MIDI Time Code

In the video or film production process, it is common to have the various audio tracks (dialog, effects, music) on individual players that are electronically synchronized with the picture. SMPTE time code is the standard used for this synchronization.

Film and video are a series of frames, and SMPTE time code consists of data for each frame telling hour, minute, second and frame number within the second. In most of the world, there are 25 frames per second. In the US there are 24 frames per second on film and 30 (approximately) frames per second in video. Color video actually runs a bit slower than the original black and white standard, so there are only 29.97 frames in a second. This creates a discrepancy between frame numbers and real time. The discrepancy is usually ignored, but it can be compensated for by the "drop frame" numbering system in which the first second in each minute, excepting minutes divisible by 10, starts with frame 2.

MIDI Time code is a specialized variant of SMPTE that allows music software to follow along. An MTC aware program should:

1. Wait for a specified "hit point"
2. Adjust its tempo (and pitch if it plays audio) if the timecode varies in speed.
3. Stop when code ends, but not too quickly, because code often has brief dropouts.
4. Restart in the middle of things if that's what the code does.

MTC Messages

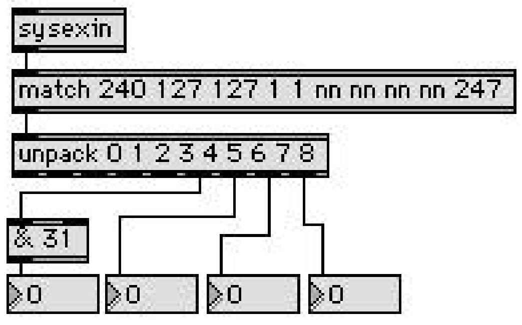
MTC comes in two forms. Full Frame Messages convey the position of a tape in a single message, which is generally transmitted after a device has executed a locate or a stop. Quarter Frame messages are transmitted continuously as tape is moving. Each QFM has a single byte of data, and it takes eight of them to covey a complete location.

Displaying Full Frame Messages

The FFM is transmitted via my favorite oxymoron, a universal system exclusive message. The format of the message is:

240 127 127 1 1 hours minutes seconds frames 247

That's a total of 10 bytes. [match 240 127 127 1 1 nn nn nn nn 247] does a good job of trapping that string out of sysexin and unpack will take it apart for us. We are interested in what unpack calls elements 6 - 9. Minutes, seconds and frames can be displayed directly.



Full Frame Messages

The hours value contains a code for the frame rate in bits 5 and 6 (the least significant bit is bit 0) and the hour is bits 0-5. To extract the hour, use & 31 to mask off the unwanted bits. To read the format (seldom necessary) use the right shift object >> 5. The rate codes are:

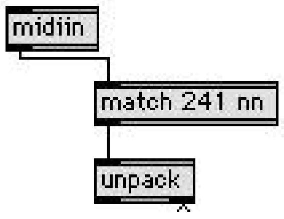
0 = 24 frames a second
1 = 25 fps
2 = 30 fps drop frame encoded
4 = 30 fps normal encoding

Displaying Quarter Frame Messages

Quarter Frame messages are only a little more complex to handle. A QFM has the format:

241 data

These are system common, not real time, so they should never turn up in the middle of another message the way Clock messages can. Match 241 nn will snag them for us, and unpack will shoot the desired numbers from the right outlet[1].

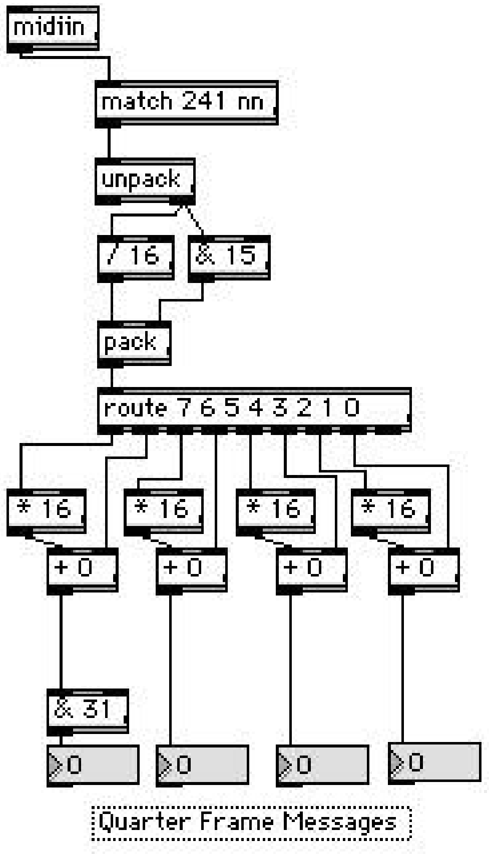


[1] You may have to look around the ports to find the Time Code. On a simple interface, midiin should show it. On an interface that has its own timing features, such as the Studio 64XTC, there will be a special port dedicated to it. (The time port on the Studio 64TXC is called studio 64 XTC.)

