Max and Rhythm

Generating Rhythms

The motive force in Max is the **metro** object. This is an object that simply emits bangs at a steady rate. Metro can easily be turned on or off by sending a 1 or 0 into the left inlet.

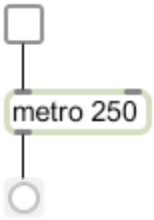

Figure 1.

The argument in the metro object is the period-- the number of milliseconds between bangs at the metro outlet. The rate of the metro is 1/periold. The period can be changed by a message in the right inlet. The period can be a float. This will result in accurate long term performance, but individual bangs will only be within one or two milliseconds of the expected time.

There are a couple of quirks to be aware of when using metro. When the period is changed, the change does not take effect until after the end of the current cycle. To get an immediate change in the rate, bang the left inlet immediately after setting the right inlet. This will restart the metro object with the new period. The exception to this rule is that if the new value arrives right on the tick, the rate will change immediately. One way to make this happen is to use the metro bang to apply the new value.

We can produce rhythms in Max by manipulating the period of a metro object. To do this properly, we must first understand the relationships between Period, Tempo, Note Value, and Note Duration.

Tempo

Tempo is the number of beats to play in a minute. In most computer music systems, the beat is assumed to be a quarter note. There is a tempo object in Max that produces a series of beat numbers, suitable for programming a visual metronome. (This odd reversal of names is probably related to the fact that the first version of Max was designed in French.) It is part of an old timing system called clockers that is seldom used any more.

Note Value

Note Value is the kind of note: half, whole, thirty-second triplet, whatever. In computer programs, these are represented by numbers. MIDI uses a system of ticks with 24 ticks to the quarter note[1]. This implies 48 ticks per half, and 12 ticks in an eighth. 24 is a convenient number, because you can represent a quarter note triplet with 16 ticks, or an eighth triplet with 8 ticks.

[1] 24 ticks is used for MIDI clocks. Files can specify a finer division to represent notes as they are actually played, but 24 ticks to a quarter is good enough for notation down to 32nd notes. You seldom see 96th notes.

Whole	96
Dotted Half	72
Half	48
Dotted Quarter	36
Triplet Half	32
Quarter	24
Dotted eighth	18
Triplet quarter	16
Eighth	12
Dotted 16th	9
Triplet 8th	8
16th	6
Dotted 32nd	4.5 [2]
Triplet 16th	4
32nd	3

Table 1. Ticks per note in MIDI

Duration

Duration is the amount of time the note will actually last, from key down to key up. We have to combine note value and tempo to find it.

To set the tempo, the first step is to find the duration of one tick in milliseconds. The formula is:

Tick Duration = 60000 / (Tempo * tick count of a beat)

For example, at tempo = 90, if the quarter note gets the beat

Tick Duration = 60000 / (90 * 24) = 27.7

To get note durations, the tick duration will be multiplied by the note values. With a tempo of 90 a quarter note will last 666 ms. The calculation can be simplified-- there are 2500 ms in 1/24th of a minute, so dividing the tempo into 2500 will give the result we need.

Figure 2 is a patcher to illustrate these calculations.

[2] Some years ago, durations were restricted to integer values for performance reasons. Modern computers can handle floats easily.

