control a polyphonic synth patch with long decays, as I did, you can have an instant melody and chord machine. Performing with a Tibetan and a Hindustani singer, both of whom used rather interesting just-intonation modes, suggested to me that having microtonal performance capabilities would be a good thing. Especially with Tenzin, I was able to match the pitches of his mode, and ornament what he was singing several octaves higher, something that without the instant scale-making possibilities of Scala, I would never have been able to do. The two scales I used most frequently in the Cathedral performances were both 6 by 6 Euler-Fokker Matrices, one consisting of horizontal 7/4s and vertical 9/8s, and the other consisting of horizontal 3/2s and vertical 5/4s. For anyone interested in microtonal performance on a laptop, I would enthusiastically recommend that they download Scala (PC now, Mac “soon”), and explore its many possibilities.

I’m finding this kind of improvisatory performance, where the use of interactive and algorithmic computer software is central, but not the whole story, very exciting. It allows me to both perform electroacoustically, but also to do other things. Further, it allows me to exercise a level of theatricality that I enjoy, adding that to the palette of resources available for electroacoustic performance. Hopefully, I’ll be doing a lot more of this kind of performing in the future. Anyone want to jam?

Websites:
Audiomulch: www.audiomulch.com

Softstep: http://algoart.com

Cathedral:
www.monorestreet.com/Cathedral

CrusherX: www.crusher-x.de

Scala: www.xs4all.nl/~huygensf/scala

Writing a Device Driver using MIDI System Exclusive Messages:

Angelo Fraietta
angelo_f@bigpond.com

Designing your own MIDI firmware can be a daunting task, particularly if your clients have to communicate with your hardware using MIDI System Exclusive (sysex) messages. Additionally, you could use SYSEX messages to communicate between two or more machines running MAX as a novel way of communicating with your patches. If you are writing firmware for MIDI, *Synthesizer Performance and Real-Time Technique* by Jeff Pressing is a great starting point for understanding the MIDI protocol. To gain a greater understanding, you can purchase the latest MIDI specification from the Midi Manufacturers Association (MMA) at http://www.midi.org/. Alternatively, you can do what I did and get the Midi Specification 1.0 for free at www.midi-classics.com/midispec.txt.

I will use the SYSEX protocol that I wrote for the Midi Controller that I presented at ACMC as an example, which in turn could empower someone to write a patch editor for it in Max.
Note: I will be writing the numbers in hexadecimal, using `0x` as a prefix to signify that the number is in hexadecimal. For example `0x81` is ($8 \times 16^1 + 1 \times 16^0$), which is equal to decimal 129. The reasons for hexadecimal notation are that the decimal value is irrelevant to the protocol, and it is easier to encode and decode using this protocol, as we may be manipulating one bit of that value

Why write device drivers?

The concept of the MIDI sysex message is that a host device can communicate information not specified by other MIDI message types to target device. For example, if you want to reconfigure your device to echo incoming MIDI data to its MIDI output (independent of the MIDI through output), what MIDI message would you send? The available types of MIDI messages would be unsuitable (although you could program your device to respond to a particular standard MIDI message). The most effective way to communicate this information would be to send your own configuration message over the MIDI cable. For example, let us say that the value of EEPROM address 1 in your hardware device determines whether it echoes the data to its output port, we have to communicate with our hardware that we need to change the value of EEPROM address 1. Using a device driver enables us to focus on the issue of communicating that we want to change the value at an EEPROM address without having to concern ourselves with what the physical transmission allows or disallows. This, effectively, is a buffer that enables you to write highly cohesive code that has a low coupling or dependency on other code modules. The device driver that I have written (and provided) ends up being two functions that encode and decode a data stream.

When writing data communications protocols, the most effective way is to start from the top layer (application) and work your way down to the lower layer (physical connection). The reason for this is that at the application level – e.g. value of EEPROM data at address 1 – it does not need to know anything about MIDI. The layers each communicate at their own level and are only able to decode information at their own level. In order to make the physical connection the information from the source is encoded and then passed down to a lower layer (closer to the hardware) and encoded by that layer. This will not be decoded until it reaches that same layer in the target device. This continues on until the information has traveled to the actual hardware connection (DIN plugs with a 5mA current loop), each layer adding its own encoding around the information encoded by the previous layer. The device driver in the source device encodes information from an upper layer, encoding it in order that it can be transmitted using the MIDI protocol. In the target device, it receives information from the MIDI protocol, decodes what the device driver in the source had encoded, and passes it to the upper layer. The following sequence describes the parts of the process that concern us:

Source Device

Upper Layer needs to send the following sequence:
```
0x00 0x80 0x70 0x00
```

(note that this sequence is illegal in MIDI as 0x80 has the MSB set, and therefore cannot be sent as part of a SYSEX message)

Upper SYSEX driver encodes information and passes to Lower SYSEX driver
```
0x00 0x01 0x00 0x70 0x00
```

