MIDIBox Sequencer vX

aka 'veX'

MIDIBox Sequencer vX

Some of you will already know about my work on developing a somewhat unusual sequencer... Well, a few recent events have made major changes to the way I can do this, and so I thought that after such a long time, and the recent advances, it might be time to let you guys know what I'm up to - both for your interest, and for the purpose of any feedback that you might give in the form of suggestions or corrections etc...

Well the latest bit of good news is the PIC18F4620. Now, you may already know that it has a problem with the UART, or EUSART to be precise, and as such has problems with MIDI. This is about the end of the downsides to this chip, and so for the purpose of conversation, I'll ignore that major problem, for now... but the really big plus with the new chip is the massive increase in both code and RAM size. Previously, due to a need for more RAM, some board members here helped me to design an external SRAM interface (THANKS GUYS!!). With the 4620 in the picture, things get much easier. The new chip, combined with TK's absolutely brilliant highly modular coding work, means that most of the hard work has now been done for me. This new sequencer concept is quite different in practice to the MBSEQ, but a great deal of the original code is re-usable, so it's been well tested too. My work in developing this idea has illustrated to me how truly great TK's code is. We're very lucky that TK has gone to such an effort :)

So anyway, enough background, lets get into the concept. This has changed a bit from the previous incarnations and may change again in the future... But this is how it works for now ;) Forgive me, this will probably be a very, very long post, as I feel I should give some insight into what I've been doing all this time, and I fear that with so many details, that it may be hard to explain well. I will do my best to explain it as clearly as I am able. The project has the working name of MIDIBox Sequencer vX - as in version X I've taken to calling it “vX” or “veX” because that's what Microsoft Word told me I was trying to spell ;) I gave it the name version X because the existing code is derived from the MBSEQ, but I don't believe that it could be a replacement or a “new version” due to it having a rather different purpose and functionality. I'm going to have two of each :D

All that said, there are three major differences between the vX and the current MBSEQ:

First, the Clock Modules:
The vX will have a master clock, exactly as the current MBSEQ, nothing new there. TK's clock interpolation algorithm for 24ppqn→96ppqn conversion is brilliant. What is different in the vX, is that there will be multiple 'slaved' clocks, which run in sync with the master and each other. These clocks tick once per step, and can run at whatever divider and/or multiplier that you want, but are slaves of the master clock, at it's resolution (EG 96PPQN). This enables us to run polyrhythms and/or polymeters, a feature available in no other (current) hardware MIDI sequencer, as well as tempo and sync independent patterns. But remember, all clock signals will be quantized to the single Master MIDI Clock at 96 ticks per quarter note. The period of the slave clock tick is set something like this:

One Master Loop, run either by incoming MIDI Clock or internal clocks run by the PIC's timer, has a length measured in MIDI Clock Ticks. The Master Loop Length is the number of MIDI Clock Ticks in one bar at the master time signature, and will be in a table which is looked up for the correct value when the song time signature is set. For eg, the MLL at 4/4 time signature is 384, at 3/4 it is 288, at 7/8 it is

336, at 5/4 it is 480.

Each clock has a pair (for simplicity of use) of multipliers and dividers. These are represented as N Steps per O Loops and P Steps per Q Loops of the Master Loop. Internally, they will be multiplied and converted to a single numerator and denominator of a master loop.

```
          O * Q * MLL
 T = ----------------
          N * P
```

Where MLL is the Master Loop Length
T is the amount of MIDI ticks between this slave clocks ticks.
(N*P)/(O*Q) is the steps/loops of the first slave clock

This will be presented as two fractions, which, for musical clarity, are inverted from the equation above. Some examples:
For a common 4 step per bar clock, you setup a clock like this:

```
 4   1    Steps
--- ---     Per
 1   1    Loops
```

The second fraction is ignored as it has the same numerator and denominator, so does nothing.
To play a polymeter, you would assign a pattern with more or less steps than this, to this same clock. For example, if you had a pattern with 5 steps (between the start and end point) assigned to this same clock along with a pattern with 4 steps, then every 20 Master Loops, they would sync up.

To have a 5:4 polyrhythm (five notes play in the same time as 4 notes), you would assign a clock such as the above to a 4 step pattern, and setup the clock assigned to a 5 step pattern clock like so:

```
 5   1    Steps
--- ---     Per
 1   1    Loops
```

Now the two patterns will loop in sync, despite having different lengths, because they are clocked by slaves to the one master loop.