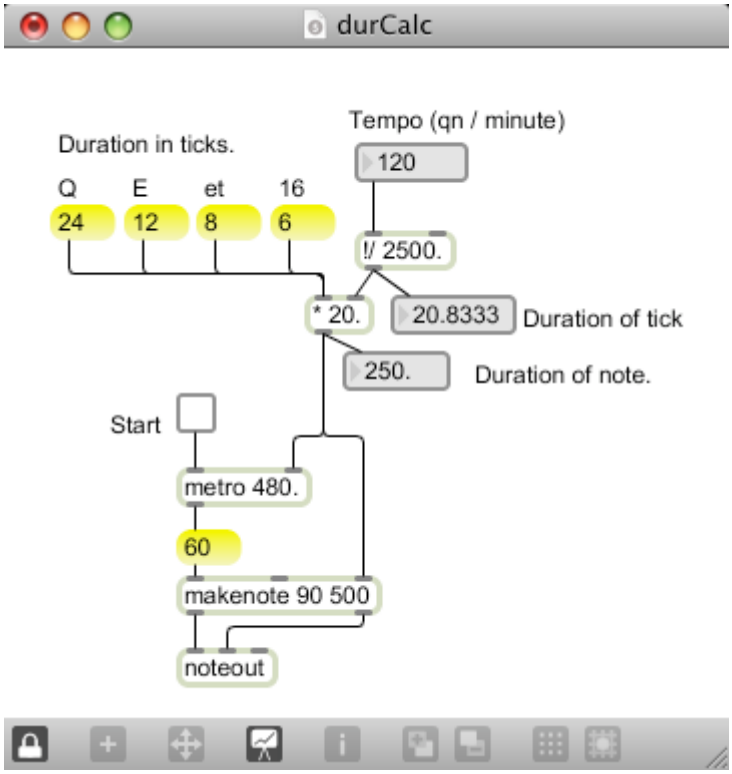

Figure 2.

This just plays Middle C at the tempo and value chosen. Set a tempo, turn on the metro object then click on the Q, E, et or 16 boxes to test various durations.

Notice that the operations are carried out as floats to maintain accuracy. Also notice that I have given an argument to the metro to set a tempo before any entry. The default metro period is 2 ms, so a patch like this would run away if it were started before any tempo was set.

There are a couple of things I should point out about generating rhythms in general. First, the duration has to be sent to makenote before the note is played.

Second, I prefer to set the duration at makenote a bit shorter than the duration at metro. This not only provides a natural breathing space, it avoids congestion in the MIDI stream that can happen when a lot of note off and note on messages arrive at the same time.

Patterns

Patterns in a Coll

Steady streams of notes quickly become boring. Most music includes a variety of durations, often in recognizable groups or patterns. Many composers generate a duration with each pitch, and then handle their notes as lists of pitch, velocity, duration. Here is how a series of note lists (stored in a coll) might be played.

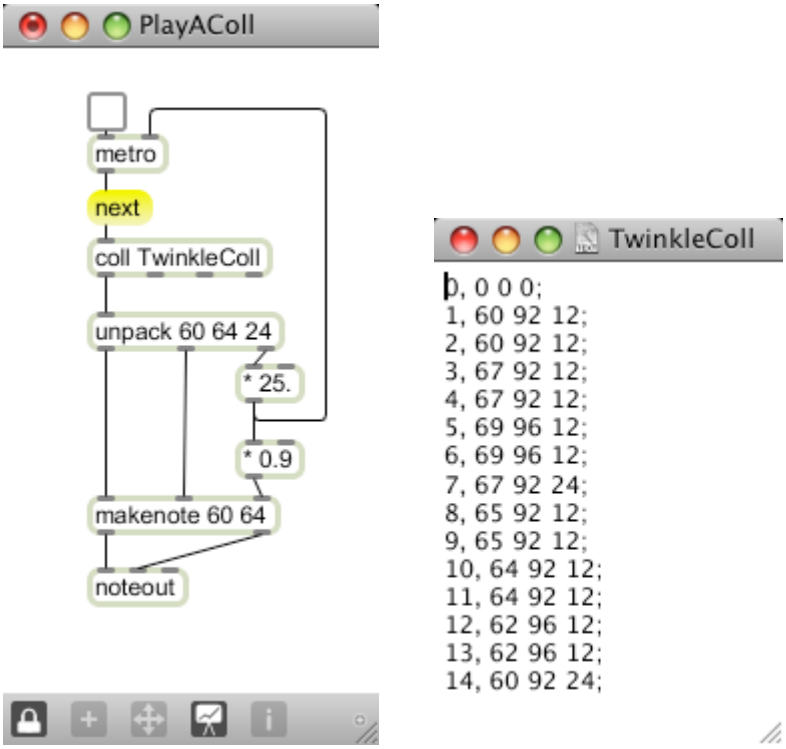

Figure 3.

