user to separate the captured data from the two MIDI streams. This is done by selecting the port number of the USB interface's MIDI port used for each stream. The application is then able to extract the data for each specific port, placing it in the data grid so that it can be easily copied into a spread-sheeting application for analysis.

Libreoffice's Calc ⁶ was used to analyse the data that was collected from MIDI-OX. The data for each USB MIDI interface were reviewed with the mean and standard deviation being calculated for each interface.

⁶Calc is a free and open source spread-sheeting application licensed under the Mozilla Public License, and is available from <https://www.libreoffice.org/>

Table 8.1: Measured Latency of tested USB MIDI interfaces (in ms)

MIDI Device	Mean	Standard Deviation
ESI MIDIMate (Unit 1) Config A	1.39	0.49
ESI MIDIMate (Unit 1) Config B	1.39	0.49
ESI MIDIMate (Unit 2) Config A	1.31	0.46
ESI MIDIMate (Unit 2) Config B	1.22	0.42
Roland UM-1	1.27	0.45
Focusrite Scarlett 2i4	1.29	0.46
M-Audio Axiom 25	1.35	0.5
Steinberg UR22	2.92	0.34

Table 8.1 shows the results of the USB MIDI interface testing. These results show the latencies (given in ms) from when a MIDI message was generated in MIDI-Ox and output, to when the message was captured by the MIDI input. As shown in the table, the mean latencies range from 1.22 to 2.92 milliseconds. The mean amongst the results is 1.51ms with a standard deviation of 0.56ms. The Steinberg UR22 then can be considered an outlier in this regard as it is the only interface that lies beyond the mean of the interfaces, with a latency of nearly twice the mean.

With the baud rate of MIDI at 31250, it should take 0.32ms to produce or read a byte⁷. Note-On and Note-Off messages consist of 3 bytes and should therefore take 0.96ms to send or receive. From this it can be estimated that with no other delays being introduced, the time taken to loop back a MIDI message should be 1.92ms.

The timestamp provided by MIDI-OX is given in milliseconds, and therefore suffers a form of aliasing where message timestamps are rounded down to the nearest millisecond. Messages that should have a latency below 2ms would be recorded as 1ms, and latencies above 2ms but less than 3ms would be recorded as 2ms. This error would be further exaggerated given the timings for 2 byte messages which should transmit in 1.28ms.

The means for these tests are all above the 1ms mark, which is to be expected with the theoretical loopback time being 1.92ms. This theoretical time falls below the 2ms marks, resulting in the skewing of the results towards the lower sampling point: a higher occurrence of 1ms latencies.

⁷The baud rate of 31250 accounts for the start and stop bits that represent the beginning and end of a byte. This results in a speed of 3125 bytes/seconds.

Despite this statistical discrepancy, the results (after the removal of the Steinberg UR22) are fairly uniform with all recorded latencies falling within the 1ms to 2ms range.

8.4 Apparatus testing

The methodology for testing the two connection management systems can be found in Section 8.2. It describes the design of an apparatus, shown in Figure 8.2, which determines the latency that a MIDI connection management system adds to a MIDI signal path.

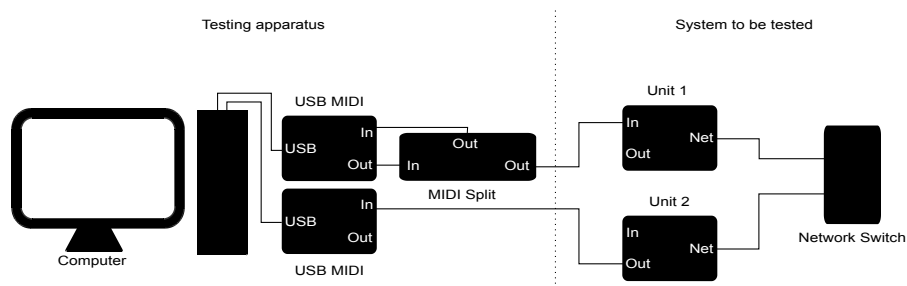


Figure 8.2: MIDI and Network testing apparatus

To prove that the apparatus would provide accurate results, the apparatus itself required testing.

8.4.1 Testing

Figure 8.3 shows the MIDI signal paths in the test apparatus. The MIDI messages are initially sent out through one of the testing computer's USB MIDI interfaces. The messages travel the first leg (denoted by label A) of the signal path to the MIDI splitting device. From here the paths are split: path B returning to the testing computer to be recorded, and path C continuing to the MIDI connection management system being tested.

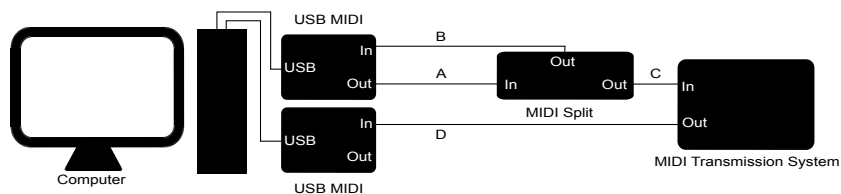


Figure 8.3: The MIDI Reader application

After the messages have passed through the connection management system, they return to the testing computer via path D. The computer will monitor the times of the messages from B and D, where the difference between the timestamps can be assumed to be introduced by the MIDI transmission system.

To test this assumption, a loopback test was constructed as shown in Figure 8.4. The test would monitor messages being returned over paths A and B, with the assumption that these should be received at the same time.

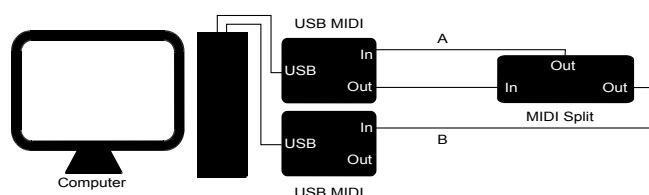


Figure 8.4: The MIDI Reader application

The tests were conducted using MIDI-OX and followed the same key press procedures described in Section 8.3.2. The recorded output from MIDI-OX was processed by MIDI-OX Reader and analysed using LibreOffice Calc.

8.4.2 Results

The 100 messages were analysed by a comparison of timestamps. No difference in timestamps was recorded, revealing that messages sent over the MIDI paths represented by A and B in Figure 8.4 experience the same latency, bearing in mind that the granularity of the timing is 1ms. This indicated that the times of signals sent over the two paths are within 1ms of each other.

8.5 Analysis of a Typical MIDI Performance

The successful transmission of MIDI messages does not provide much insight into a MIDI connection management system's performance capabilities. As noted in Section 8.2, tests were planned to establish the impact that the two MIDI Connection Management systems have on MIDI message latency.

It was predicted that there would be latency introduced by the connection management systems, and it was assumed that this latency could be affected by the load placed on the

system. In order for the MIDI Streaming system to be tested, a repeatable set of MIDI messages was required that would represent a MIDI performance. There is no set definition for what constitutes a complete MIDI performance, so MIDI files were investigated for a suitable song to replay as a MIDI performance.

Evangelos Odysseas Papathanassiou, widely known as Vangelis is a composer known for the award winning Chariots of Fire sound track[39]. Vangelis has used MIDI sequencing to control synthesisers in productions such as the Chariots of Fire and so the decision to use the MIDI score of the Chariots of Fire track Titles was fitting.

The MIDI file for Chariots of Fire was inspected to ascertain whether it would be a suitable test file. The file showed a suitable amount of instrumentation, and made use of multiple MIDI channels for instrument control. As shown later in this section, the channel use and message density of the file made it suitable for testing. The MIDI version of Vangelis's Titles that was used in this testing has a total runtime of 206 seconds.

The MIDI file format allows for MIDI messages to be stored and played back at specific times. MIDI files include all information on how synthesisers are configured, how levels are set, which notes are played and how these notes are modulated. Although it would have been possible to analyse the MIDI file itself, the transmission systems make use of MIDI messages, not the MIDI file format, thus making the performance of these messages more pertinent.

When a MIDI file is played, it recreates the performance as it would have been played live. For this reason the MIDI file was played and the messages created during the performance captured by MIDI-Ox for analysis as seen in Section 8.5.1.

8.5.1 Analysis Software

MIDI files were played using a MIDI playback and editing tool called MidiEditor ⁸, with messages being output using the USB MIDI interface. This allowed the produced messages to be captured as if they were played in live through a keyboard, simulating the capture of a MIDI performance.

This capture was done using MIDI-OX. As mentioned in Section 8.3, MIDI-Ox is a MIDI tool that allows for MIDI that is being sent out of a MIDI interface to be captured, as well as recording the MIDI being received on another interface.

⁸MidiEditor is a free and open source tool licensed under the GNU General Public Licence. Available from <https://sourceforge.net/projects/midieditor/>

MIDI-OX's monitoring has a limitation on the number of messages that are stored while inputs or outputs are being monitored. For this reason, the whole MIDI performance could not be analysed.

The first 45 seconds of Vangelis's Main Theme to Chariots of Fire was chosen, as there would be an upward spike in the number of MIDI messages at the start of the song as set up of instruments and channels is performed. The monitored file was processed using the MIDI-Ox Reader to prepare it for analysis, with the retrieved MIDI messages being placed in a spread-sheet.

8.5.2 Analysis

Simple quantitative testing was done on the performance data generated from the MIDI file to investigate the distribution of messages within the performance. This was done by totalling the number of MIDI messages received, based on the type (status byte and channel number) of each message and also by determining the number of messages in each second time interval. This can be used to develop a profile for a typical MIDI performance, and gives insight into the use of the various MIDI message status bytes.

It was assumed that if the number of MIDI messages being produced per second increased, there might be an impact on a connection management system's ability to transmit the messages, or the latency of the reproduction.

The MIDI messages were processed to produce a graph (Figure 8.5 in Section 8.5.3) showing the number of MIDI bytes produced per second⁹ by the system over the duration of the MIDI performance. This provided insight into density of the MIDI performance in terms of bytes per second, thus proving an overview of which parts of the MIDI performance would place higher strain on the systems that were tested.

8.5.3 Findings

Analysis of the totals for each type of status message revealed that the MIDI file was largely comprised of NoteOn and NoteOff messages with around 20% of messages being used for control changes. Table 8.2 shows the distribution of message types.

⁹The decision to make use of bytes per second rather than MIDI messages per second was made due to variation in the length of MIDI messages, being either 2 or 3 bytes.

Table 8.2: Distribution of MIDI messages by Status Byte

Note Off	Note On	Key Pressure	Control	Program	Channel Pressure	Pitch Bend
435	435	0	199	9	0	0

The table shows a high number of 3 byte messages, as few program changes were made throughout the score with no after touch used. Since this is a MIDI file intended to feed synthesisers (which in this case conform to the General MIDI extension), there are no SysEx messages. The analysis and testing of the connection management systems aimed to simulate a live MIDI performance. For this reason SysEx messages were not included in the testing.

The distribution of messages over different channels would not have any impact on the timings or reproduction of this MIDI performance, but serve as an illustration of the multi-channel nature of the MIDI file being used for testing. Table 8.3 shows this distribution, with almost no utilisation of channel 3 as well as channels 10 to 15, other than initial set-up.

Table 8.3: Distribution of MIDI messages over MIDI channels

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
59	588	12	2	38	14	117	16	12	208	2	2	2	2	2	2

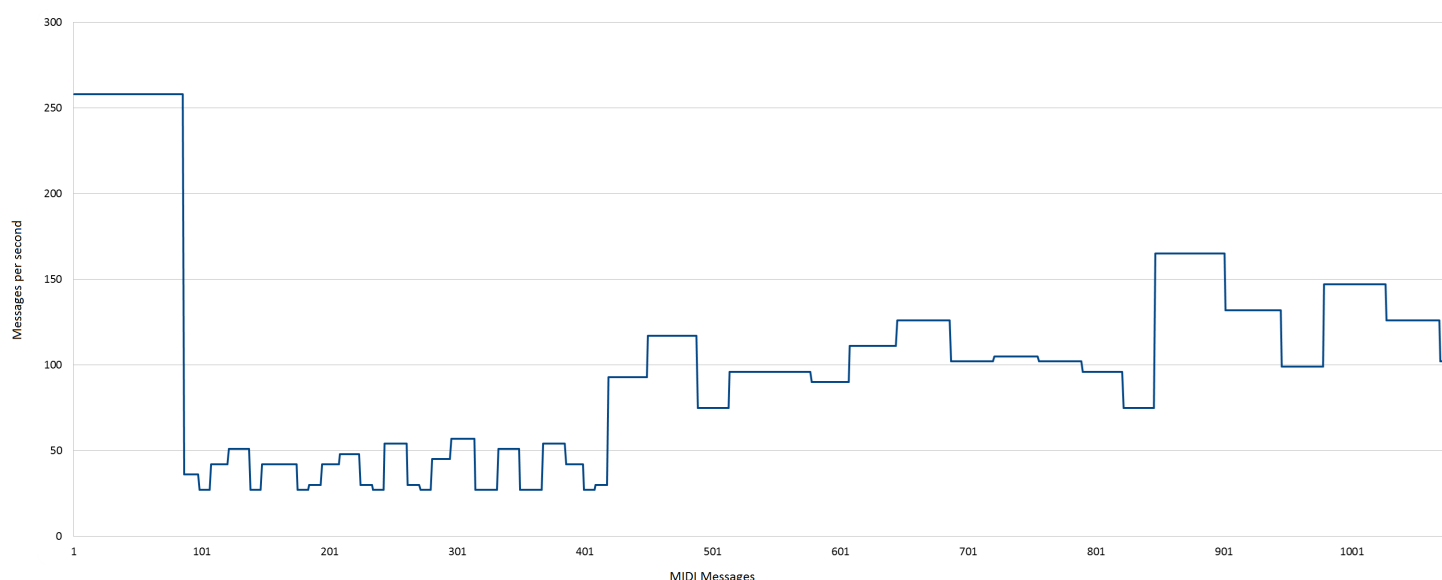


Figure 8.5: Bytes per second during MIDI performance

Figure 8.5 shows the complexity of the orchestration as the song progresses. At the

start of the song there is a spike in the number of messages per second. This is due to the initialisation of the various synthesisers attached to each channel, with a total of 86 messages being played within the first 640 milliseconds.

This set-up included Program changes to select the patches¹⁰ used on channels 0 through 10, and Control Changes for the synthesisers on these channels. The Control Changes included the set-up of parameters such as Reverb and Chorus, and adjustments to the levels.

Other than the initial set-up, the MIDI message density peaks after 687 messages, with a total of 42 messages being between the 36th and 37th seconds. This shows that the MIDI bandwidth¹¹ was not close to being fully utilised, although this was not necessary for the latency analysis, where a variation of message density could be compared with the recorded latency.