Lower SYSEX driver adds manufacturer ID and our own device ID (first 2 bytes)
```
0x7D 0x01 0x00 0x01 0x00 0x70 0x00
```

Data packed into SYSEX message (first and last byte)
```
0xF0 0x7D 0x01 0x00 0x01 0x00 0x70
0x00 0xF7
```

Data sent out of MIDI port

Target Device

Data received from MIDI port
```
0xF0 0x7D 0x01 0x00 0x01 0x00 0x70
0x00 0xF7
```

Determined a SYSEX message and sent to Lower SYSEX driver
0x7D 0x01 0x00 0x01 0x00 0x70 0x00

Lower SYSEX driver removes manufacturer ID and our own device ID and passes it to our Upper SYSEX driver.
0x00 0x01 0x00 0x70 0x00

Upper SYSEX driver decodes data and passes to upper layer
0x00 0x80 0x70 0x00

You will notice that the device driver was split in two –Upper and Lower. This has been done so a different manufacturer can use the upper layer and modify the lower layer, using their own manufacture ID and other machine specific data without modifying the encoding / decoding algorithm.

Lets take a closer look at what actually happened in the decode / encode stages.

The original data presented was:
0x00 0x80 0x70 0x00

We stated that 0x80 was invalid as a SYSEX data byte. We need to convert 0x80 into two bytes that do not have the MSB set (i.e. the number must be 0x7F or less). We do this by “byte stuffing” the unacceptable character. This is accomplished by defining a control character that notifies the decoder that the byte following requires decoding. We accomplished this by using the character 0x01 as a control character and then encode the data byte by clearing the MSB. The byte 0x80 therefore becomes a two byte sequence: 0x01 0x00. When the decoder sees the 0x01, it knows that it must set the MSB of the following byte. This introduces a second problem: what if we need to send 0x01 as a data byte. e.g. the required data is:

0x01 0x80

We overcome this by creating another control character that notifies the decoder that the following character does not require decoding. In this case, I have used the control character 0x02 to signify that the character following it does not require decoding. Data byte 0x01 therefore becomes two bytes: 0x02 0x01. When the decoder sees the 0x02, it knows that it is not a data byte, but a control character that signifies that the following 0x01 is not a control character. So what happens if we want to send 0x02 as a data byte? We byte stuff it with the same control character that we used to byte stuff 0x01—we place a 0x02 in front of it. Data byte 0x02 therefore becomes two bytes: 0x02 0x02. We now have two control characters: 0x01, which signifies that the following character requires the MSB to be set; and 0x02, which signifies that the following character is not a control byte.

Now that we have that sorted, let us encode and then decode a series of data bytes for transmission in a MIDI SYSEX stream.
0x01 0x20 0x00 0x81 0x00 0x02

I will show in bold the characters that require encoding
0x01 0x20 0x00 0x81 0x00 0x02

Now encode them
0x02 0x01 0x20 0x00 0x01 0x01 0x00 0x02 0x02

We can now decode the stream. In the following table, each row signifies a change of Machine State, occurring as a result of the incoming byte shown in the Current Byte
column. The current control character value is stored in the Current Ctl val column. If the Current Byte is a control character, this value is stored as the current control character in the next row. The action performed on the current byte is determined by the value of the current control character.

Now let us decode them one character at a time

Current Byte	Current Ctl val.	Comment	Decoded bytes
0x02		0x02 is a control character. The next character is not a control character and does not require decoding	
0x01	0x02	0x01 is not a control character in this instance because the current Ctl val. is 0x02. data value = 0x01	0x01
0x20		data	0x01 0x02
0x00		data	0x01 0x02 0x00
0x01		0x01 is a control character. The next byte requires its MSB set	0x01 0x02 0x00
0x01	0x01	0x01 is not a control character because the current Ctl val. is 0x01. data byte requiring MSB set 0x81	0x01 0x02 0x00 0x81
0x00		data	0x01 0x02 0x00 0x81 0x00
0x02		0x02 is a control character. The next character is not a control character and does not require decoding	0x01 0x02 0x00 0x81 0x00
0x02	0x02	0x02 is not a control character in this instance because the current Ctl val. is 0x02. data value = 0x02	0x01 0x02 0x00 0x81 0x00 0x02

We can see the final result is that which we started at before we encoded the data at the source.

The following truth tables can be used to encode and decode the data stream:

Encode MIDI Data Truth Table			
Data Byte value B	Control character	Encoded Data byte value	Number of Bytes transmitted
0x00	Nil	B	1
0x01 to 0x02	0x02	B	2
0x03 to 0x7F	Nil	B	1
0x80 to 0xFF	0x01	B 0x80	2

Decode MIDI Data Truth Table				
Current Control Char. value	Current Byte	New Control Char. value	Decoded Data value	Valid Data
0x01	B	Nil	B + 0x80	Y
0x02	B	Nil	B	Y
Nil	0x01	0x01	N/A	N
Nil	0x02	0x02	N/A	N