If you wanted to play 4/4 triplets, you would set the clock parameters like so:

```
 4   3    Steps
--- ---     Per
 1   1    Loops
```

This is the same as 12/1 or 24/2 etc ...you can decide which is a more clear representation for your needs, the core takes care of the math.

Or for 4/4 with a 'divider' of 8:

```
 4   1    Steps
--- ---     Per
 1   8    Loops
```

(This is the same as 4 steps per 8 loops or 1 steps/2 loops)

And so on… you could do weird nonexistent (until now) things like synchronized 7th note fifthlets if it suits you ;) The clocks are limited to keep the math involved practical, and timings reliable. The fastest clock available would tick 32 steps per loop (1/32 notes), and the slowest would be 1 step per 32 loops.

Normally, doing this would mean either clocking externally and wasting IO pins, which can also be slow to drive, or doing a big nasty multiply/divide and rounding operation at each step, which would be too CPU intensive for our purposes. In order to implement this feature, I use an algorithm which does just one multiply/divide when the clock is initialised, and after playing each step, it does only simple addition or subtraction, to calculate the number of MIDI clock ticks until the next step.

This is done by calculating the integer number of ticks required between steps, which is essentially three multiplies (the pair of numerators and the clock resolution, and the pair of denominators) followed by a divide operation which returns the quotient and modulus. The quotient is used as the default number of ticks to count between steps. Where there is a remainder/modulus, the modulus is automatically distributed evenly throughout one full cycle of the clock. This is done by addition and/or subtraction only (OK, there is one multiply by 2, but that's just a left shift of the byte), so is much faster than multiplying/dividing at each step, to figure out the amount of ticks that will pass until the next step. It also means that song position, which step we are on, etc, is irrelevant, and the clocking engine will always distribute the modulus evenly, regardless of location within the song. These countdowns are calculated in advance when the PIC is not busy, but of course, all this fancy stuff is skipped if the modulus is 0, as the countdown will always be the same (the quotient).

So… each clock has a set of parameters which it will use to calculate the number of midi clock ticks until the next step. With each 96ppqn tick of the master/MIDI clock, this 'Next Tick Countdown' is decremented on each active slave clock. When it reaches 0, a core 'slave tick' is added to a counter. This is pretty much the same thing that happens with the MBSEQ, except that currently there is only one counter (it's a counter, not a bit, so that no ticks are lost. Very clever, TK!). With the vX, the master clock drives a number of clock dividers which increment multiple counters.

Next, Pattern Modules:
With the MBSEQ, each track is scanned and the next step is played, if the clock has ticked. The same thing happens with my design, only the tracks check different clocks which are assigned in their parameters. The existing MBSEQ code is so modular, I could easily believe that the concept of multiple clocks was already intended in the current design :)

Each pattern is a single track of 16 steps, is independent of the other patterns, and can use an independent clock, or share a clock with other tracks. They have independent Loop Start and End Points, can be independently set to Play/ Stop/ Pause/ Record, individually set to loop in different modes, independently set to a different current step, etc.

Pattern parameters include Pattern Play Sync, which works like this: You make a pattern and you have it triggered to play and step by clock #7. Now, you set the pattern to enter Play Mode after the next tick of clock #3. When you press Play for this pattern, it sits idle, until after clock #3 ticks, when the pattern is triggered to enter play mode. But it will not process and play the first step yet… until clock #7 ticks, because that is the clock which is set to trigger the steps to play.

Other pattern parameters: Direction:

- Forward - Plays from start point to end point

- Reverse - Plays from end point to start point

Loop Mode:

- Looped - Normal looped play in the above direction
- PingPong - Plays forward, reverse, then loops. Can be reversed as above
- OneShot - Plays once only in direction above. Stops at end point in forward mode, or at Start point in Reverse mode

Clock Patch Mode:

- Step,Trigger - Clock tick plays the step and then advances to the next step according to the above. This is the way MBSEQ works
- !Trigger,Step - Clocks advance to the next step according to the above, but do not play the step before they advance. Similar to muting the pattern, but the steps can be triggered to play by remote control. When a pattern is muted, no steps will play, regardless of whether they are triggered by remote control, or a clock.
- Trigger,!Step - Clocks trigger current step to play, but do not advance to the next step – the pattern is Stepped by remote control (either an absolute step number or an increment/decrement according to the pattern direction)
- !Trigger,!Step - No Clocks will be checked to trigger step to play or to advance to the next step. The pattern must be entirely remote controlled.

While any pattern can be stepped or triggered regardless of these parameters, patterns set to these modes will need to be stepped/triggered by remote control.

