Innovation through MIDI: Developing Windows MIDI Applications

Andrew Eales
Central Institute of Technology
Upper Hutt, New Zealand
andrew.eales@cit.ac.nz

ABSTRACT

MIDI capability is a standard feature of audio hardware on Intel platforms. Interesting and innovative MIDI applications running under Microsoft Windows can be developed using the multimedia capabilities of Windows. This paper examines Windows MIDI development, and mentions the non-trivial difficulties associated with developing robust MIDI applications. Two interesting applications of MIDI software are also discussed.

KEYWORDS

MIDI, Windows Programming, Audio Applications.

1. INTRODUCTION

MIDI (Musical Instrument Digital Interface) is a standardized, real-time communications protocol that facilitates communication between hardware devices that support MIDI. Communicating devices typically include computer sound cards, synthesizers, recording equipment, lighting systems and stage machinery. Standard commands that perform common tasks associated with these devices form the core of the MIDI protocol. However, MIDI is an open-ended protocol, supporting the transmission of user-defined data. This feature provides unlimited possibilities for the development of unusual MIDI-based systems. Application areas include robotics, spatial sound (Eales, 1994), animation (Riemersma, 1999) and real-time process control (Stevens, 1997). Another area of interest is the sound produced by abstract mathematical worlds such as cellular automata (Eales, 1996) and biological worlds (Dunn, 1997). This paper does not discuss the extensively documented MIDI standard, but focuses on the development of Windows MIDI applications and the innovative application of MIDI. Detailed descriptions of MIDI are provided by (Lehrman, 1993), and Rona (1994). Extensions to MIDI are discussed by Selfridge-Field (1998).

2. THE MIDI PROTOCOL

MIDI is a serial communications protocol that operates at 31.25K baud. This protocol consists of a variety of predefined messages that accomplish different tasks, such as playing a single note on a synthesizer

Byte1 (MSB)	Byte2	Byte3	Byte4 (LSB)
Not used	Data2	Data1	90H (All values in Hex)
	Velocity	Pitch	NoteOn (9nH) Channel 1 (0H)

Figure 1

or exchanging data blocks between MIDI devices. Two broad categories of MIDI messages exist. The first category consists of commands made up of one, two or three bytes of data. Typically, a status byte defining the type of message and one or two data bytes that serve as parameters to the message. The status byte consists of a four-bit command and a four-bit channel assignment, as MIDI uses sixteen distinct communication channels. As an example, a *NoteOn* message, which plays a single note on a synthesizer, is represented by four bytes as shown in Figure 1.

Pitch is encoded as a keyboard key number and velocity affects the loudness of a note. Sending the four bytes to a MIDI soundcard plays a middle-C note on channel one.

A second category of MIDI messages, *system exclusive* (*sysex*) messages support user-defined data by allowing any number of data bytes to be transmitted between a start and an end byte represented by F0H and F7H. System exclusive messages are typically used to transfer variable-length data blocks between a computer and a MIDI device.

3. DEVELOPING WINDOWS MIDI APPLICATIONS

Windows 3.11 and later versions of Windows provide a uniform software layer that MIDI hardware can communicate with using hardware-independent virtual device drivers. There are three different ways to implement Windows MIDI applications, by using the Media Control Interface (MCI) functions, using MIDI streams, or via the low-level Application Programmer's Interface (API) functions.

MCI functions encapsulate the low-level multimedia API providing simple solutions to using MIDI. Visual development environments such as Visual Basic, Delphi and C++ Builder provide multimedia components that use MCI functions. Streams are a recent addition to Windows, included with Windows98 and supported by later versions of Windows95. Streams provide a compromise between the limited functionality of the MCI and the programming difficulties associated with the low-level functions. Streams provide most of the functionality of the low-level API but do not support timestamped input of MIDI data. Robust applications that require timestamped input and output must use the low-level API. Support for MIDI is gradually appearing in Microsoft's DirectMusic, which may become the standard for future MIDI development.

