Lab 13 Controllers and Modulation

Alesis QX25 MIDI Controller

Connect the USB cable to the QX25 MIDI Controller and then to the computer. Make sure the rear panel switch is in the USB position.

Setup

If you haven't already done so, download and unzip m208Lab13.zip to your Desktop.

Mac Only - Open Au Lab

In order to use MIDI output on the Mac you must open Au Lab before opening miniAudicle. This is not necessary if you are using Windows.

Open miniAudicle

Quit and restart miniAudicle. Restarting miniAudicle may not always be necessary, but if your program stops responding to MIDI input that's the first thing to try. Set m208Lab13 to the working directory in miniAudicle. If you've been working with other MIDI devices, remove all shreds and save any files you've been working on.

Identify the MIDI Input and MIDI Output Device Numbers

Open Terminal and run the chuck --probe command to list the available Audio and MIDI input and output devices. This command works on both Mac and Windows but must be run from a Terminal window. The MIDI devices are shown at the end of the list and the device number is shown in brackets. There is a second Mac only option: choose Device Browser from the miniAudicle Window menu.

If you've already done the nanPAD2 lab you can reuse some of the following code.

ChucK MIDI Event Documentation

ChucK contains built-in MIDI classes to allow for interaction with MIDI based software or devices.

```
MidiIn min;
MidiMsg msg;

// open midi receiver, exit on fail
if ( !min.open(0) ) me.exit();

while( true )
{
    // wait on midi event
    min => now;

    // receive midimsg(s)
    while( min.recv( msg ) )
    {
        // print content
      <<< msg.data1, msg.data2, msg.data3 >>>;
    }
}
...
```

MidiIn is a subclass of **Event**, and as such can be ChucKed to **now**. MidiIn then takes a MidiMsg object to its **.recv()** method to access the MIDI data.

As a default, MidiIn events trigger the **broadcast()** event behavior.

http://chuck.cs.princeton.edu/doc/language/event.html#midi

MIDI Documentation Example

Run the above code in miniAudicle. It's called midiIn.ck and it's in the m208Lab13 folder.

```
144 60 36
128 60 7
144 62 33
128 62 1
144 64 1
128 64 5
144 65 14
128 65 8
144 67 22
128 67 2
```

144 decimal is 90 hex and 128 decimal is 80 hex. These are the staus bytes for a note on (NON) and a note off (NOF).

Problems

If you don't see anything MIDI message printouts in the Console Monitor maybe you're trying to connect to an invalid MIDI device number. On my computer the QX25 keyboard was input device 1, not 0. Change this line and run the program again.

```
// open midi receiver, exit on fail
if ( !min.open( 1 ) ) me.exit();
```

Play some keys, turn some knobs, move the pitch wheel, mod wheel, and press some pads. You should see messages like this appear in the Console Monitor window.

```
[chuck](VM): sporking incoming shred: 1 (midiExample.ck)...
144 59 6
128 59 1
144 63 20
128 63 2
144 65 12
128 65 4
144 66 28
128 66 3
```

Remove all shreds. Save the file. Quit and restart miniAudicle. Restarting miniAudicle may not always be necessary, but if your program stops responding to MIDI input that's the first thing to try.

MIDI Messages

The majority of MIDI messages consist of three numbers called the status byte, data 1 byte, and data 2 byte. Chuck calls them msg.data1, msg.data2, and msg.data3. All three numbers are 8 bit numbers that range from 0 to 255 ($2^8 - 1$). Status bytes range from 128 – 255 (80-FF hex) and data bytes range from 0 – 127 (0-7F hex).

Status byte messages are divided into eight categories, the most common being Note Off, Note On, Control, Program Change, and Pitch Bend messages. Each of the eight status categories can send messages to 16 different MIDI channels independently. Each of the 16 channels can play a different instrument.

The output from the first example program displayed the MIDI status bytes in decimal format. It is MUCH easier to decipher MIDI messages when the status byte is displayed in the hexadecimal (base 16) number system. Create this program and you'll see what I mean.