This gives a brief overview of the modular patterns, but so far I have left out one major detail – I hinted that the above parameters, as well as all the other Pattern parameters including parameters for each step, and also all of the Clock parameters, accept input from remote control, but all the parameters can also be from other patterns. How does one pattern modify or trigger a clock or another pattern, I hear you ask?....

Finally, Step Functions:
In the vX, a step is no longer just for a MIDI message. Each step contains a parameter which selects a Function to use, and a set of parameters which are passed to the function when it is 'played'. An obvious function is to play a note, which is one thing the steps currently do in the MBSEQ. The parameters of the step would be interpreted as the MIDI Note#, Velocity, Step Length, etc. Functions could also do other things like stepping or triggering patterns, as indicated above, or starting, pausing, muting or stopping other patterns, changing clock parameters, sending NRPN's, changing a note value on another pattern, reversing the play direction on another pattern, increasing the sustain time on a MIDI CC envelope generator,changing loop start or end points, loop type or direction, one function could call a number of other functions, the list goes on..... Basically, they can change any parameter on any object (clock, pattern, step, etc) or perform any function supported by MIOS... You just hardcode the functions to your requirements before you upload the app. Of course, there will be a set of standardised functions, like playing a midi note, which you would be unlikely to remove. As many functions as memory allows can be customised and uploaded per your requirements.

This “function” feature, combined with floating patterns, means that song mode is now obsolete. If you want to play say, 3 bars of drum break #1, then play Drum Fill #1 in the 4th bar, it's easy... First, you create your drum break and fill patterns, and you set them to sync 4 steps with the master clock. You set up a pattern 4 steps long, and attach it to a clock in one-shot mode which triggers one step for each bar (So, 1/4 of the master clock speed). Steps 1-3 run a function which Plays the pattern

containing drum break #1, and step 4 runs a function which Plays the pattern containing drum fill #1... See how it goes?

Then, you can do tricky things like having another pattern, which changes step 4's parameters, so that it plays drum fill#1, then next time, it plays drum fill #2, then fill #3, etc etc. It opens up various algorithmic composition possibilities... I hope you can all see where I'm going with this.

Also, patterns of extended lengths are easily implemented. If you want a pattern 32 steps long, you create two, 16-step patterns, and link them, using a single 2-step pattern... Or 4,8-step patterns, linked by a single 4-step pattern, etc

All of these examples would require functions which can manipulate tracks in ways such as the examples in the 'Pattern Modules' section. They'll be standard issue :)

The functions can also be called externally, so for example, a slider on your midibox could select a pattern by setting the parameter for a given function, and the C4 key could be used to trigger the pattern selected by the slider... Another example is that you could setup keys on your controller keyboard, or trigger pads, that will have a pattern jump to a certain step in the sequence. DJ's reading this who can beat juggle will immediately see the use of a function like this, and those who are familiar with modular sequencers with a step# input will see a similarity also.

These mappings of external input to internal functions and their parameters could be hardcoded like the functions, or could possibly be included as a set of parameters which would be part of each song... I have not worked on the remote control very much at all just yet.

General Info

Those of you who have used an analog modular sequencer will probably see a great deal of resemblance between them, and the vX... Only in our case, multiple synced clocks are possible, enabling multipliers and dividers limited only to the resolution of MIDI clock; and the steps can trigger far more complex features than just sending CV out, they can dynamically repatch the entire rack :)

Similar to the Notron, the vX can pass values generated by functions as a relative number. In these functions, values of 0 to 127 are interpreted as -63 to +64, so 63 is interpreted as a 0. For example, a pattern has a parameter for a “Base” note number. This number is then summed with the value of the step parameter for Note Number. Using note number as an example:

For “normal” use, the “base” note number for a pattern is assigned as D#4 (note 63)

- If the step has a value of 63, it is interpreted as 0. 0 + 63 = 63 = D#4
- If the step has a value of 64, it is interpreted as +1. +1 + 63 = 64 = E4
- If the step has a value of 62, it is interpreted as -1. -1 + 63 = 62 = D4

...All “normal” there :) But say we use a function to change the base note to say, A2 (note 45)....

- If the step has a value of 63, it is interpreted as 0. 0 + 45 = 45 = A2
- If the step has a value of 64, it is interpreted as +1. +1 + 45 = 46 = A#2.
- If the step has a value of 62, it is interpreted as -1. -1 + 45 = 44 = G#2

I hope to include another feature from the Notron (and about 1000 other algorithmic sequencers) where notes will be locked to a scale. This function would use a table lookup to map note numbers to the scale set in the pattern parameters.

Similarly, a function could modify a parameter's value in a relative manner, by adding or subtracting the value as above.