On each metro bang, the next message pops a list of numbers out of the coll. This is taken apart by unpack. Each line contains pitch, velocity, and duration for one note. The durations are stored as tick counts, which are multiplied by tick duration (25.) to get a time in ms. This is applied to the metro and makenote. Velocity and pitch are fed directly to the makenote. The pitch, at the leftmost inlet, will trigger the note.

Coll is a complex object, but in cases like this it is easy to use. You can open the data editing window by holding the command key and double clicking on the coll in the patcher. Pay close attention to the punctuation in the coll file. There must be a comma after the line number and a semicolon at the end of each line[3].

When you close the data editing window, you will be asked to save the data. At this stage, you are simply saving the data in the coll object . This will not necessarily be saved when you close the patcher. If you give the coll object a name, you can save the contents of the coll in a file with the write message. The file will be loaded when you open the patcher. Alternatively, you can save the data in the patcher itself. To do this, open the coll inspector (cmd-I with the coll object selected) and set the Save Data with Patcher option[4]:

Figure 4.

[3] If you leave out a comma or semicolon, all data from then on will be lost when you save the coll.
[4] I've seen so much heartbreak caused by failure to do this that I'm tempted to write a country song about it.

unlist

Another approach is to think of rhythm patterns as basic units that get attached to pitches.
The problem presented to Max is to choose patterns at appropriate times. To facilitate this, I wrote an object called unlist. The function of unlist depends on the arguments. The operation when there are no arguments is simple. When a list is received at the left inlet, the first item in the list is passed through immediately. Subsequent bangs send the other items one at a time. When the list is empty, the right outlet bangs. If this bang fetches a new list its data will be smoothly passed along.

This patcher illustrates using unlist to produce rhythm patterns:

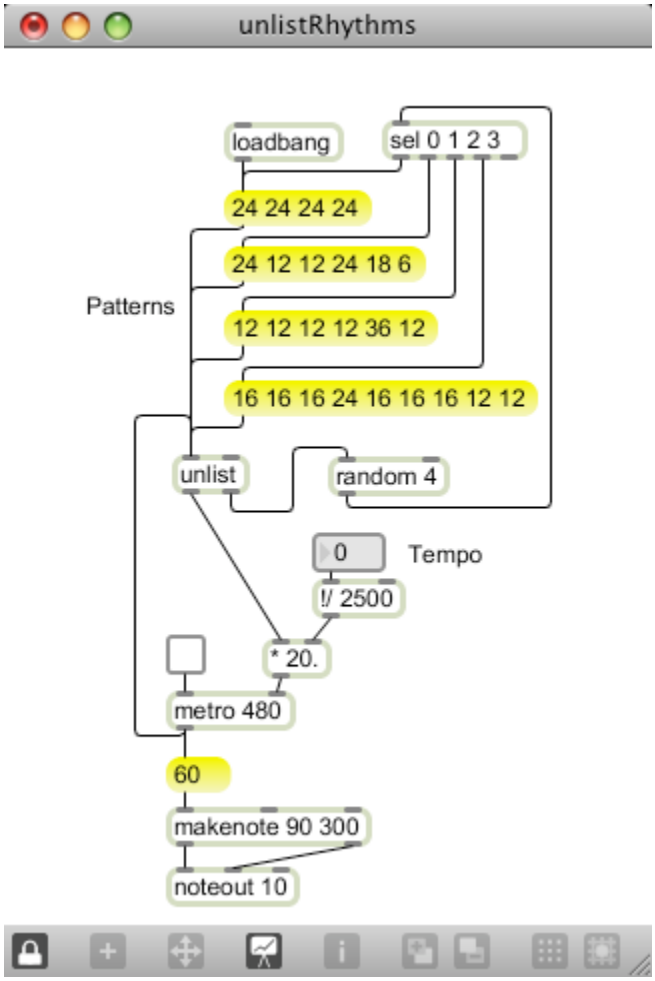

Figure 5.