byte2hex.ck

```
// byte2hex.ck
// John Ellinger Music 208 Winter2014
// converts a MIDI status byte to hexadecimal display

function string byte2hex( int num )
{
    // in a two digit hex byte each digit is called a nibble
    num / 16 => int msn; // most significant nibble, digit on left
    num % 16 => int lsn; // least significant nibble, digit on right

    ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
    "A", "B", "C", "D", "E", "F"] @=> string h[];
    return h[ msn ] + h[ lsn ];
}

// range of MIDI status bytes
```

```
<<< "==== msg.data1 STATUS BYTES in hex", "" >>>;
<<< byte2hex( 128 ), "-", byte2hex( 143 ), "is NOTE OFF" >>>;
<<< byte2hex( 144 ), "-", byte2hex( 159 ), "NOTE ON" >>>;
<<< byte2hex( 160 ), "-", byte2hex( 175 ), "AFTER TOUCH POLY" >>>;
<<< byte2hex( 176 ), "-", byte2hex( 191 ), "CONTROL MESSAGE" >>>;
<<< byte2hex( 192 ), "-", byte2hex( 207 ), "PROGRAM CHANGE" >>>;
<<< byte2hex( 208 ), "-", byte2hex( 223 ), "AFTER TOUCH CHANNEL" >>>;
<<< byte2hex( 224 ), "-", byte2hex( 239 ), "PITCH BEND" >>>;
<<< byte2hex( 240 ), "-", byte2hex( 255 ), "SYSTEM EXCLUSIVE" >>>;
// range of MIDI data bytes
<<< "\n==== msg.data2 and msg.data3, DATA BYTES in hex", "" >>>;
<<< byte2hex( 0 ), "is MINIMUM data byte in hex is 0 decimal" >>>;
<<< byte2hex( 127 ), "is MAXIMUM data byte in hex is 127 decimal" >>>;
```

You should see these results in the Console Window.

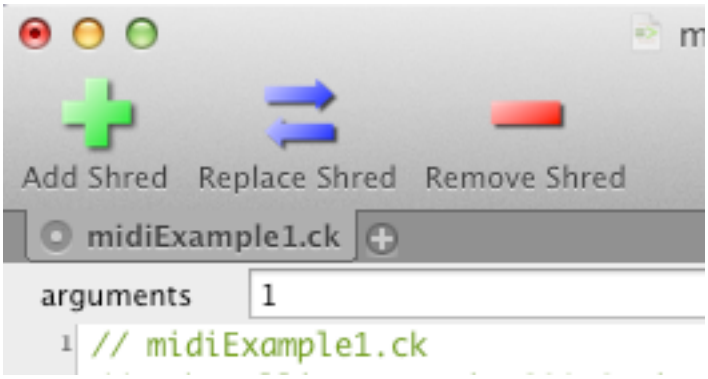
```
[chuck](VM): sporking incoming shred: 1 (byte2hex.ck)...
==== msg.data1 STATUS BYTES in hex
80 - 8F is NOTE OFF
90 - 9F NOTE ON
A0 - AF AFTER TOUCH POLY
B0 - BF CONTROL MESSAGE
C0 - CF PROGRAM CHANGE
D0 - DF AFTER TOUCH CHANNEL
E0 - EF PITCH BEND
F0 - FF SYSTEM EXCLUSIVE

==== msg.data2 and msg.data3, DATA BYTES in hex
00 is MINIMUM data byte in hex is 0 decimal
7F is MAXIMUM data byte in hex is 127 decimal
```

midiInExample.ck

Save midiDocExample.ck as midiInExample.ck. Modify the opening lines to read the input args() and add the byte2hex() function. Then enter this code.

```
// midiInExample.ck
// John Ellinger Music 208 Spring2014
//  modified: http://chuck.cs.princeton.edu/doc/language/
event.html#midi

// Input device number to open (see: chuck --probe)
0 => int device;
// get alternative device from the command line
if( me.args() ) me.arg(0) => Std.atoi => device;

MidiIn min;  // the midi event
MidiMsg msg; // the message for retrieving data
// open the device
if( !min.open( device ) ) me.exit();
// print out device that was opened
<<< "MIDI device:", min.num(), " -> ", min.name() >>>;

function string byte2hex( int num )
{
    // in a two digit hex byte each digit is called a nibble
    num / 16 => int msn; // most significant nibble, digit on left
    num % 16 => int lsn; // least significant nibble, digit on right

    ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
    "A", "B", "C", "D", "E", "F"] @=> string h[];
    return h[ msn ] + h[ lsn ];
}

// never ending loop
while( true )
{
    // wait on the event 'min'
    min => now;
    // get the message(s)
    while( min.recv(msg) )
    {
        // print out midi message
        <<< byte2hex(msg.data1), msg.data2, msg.data3 >>>;
    }
}
```

Restart miniAudicle but don't run the code just yet. Instead of changing the MIDI Input device number in code, enter its number in miniAudicle's arguments text field.



If you're running midiInExample.ck from the Terminal, add :1 after .ck like this.

```
502,je@jemac:~/Desktop/m208Lab13$ chuck midiExample1.ck:1
MIDI device: 1  ->  QX25
90 60 15
80 60 11
90 62 4
80 62 4
```

QX25 Controls

Besides the piano keys, there are several other knobs, pads, switches, sliders and wheels that send MIDI messages.