8.6 Stream Testing and Analysis

In order to gauge the ConMan system's performance it should be compared alongside an existing system. The analysis of the MIDI file-based performance in Section 8.5 provides an indication of the load placed on the system during the performance and hence enables the determination of the throughput of the two MIDI connection management systems at varying levels of load.

8.6.1 MIDINet test system specification

Testing of the MIDINet system was done using two low cost Intel systems that at the time of testing had a replacement value of about \$200¹², which makes them comparable in terms of cost to the XMOS SliceKit development kits. The full hardware specification and software setup of the machines can be found in Appendix C.

Each machine was set up with Windows 7 Enterprise (64 bit) as their operating system, above which the MIDINet system was running. Since the machines are not natively

¹⁰Selected sound to be produced by the synthesiser.

¹¹The MIDI specification allows for the transmission of 3125 bytes per seconds. Messages per second would not be an adequate metric as the number of bytes in a MIDI message can vary based on the type of message.

¹²This cost assignment is based purely on the hardware of the tower, and does not include the cost of a monitor or other peripherals.

equipped with MIDI ports each was paired with an ESI MIDIMate USB MIDI interface. They were then connected to a 1000Mbps network connection attached to a central D-Link DGS-1024D switch.

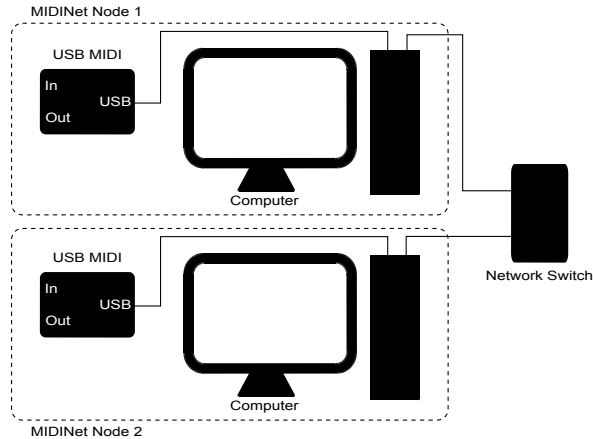


Figure 8.6: MIDINet Test system

As shown in Figure 8.6, each computer with its MIDINet software and ESI MIDI interface is referred to as a 'MIDINet node'.

8.6.2 ConMan test system specification

The ConMan test system comprised two ConMan prototyping devices. Each device included an XMOS SliceKit Core board, equipped with an Ethernet slice and an Audio Slice together referred to as a 'ConMan Node'. To eliminate any possible bottlenecks or speed inconsistencies, each board was flashed with the ConMan MIDI Stream firmware and was disconnected from the programming computer¹³. The specifications for the hardware are fully detailed within the design chapter in Section 6.3.

¹³The XMOS development environment allows for XMOS device to be monitored over the USB cable when running an application. This is useful for debugging as it allows for messages to be sent from the XMOS device to the computer. It was noted during testing that the USB connection to the computer appeared to have an effect on timing during run-time, and as such, flashing the devices was seen to be a safer option.

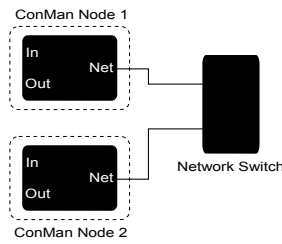


Figure 8.7: ConMan Test System

Each ConMan node was connected to a central D-Link DGS-1024D switch via its Ethernet slice as shown in Figure 8.7. The XMOS Ethernet slice provides a 100mbps connection. The XS1 microcontroller is not capable of providing a 1000mbps connection, but the choice to make use of a 1000mbps switch allowed for testing to be identical except for the nodes themselves, and removed any possibility for latency variance due to differing switch speeds.

8.6.3 Monitoring system specifications

To ensure that the initial production of MIDI messages, as well as the monitoring and recording of results, were not influenced by the monitoring system, the machine chosen was specified to have a higher performance than that of all systems to be tested. An Intel based system with a 4 core processor and 16Gb of memory was employed to perform all software based production and monitoring tasks. The machine ran a Microsoft Windows 8.1 (64bit) operating system, and made use of the previously mentioned MidiEditor software to produce the MIDI, and MIDI-OX to record received messages.

The monitoring machine was equipped with two MIDI interfaces which provided the systems that were to be tested with the necessary connections. As per the USB MIDI interface testing, a Focusrite Scarlett 2i4 audio interface with MIDI connections and an M-Audio Axiom 25 USB MIDI controller were chosen as monitoring interfaces.

As indicated in Section 8.2 an apparatus was constructed to enable testing of the latency of the two different connection management systems.

8.6.4 System testing

With the apparatus proven to work as seen in Section 8.4, the system was tested using a performance of Vangelis's Chariots of Fire, which was analysed in Section 8.5. The testing

was performed using MidiEditor¹⁴ to play MIDI messages from the MIDI file out of the USB MIDI interface, where they are split and sent through the two loops. The returning MIDI messages were recorded using MIDI-OX¹⁵.

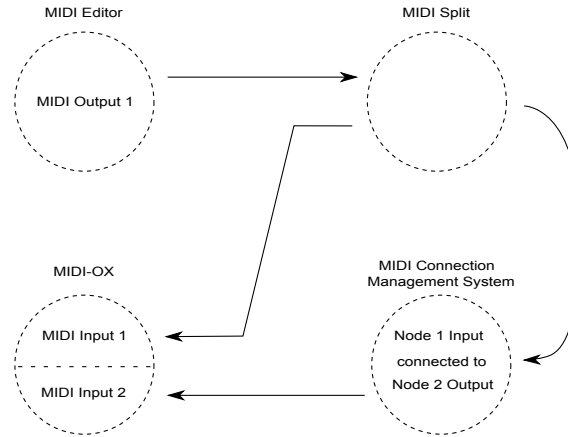


Figure 8.8: Abstract overview of testing

Figure 8.8 shows the testing setup, where the timestamps for MIDI Input 1 and MIDI Input 2 can be compared to determine the latency of the MIDI connection management system being tested. Initially the MIDINet system was tested, with the ConMan system being tested next. The systems being tested were the only factors in the tests that changed between tests, with the same network switch being used in both tests.

8.6.5 Analysis

The recorded data from MIDI-OX was processed using MIDI-OX Reader, and placed into Libreoffice Calc to be analysed. The data was first checked for validity and this was followed by a latency check. The latency data collected from the testing was analysed and compared to the message density of the original MIDI performance to ascertain if the latency on the system being tested had been influenced by the load on the system.

8.6.5.1 Validity check

The first check took the data points that were recorded and checked for validity. This was done by comparing the data received from the loopback to that which was received from the system being tested.

¹⁴MidiEditor is discussed in Section 8.5.

¹⁵MIDI-OX is discussed in Section 8.5.

The aim of the test was to ensure that the MIDI that had passed through each of the systems matched the original MIDI messages. Both systems had a 100% transmission throughput. Neither the MIDINet system nor the ConMan system experienced message loss or corruption, with all messages being received in the correct order.

8.6.5.2 Latency check

The subsequent latency check was used to test the latency of each system. Latency was determined by subtracting the timestamps of the looped back MIDI messages from the timestamps of the messages that had passed through the system being tested.

The transmission time varies depending on the length of the MIDI message being sent. This means that for each message a length metric (based on the number of bytes in the message) needed to be included, so that the latencies could be calculated per byte. The data obtained from testing the ConMan and MIDINet system was then processed to ascertain the average latency per byte, and the standard deviation.

8.6.6 Findings

Musicians, depending on their chosen instrument, are capable of playing with varying instrument latencies. Pianists and keyboardists, musicians most likely to make use of MIDI in live performance, experience difficulty while performing with a latency above 10ms. [29]

The testing and analysis found the two systems to have substantially different latencies. Table 8.4 shows the mean latencies and standard deviations for the ConMan and MIDINet system in ms, as well as the maximum and minimum latency experienced¹⁶.

Table 8.4: System latencies per message (ms)

System	Mean	Std. Dev.	Max.	Min.
ConMan	0.331	0.054	2	0
MIDINet	1.211	1.362	26	1

¹⁶The figures shown in the table are in ms, but as described in Section 8.3.3, are rounded down by MIDI-OX. The minimum latency achieved by the ConMan system is shown for be 0ms, but could be as high as 0.99ms due to the margin of error in the capture of the MIDI messages.

The MIDINet system had a mean latency of 1.211ms per byte for the measured performance. This latency corresponds to a 3.633ms latency when producing a Note-On message. Such latency is considered acceptable[35].

Analysis revealed the ConMan system to have a latency of 0.331ms per byte, which results in a 0.993ms latency when producing a Note-On message. This too meets the latency requirements for performance, while being considerably lower than the MIDINet system.

Figure 8.9 shows the latency per message of the MIDINet connection management system over duration of the MIDI performance. The figure shows an increasing latency in the beginning of the performance, followed by a drop in latency. The latency over the performance then varies slightly as the performance continues.

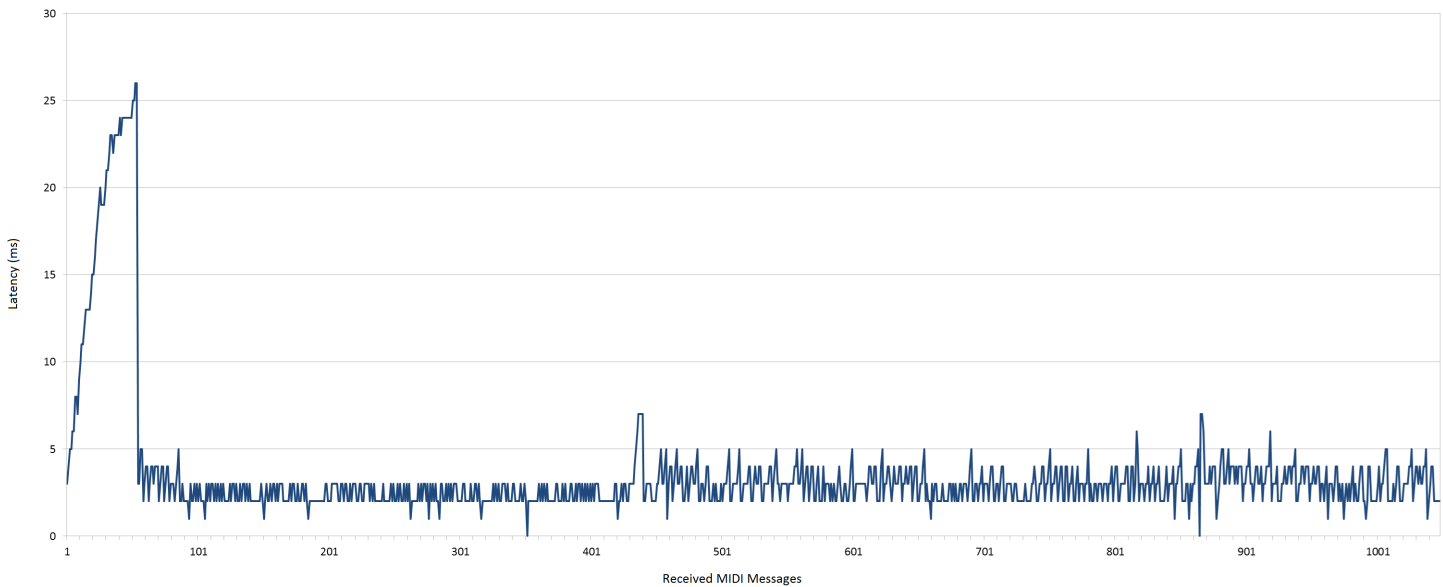


Figure 8.9: MIDINet system latencies

In Figure 8.10 the latency of the ConMan system is shown to have minimal variance, supported by the low standard deviation. The latency remains predictable and consistent throughout the performance.

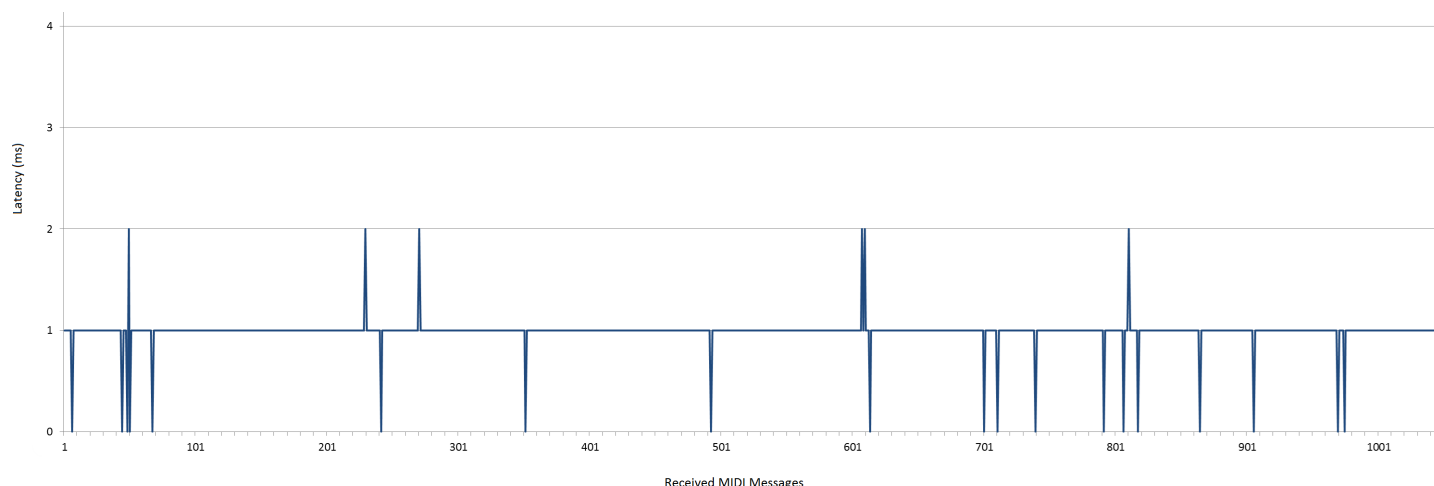


Figure 8.10: ConMan system latencies

The MIDI performance data used in the testing was examined in Section 8.5. The analysis of the performance revealed that the MIDI file had a high density of MIDI messages at the start of the performance. This is due to the setup¹⁷ that occurs at the beginning of the performance, resulting in the first 640ms containing 86 MIDI messages (258 bytes).

Figure 8.11¹⁸ shows the message per second density of the MIDI performance. As seen in Figure 8.9, there is an increase in the latency of MIDI messages at the beginning of the track during the period of high density. Toward the middle of the MIDI performance as the density increases the latency starts to rise again, becoming less predictable.

The latency of the ConMan system did not suffer any increases in latency due to the increases in message density from the MIDI file.