There are four patterns here. The top one is chosen initially by the loadbang object. The first 24 in that list is sent through the multiplier to set the metro period. When the metro bangs, the next value from the list is brought down. When the list is used up the right outlet of unlist bangs to pick a random number, which prompts the select (sel) object to choose one of the patterns.

A simpler patch will make repeating patterns

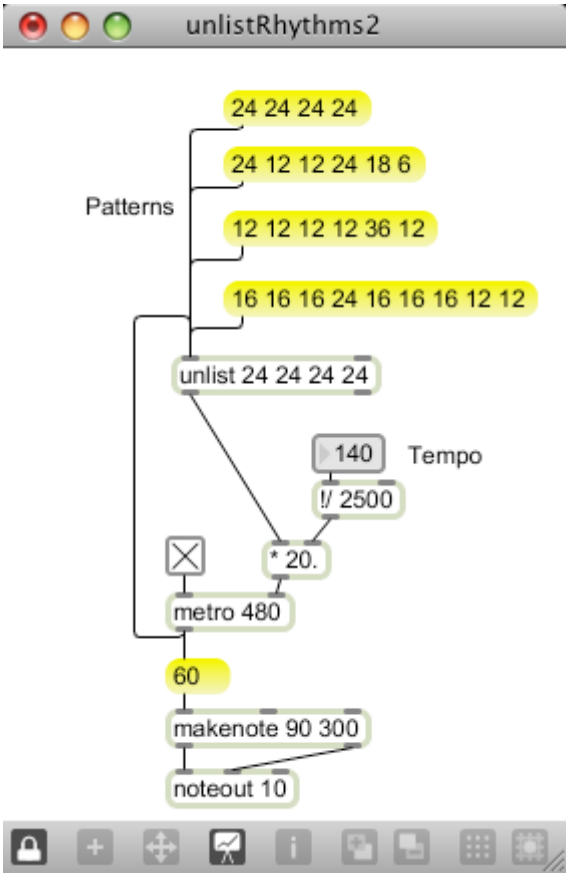

Figure 6.

When unlist has an argument list, it steps through the list over and over as the inlet is banged. If a new list is received, it replaces the stored list. This patcher will play straight quarter notes to start. If a list is clicked, the pattern will change. This approach is useful for interactive performance patches.

Rhythms by Subdividing

Another way of looking at rhythm is to think of notes as happening on a subdivision of a rapid pulse. This fragment of a patcher generates a continuous stream of bangs at the rate of 96 per quarter note. These are sent to receive objects labeled ticks:

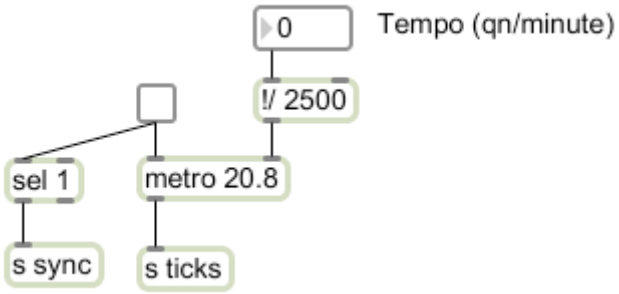

Figure 7.