Here's a map of the keyboard controls and the messages they send.

	STATUS HEX	DATA 1	DATA 2	
	msg.data1	mgs.data2	msg.data3	Comments
A	90	60	0-127	Note On
	80	60	ignored	Note Off
B	B0	14	0-127	Control 14
C	B0	15	0-127	Control 15
D	B0	16	0-127	Control 16
E	B0	17	0-127	Control 17
F	B0	18	0-127	Control 18
G	B0	19	0-127	Control 19
H	B0	20	0-127	Control 20
I	B0	21	0-127	Control 21
J	90	48	0-127	Pad 1 Note On
	80	48	0	Pad 1 Note Off
K	90	49	0-127	Pad 2 Note On
	80	49	0	Pad 2 Note Off
L	90	50	0-127	Pad 3 Note On
	80	50	0	Pad 3 Note Off
M	90	51	0-127	Pad 4 Note On
	80	51	0	Pad 4 Note Off
N	E0	0	0-127	Pitch Wheel
O	B0	1	0-127	Mod Wheel
P	B0	22	0-127	Control 22
Q	C0	2	65	Program Change

qx25Example1.ck

Save midiInExample.ck as qx25Example1.ck and enter this code.

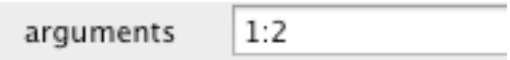
```
// qx25Example1.ck
// John Ellinger Music 208 Spring2014
// modified: http://chuck.cs.princeton.edu/doc/language/
event.html#midi

// number of the device to open (see: chuck --probe)
0 => int deviceIn;
0 => int deviceOut;
// get command line
<<< me.args() >>>;
if( me.args() == 2 )
{
    me.arg(0) => Std.atoi => deviceIn;
    me.arg(1) => Std.atoi => deviceOut;
}

// the MIDI Input event
MidiIn min;
// open the device
if( !min.open( deviceIn ) ) me.exit();
// print out device that was opened
<<< "MIDI In device:", min.num(), " -> ", min.name() >>>;

// the MIDI Output event
MidiOut mout;
// open the device
if( !mout.open( deviceOut ) ) me.exit();
// print out device that was opened
<<< "MIDI Out device:", mout.num(), " -> ", mout.name() >>>;

// the message for retrieving data
MidiMsg mmsg;

// display status byte in hex
function string byte2hex( int num )
{
    num / 16 => int msb; // most significant byte
    num % 16 => int lsb; // least significant byte
```

```
    ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
    "A", "B", "C", "D", "E", "F"] @=> string h[];
    return h[ msb ] + h[ lsb ];
}

// MIDI Utilitiy functions
function int isNON( int data1 ) // is status byte a NON (Note On)
{
    // && is logical AND
    if ( data1 >= 0x90 && data1 <= 0x9f )
        return 1;
    else
        return 0;
}

function int isNOF( int data1 ) // is status byte a Note Off (NOF)
{
    if ( data1 >= 0x80 && data1 <= 0x8f )
        return 1;
    else
        return 0;
}

function int isController( int data1 )
// all Control messages are from B0-BF hex
{
    if ( data1 >= 0xB0 && data1 <= 0xBf )
        return 1;
    else
        return 0;
}

// infinite time-loop
while( true )
{
    // wait on the event 'min'
    min => now;
    0 => int padDown;
    // get the message(s)
    while( min.recv(mmsg) )
    {
        // print out midi message
```

```
            <<< byte2hex(mmsg.data1), mmsg.data2, mmsg.data3 >>>;
            // check to see if it came from a pad
            // || is logical OR
            if ( isNON( mmsg.data1 ) || isNOF( mmsg.data1 ) )
            {
                mout.send( mmsg ); // echo to MIDI out

            }
            else if ( isController( mmsg.data1 ) )
            {
                <<< "Controller", mmsg.data2, "\tvalue\t", mmsg.data3 >>>;
            }
        }
    }
```

Command Line Arguments

If you need MIDI Input device 1 and MIDI Output device 2 enter these arguments in miniAudicle…



or in Terminal:

```
je@jemac:~/Desktop/m208Lab13/2014$ chuck qx25Example1.ck:1:2
```