¹⁷This is discussed in Section 8.5

¹⁸This figure is a duplicate of Figure 8.5 discussed in Section 8.5.3. The figure has been repeated on this page for ease of reference.

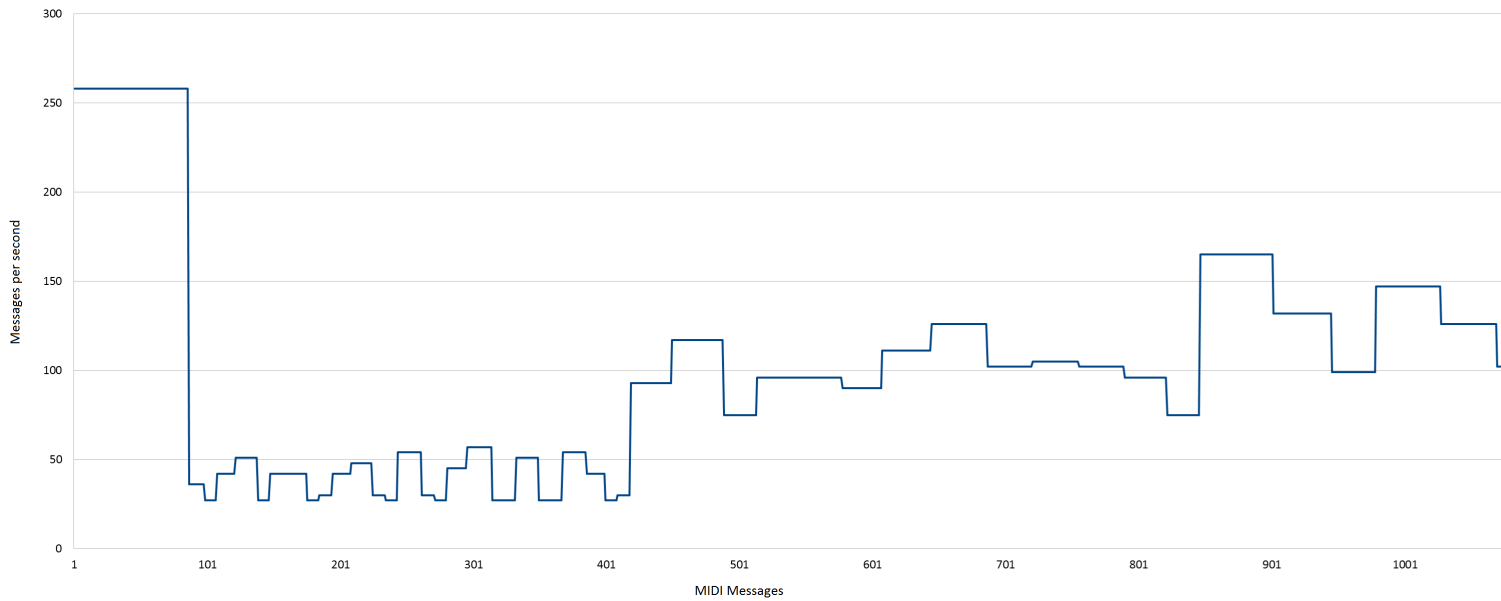


Figure 8.11: Density of MIDI performance in messages per second.

8.7 Interpretation

The results obtained from the stream testing showed the two connection management systems to behave differently when under load. The MIDINet system appeared to suffer an increase in latency as the number of messages per second increased.

The MIDINet testing system made use of desktop processors with 1Gb/s Ethernet NICs. In terms of raw performance these should out perform the SliceKit Core board with its slower speed processor and 100mb/s Ethernet slice.

Since the interfaces were tested in Section 8.3 and did not show a proportional latency response to load, it is clear that neither these interfaces nor their drivers could have introduced the latency.

It can only be assumed that the latency that the MIDINet test system experienced was due to either the MIDI/Networking implementation of the MIDINet application (discussed in Chapter 5), or that the latency is due to the overhead associated with running the application under a Windows operating system.

The implementation of the ConMan system, while equipped with lower end hardware, has yielded consistently low and predictable latencies. The real-time design and implementation of the ConMan system, coupled with the deterministic execution provided

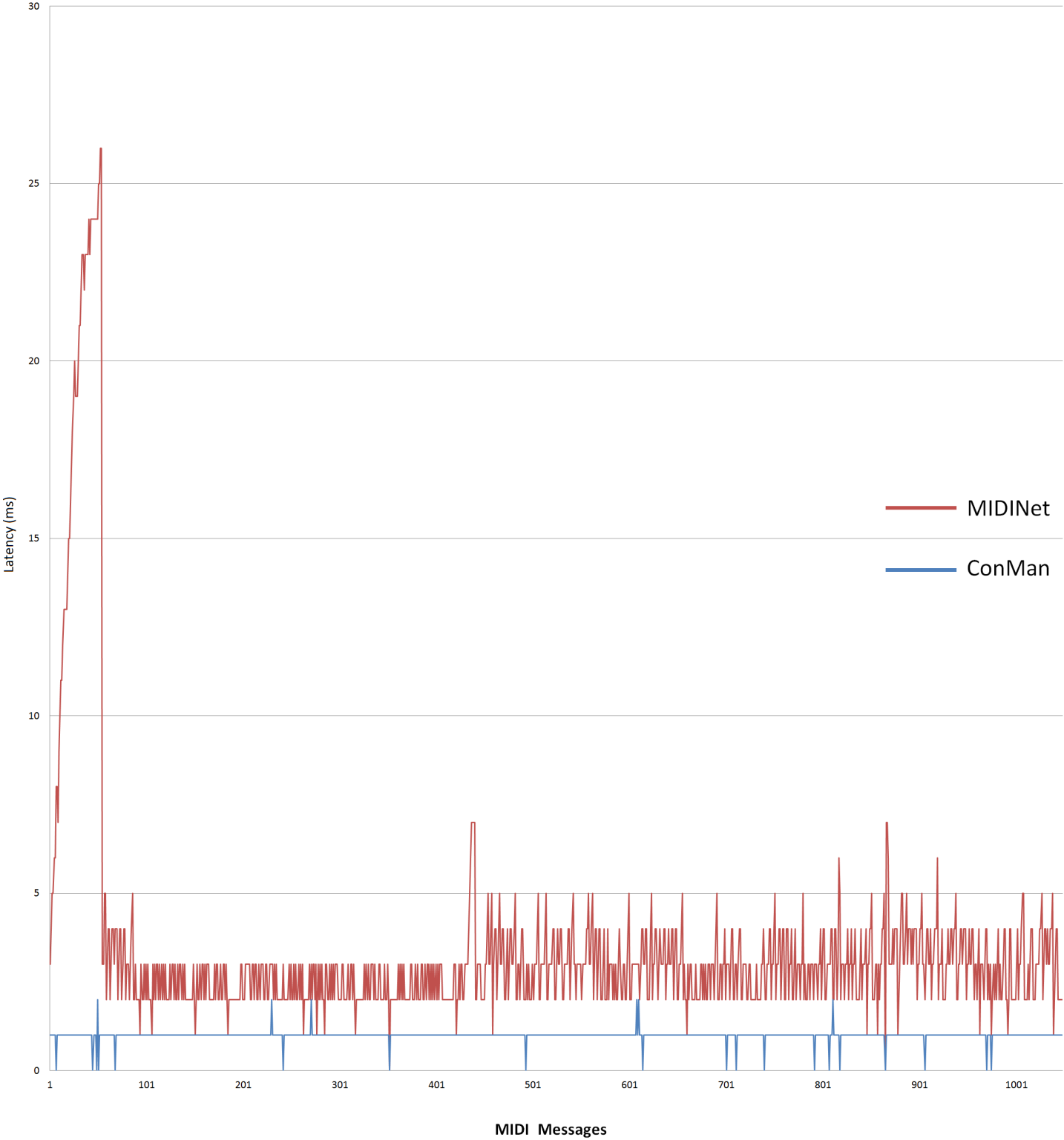


Figure 8.12: Comparison of latency: MIDINet and ConMan

by XMOS's microcontrollers, has led to the development of a system that manages to provide deterministic, low latency MIDI transmission over a network.

8.8 Conclusion

A methodology for the testing of MIDI connection management systems was proposed and tests were performed to ascertain whether the methodology would provide accurate results. The apparatus used in the testing was found to provide an adequate degree of monitoring that could be used in the comparison of MIDI connection management systems.

An investigation was also conducted into the use of a MIDI file to simulate a MIDI performance. The investigation made use of Vangelis's theme to Chariots of Fire, which exhibited a sufficient range of message density over the duration of the performance as well as high levels of channel utilisation.

The ConMan system that was implemented in Chapter 7 was compared (Figure 8.12) to the MIDINet System investigated in Chapter 5. The tests revealed the latency of the MIDINet system to increase under increased load. The results of the testing showed the ConMan system to hold a constant latency, with increases in load having no noticeable effect on the latency of the system.

The ConMan system's low latency performance has been attributed to real-time design methodologies coupled with deterministic multicore microcontrollers.

Chapter 9

Conclusion

9.1 Problem statement and Hypothesis

The MIDI protocol was initially developed to provide a standardised way to control synthesisers and audio production equipment. Although in need of a revision, MIDI remains a widely used control protocol for stage, studio and show control.

Despite the limited message space available to MIDI, the protocol has seen widespread utilisation for mixing consoles, audio processors and show control. This is largely attributed to the SysEx messaging format, which has enabled manufacturers to create their own messaging sub-formats, thus allowing for MIDI to be used in scenarios beyond the protocol's original scope.

However, as discussed in Chapter 2, MIDI has some limitations, which can largely be attributed to the age of the protocol, and its lack of modernisation. The more problematic limitations of the protocol include the unidirectional transmission system, restricted cabling length and lack of remote connection management.

The hypothesis of this thesis, stated in the introduction, is that a low cost embedded system can be used to provide connection management and deterministic transmission of MIDI over wireless networks.

9.2 Existing solutions to the problem

Due to the limitations in MIDI's specification, solutions have been developed to extend the usability of the protocol and overcome some of these limitations. Many solutions have been developed to overcome the cable length limitation, allowing MIDI messages to be transmitted over greater distances, enabling wider use of the protocol for applications such as show control.

A survey of existing solutions showed that none met the requirements of the hypothesis, but showed that some provide a feature-set that could be improved upon to prove the hypothesis.

Chapter 2 examined some available solutions, and compared these to identify issues in their feature sets and usability. Many of the solutions provided increased coverage through the removal of the cable length limitation and many featured connection management. The complicated set up and configuration of some solutions leads to them ultimately being cumbersome to use, with some requiring the user to have technical knowledge of networking to configure MIDI connections.

The MIDINet system provides a flexible connection management system with MIDI transmission over Ethernet networks. However, the system runs on desktop computers, and requires USB MIDI interfaces to enable hardware MIDI connectivity. This increases the cost of implementing the system and results in full size computers being used as control interfaces for routing, not granting engineers the freedom of mobility or the ability to manage connections remotely.

9.3 Embedded platform and reasons for its choice

The real-time nature of MIDI performance and show control required the use of a deterministic microcontroller as a hardware platform to create a solution that could fulfil the hypothesis. Hardware platforms such as Raspberry Pi, Arduino and XMOS were investigated as possible development platforms.

The Raspberry Pi was seen to not be suitable due its use of an operating system, which would make the platform non-deterministic. The Arduino development platform offers low level hardware development, and allows for less operating overhead than the Raspberry Pi, but has low clock speeds and only single core execution.

The deterministic execution of the XMOs microcontrollers, coupled with the availability of hardware development kits that provided the necessary connectivity for the fulfilment of the hypothesis, prompted the decision to use the XMOs Slicekit as the development platform.

The ability of the platform to include C and C++ libraries allowed for the design of the ConMan system to make use of both real-time and object-oriented design methodologies, without sacrificing the deterministic routing and transmission components.

The discovery that the XMOs WiFi slice was not fully supported made it impossible to complete the development as a wireless system. However the modular design of the Slices from XMOs and the interchangeability of the physical layer implementation within the OSI model would allow for the replacement of the Ethernet Slice with the WiFi slice once a functional component is available.

The use of the Ethernet slices coupled with wireless access points allows for the system to be managed remotely from a wireless device, such as a tablet or smartphone, providing the advantages of wireless freedom for connection management and control.

9.4 The design and its part in lowering latency

The ConMan system was designed using both real-time systems design and object-oriented design concepts. The flexible connection management of the MIDINet system allows for devices to be easily connected. Minimal set up is required due to the discovery process within MIDINet, which automatically identifies other MIDINet enabled devices on a network and populates the connection management system.

The non-deterministic operation of the MIDINet system required that the design move time-sensitive aspects of the transmission system to a real-time design. The connection management was redesigned as an object-oriented module that could be used within the ConMan system, while allowing the real-time system to handle MIDI transmission and routing.

The focus of the design on lowest latency transmission created an efficient and deterministic routing and transmission mechanism that remains decoupled from the connection management system.

9.5 Web based control over connections

Embedded systems can decrease the size and cost of solutions, through the removal of hardware used to provide physical controls to a device, such as displays and hardware controls. These embedded systems still require control, making use of desktop or web based applications to remotely control the devices.

The use of a web application enables control to be performed from any device that has a web browser, allowing for remote control from a wide range of devices, while only requiring the implementation of a single control application.

A combination of embedded web server and modern web technologies removes the need for a display and controls to be present on ConMan devices, while providing users the mobility and freedom to control devices without requiring physical access to a ConMan node.

This aids in the reduction of design and implementation costs while increasing the usability of the system through the use of remote control for distributed connection management.

9.6 Discoveries arising from testing

An apparatus was created and tested to aid in the evaluation of the ConMan system. Testing was also performed on USB MIDI interfaces to ensure fairness and accuracy of the testing to be performed on the connection management systems. The ConMan system was tested against MIDINet, as MIDINet incorporated all the desired connection management features.

While testing the systems, it was revealed that under heavier loads the MIDINet system suffers an increase in latency. The tests showed the ConMan system to hold a constant latency, with no latency spikes when placed under heavy loads.

Testing proved that the computationally efficient design and hardware features for deterministic execution can have more of an impact on the performance of the system than the use of hardware with greater computational capability.

The ConMan system has been shown to perform predictably with the MIDI load having no discernible impact on the system's latency. The latencies revealed by the testing are

predictable and acceptable in use cases involving musicians, and should be adequate for control.

9.7 Confirmation of hypothesis

Currently available multicore microcontrollers when used in conjunction with real-time design techniques and appropriate network support structures show that a low cost embedded system can indeed be used to provide flexible connection management and deterministic MIDI transmission over networks. The ConMan system overcomes limitations of MIDI such as the cable length limit and the protocol's lack of connection management.