Any number of simple sections can read this stream and produce notes that are triggered by counting ticks:

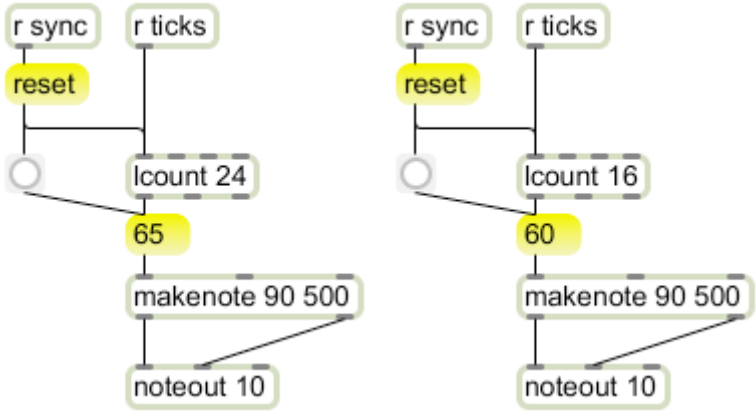

Figure 8.

The Lcount object will bang at the left outlet after the indicated number of bangs in. In these patches Lcount 24 will produce quarter notes, and Lcount 16 will produce triplets. The count size of Lcount can be manipulated in the same way as a metro period. This structure will produce dotted eighths and sixteenths:

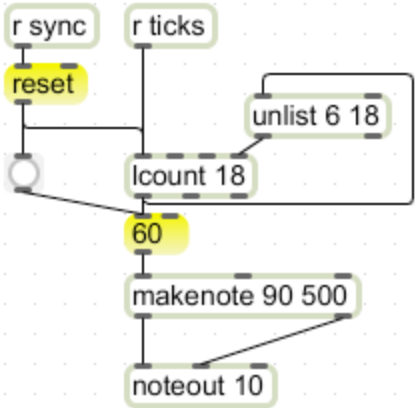

Figure 9.

Notice how the bang distributed through the send sync object both resets the Lcount objects and triggers the first note. The main advantage of this approach is that complex pattern generators are easily coordinated.

Rhythm Control by Transport

Figure 10.

The subdivision system is rather inefficient-- tick messages are constantly flying around and getting counted, which can clog up a patch with lots to do. The **transport** object keeps the tick counting within one object. Other objects, called **timepoint** can send transport a request to be notified when a specific time arrives. All of this happens behind the scenes, you need no patch cords between transport and timepoint. Instead, they are linked by a name attribute.

Attributes are an alternative way of setting arguments in certain Max objects. Instead of writing values in a specific order, each attribute has a name, indicated by the @ sign as in @clocksource. This is followed by the value (symbol in this case) to attach to the clocksource attribute. You can see the attribute names for an object, and the current settings by clicking and holding the left inlet. You can also set many attributes with a message that begins with the attribute name[5]. If an object requires both ordered arguments and attributes, put the attributes last.

The default name for transport and timepoint objects is <default>, which means they are linked to the "global transport" found in the extras menu. (Figure 10.) This transport is tricked out with features to set tempo and time signature and display the current time in bars and beats as well as H:M:S:ms format. There is even a tap tempo feature and a simple metronome.

All you need to use the global transport is a timepoint[6].

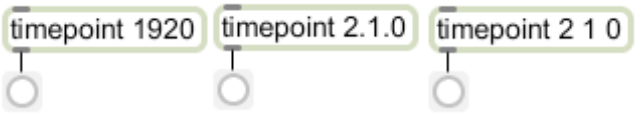

Figure 11.

The argument to timepoint defines the transport time for timepoint to bang. This time may be specified in various ways. A single number is the time in ticks. A tick is 1/480th of a quarter note (that's 17 microsecond resolution at a tempo of 120). Table 2 at the end shows the number of ticks in various durations. A duration of 1920 ticks is four quarter notes, so we would expect the first object to bang on the first beat of measure two (assuming 4/4). Max offers two more convenient ways of specifying the same time in bars, beats and units

[5] The fact that one of the most popular attribute names is @name is a bit confusing to explain, but you will soon get used to setting names for all sorts of things. You can also set names in the inspector window, but except for UI objects it's better if they show in the patch.

[6] You can set up your own named transport and timepoints if you want independent time streams.