Play Chords With Each Key Press

Make these changes and you'll hear a four note chord on each note played.

```
if ( isNON( mmsg.data1 ) || isNOF( mmsg.data1 ) )
{
    mmsg.data2 => int note1;
    mout.send( mmsg ); // echo to MIDI out
    note1 + 7 => mmsg.data2;
    mout.send( mmsg );
    note1 + 10 => mmsg.data2;
    mout.send( mmsg );
    note1 + 16 => mmsg.data2;
    mout.send( mmsg );
}
```

qx25Example2.ck

Save midiInExample.ck as qx25Example2.ck. This example uses a STK instrument. It's monophonic so you can only play one note at a time. Try pressing the octave up down buttons on the QX25 to change octaves.

```
// qx25Example2.ck
// John Ellinger Music 208 Spring2014
// modified: http://chuck.cs.princeton.edu/doc/language/
event.html#midi

// number of the device to open (see: chuck --probe)
0 => int device;
// get command line
if( me.args() ) me.arg(0) => Std.atoi => device;
// the midi event
MidiIn min;
// the message for retrieving data
MidiMsg msg;
// open the device
if( !min.open( device ) ) me.exit();
// print out device that was opened
<<< "MIDI device:", min.num(), " -> ", min.name() >>>;

Moog moogie => dac;

function string byte2hex( int num )
{
    // in a two digit hex byte each digit is called a nibble
    num / 16 => int msn; // most significant nibble, digit on left
    num % 16 => int lsn; // least significant nibble, digit on right

    ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
    "A", "B", "C", "D", "E", "F"] @=> string h[];
    return h[ msn ] + h[ lsn ];
}
```

```
// infinite time-loop
while( true )
{
    // wait on the event 'min'
    min => now;

    // get the message(s)
    while( min.recv(msg) )
    {
        // print out midi message
        <<< byte2hex(msg.data1), msg.data2, msg.data3 >>>;
        // we'll hard code for channel 0
        if ( msg.data1 == 0x90 ) // NoteOn
        {
            Std.mtof( msg.data2 ) => moogie.freq;
            moogie.noteOn(.7);
        }
        else if ( msg.data1 == 0x80 ) // NoteOff
        {
            moogie.noteOff(0);
        }
    }
}
```

The ChucK documentation for Moog shows several control parameters. We can use the scale data function above to map the QX25 controls to these parameters.

[ugen]: **Moog** (STK Import)

- *STK moog-like swept filter sampling synthesis class.*
- *see examples: moogie.ck*

```
This instrument uses one attack wave, one
looped wave, and an ADSR envelope (inherited
from the Sampler class) and adds two sweepable
formant (FormSwep) filters.

Control Change Numbers:
   - Filter Q = 2
   - Filter Sweep Rate = 4
   - Vibrato Frequency = 11
   - Vibrato Gain = 1
   - Gain = 128

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.
```

extends StkInstrument

(control parameters)

- **.filterQ** - (float , READ/WRITE) - *filter Q value [0.0 - 1.0]*
- **.filterSweepRate** - (float , READ/WRITE) - *filter sweep rate [0.0 - 1.0]*
- **.vibratoFreq** - (float , READ/WRITE) - *vibrato frequency (Hz)*
- **.vibratoGain** - (float , READ/WRITE) - *vibrato gain [0.0 - 1.0]*
- **.afterTouch** - (float , WRITE only) - *aftertouch [0.0 - 1.0]*

(inherited from StkInstrument)

- **.noteOn** - (float velocity) - *trigger note on*
- **.noteOff** - (float velocity) - *trigger note off*
- **.freq** - (float frequency) - *set/get frequency (Hz)*
- **.controlChange** - (int number, float value) - *assert control change*

Range Scaling

The MIDI knobs, wheels, and sliders output data in the range from 0 – 127. Many ChucK methods expect numbers in the range of 0 – 1.0. It would be nice to have a generic function that could take a value from on range and scale it to another range.

```
// scaleDataTest.ck
// John Ellinger Music 208 Winter2014

function float scaleData( float inValue, float inMin, float inMax,
float outMin, float outMax )
{
    inMax - inMin  => float inRange;
    outMax - outMin => float outRange;
    return  (outRange * inValue / inRange) + outMin;
}

<<< "UNIPOLAR OUT scale 0-127 in to 0-1.0", "" >>>;
<<< "0\t", scaleData( 0, 0, 127, 0, 1.0) >>>;
<<< "64\t", scaleData( 64, 0, 127, 0, 1.0) >>>;
<<< "127\t", scaleData( 127, 0, 127, 0, 1.0) >>>;

<<< "AMPLITUDE OUT scale 0-127 in to -1.0-1.0", "" >>>;
<<< "0\t", scaleData( 0, 0, 127, -1.0, 1.0) >>>;
<<< "64\t", scaleData( 64, 0, 127, -1.0, 1.0) >>>;
<<< "127\t", scaleData( 127, 0, 127, -1.0, 1.0) >>>;
<<< "DECIBELS OUT scale 0-127 in to -100-0", "" >>>;
```

```
<<< "0\t", scaleData( 0, 0, 127, -100, 0) >>>;
<<< "64\t", scaleData( 64, 0, 127, -100, 0) >>>;
<<< "127\t", scaleData( 127, 0, 127, -100, 0) >>>;

<<< "FREQUENCY RANGE OUT scale 0-127 in to 457-1234", "" >>>;
<<< "0\t", scaleData( 0, 0, 127, 457, 1234) >>>;
<<< "64\t", scaleData( 64, 0, 127, 457, 1234) >>>;
<<< "127\t", scaleData( 127, 0, 127, 457, 1234) >>>;
```