3.1 Basic MIDI Input and Output

Low-level API functions are documented in the Microsoft SDK, which also exists as a set of help files shipped by compiler vendors with their products. Despite recent improvements, the SDK is not a clear introduction to MIDI programming. The event-driven architecture of Windows does not allow the polling of input ports. Windows sends an application a message or *callback* when a specific event occurs. A callback window procedure or a callback function process Windows messages for an application. Standard event-handlers such as *OnMouseDown* that use a callback window procedure are familiar to Windows programmers. Windows posts a message to a callback each time a complete MIDI message is received from an input port. To read MIDI data from an input port:

1 Determine the number of input ports available to Windows using *GetNumIntputDevices()*
2 Define a callback window or function.
3 Open an input port using *MidInOpen(. .)* and attach the callback to the input port by specifying the callback as one of the parameters.
4 Respond to the callback message, which will be a MIM_DATA message when the input port receives a complete MIDI message.
5 Close the input port before the application terminates with *MidiInClose()*

The specified callback handler will also be sent messages when the port is opened and closed. The output of MIDI data is easier as the process does not use a callback procedure. A port is opened and then a byte string sent to the output port.

3.2 Timestamped Data

Real-time applications such as musical performances, lighting sequences or the labeling of tomato cans require exact timing information for the sequencing of events. Windows95/98 provides multi-tasking using separate execution threads. Unfortunately, this design decision means that real-time applications cannot obtain accurate timing information (Messick, 1998). Code that attempts to obtain the time may become delayed in an application's message queue, or may have to wait for other threads to finish executing.

3.2.1 The Windows 3.11 Legacy

Windows 3.11 is a sixteen-bit operating system, designed to run on Intel processors having a word size of sixteen bits. Windows95, WindowsNT and all later versions of Windows provide thirty-two bit environments. However, Windows95 and 98 have retained the sixteen-bit multimedia subsystem of Windows 3.11. Only WindowsNT/2000 are full thirty-two bit operating systems. The sixteen-bit multimedia extensions provide a timer that operates using processor interrupts guarded by a mutual exclusion semaphore to provide extremely accurate timing information. Unfortunately, thirty-two bit code and sixteen-bit code cannot communicate directly, as sixteen-bit memory addresses must be converted to thirty-two bit memory addresses and vice-versa. A *thunk* layer that translates code between these different word sizes is required. Thunking is described by the Microsoft SDK, and the use of a thunk to provide access to the sixteen-bit multimedia subsystem is described by Messick (1998).

The correct sequencing of MIDI data is the programmer's responsibility with one exception; Windows will timestamp incoming MIDI data that is handled by a function callback. The timestamped input data must usually be processed by a window procedure for the data to be used effectively by an application. Although the timestamped data may be delayed in a message queue, the timestamping will be accurate.

3.3 System Exclusive Messages

The transfer of system exclusive data requires different low-level functions from the API functions used to transfer fixed-length messages, as the size of data blocks is the critical factor. Sysex data does not require timestamping as the data is processed as soon as it is received. System exclusive output requires a MIDIHDR structure to be created which provides the size and address of a data block to Windows. System exclusive input requires a callback procedure to process data buffers filled by Windows from the input port.

Descriptions of the various options and excellent code examples are provided by Messick (1998), and by various contributors to 'The MIDI Fanatics Brainwashing Centre' referenced at the end of this paper.

4. INNOVATIVE MIDI APPLICATIONS

The creative use of MIDI is only limited by the availability of hardware and the imagination. Two examples of interesting applications are spatial sound processing and the sonification of abstract and real-world processes.

4.1 Spatial Sound

Projecting sounds into a virtual space is of interest to composers, researchers in acoustics and the designers of virtual reality systems. Multiple loudspeakers are used to create a virtual sound space. Audio signals are mixed and routed to the different loudspeakers to create the illusion of movement within the virtual sound space. A dedicated hardware unit at Rhodes University that mixes and routes analog audio under MIDI control is documented by Wilkes (1991). MIDI software that controls sound within a virtual space is discussed by Eales (1994).