The data byte of the QFM is encoded like this: (0 x x x y y y y) the x bits indicate which part of the message you are looking at, and the y bits are either the low four or upper three bits of the numbers we want. The coding is like this:

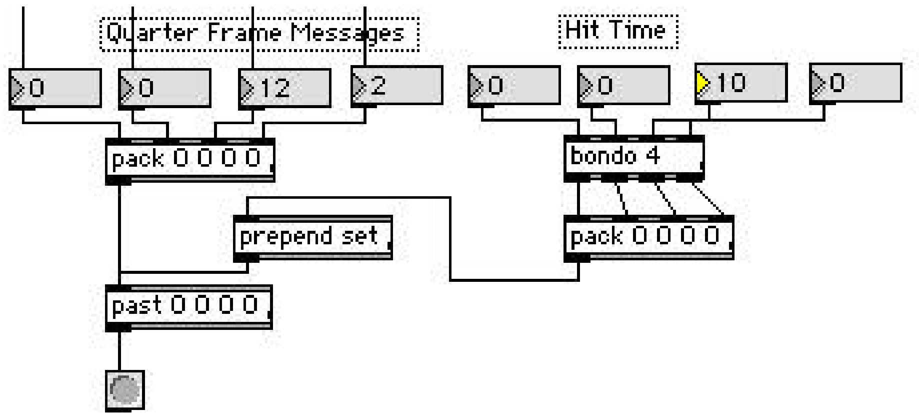
xxx	yyyy
0	Least significant 4 bits of frame number
1	Most Significant bits of frame
2	LSb of seconds
3	MSb of seconds
4	LSb of minutes
5	MSb of minutes
6	LSb of hours
7	MSb of hours

To decode this, break the data byte with / 16 and & 15. Pack these two into a list, and run it through a route 7 6 5 4 3 2 1 0. The outlets for 7 5 3 and 1 get multiplied by 16 and added to the outlet on their right. Frames, seconds and minutes are ready to display, but the hours value includes frame format just like the full frame message.