The MIDI protocol remains relevant to the music production and entertainment industries. MIDI still exhibits limitations such as the limited message space, channels and speed. Further research may aid in the enhancement of the protocol and removal of such limitations.

9.8 Possible future work

This section identifies and addresses known issues within this thesis where an increase in scope and further investigation would benefit the research.

9.8.1 XMOS WiFi Slice

Initially the decision to use the XMOS development kit was largely predicated on the availability of hardware add-on boards called slices. XMOS had available an Audio slice and a WiFi slice, the combination of which was seen to be idea for the project. During the project is became apparent that the WiFi slice's support library was not fully implemented. Ideally the usage of this slice would have yielded a wireless MIDI transmission system, which was the initial goal.

Development of the WiFi slice support library would be beneficial to other researchers and would would have allowed for the completion of the originally proposed project. While the further development of the WiFi slice or investigation into the creation of a new, more

capable WiFi slice was outside the scope, this would be a valuable contribution to the area in the future.

9.8.2 Compared transmission systems

As mentioned in Chapter 2, numerous MIDI transmission systems are available. The testing performed in Chapter 8 only considered the ConMan and MIDINet systems, due to their connection management capabilities.

However, for purely transmission based testing, a wider field of test subjects could be surveyed. This would provide a more thorough set of results and could further be used in other research for performance comparison.

9.8.3 MIDI message source

The use of a preselected song in the testing was originally seen to be representative of an average MIDI performance, but this can not simple be assumed. To do so would require a comprehensive survey of MIDI scores with regards to message density, length and channel utilisation. This was seen to be outside the scope of the project, but can certainly be addressed as a separate body of work.

A more suitable form of testing might have been to utilise randomly generated MIDI, so that channel utilisation and message density could be more thoroughly controlled.

9.8.4 Testing methodology

While the testing methodology bares some resemblance to other methods [65, 59], and is not entirely flawed, the use of MIDI-OX as a method for MIDI monitoring and capture is problematic.

Other than redoing the testing using previously mentioned methods such as an oscilloscope or MIDI-Wave setup, the testing might benefit from simply replacing MIDI-OX with a higher resolution monitoring application. A simple MIDI monitoring application could be developed that is capable of recording sub-millisecond timestamps. This was seen to be outside the scope, but if developed, could simply be used to replace MIDI-OX within the testing apparatus.

Further development of a system purely designed for this form of testing could be performed, removing the need for MIDI splitters or other hardware devices. This too was beyond the scope of the project, but would be of great future benefit.

9.8.5 SysEx messaging

By nature SysEx messages are unique to use-case, manufacturer or device, and as such it was initially determined that transmission testing of these messages would not be beneficial.

However, it should be stated that the handling of these messages is different to those of MIDI Voice messages and as such testing of SysEx messages might provide further insight into various transmission systems.

As with the proposed future work for the MIDI Message Source, a survey of MIDI SysEx messaging would be useful in identifying common traits and developing a system that can produce arbitrary SysEx messages for testing.

9.8.6 Connection Management

The time taken to create, modify or destroy connections would benefit from future testing. Work would need to be done so that this could be compared to other systems, which would require the testing methodology be modified or a new methodology be selected.

9.8.7 User interface

The user interface could in future work be tested for usability, completeness and bugs. This could be achieved through a survey carried out by computer users. The diversity of the group in terms of both computer (and network) literacy and knowledge of MIDI would need to be tested too.

This form of testing was not seen to be beneficial as the focus of the project was primarily on performance. However, future connection management applications, developers and researchers would benefit from testing into the usability of such applications.

Bibliography

- [1] ABDALLAH, A. E., JONES, C. B., AND SANDERS, J. W. *Communicating Sequential Processes. The First 25 Years: Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004. Revised Invited Papers*, vol. 3525. Springer Science & Business Media, 2005.
- [2] ABOBA, B., GUTTMAN, E., AND CHESHIRE, S. Dynamic configuration of ipv4 link-local addresses.
- [3] BECKHOFF AUTOMATION GMBH. Fast communication with the DMX protocol. On-line, 2011. Available from: http://download.beckhoff.com/download/document/application_notes/DK9222-0311-0029.pdf.
- [4] BUS, U. S. 3.0 specification. *Hewlett-Packard Company, Intel Corporation, Microsoft Corporation, NEC Corporation, ST-NXP Wireless, Texas Instruments*, rev 1 (2008).
- [5] CAIN, B., HABERMAN, B., AND HOLBROOK, H. Using internet group management protocol version 3 (igmpv3) and multicast listener discovery protocol version 2 (mldv2) for source-specific multicast.
- [6] CHESHIRE, S., AND KROCHMAL, M. Rfc 6762: Multicast dns. *Internet Engineering Task Force (IETF) standard* (2013).
- [7] COMPAQ, H.-P., INTEL, L., ET AL. Universal serial bus specification revision 2.0. In *USB Implementers\ Forum* (2000).
- [8] DEFURIA, S., AND SCACCIAFERRO, J. *MIDI Programmer's Handbook*. IDG Books Worldwide, Inc., Foster City, CA, USA, 1990.
- [9] DIBLEY, J. *An Investigation of the XMOS XSI Architecture as a Platform for Development of Audio Control Standards*. 2013.

- [10] DROMS, R., BOUND, J., VOLZ, B., LEMON, T., PERKINS, C., AND CARNEY, M. Dynamic host configuration protocol for ipv6 (dhcipv6). Tech. rep., 2003.
- [11] DUNKELS, A. uip-a free small tcp/ip stack. *Technical report* (2002).
- [12] EALES, ANDREW; FOSS, R. Service discovery using open sound control. In *Audio Engineering Society Convention 133* (10 2012).
- [13] ESI. Esi midimate ii.
- [14] FARAGO, B. Xmos midi1. https://github.com/bfarago/xmos_midi1, 2016.
- [15] FOCURITE. Focusrite scarlett 2i4.
- [16] FOSS, R. Object oriented design part 3: Midinet complete design and implementation. Rhodes University Computer Science III course notes.
- [17] FOSS, R. J. A networking approach to sharing music studio resources. 1996.
- [18] HOARE, C. A. R. Communicating sequential processes. In *The origin of concurrent programming*. Springer, 1978, pp. 413–443.
- [19] HOSKEN, D. *An Introduction to Music Technology*. Taylor & Francis, 2010.
- [20] HUBER, D. M. *The MIDI manual: a practical guide to MIDI in the project studio*. Focal Press, 2012.
- [21] HUNTINGTON, J. *Control systems for live entertainment*. Taylor & Francis, 2007.
- [22] IEEE. Part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications, 2012.
- [23] JACOBS, G., AND GEORGHIADES, P. *Music and new technology: the MIDI connection*. Sigma Pr, 1991.
- [24] JUN-CHI, FUJIMORI; FOSS, R. mlan: Current status and future directions. In *Audio Engineering Society Convention 113* (5 2002).
- [25] KIRK, R., AND HUNT, A. *Digital sound processing for music and multimedia*. Focal Pr, 1999.
- [26] KUROSE, J. F., AND ROSS, K. W. *Computer networking: a top-down approach*. Addison Wesley, 2007.

- [27] LAZZARO, J., AND WAWRZYNEK, J. Rfc 4696: An implementation guide for rtp midi. *InterNet Network Working Group* (2006).
- [28] LONE WOLF. *MIDITap User Manual*, 1990.
- [29] LOY, G. Musicians make a standard: The midi phenomenon. *Computer Music Journal* 9, 4 (1985), pp. 8–26.
- [30] LSR. TiWi-SL Module Datasheet. Online. Available from: <https://www.lsr.com/downloads/products/330-0085.pdf>.
- [31] M-AUDIO. M-audio axiom 25.
- [32] MAY, D., MULLER, H., HEDINGER, P., DIXON, A., OWEN, R., AND RICHARDS, N. Xs1 ports: use and specification. *XMOS Ltd 1* (2008).
- [33] MESSICK, P. *Maximum MIDI: Music Applications in C++*. Manning Pubs Co Series. Manning, 1998.
- [34] MIDI MANUFACTURERS ASSOCIATION. MIDI Show Control (MSC) 1.0. Online, 1991. Available from: <http://www.richmondsounddesign.com/docs/midi-show-control-specification.pdf>.
- [35] MIDI MANUFACTURERS ASSOCIATION. Complete MIDI 1.0 Detailed Specification. Online, 1999/2008. Available from: <http://www.midi.org/techspecs/gm.php>.
- [36] MOOG, R. A. Midi: Musical instrument digital interface. *J. Audio Eng. Soc* 34, 5 (1986), 394–404.
- [37] MOSALA, T. J. Routing midi messages in a shared music studio environment. 1995.
- [38] MYNDIDEAL. midil. <http://www.xcore.com/viewtopic.php?f=21&t=5444&p=27703&hilit=MIDI>.
- [39] PAPATHANASSIOU, E. O. Chariots of fire, 1983. Interscope / Polydor / Universal Motown.
- [40] PETERSON, L., AND DAVIE, B. *Computer Networks: A Systems Approach*. The Morgan Kaufmann series in networking. Morgan Kaufmann Publishers, 2003.
- [41] PLUMMER, D. C. Rfc 826: An ethernet address resolution protocol. *InterNet Network Working Group* (1982).
- [42] POSTEL, J. User datagram protocol. Tech. rep., 1980.

- [43] POSTEL, J. Rfc 792: Internet control message protocol. *InterNet Network Working Group* (1981).
- [44] POSTEL, J. Transmission control protocol.
- [45] POSTEL, J., AND PROTOCOL, U. D. Rfc 768. *User datagram protocol* (1980), 1–3.
- [46] POSTEL, J., AND REYNOLDS, J. K. Standard for the transmission of ip datagrams over ieee 802 networks.
- [47] PROSISE, J. *Programming Windows with MFC*. Microsoft programming series. Microsoft Press, 1999.
- [48] ROLAND. Roland um-one.
- [49] RONA, J. C., AND SCHIFF, R. S. *MIDI, the INS, Outs and Thru's*. Hal Leonard Books, 1990.
- [50] ROTHSTEIN, J. *MIDI: A comprehensive introduction*, vol. 7. AR Editions, Inc., 1995.
- [51] RUMSEY, F. *MIDI Systems and Control*, 2nd ed. Butterworth-Heinemann, Newton, MA, USA, 1994.
- [52] RUMSEY, F., AND WATKINSON, J. *Digital interface handbook*. Taylor & Francis, 2004.
- [53] SCHMITT, D. ipMIDI. Online. Available from: <http://www.nerds.de/en/ipmidi.html>.
- [54] SCHULZRINNE, H., CASNER, S., FREDERICK, R., AND JACOBSON, V. Rfc 3550. *RTP: A transport protocol for real-time applications 7* (2003).
- [55] SCHULZRINNE, H., CASNER, S., FREDERICK, R., AND JACOBSON, V. Rtp: A transport protocol for real-time applications. Tech. rep., Internet Engineering Task Force, 2003.
- [56] SELFRIDGE-FIELD, E. *Beyond MIDI: the handbook of musical codes*. MIT press, 1997.
- [57] STARR LABS. AirPower2 Wireless MIDI: Wireless MIDI to USB + MIDI Stand-alone Solution. Online. Available from: http://www.starrlabs.com/index.php?route=product/product&product_id=79.

- [58] STEINBERG. Steinberg ur22.
- [59] THOM, B., AND NELSON, M. J. An in-depth analysis of real-time midi performance. In *ICMC* (2004).
- [60] VEGODA, L., MEYER, D., AND COTTON, M. Iana guidelines for ipv4 multicast address assignments.
- [61] WARD, P., AND MELLOR, S. J. *Structured Development for Real-Time Systems, Volume 1: Introduction & Tools; Volume 2: Essential Modeling Techniques; Volume 3: Implementation Modeling Techniques*. Yourdon Press, New York, 1985.
- [62] WATT, D. *Programming XC on XMOS devices*. XMOS Limited, 2009.
- [63] WATT, D., OSBORNE, R., AND MAY, D. Xc reference manual (8.7). website, 2008.
- [64] WING, D. Symmetric rtp/rtp control protocol (rtcp).
- [65] WRIGHT, J. L., AND BRANDT, E. System-level midi performance testing. In *ICMC* (2001).
- [66] WRIGHT, M. Open sound control 1.0 specification, March 2002.
- [67] WRIGHT, M. Open sound control: an enabling technology for musical networking. *Organised Sound 10* (2005/12/01 2005), 193–200.
- [68] WRIGHT, M., AND FREED, A. Open sound control: A new protocol for communicating with sound synthesizers. In *International Computer Music Conference* (Thessaloniki, Hellas, 1997), International Computer Music Association, pp. 101–104.
- [69] XMOS. Embedded Webserver Library Programming Guide. Online. Available from: <https://www.xmos.com/download/public/Embedded-Webserver-Component-Programming-Guide-%28documentation%29%281.0.0rc0.a%29.pdf>.
- [70] XMOS. TCP/IP Library. Online. Available from: https://www.xmos.com/download/private/lib_xtcp-%5Buserguide%5D%284.0.1rc1%29.pdf.
- [71] XMOS. xCORE Multicore Microcontrollers. Online. Available from: [https://www.xmos.com/download/public/xCORE-Multicore-Microcontrollers-Overview\(1.2\).pdf](https://www.xmos.com/download/public/xCORE-Multicore-Microcontrollers-Overview(1.2).pdf).

-
- [72] XMOS. XMOS Programming Guide. Online, 2014. Available from: [https://www.xmos.com/download/private/XMOS-Programming-Guide-\(documentation\)\(E\).pdf](https://www.xmos.com/download/private/XMOS-Programming-Guide-(documentation)(E).pdf).
- [73] XMOS. Using XMOS TCP/IP Library for UDP-based Networking. Online, 2015. Available from: [https://www.xmos.com/download/private/AN00121:-Using-XMOS-TCP-IP-Library-for-UDP-based-Networking\(2.0.0rc1\).pdf](https://www.xmos.com/download/private/AN00121:-Using-XMOS-TCP-IP-Library-for-UDP-based-Networking(2.0.0rc1).pdf).

Appendices

Appendix A

xC Language Overview

A.1 Parallel tasks

The xC language utilised a statement called a 'par' block[63] to declare two tasks that will be run in parallel. The format of a par block is shown in Figure A.1.

Listing A.1: Example usage of par block.

```
main()
{
    channel internal; //Channel over which data is sent.
    par
    {
        Producer(internal); //A function that produces data
        Consumer(internal); //A function that consumes data
    }
}
```

In Listing A.1 the two tasks running in parallel communicate of the channel that is passed to each of them, the 'internal' channel[63].

A.2 Channels

Channel messaging on the XMOS XS1 range of microcontrollers is done using a set of software controlled bridges between logical cores. When data is to sent from one core (or process) to another over a channel, the route is negotiated. This route is temporary and will be removed over the transmission is complete[62].