Order of execution

The order of execution goes like so:

- Clocks - Have to be a priority for timing to be accurate
- Patterns: -
- Start Sync - So that tracks are started for these other procedures...
- Control Functions - So that parameters are modified in time to be read for MIDI Functions
- MIDI Functions - Then the notes get sent

All data is buffered with code based on that within the existing MBSEQ, so all this seems to happen at the same time :)

- UI - Not so time-sensitive

EUSART

Previously, I mentioned the PIC18F4620, and it's additional memory allowing for these features to be implemented practically. I also mentioned that there is a EUSART fault (for those who haven't heard, it's got a bug and it causes problems with MIDI). If Microchip do not release a fixed silicon revision for this, I will implement a way to send MIDI out through the built-in I2C on PORTC pins of the PIC, to a second, older model PIC. This second PIC will then pass the incoming I2C data as MIDI out. In line with TK's prototypes, I will use a PIC16F88. Any extra power in the chip will be used to perform MIDI Effects such as transposition, delays, filtering, and LFO's and Multi-Point Envelopes to sent MIDI CC's, NRPN's, SysEx, etc. The reason behind this is just because I feel that if I have to write the special code and possibly make minor modifications to circuitry to send MIDI over I2C, then I will capitalize on it and make the most of it... and the good news is, it will easily be converted to a standalone unit :) Also, I have thought about using several MIDI over I2C slaves, to allow for multiple MIDI Outs, so you could run 32 channels.

A Touch of Normalcy ;)

Despite all of the differences, and that this sequencer is obviously something that will appeal to those interested in tribal/ethnic/algorithmic music, or just those with weird taste in music (like me ;)), the default values for a new song will be pretty standard stuff... mostly in lots of 4 steps, 4/4 time, etc. This is just to give a starting point that is fairly neutral as a skeleton for a song, but as a side effect, the vX will be well configured to do traditional step-sequencing – but of course because the MBSEQ specialises in that field, it will always be superior for that purpose.

User Interface

The user interface will be reminiscent of the Doepfer Schaltwerk, with buttons for each of the steps arranged in a row for each pattern, and a single large character LCD, jog wheel and slider for fast data entry, buttons, and set of rotary controllers to edit parameters of the selected object.

There will be a set of buttons which will be used for selecting one of the clocks for configuration using the same LCD. Each button has an LED which flashes when that clock ticks. When a clock is selected for editing, the LED is set on, and flashes off for each tick (The LED is inverted).

I hope that eventually the grid of buttons for the steps will be arranged in an 8 row x 12 column grid. This will allow for two modes of operation. The first would be one that we are already familiar with, where each row is a pattern and each button within that row representing a step in the sequence. In this mode, there would be left/right buttons which would control which 8 of the 16 steps are displayed, and each row can be freely assigned to any pattern. Pressing a button selects that step, and then you set the parameters for the step

The second mode would be Grid Edit, where all of the buttons are used to set parameter values for a single pattern. In this mode, each column of buttons represents a single step, and each button in that vertical column represents a separate value for the selected parameter, so the 12 rows become, for example, an octave worth of notes. This is basically like a piano roll, but of course you can only edit one pattern at a time in this manner. When you press a button, it sets the selected parameter (eg Parameter 1, which is used for MIDI Note Number in the 'Send MIDI Note' Function) according to the row. In this mode, there would be left/right buttons which would control which 8 of the (up to) 16 steps are displayed, and pitch up/down buttons to 'scroll' up and 'scroll' down the grid.

In the meantime, the User Interface will be driven by PC software... This is just until the core features are all done. I strongly suspect that a UI such as this may require a second core module, which would pass data to the second core, to drive the LCD and DOUTs. This has been discussed previously on the forum, and I am keen to implement it, as moving the control surface onto a second core will relieve the primary core of extra load driving IO.

Prototype Implementation

Implementation will be via a series of prototype stages:

- Flowcharts and pseudocode (Nothing real, but lots of planning time)
- Seq with multiple single-track patterns, one clock (very boring, will mostly be a stripped-down MBSeq v2)
- Add multiple clocks (way more interesting)
- Add functions (yee hah!)
- Remote control
- Hardware UI

I'm sure there's enough to give an idea of what I've been up to, I just hope I've gotten it right :)

Please share any thoughts or ideas or concerns or criticism or whatever!! :)

From:
https://wiki.midibox.org/ - **MIDIbox**

Permanent link:
https://wiki.midibox.org/doku.php?id=midibox_sequencer_vx&rev=1143225323

Last update: **2006/10/15 09:35**