(bbu). The notation 2.1.0 refers to measure 2 beat 1 tick 0, as does the list 2 1 0. Note that time in ticks begins at 0, whereas time in bars and beats begins at 1. Since there are 480 ticks in a beat, the highest beat number you see is 479.

The Max documents refer to the current time as position. Time in bars may also mean an interval- the time between events or some amount of time to add to the current position. Math in the bbu format can be tricky, so it is usually done in the equivalent ticks. Converting from ticks to bbu could also be tricky, but the translate object will do this for you. The precise behavior of translate is set by attributes @in @out and @mode. In and out can be a variety of time formats. It is important to set the proper mode -- position when you want to know where something is relative to the start, interval when you want to add times. Translate is illustrated in figure 12.

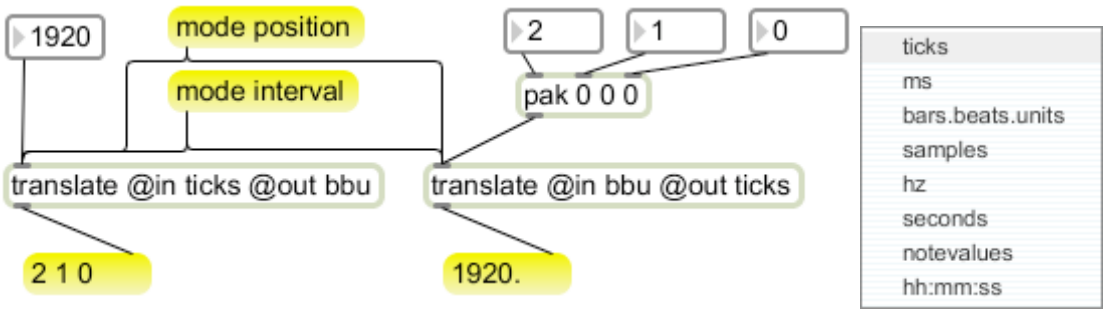

Figure 12.

The secret of using timepoint lies in figuring out times to ask for. The patch in figure 13 will behave like a metro, playing a note once per measure.

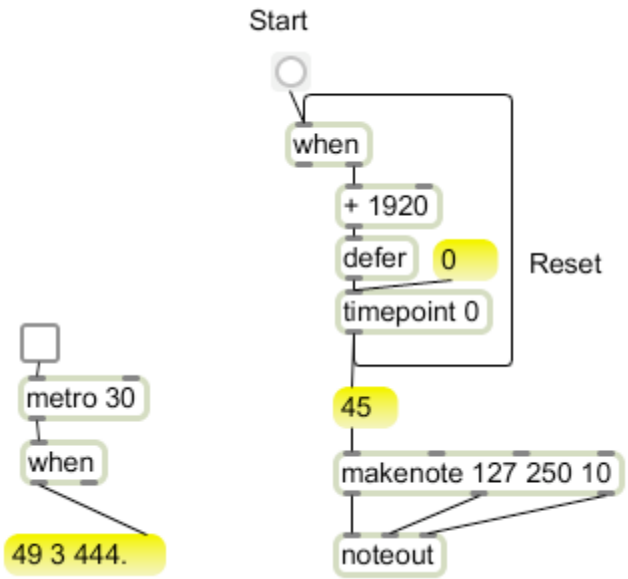

Figure 13.

The key to figure 13 is the **when** object, which provides the current time (ticks from the right, a list from the left) each time it is banged. The left section of figure 13 shows how to use when for a constant time display. In the right section, when needs an initial bang to grab the current time. This is increased by 1920 to set the next time a whole note into the future. When timepoint fires, the when object will be banged again. Note the defer object above the timepoint. This ensures that the next time is set after certain internal operations of timepoint are finished-- without this, timepoint occasionally gets stuck in the past[7].

Figure 13 will stop playing if the transport is rewound. It will start again when the transport gets back to where it left off, but if you want the patch to begin as the transport starts, send timepoint a 0. Applying a bang to the when object will restart this at any time.