Output

```
[chuck](VM): sporking incoming shred: 30 (scaleDataTest.ck)...
UNIPOLAR OUT scale 0-127 in to 0-1.0
0     0.000000
64    0.503937
127   1.000000
AMPLITUDE OUT scale 0-127 in to -1.0-1.0
0     -1.000000
64    0.007874
127   1.000000
DECIBELS OUT scale 0-127 in to -100-0
0     -100.000000
64    -49.606299
127   0.000000
FREQUENCY RANGE OUT scale 0-127 in to 457-1234
0     457.000000
64    848.559055
127   1234.000000
```

MIDI Example 4

Save qx25Example2.ck as qx25Example3.ck and add these lines.

```
// qx25Example3.ck
// John Ellinger, Music 208, Spring2013
// modified from: http://chuck.cs.princeton.edu/doc/language/event.html#midi

// number of the device to open (see: chuck --probe)
0 => int device;
// get command line
if( me.args() ) me.arg(0) => Std.atoi => device;
// the midi event
MidiIn min;
// the message for retrieving data
```

```
MidiMsg msg;
// open the device
if( !min.open( device ) ) me.exit();
// print out device that was opened
<<< "MIDI device:", min.num(), " -> ", min.name() >>>;

Moog moogie => dac;

function string byte2hex( int num )
{
    // in a two digit hex byte each digit is called a nibble
    num / 16 => int msn; // most significant nibble, digit on left
    num % 16 => int lsn; // least significant nibble, digit on right

    ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
    "A", "B", "C", "D", "E", "F"] @=> string h[];
    return h[ msn ] + h[ lsn ];
}

function float scaleData( float val, float inMin, float inMax,
                          float outMin, float outMax )
{
    inMax - inMin + 1 => float inRange;
    outMax - outMin => float outRange;
    return   outMin + (outRange * val / inRange);
}

// infinite time-loop
while( true )
{
    // wait on the event 'min'
    min => now;

    // get the message(s)
    while( min.recv(msg) )
    {
        // print out midi message
        // we'll hard code for channel 0
        if ( msg.data1 == 0x90 ) // NoteOn
        {
            Std.mtof( msg.data2 ) => moogie.freq;
            moogie.noteOn(1.0);
        }
        else if ( msg.data1 == 0x80 ) // NoteOff
        {
            moogie.noteOff(0);
```

```
        }
        else if ( msg.data1 == 0xB0 && msg.data2 == 22 ) // slider S1
        {
            scaleData( msg.data3, 0, 127, 0, 1 ) => moogie.gain;
            <<< "volume", moogie.gain() >>>;
        }
        else if ( msg.data1 == 0xB0 && msg.data2 == 14 ) // knob 1
        {
            scaleData( msg.data3, 0, 127, 0, 1 ) => moogie.filterQ;
            <<< "filterQ", moogie.filterQ() >>>;
        }
        else if ( msg.data1 == 0xB0 && msg.data2 == 15 ) // knob 2
        {
            scaleData( msg.data3, 0, 127, 0, 1 ) => moogie.filterSweepRate;
            <<< "filterSweepRate", moogie.filterSweepRate() >>>;
        }
        else if (msg.data1 == 0xB0 && msg.data2 == 16 ) // knob 3
        {
            scaleData( msg.data3, 0, 127, 0, 255 ) => moogie.vibratoFreq;

            <<< "vibratoFreq", moogie.vibratoFreq() >>>;
        }
        else if ( msg.data1 == 0xB0 && msg.data2 == 17 ) // knob 4
        {
            scaleData( msg.data3, 0, 127, 0, 1 ) => moogie.vibratoGain;
            <<< "vibratoGain", moogie.vibratoGain() >>>;
        }
    }
}
```

Twiddle the knobs as you play a note, especially knobs 3 and 4. Experiment with the scaleData output range for vibratoFreq.