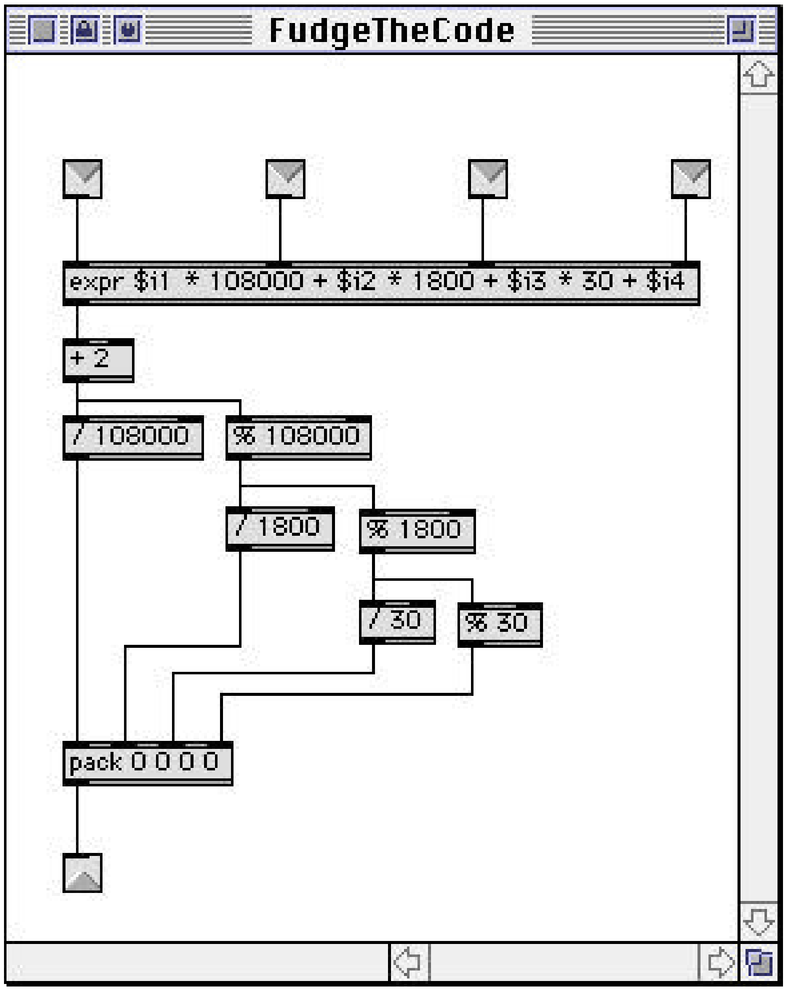
Triggering Events

Triggering events from Quarter frame messages is a bit tricky. For one thing, you only get every other number, and there's no guarantee that the even or odd frames numbers are being sent (it depends on where the tape stopped), so if you just watch for a particular frame number, you may miss the cue. This is a job for the past object:

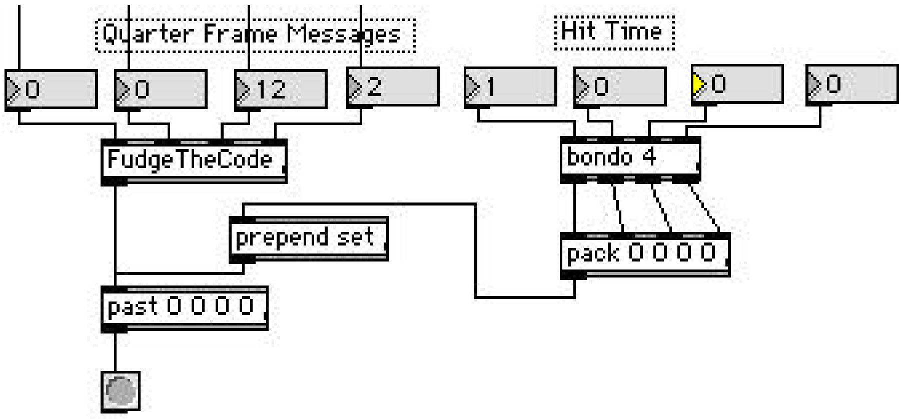


Set the time you want on the right set of number boxes (bondo ensures that the entire list is sent out when you change any digit.) Past will trigger when the incoming time is equal to or greater than the hit time. For most musical purposes, this is a good enough solution.

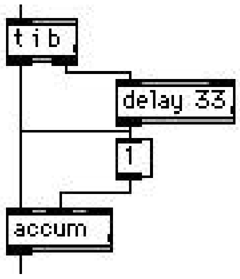
It's not exact though. The number you are decoding is the number of the frame that started at the same time the first part of the frame number was sent. That means by the time you have received the entire location, it's almost two frames later. I'll fix that with a subpatcher:



In this subpatch the individual inputs for hours, minutes, seconds and frames are converted into a total number of frames (assuming 30 frames per second). Two frames are added and the result is reformatted into a list. It fits into the previous patch like this:



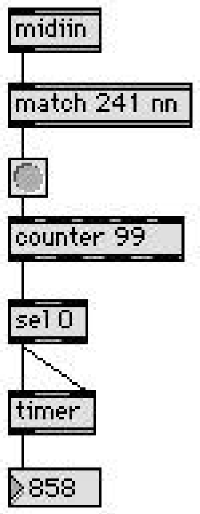
We still need to deal with the fact that MTC is only giving every other frame. I'll add this mechanism at the expr outlet within FudgeTheCode:



This passes the received time through, then, 33 ms later, adds a fake frame. Once this is done, the patch will run one frame past wherever the time code stops, but that is normal behavior for synchronized systems. In fact, most devices run an extra 20 frames or so to allow for dropouts in the code.

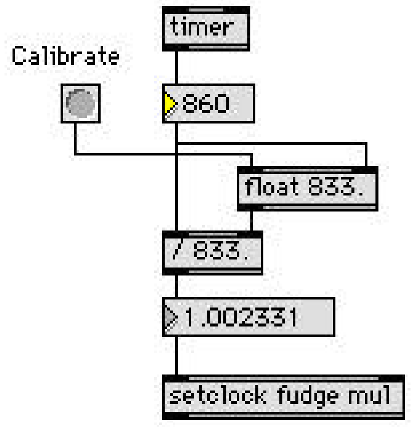
Adjusting Tempo

If OMS is installed, the various metro and clock objects in a patch can be made to vary their rate with the incoming time code. This is done using the setclock object. First we must time the quarter frame messages:

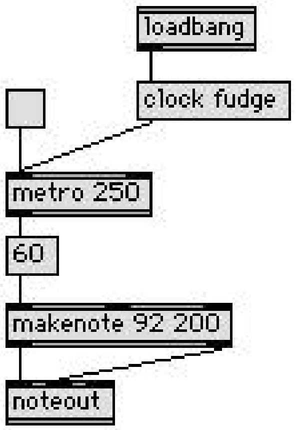


This will give us a period in milliseconds for 100 QFMs. Of course, the relationship between the Macintosh internal clock and reality is rather vague. The output of the timer should be 833, as a QFM is produced every 8.33 ms.

We have to include a calibration feature when we compute the value to send to setclock:

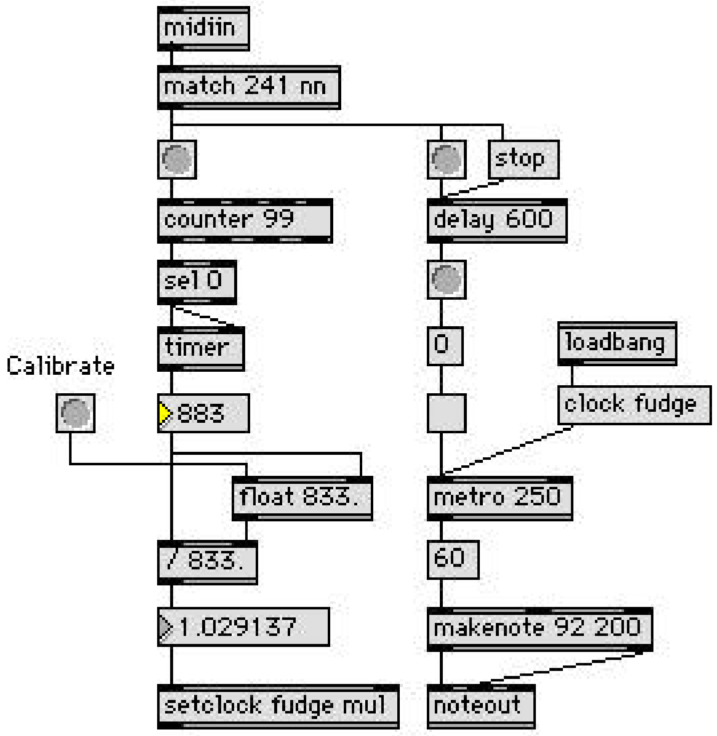


Here, the nominal timing can be captured as the divisor if the button is clicked when the timecode is running at correct speed. (i.e. straight out of the generator.) The current period divided by the correct period will give the deviation, suitable for feeding the setclock object in mul mode. Fudge is the name of the setclock. Here is a simple patch that is controlled by fudge:



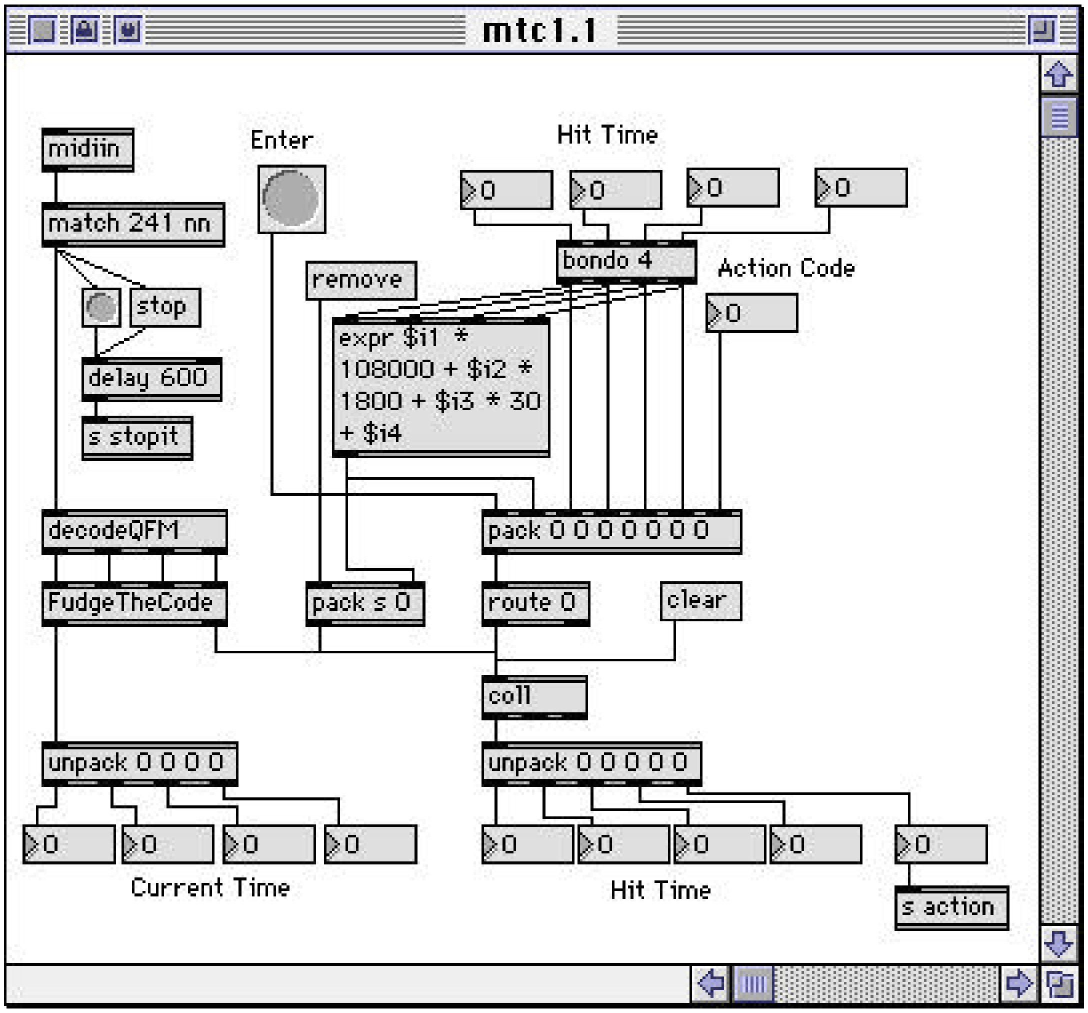
Detecting End of Code

When the timecode stops, the patch should stop. However, code is often briefly interrupted because of tape dropouts and other problems. Here is the entire patch including a delay object to shut the metro off after a bit of "freewheeling".



Multiple Hits

Everything so far was designed to trigger one event per patch. Here's a patch that allows many hits, and features some editing of the times and events. It might be useful for playing sound effects from a sampler:



The left side of this should be familiar. I've consolidated a lot of the logic boxes in a single subpatcher "decodeQFM". I've made one change to FudgeTheCode: I added an outlet that will provide the time as a running frame count (by tapping off just below the + 2 box).

The frame count can be used to pop data out of a coll. If we store lists with the frame count as the address, they will be produced when the timecode reaches the desired point. This will only be as reliable as the timecode, but if it is coming from a digital source such as Vision or an ADAT, it'll be fine.

The data that goes into the coll has the format:

framecount, hours minutes seconds frame action;