Figure 14 shows how to use unlist with timepoints. If you don't want a time point to play, set it far in the future.

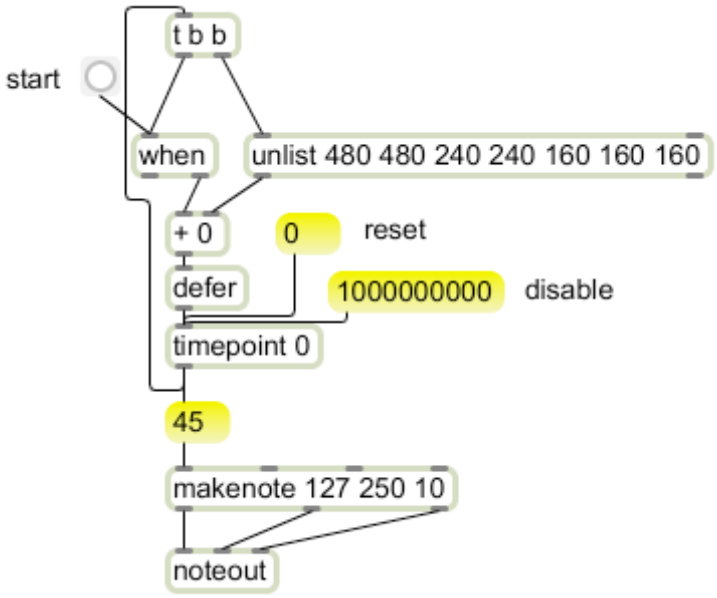

Figure 14.

Figure 15 shows how to play from a coll. With the exception of replacing an ordinary metro with the when and timepoint mechanism, this works like figure 3. The times in the coll are now expressed in transport compatible values (see table 2 at end.)

[7] I'm not sure if this is a temporary bug or a permanent feature. There's no harm done, because we are asking for times in the future anyway.