To allow for channel communication, xC allows for parallel functions to be assigned channel ends (chanends) as parameters. A consequence of the nature of the parallel processing

of the language is that function declared in a parallel execution block will be declared with the channel they will use to communicate between them, while the function will be defined with the chanends. So there is not a simple way to see where a channel is used throughout the application, as each channel is also represented as two chanends[62, 63].

Listing A.2: Example producer function

```
void Producer(chanend out)
{
    //Loop sends integers 0 to 9 through the out channend
    for (int i = 0; i < 10; i++)
    {
        i :> out;
    }
}
```

Listing A.3: Example consumer function

```
void Consumer(chanend in)
{
    //Loop receives and prints 10 integers received from the in channend
    for (int i = 0; i < 10; i++)
    {
        int x <: in;
        println(x);
    }
}
```

Listing A.4: Example main function

```
main()
{
    channel internal; //Channel over which data is sent.
    par
    {
        Producer(internal); //A function that produces data
        Consumer(internal); //A function that consumes data
    }
}
```

The channel declared in the main method is called “internal”. This channel is used in the par block and is given to both the “Producer” and “Consumer” functions as the channel over which they will communicate. But as seen in the listing above, the “Producer” is actually defined as only having a chanend as a parameter “out” and the “Consumer” only takes a chanend “in”. These appear to be different in type to the channel “internal” that each function is called with[9].

A channel is declared, and is then connected to two chanends. It may not be set up with more than two chanends. The channel can be used for bidirectional communications and there is a certain amount of buffering that can occur on a channel, but data passed to a channel must be removed or an illegal resource exception will occur[32].

Streaming channels are used to create asynchronous connection between two tasks that are permanent and are not renegotiated after each transmission. This is faster than

transactions and can be used concurrently. But since these are not freed after use, they consume a channel permanently, reducing the resources left of the microcontroller[71, 62].

A.2.1 Channel Transactions

Channel transactions allow for data to be transmitted in a way that does not result in the channel being renegotiated between every transmission. This is done by declaring a master and slave task that allows for channels to remain active during transmission. This is generally used to communicate large quantities of data. The master thread will continue to send uninterrupted, unless the channel buffer is full, while the slave thread will pull from data from the channel unless the channel buffer is empty[72].

Listing A.5: Example producer using channel transactions

```
void Producer(chanend out)
{
    master
    {
        //Loop sends integers 0 to 9 through the out channend
        for (int i = 0; i < 10; i++)
        {
            i :> out;
        }
    }
}
```

Listing A.6: Example consumer using channel transactions

```
//Task to run in parallel
void Consumer(chanend in)
{
    slave
    {
        \\Loop receives and prints 10 integers received from the in channend
        for (int i = 0; i < 10; i++)
        {
            int x <: in;
            println(x);
        }
    }
}
```

A.3 Select Statement

xC provides a manner of declaring interest in events through the use of 'select' statements. Select statements can act upon events such as data being passed over a channel, ports, or a timer. You can also construct functions with a "select" return type that can return a case for a select statement as to keep the select statements neat and tidy[63].

To do this a function must first be created:

Listing A.7: Example of a merge using the select statement

```
//Function with select return type
select merge(chanend left , chanend right , chanend out)
{
    case left :> v: //Data received from left channend
        out <: v;    //Send data through out channend
        break;

    case right :> v: //Data received from right channend
        out <: v;    //Send data through out channend
        break;
}
```

Listing A.8: Example usage of merge function

```
//Select Statement with parameterized function
select
{
    merge(left , right , out);
}
```

A.4 Ports and Pins

A.5 C and C++ integration

If a function is defined in a C or CPP file, the function will need to be prototyped in the xC file where you plan to use it. This is done by implementing the “extern” directive[62].

The listing below shows a C function that has been written and placed in an xC file.

Listing A.9: function-c.c

```
int f()
{
    printf("This_is_a_C_function\n");
}
```

The C function can then be called from an xC function using the “extern” directive. The listing below demonstrates how an xC function can thus be set up to use C functions.

Listing A.10: function-xc.xc

```
extern "c" { extern int f(); }

// The function can now be used in xC files and in parallel statements

void g()
{
    par
    {
```



```

        f();
        f();
    }
}

```

The example above provides the most basic usage of the inclusion of C in xC. The example below makes use of a most classic C++ style file structure, and shows the integration of C++ (or C) files in an xC application.

The listing below shows the main file of an xC application. The file contains a producer (main_par_function) that is to run in parallel with another function that exists with a module. The par block calls the producer (main_par_function) and consumer (module_par_function) which communicate over the internal_chan channel.

Listing A.11: main.xc

```

main_par_function( chanend out )
{
    for (int i = 0; i < 10; i++)
        out <: i;
}

main(void)
{
    chan internal_chan;

    par
    {
        main_par_function(internal_chan);
        module_par_function(internal_chan);
    }
}

```

The listing below shows a module that provides a wrapper function for the C++ function that will be used by the xC application. The wrapper is written in xC and calls the C++ function from within a loop. In this case the xC is the consumer as it deals with the actual channel interactions.

Listing A.12: module.xc

```

#include <xccompat.h>
#include <module.h>

void module_par_function(chanend in)
{
    for (int i = 0; i < 10; i++)
    {
        int x <: in;
        cpp_function(x);
    }
}

```

The listing below shows the header file for the C++ code.

Listing A.13: module.h

```

void module_par_function(chanend in);

void cpp_function(int x);

```

The listing below shows the C++ file where the function is actually written. The function is prefixed with an “extern” directive to allow it to be called from xC functions. The function here only prints integers passed to it by the consumer.

Listing A.14: functions.cpp

```
extern "C" void cpp_function(int i)
{
    printintln(i);
}
```

The listings and examples above have been placed here as a summary of documentation provided by XMOS on the usage of C and C++ in xC applications[62, 63, 72].

Appendix B

ConMan System Design

B.1 ConMan: Essential Model

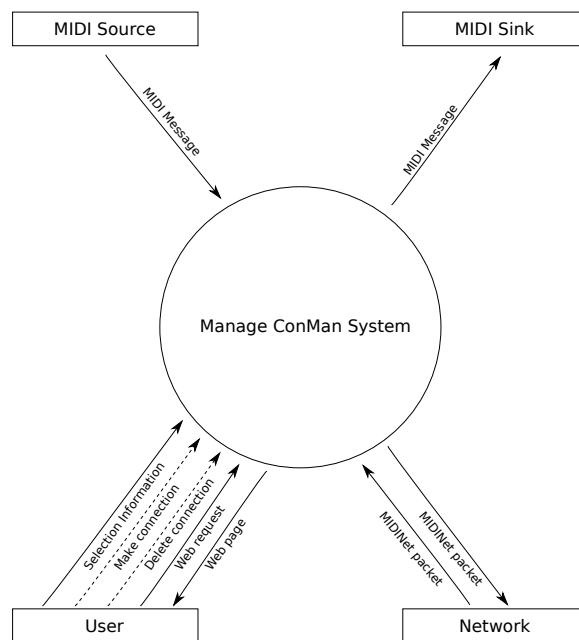


Figure B.1: Context diagram for MIDI ConMan

B.1.1 Level 0

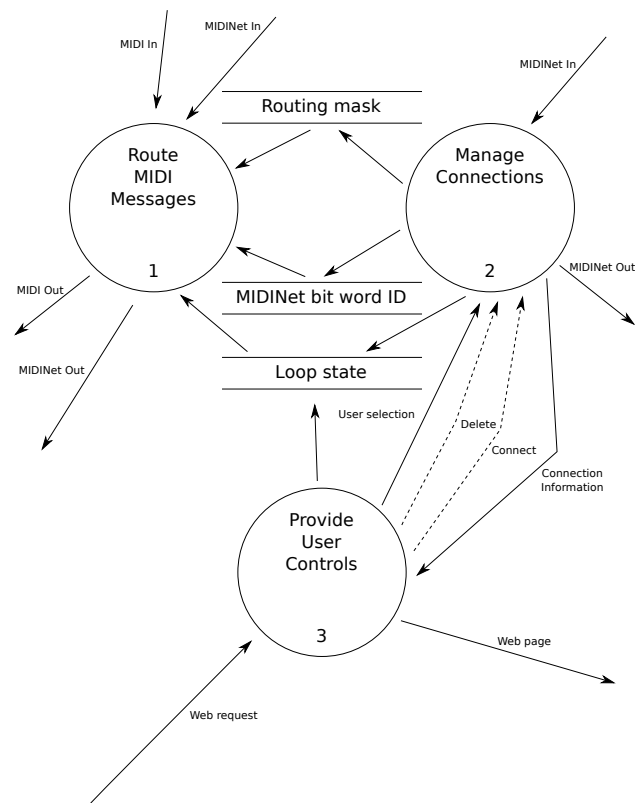


Figure B.2: Level 0 - Manage ConMan System.

B.1.2 Level 1

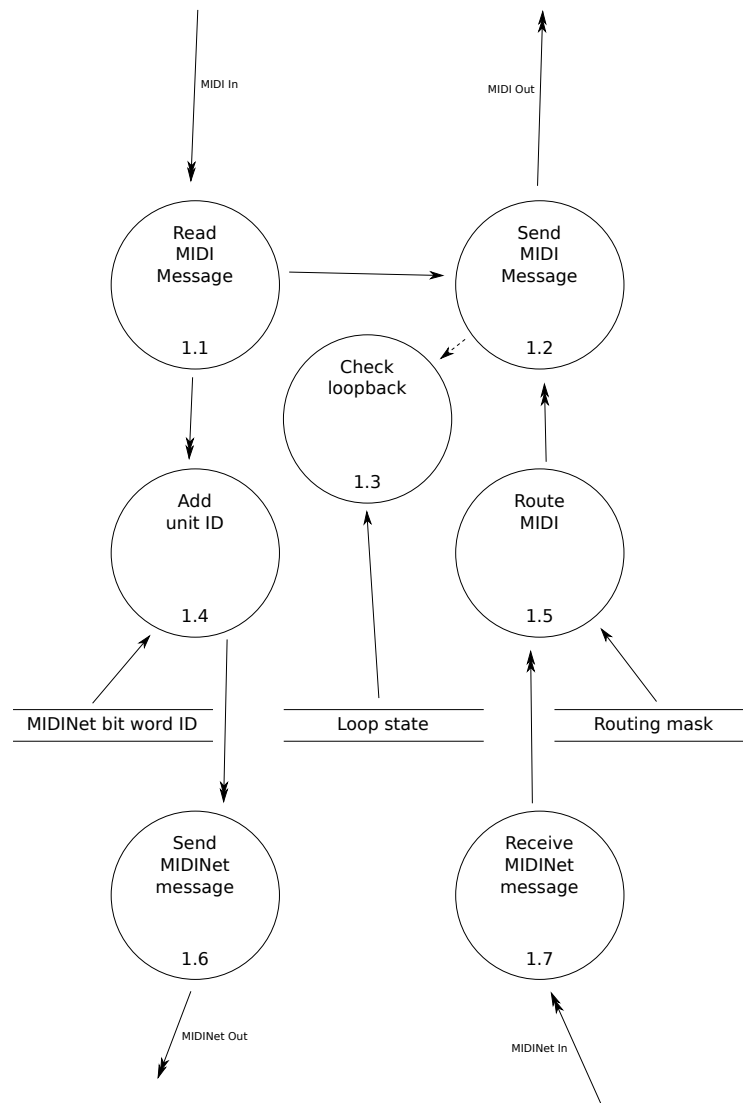


Figure B.3: Task 1 - Route MIDI messages.

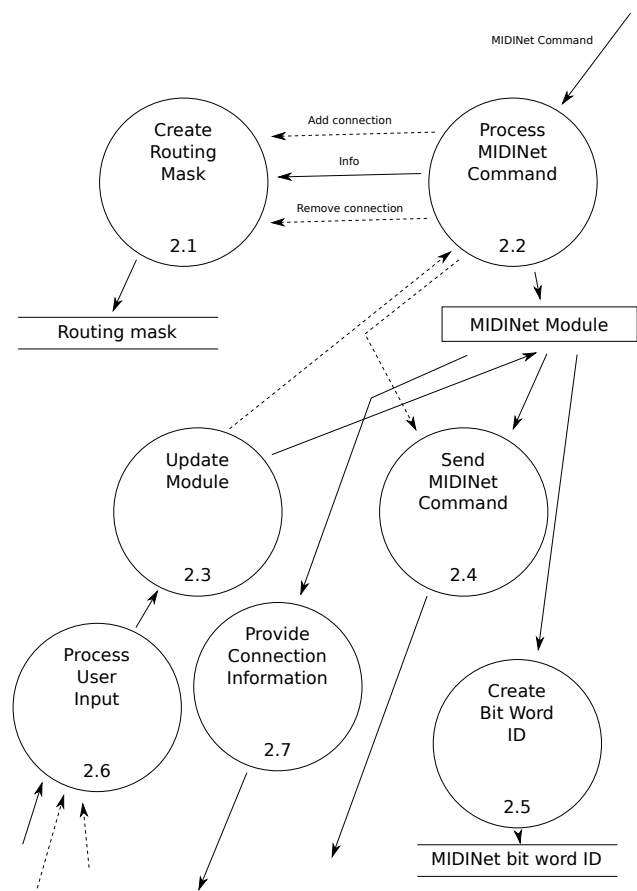


Figure B.4: Task 2 - Manage connections.

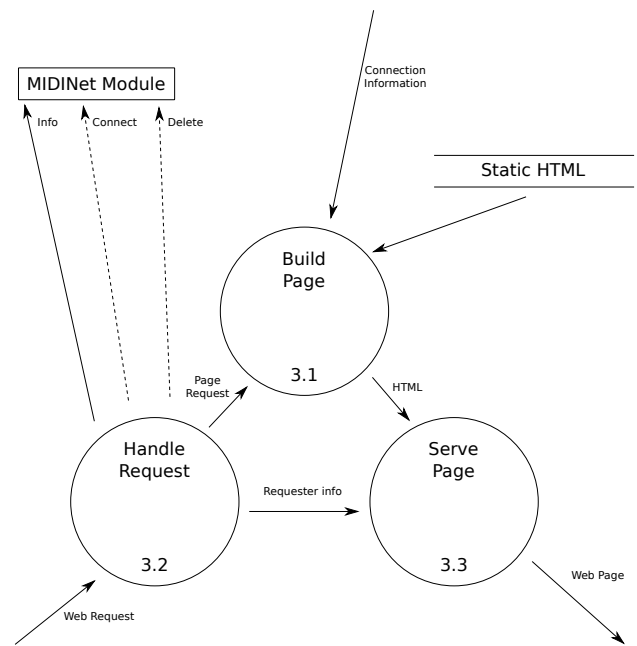


Figure B.5: Task 3 - Provide User Interface.