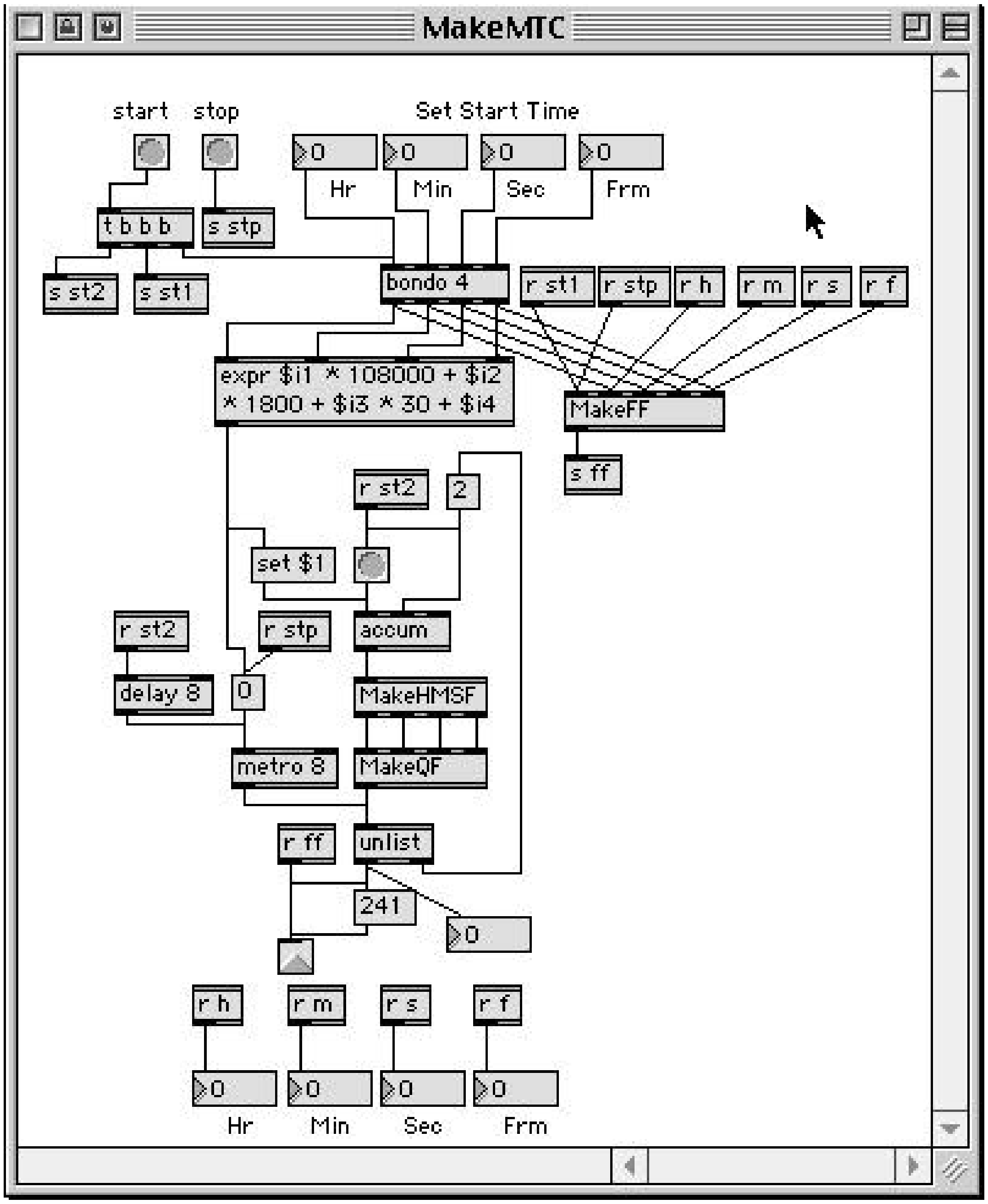
The frame count is calculated by the expr. The time is also stored as hmsf for editing purposes. (You can open up the coll to delete an action or change the code.) The action code is an arbitrary int that will be used to trigger events. All this is stored when the enter button is clicked. Clicking remove will delete the current time data, and clear will empty the coll.

There's potential for lots of interesting features on this patch. A button to transfer the current time as a cue, using next to step through the coll, and so forth.

Faking MTC

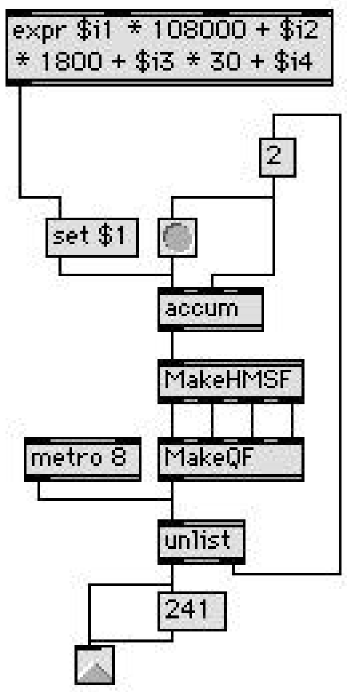
There isn't much point in using Max to generate MIDI Time Code. It won't have accurate timing, and OMS won't output it reliably. It will sort of work in a setup without OMS, but without OMS, you can't adjust the timing at all! Max is pretty much limited to follow an external source. (It will follow timecode from Vision via the aic buss.)

On the other hand, you may find yourself wanting to test a patch with no source of MTC handy. Here's a patch that will do it; it's designed to be installed as a bpatcher within some other patch.