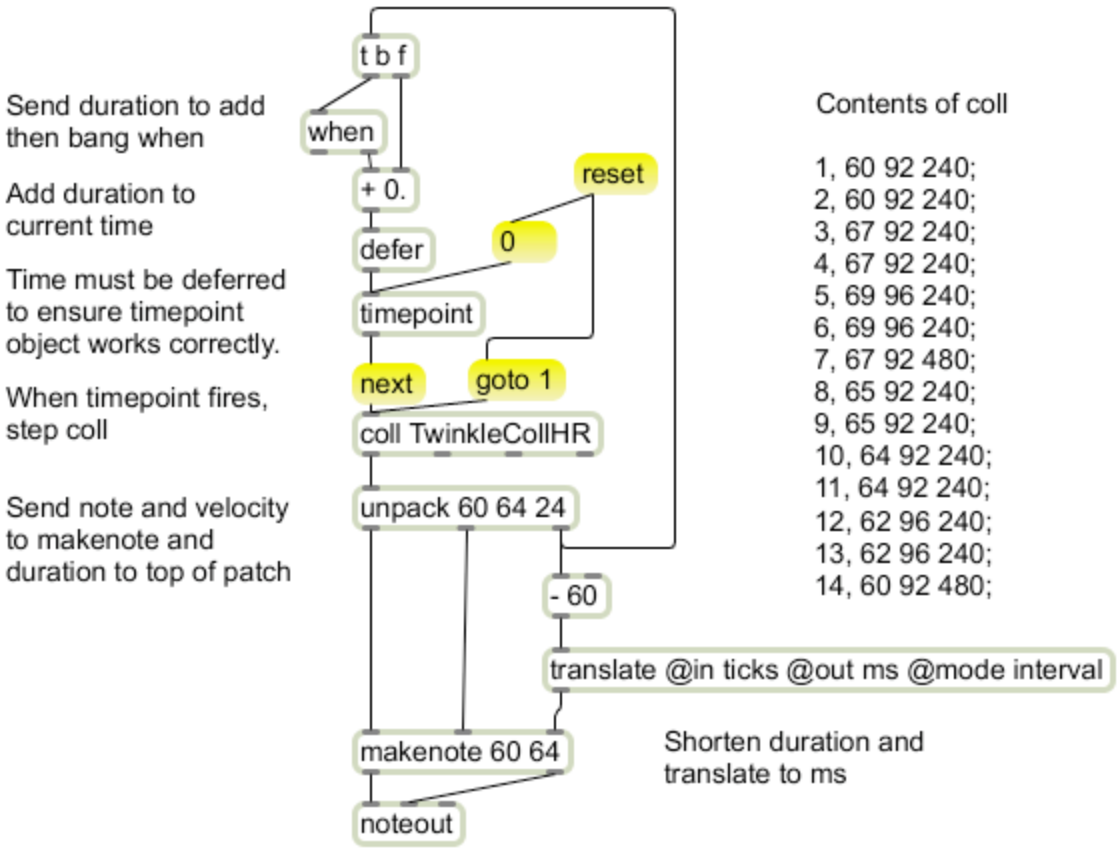

Figure 15.

Metros and Transport

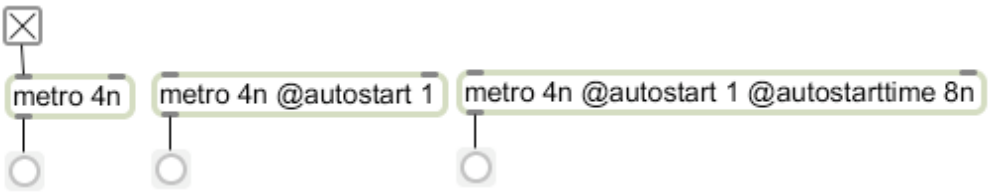

Figure 16.

Metro objects can be synchronized with the transport system. All you do is enter the metro period as a time relative to the tempo. The notation 4n 4nd and 4nt specify quarter note, dotted quarter and quarter triplet, with similar terminology from 128n up to 1nd. When a metro is getting time values in this way, it will only run if the transport is running. It is not synchronized to the transport, it is merely ticking away at its own rate, but will pause when the transport pauses. If it is started off the beat, it will remain off the beat.

The @autostart attribute will start the metro with the transport. The second metro in figure 16 bangs on every quarter note. It will not start unless the transport is set to 1.1.0, because the time the metro is waiting for is 0 ticks. The @autostarttime attribute can be used to set a different time point for the metro to be started. The third metro in figure 16 will tick on the afterbeats. Note that once a metro has been started it will continue to bang, even if the transport is rewound before the @autostarttime. It will pause with the transport if a time interval is used as the argument, but you have to send it a 0 to shut it up when the transport is running.

It is possible to get metros out of sync with the transport by stopping and starting them with 1 and 0 toggle. The @quantize attribute will force the metro to wait to the next specified division to start after it is sent a 1. @quantize 4n will make the metro wait for the next beat to start.

If a qmetro is required (as in jitter operations) you can make a metro behave like qmetro with the @defer attribute.

Here are the rhythm values defined for transport.

Name	MIDI ticks	Max ticks	Max name
Dotted Whole	144	2880	1nd
Whole	96	1920	1n
Dotted Half	72	1440	2nd
Triplet Whole	64	1280	1nt
Half	48	960	2n
Dotted Quarter	36	720	4nd
Triplet Half	32	640	2nt
Quarter	24	480	4n
Dotted eighth	18	360	8nd
Triplet quarter	16	320	4nt
Eighth	12	240	8n
Dotted 16th	9	180	16nd
Triplet 8th	8	160	8nt
16th	6	120	16n
Dotted 32nd	4.5	90	32nd
Triplet 16th	4	80	16nt
32nd	3	60	32n
Dotted 64th	-	45	64nd
Triplet 32nd	-	40	32nt
64th	-	30	64n
128th	-	15	128n

Table 2. Time values for transport.