B.1.3 Level 2

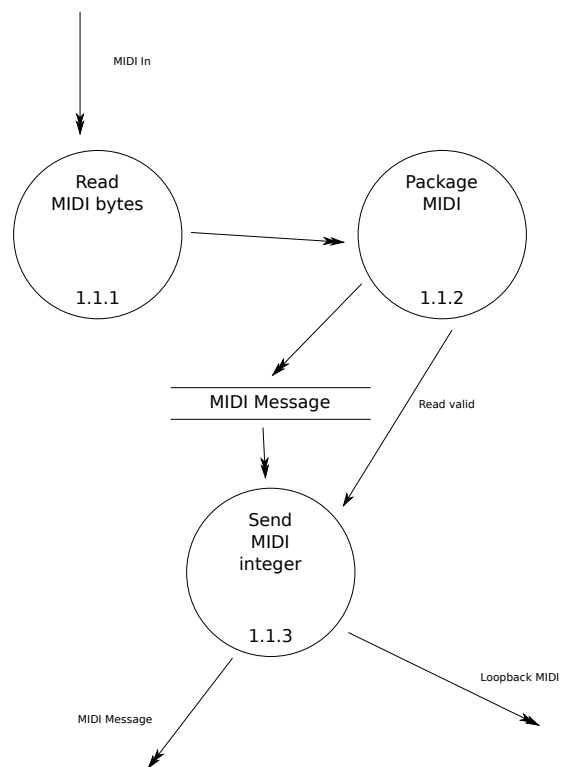


Figure B.6: Task 1.1 - Read MIDI In.

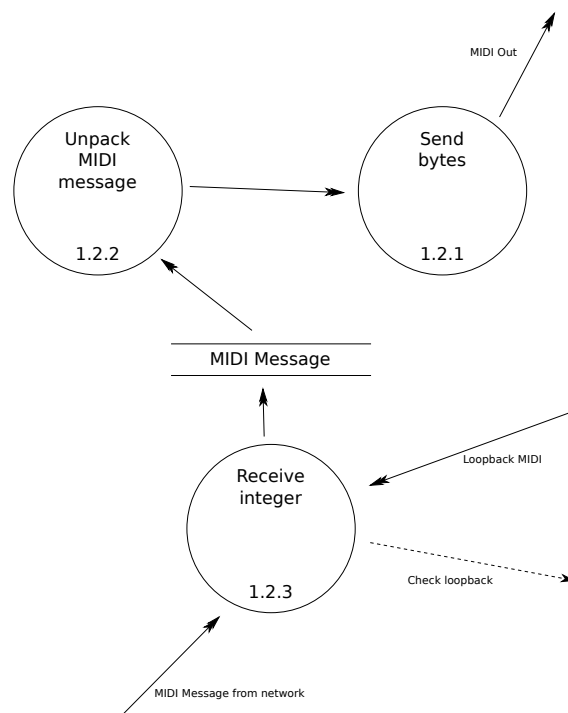


Figure B.7: Task 1.2 - Send MIDI Out.

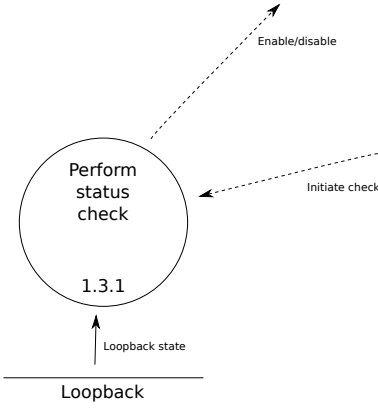


Figure B.8: Task 1.3 - Check Loopback flag.

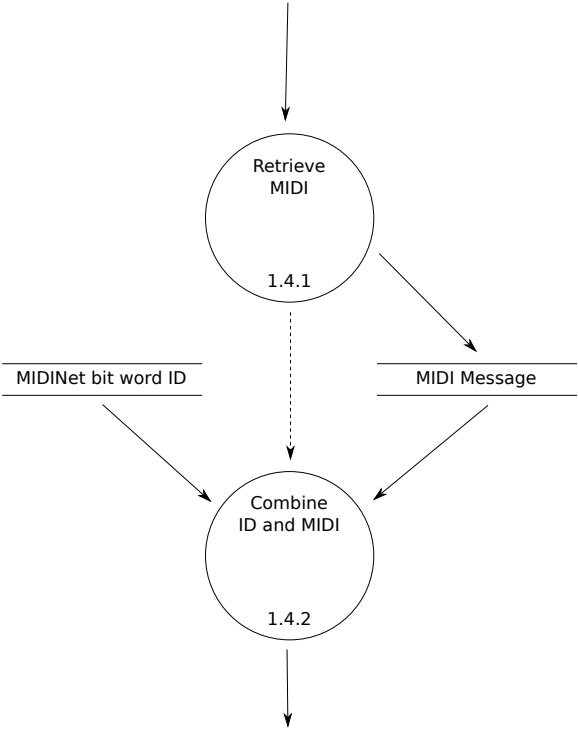


Figure B.9: Task 1.4 - Add Unit ID.

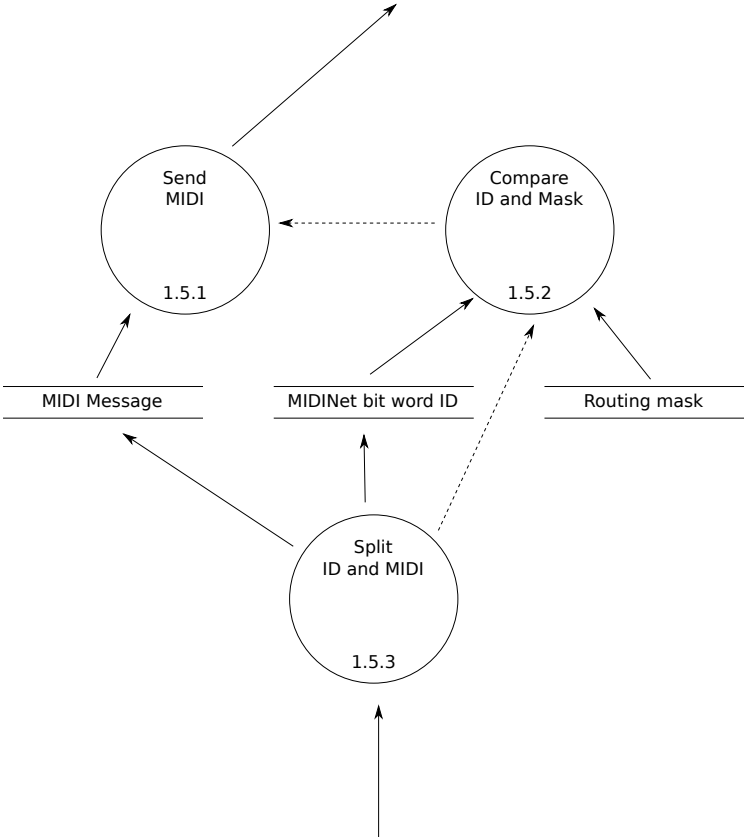


Figure B.10: Task 1.5 - Route MIDI.

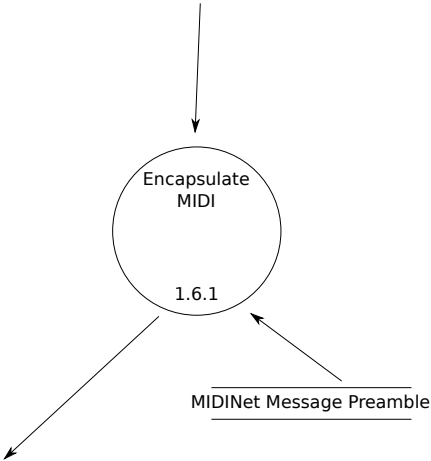


Figure B.11: Task 1.6 - Send MIDINet Message.

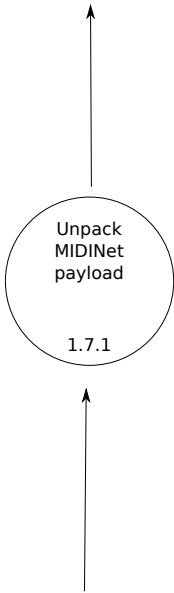


Figure B.12: Task 1.7 - Receive MIDINet Message.

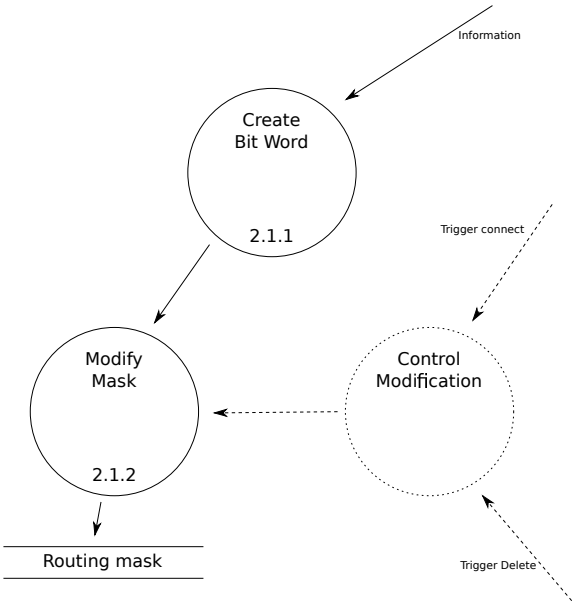


Figure B.13: Task 2.1 - Create Routing Mask.

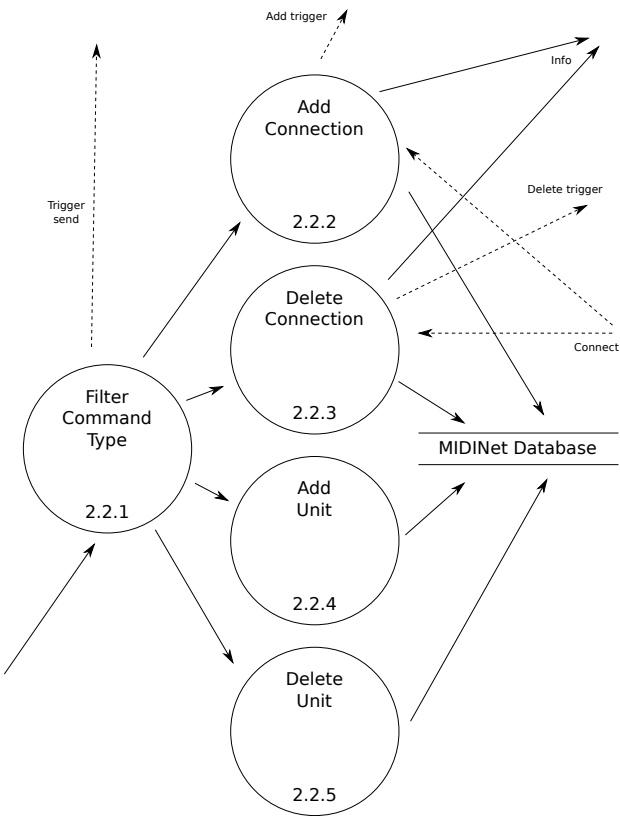


Figure B.14: Task 2.2 - Process MIDI Command.

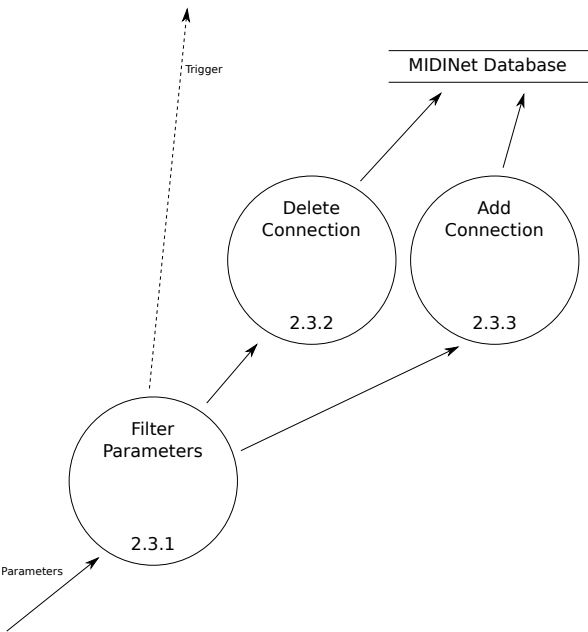


Figure B.15: Task 2.3 - Update MIDINet system.

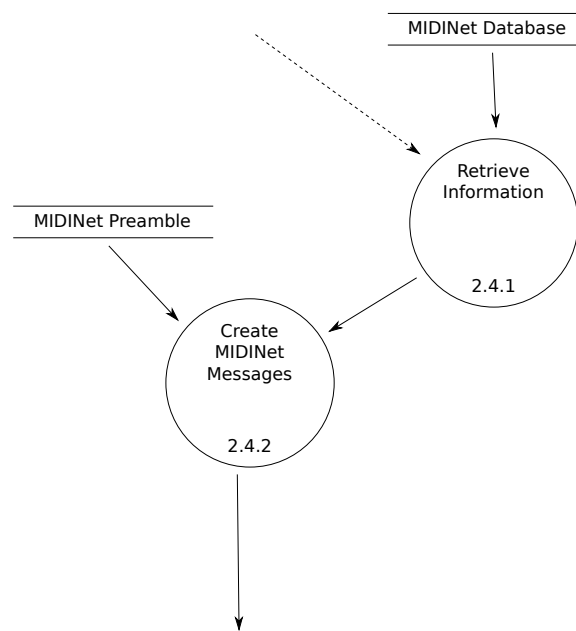


Figure B.16: Task 2.4 - Send MIDINet Command.

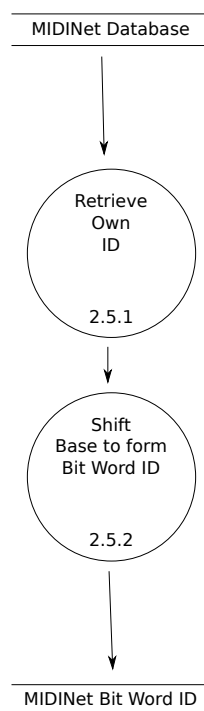


Figure B.17: Task 2.5 - Create ID.