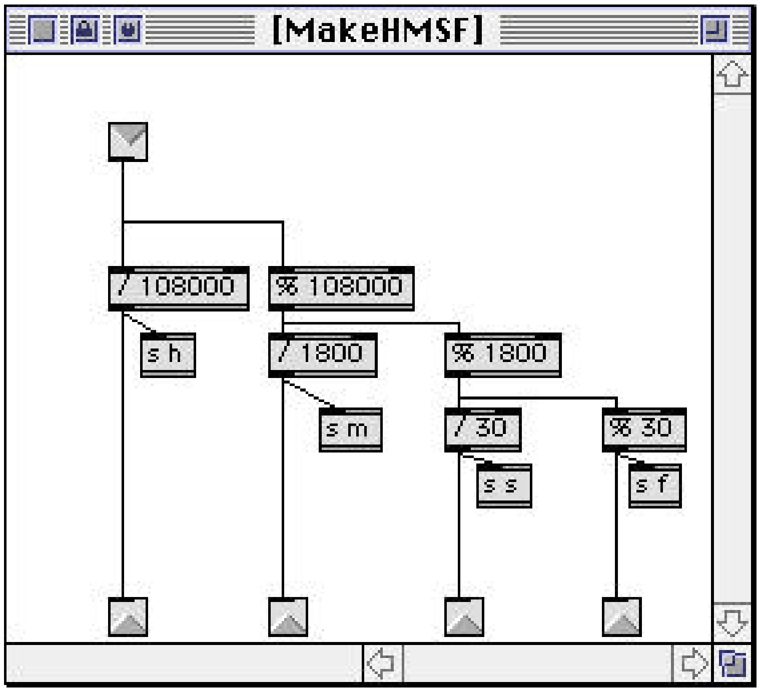
When the start button is clicked, it produces a full frame message, then a series of quarter frame messages until the stop button is clicked, when a full frame message with the final location is sent. Any desired start time can be set with number boxes, and another set of boxes displays the current time.

The heart of the patch is the accum object, which contains the current total frame count. The initial value of the accum is calculated by the expr object. (These constants are for 30 fps.)



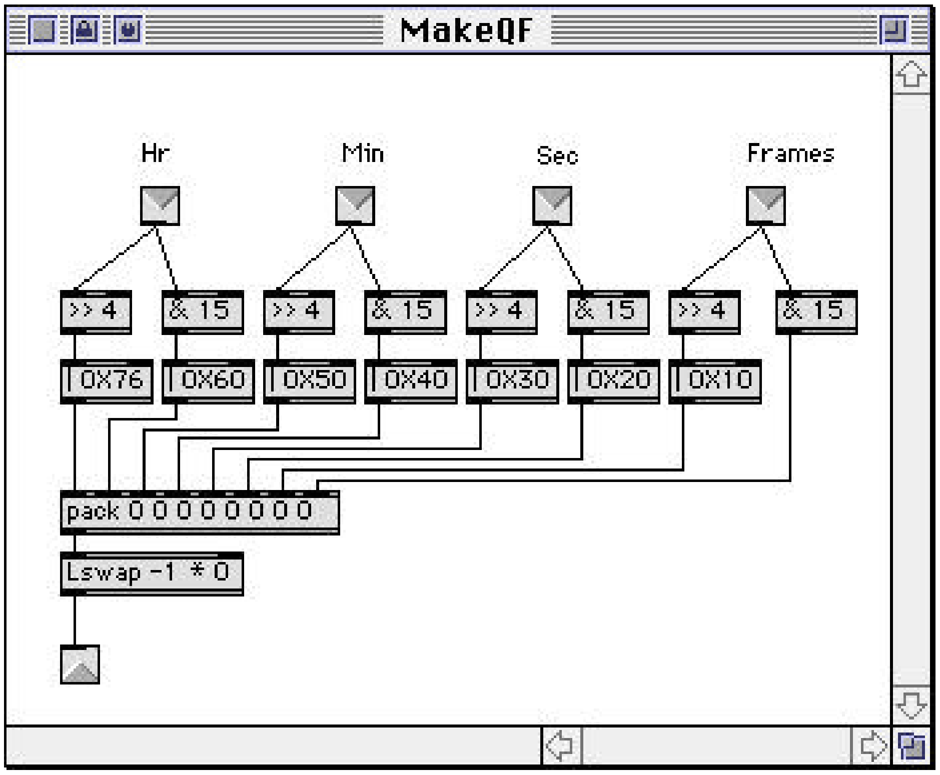
The subpatcher MakeHMSF converts the count to hours, minutes, seconds and frames. The MakeQF subpatcher converts this data into a list of 8 data bytes for quarter frame messages. The unlist object accepts this list, sending the first message to the outlet immediately. (The status byte 241 is sent first due to right/left precedence.) As the metro object bangs unlist, the remaining messages for the frame are sent out. When the unlist object is empty, the right outlet bangs, which will add 2 to the accumulator and start the next cycle.