An interesting possibility provided by the recent increase in network performance is to use an entire computer laboratory as a spatial sound system. Such a system can be programmed using C++ or JavaMIDI (Marsanyi, 2001), where one or more server applications control the routing of MIDI data to client

machines. Clients can translate MIDI data into audio using standard soundcards installed on the client machines. Support for distributed applications makes Java a suitable language for such an application.

Future computer sound cards are likely to facilitate MIDI control of the digital audio, providing cost-effective environments for spatial sound systems.

4.2 The Sonification of Real and Abstract Worlds

MIDI can be used to investigate the sounds formed by data and relationships that exist in various real and abstract worlds. These worlds include descriptive data or organisational relationships that occur within chemistry, mathematics, artificial life and biology. Descriptive data or data from any process can be converted into MIDI data and performed in real-time. The sounds of cellular automata have been explored by the author (Eales, 1996), and the sounds of protein synthesis is described by Dunn (1997). Analysis of the sound of complex systems may lead to new insights regarding the organisation of such systems.

5. CONCLUSIONS

MIDI is an extremely versatile protocol due to the freedom provided by system-exclusive messages. Dedicated hardware such as the audio patcher-mixer unit can be designed to operate under MIDI sysex control. MIDI has been ignored as a development environment for process-control because of the absence of error-checking and error-correcting mechanisms. However, a MIDI link between two computers provides over a thousand events per second and has been shown to operate with no event loss. (Stevens, 1997).

Developing software that uses MIDI to control hardware or other software is a fascinating experience. Unfortunately, developing MIDI software from scratch is a difficult undertaking. Applications that only require the playback of MIDI files should use the MCI commands. Application requiring synchronized output can be developed using the MIDI stream API. Robust applications requiring input and output synchronization are forced to use the low-level API. Accurate timing under Windows95/98 requires a thunk layer to access the sixteen-bit multimedia subsystem. The MaxMIDI toolkit developed by Messick (1998) implements a thunk layer and provides all of the source code used by the toolkit. Resources such as the MaxMIDI toolkit and JavaMIDI (Marsanyi, 2001), solve the non-trivial implementation problems and provide valuable examples of advanced Windows programming techniques.

The two application areas of MIDI that were mentioned provide examples of unusual, innovative application areas that can be explored using the MIDI protocol.

REFERENCES

Dunn, J., Clark, M.A. Life Music: The Sonification of Proteins. Leonardo Online, 1997. Accessed 21 September 2000. <http://mitpress.mit.edu/e-journals/Leonardo/isast/articles/lifemusic.html>

Eales, A.A. A Windows SurroundSound System B.Sc.(Hons) Research Project, Rhodes University Dept. of Computer Science, 1994.

Eales, A.A. **Dances of Life:** Studies in Cellular-Automated Music. Part of unpublished M.Mus. Thesis, University of Pretoria, 1996.

Lehrman, P.D. and Tully, T. MIDI for the Professional. Amsco Publications, New York, 1993.

Marsanyi, R. JavaMIDI Accessed May 7, 2001. <http://www.softsynth.com/javamidi/>

MIDI Fanatics Brainwashing Centre Accessed May 7, 2001. <http://www.borg.com/~jglatt/tech/winapi.htm>

Messick, P. Maximum MIDI: Music Applications in C++. Manning Publications, Greenwich, 1998. also <http://www.maxmidi.com>

Riemersma, T. Synchronizing Animation with MIDI Audio, 1999. Accessed August 16, 2000. <http:\\www.compuphase.com>

Rona, J. The MIDI Companion Hal Leonard Publishing, 1994.

Selfridge-Field, E. (ed) Beyond MIDI: The Handbook of Musical Codes. MIT Press, 1997.

Stevens, A. Space Shuttles, Tomato Cans, and Teenage Daughters.Dr. Dobbs Journal, vol22 no2, February, 1997. p.97

Wilkes, A. An Audio Patcher-Mixer. Department of Computer Science Technical Report.Rhodes