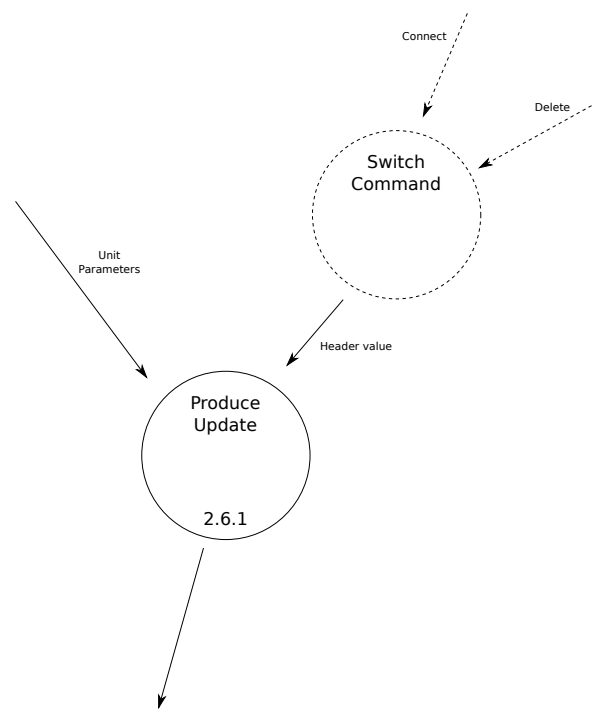


Figure B.18: Task 2.6 - Process User Input.

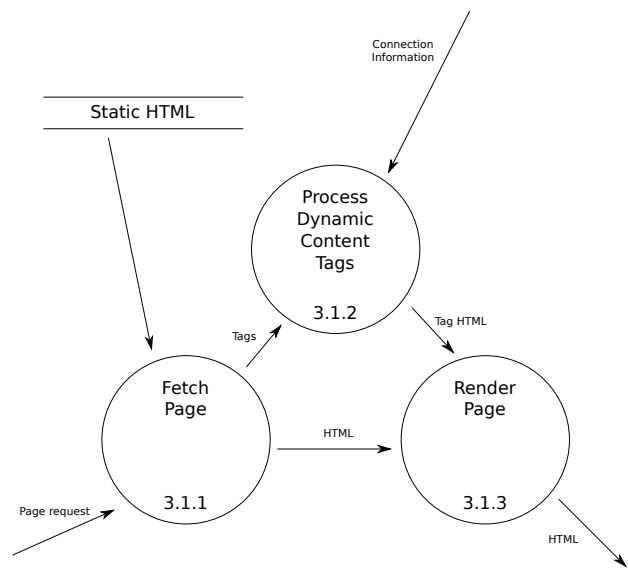


Figure B.19: Task 3.1 - Build page.

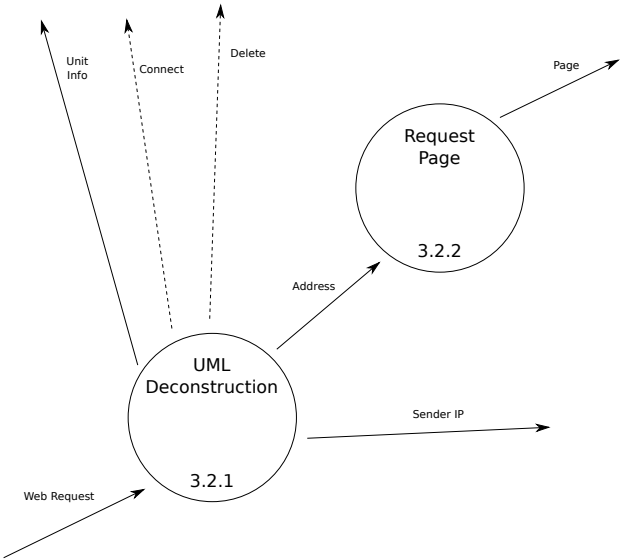


Figure B.20: Task 3.2 - Parse request.

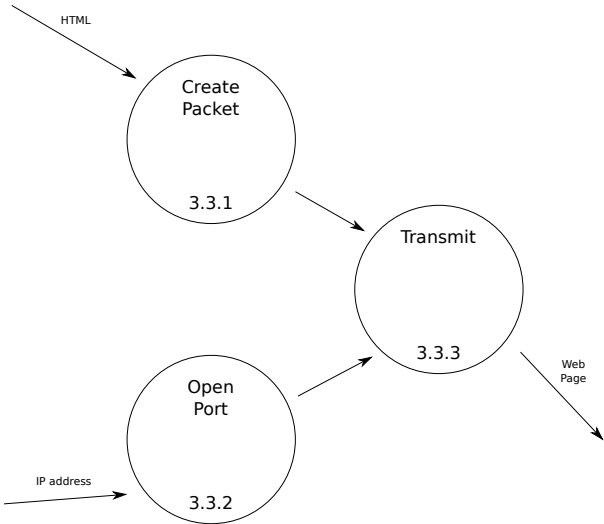


Figure B.21: Task 3.3 - Serve Page.

B.2 ConMan: Implementation Model

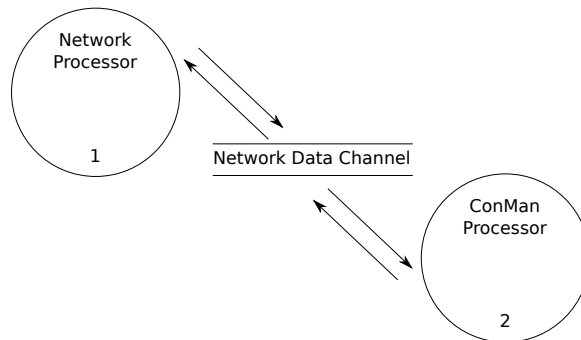


Figure B.22: System Processor View

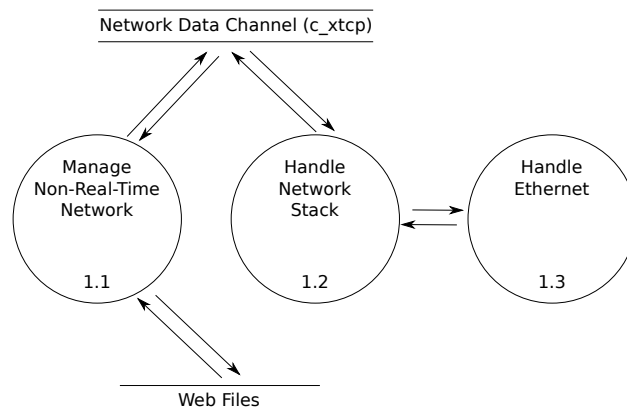


Figure B.23: Overview of the parallel task structure for Network Processor

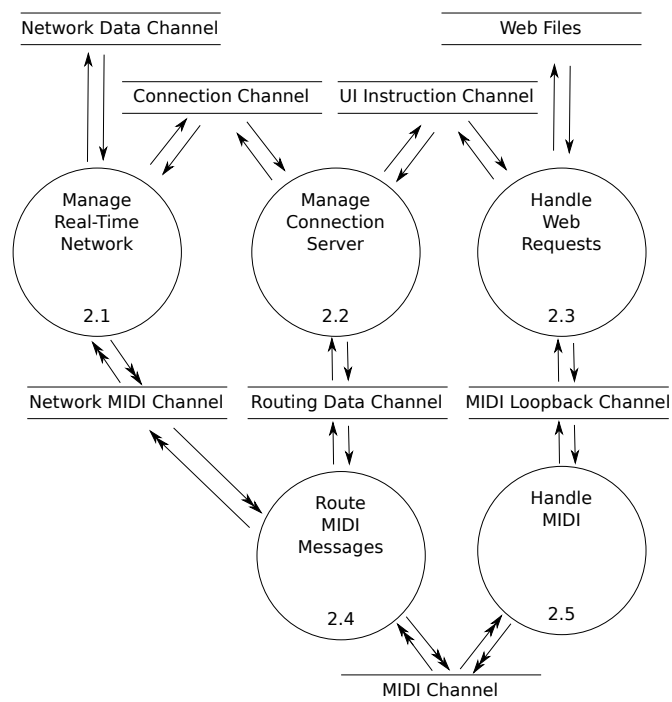


Figure B.24: Overview of the parallel task structure for ConMan Processor

B.2.1 ConMan Processor

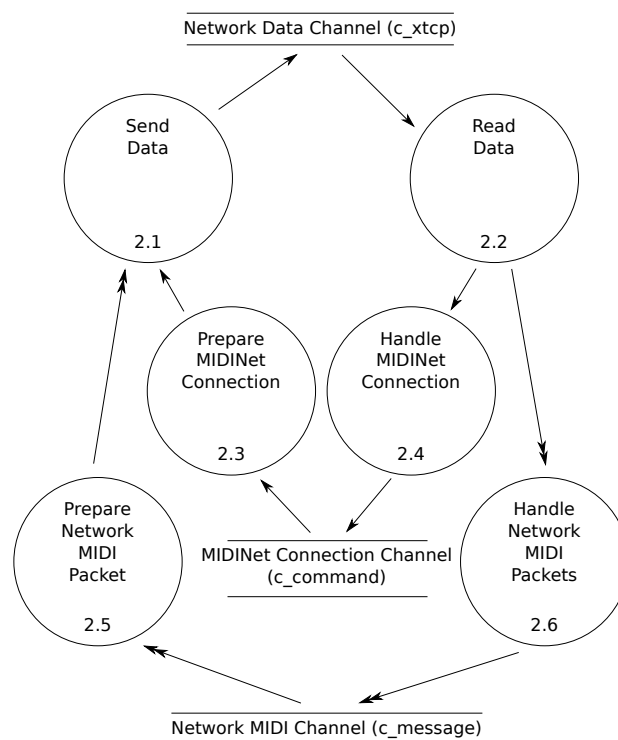


Figure B.25: Task 2.1 - Manage real time network

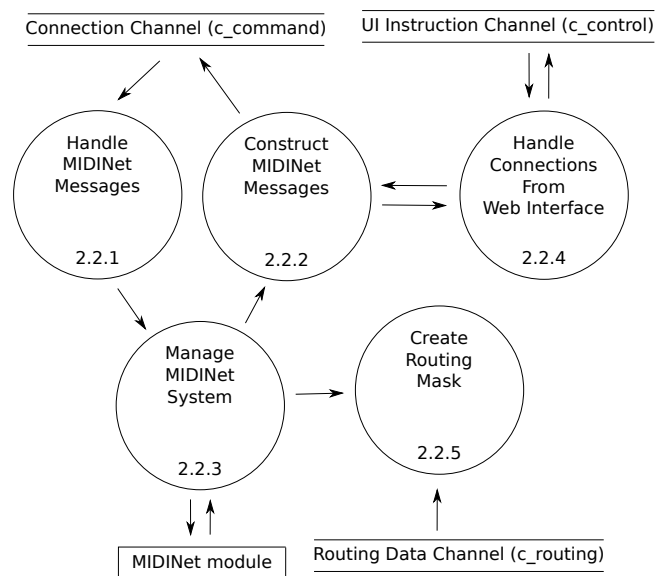


Figure B.26: Task 2.2 - Manage Connection Server

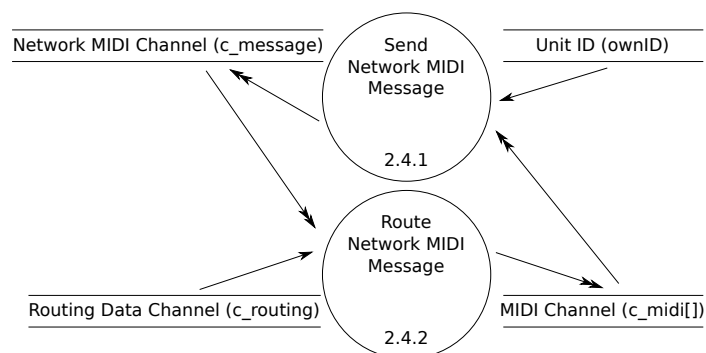


Figure B.27: Task 2.4 - Route MIDI Messages

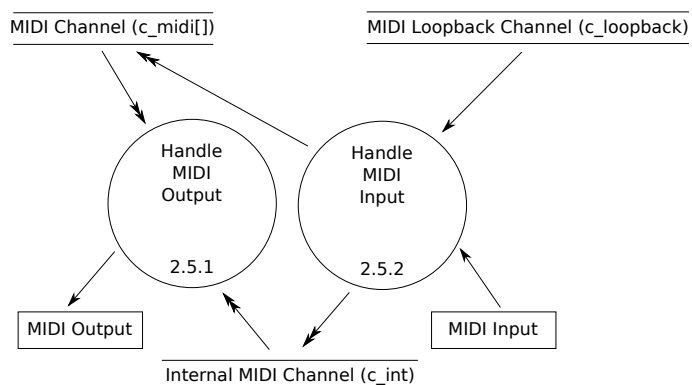


Figure B.28: Task 2.5 - Handle MIDI Ports

B.2.1.1 Handle MIDI Ports

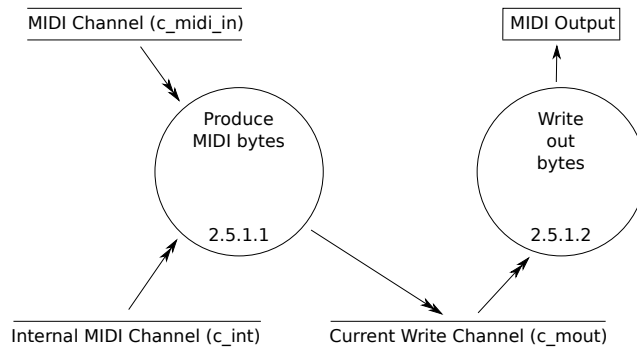


Figure B.29: Task 2.5.1 - MIDI Output handling

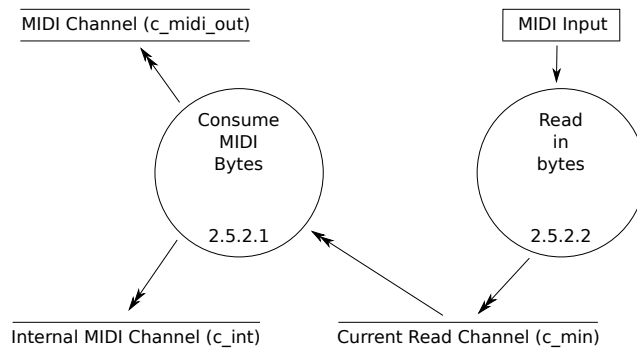


Figure B.30: Task 2.5.2 - MIDI Input handling

B.3 MIDINet: Object-Oriented Design

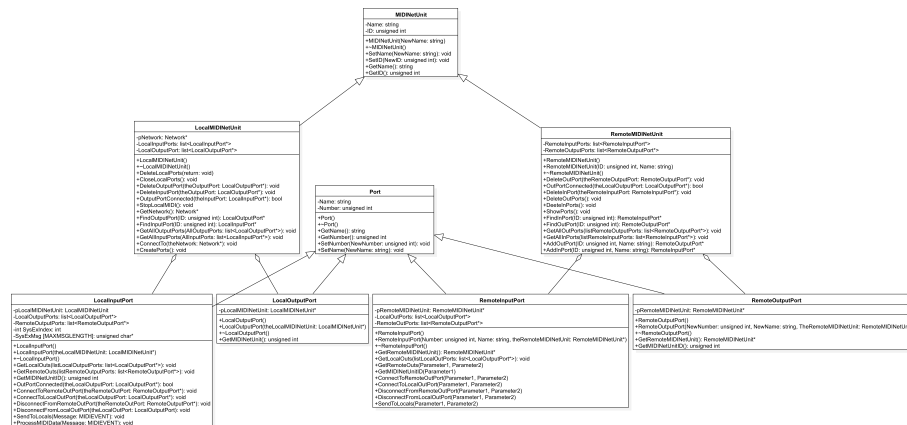


Figure B.31: MIDINet classes within the Connection Server component.