The MakeHMSF subpatcher is the reverse of the expr:



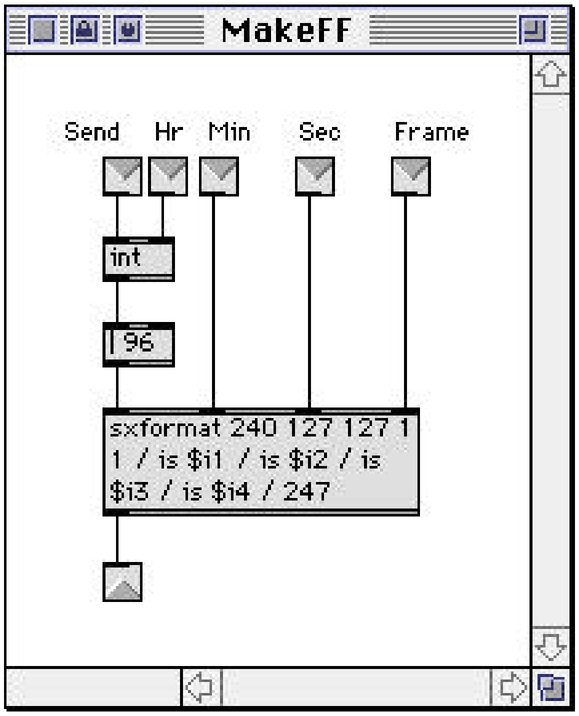
Note that the data bytes are sent to remote receive objects as well as output from the subpatcher.

The real conversion is in the MakeQF subpatcher:



Here the data bytes for hours, minutes, seconds, and frames are each split into two nibbles. The right shift produces the MSB and the mask & 15 leaves the LSB. The eight resulting bytes are each ORed with the value required to mark their place in the quarter frame scheme. I have used hex notation here for clarity. You can see how the four bits of data will replace the 0s in the OR boxes. The MSB for hours is ORed with 0x76 to include the frame rate as well as the byte number. The bytes are then packed into a list. They have to be sent out starting with the frames LSB, so the Lswap object is used to reverse the list.

The makeFF object formats the same data into the full frame message:



```
sxformat 240 127 127 1
1 / is $i1 / is $i2 / is
$i3 / is $i4 / 247
```

Nothing tricky here. The hours data is captured in an int object to prevent sending extra messages when the start time setting is changed. It takes a bang in the left inlet to produce the message. The hours value is ORed with 96 (0x60 in hex) to set bits 5 and 6, indicating 30 fps time code.

Now look at the MakeMTC patch again and follow the whole sequence:

When the user enters a start time, it is fed via bondo to the frame expr, which sets the accum without causing output. The bondo also passes the setting to the MakeFF subpatcher, where it is held. Entering data also stops the metro if it is running, to prevent sending out of sequence time code.

When the start button is clicked, the trigger object (t) will first send a bang to the bondo, resetting the start time in case the user has not chaged setting since the last run.

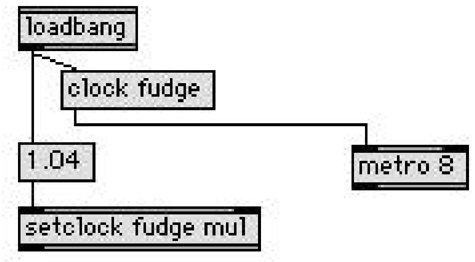
Next, the trigger sends a bang via st1 to the MakeFF object, which sends the starting full frame message.

Finally, the trigger bangs send st2, which goes to two locations: it bangs the accum which causes the first quarter frame message as described above, and it begins an 8ms delay which will start the metro when the second QFM is required.

As the patch runs, the time is displayed in the number boxes on the right. The time data gets there from the send objects inside of MakeHMSF via matching receive objects. Another set of receive objects keeps the time in MakeFF current.

When stop is clicked, a bang sent through s stp stops the metro and causes MakeFF to send the last location as a full frame message.

As it stands, the patch runs too fast. The 8 ms period of the metro object should be 8.33 ms. A setclock in mul mode, with a multiplier of 1.04 will bring the timing a little closer to what it ought to be, but it still won't be accurate enough for any but the most forgiving situations. [2]



If you use 25 frames per second as your frame rate, a quarter frame gets exactly 10 ms. If you can use this, and have msp set to “scheduler in audio interrupt”, you will get accurate timing. I will leave the conversion of this patcher to 25 fps as an exercise for the reader.

[2] And the scheduler configuration has to be set to 1 ms granularity.