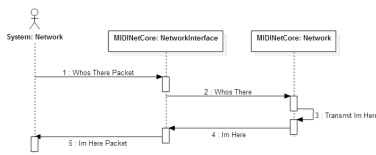


Figure B.32: Request Configuration Sequence Diagram.

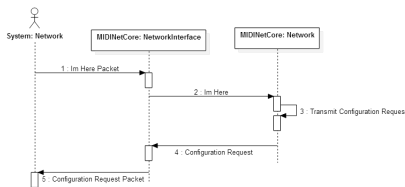


Figure B.33: Respond to Configuration Request Sequence Diagram.

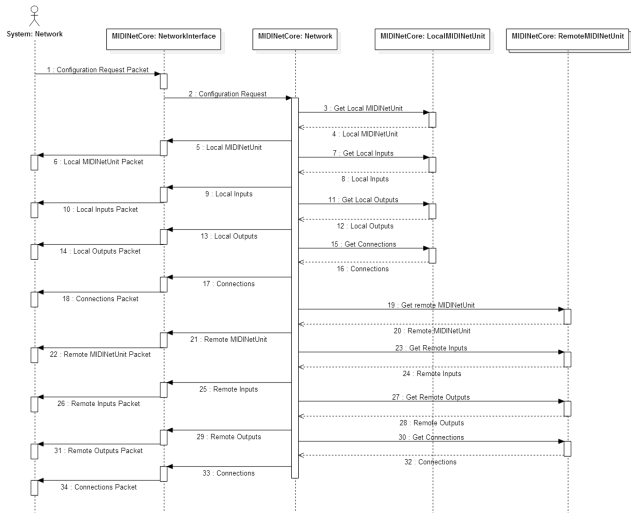


Figure B.34: Provide Configuration Sequence Diagram.

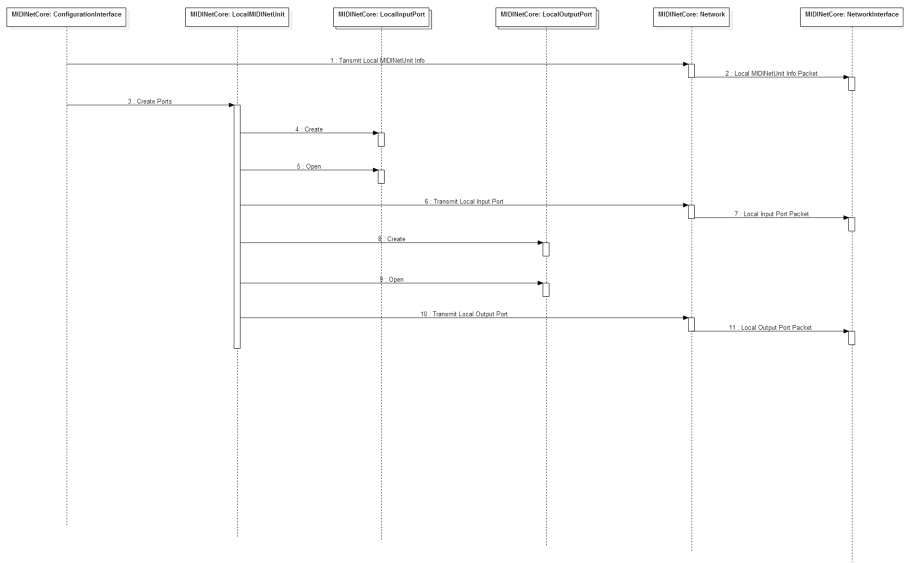


Figure B.35: Transmit Local MIDINet Unit Info Sequence Diagram.

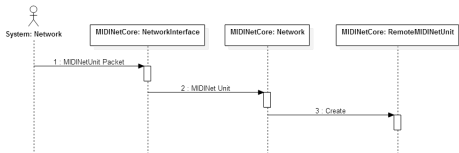


Figure B.36: Add MIDINet Sequence Diagram.

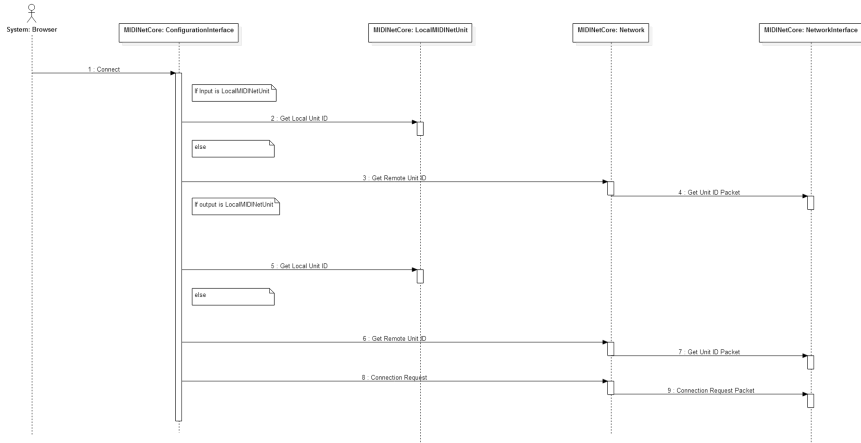


Figure B.37: Make Connection Sequence Diagram.

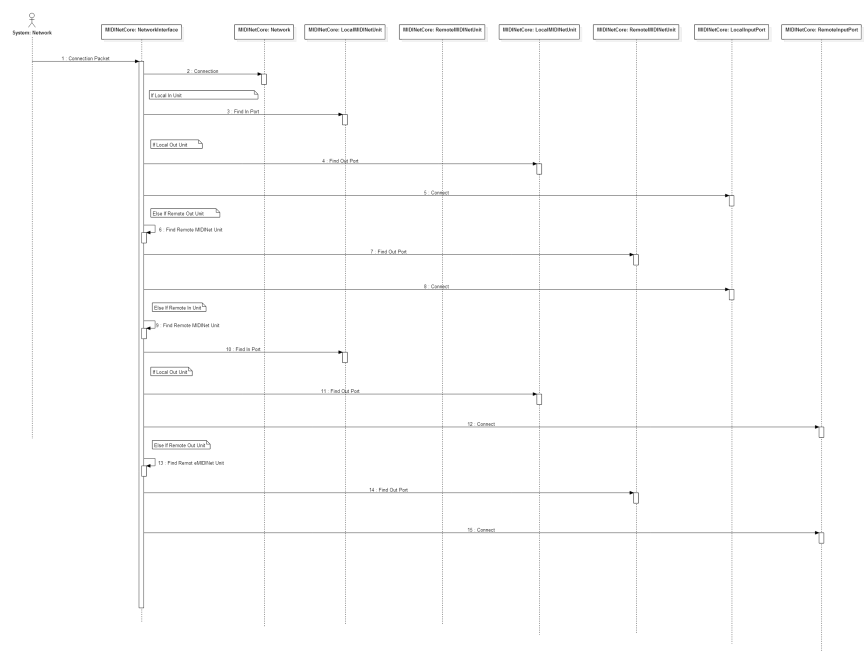


Figure B.38: Add Connection Sequence Diagram.

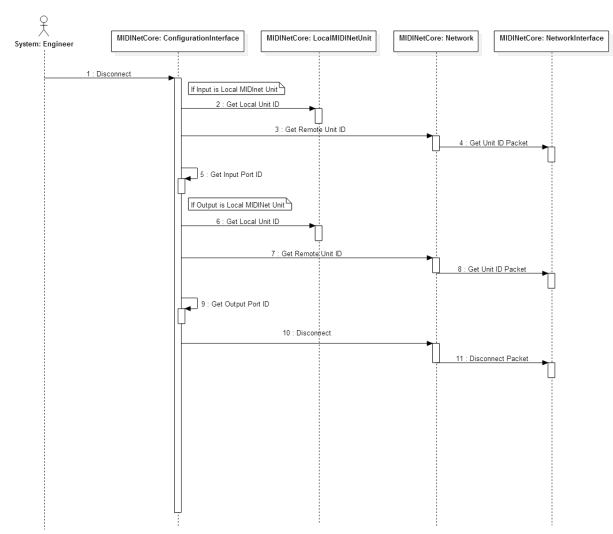


Figure B.39: Destroy Connection Sequence Diagram.

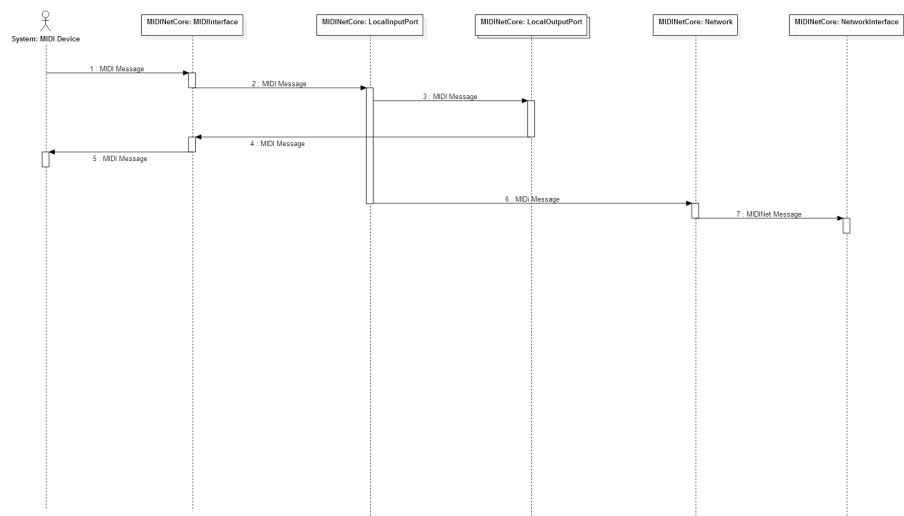


Figure B.40: Transmit MIDINet Sequence Diagram.

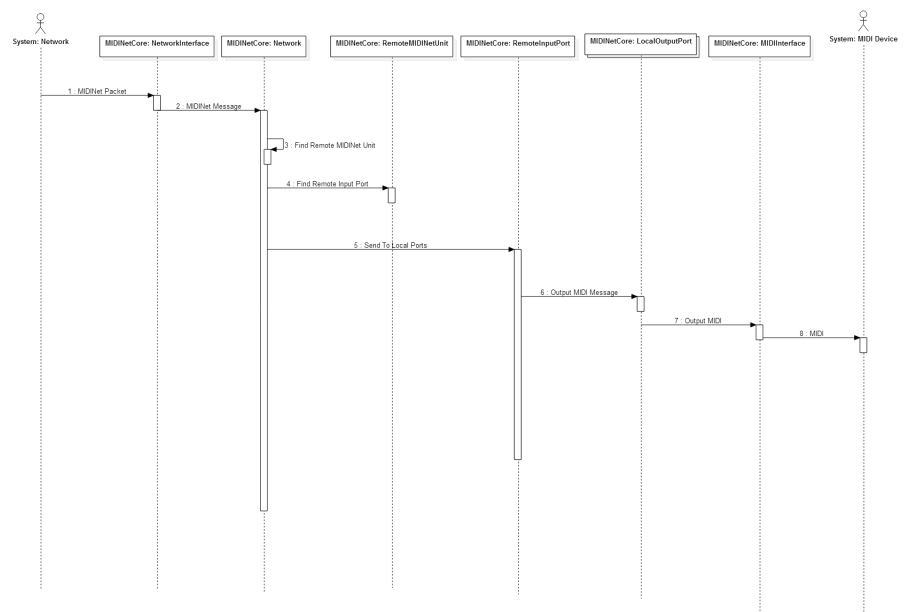


Figure B.41: Route MIDINet Message Sequence Diagram.

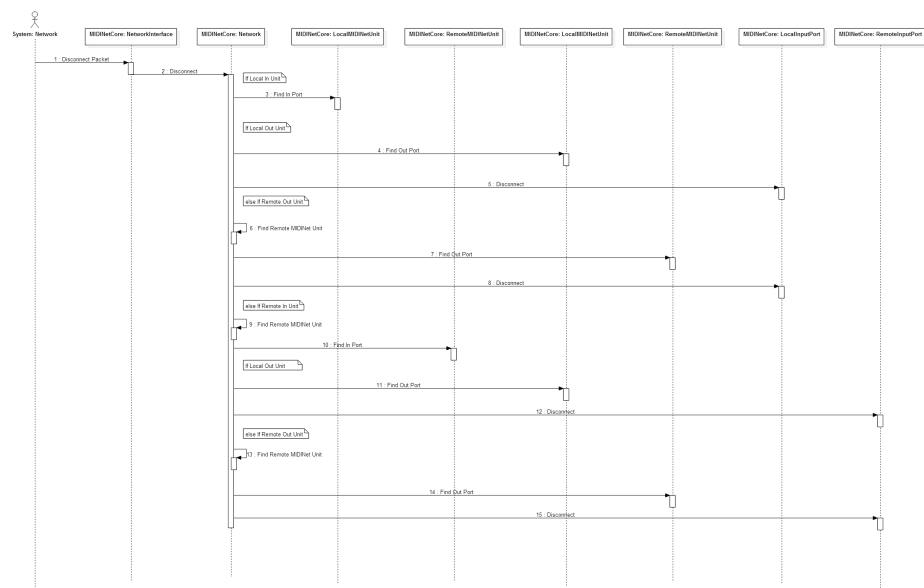


Figure B.42: Network Disconnect Sequence Diagram.

Appendix C

MIDINet test system hardware

The list below contains the hardware used in the desktop computers used to test the MIDINet system. Two machines with identical hardware were used in the testing.

Motherboard Foxconn 45CSX

CPU Intel Atom 230

RAM 2Gb DDR2 800MHz

Hard drive Western Digital 80Gb

NIC D-Link DGE-528T Gigabit PCI Card