Max and Numbers

Integer Arithmetic

Most of the time, we do math in Max using the int data type. This will work well, unless we want to divide 5 by 3. That'll give 1 as a result- as long as you expect it it's OK, but you might be fooled sometimes. If you really need 5/3, use floats. Floats are indicated in objects by the decimal point. 1 is an int, 1. is a float. When a float is applied where an int is expected, like a plain number box, the fractional part is discarded. That means 1.9999 = 1. To prevent this, round off floats when converting them to ints. The Lround object with no argument does this nicely, following the convention of rounding final 5s to the even digit.

The Lowdown On Floats

The float data type seems like a familiar sort of number- most of us have been writing decimals since grade school. However, there are some features of the way computers deal with floating point numbers that you should be aware of.

Even if a number is shown on the screen as a decimal number, the computer is still working with bits, and has a limited number of them at its disposal. In the case of Max, that's 32 bits. These are used to represent the infinite range of decimal numbers by a scheme called floating point notation. You may be familiar with a version of this known as scientific notation- to represent 4823, you write $4.823 \text{ X } 10^4$.

To encode floating point numbers, Max follows a convention known as IEEE single precision floating point format. This uses 32 bits as follows:

1 bit to indicate sign (0 for positive)
8 bits for the exponent
23 bits for the significand, which has the form b_0.bbbbbbbbbbbbbbbbbbbbbbb
where each b is either 1 or 0

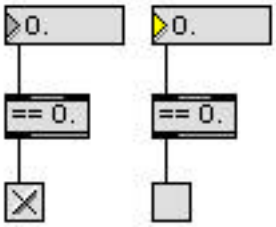
The actual value of the number is ± significand x 2^{exponent}

There is further massaging to save bits or processing time where possible. For instance, the exponent can be positive or negative, but instead of having a sign bit in the exponent, 127 is subtracted from the exponent when the number is converted. With 8 bits, the exponent may range from –127 to 128. In addition b_0 is unnecesary since it's always the same. (1 for most numbers, 0 if the exponent is –127.)

The result of all this is that floats can sometimes surprise you. For instance, you cannot represent all possible numbers with this scheme. In fact there are fewer floating point numbers than there are ints. The differece is that while ints are simply limited in magnitude, floats are spotty, jumping from value to value. With large numbers, the jumps can be pretty big, say from 134217712 to 134217720.

With small numbers some odd things happen too. You can see this by stretching a float box, typing in 0, and scrubbing the value up. You'll see the numbers change by steps of 0.01 up to 1.14, but the next value is 1.149999. That's because you can't actually represent 0.01000000000000000 as a binary number times some binary power of two. The value is really something like 0.0099999999999, which will round up to 0.01, but if you keep adding it in, eventually the rounding error catches up with you and 1.149999 pops up.

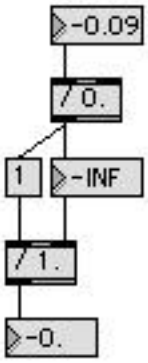
Here's another one. Tack a == 0.0 on that flaot box and watch the output with a toggle. If you type 0 into the box, the toggle shows that 0 in fact equals 0.0. However, if you scrub the box away from zero and back, you won't see an equality again. The number is probably 0.00000000002 or something.



The float box rounds off what is displayed, but sends the more precise value out. You can get a handle on this by rounding floats off yourself. The easy way to do this is use Lround.

The are other oddities- for instance, if you ever try to divide something by 0, you will see "INF" or perhaps "-INF" as the result. As you might guess, this means infinity.

Surprisingly, you can still do math with INF: 1/ -INF = -0!



Another peculiar result you may see is "NaN". This stands for "not a number", and happens if you take the square root of –1, divide 0 by 0 or do anything else that is not defined.

Working With Bits

We are used to dealing with two number formats in Max, int and float, and there are plenty of tools for manipulating each. Sometimes we come across other formats, especially when we try to work in the wild and wacky world of system exclusive code. There we are likely to encounter 4 and 7 bit nibblized values, 2s complement and constant offset signed numbers, packed bytes, and other wonders.

Hexadecimal and Binary

Hex and binary are not really distinct formats, they are simply ways of looking at ordinary numbers. All you have to do to see the hex or binary version of a number is set an option in a number box. I mention them here because sysex documentation is usually written in hex and binary. To make conversion easy, I keep a gadget like this in my patcher:

Incidentally, although most texts make a point of explaining hex to decimal conversion, we almost never do that. Hex notation is mostly used as shorthand for binary, so it is worth learning the binary equivalents of the hex characters:

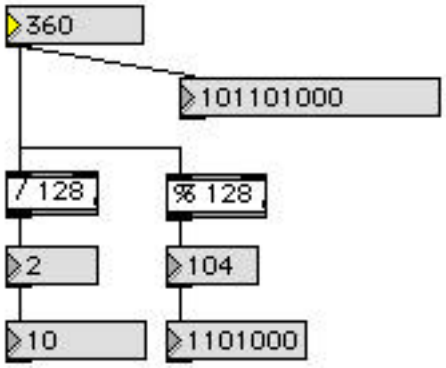
Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Notice that a hex character always encodes 4 bits. When decimal and hex notation is mixed, the hex numbers are marked in various ways, such as 345h or 0x345. The latter can used in Max objects, but will probably revert to decimal notation when the patcher is saved. (The xxh notation can be used with a few Lobjects, as I'll explain later.)

Nibblized Numbers

Since the most significant bit of MIDI data bytes is always 0, the largest value that can be directly represented is 127. Larger numbers must be broken into less than byte sized chunks called nibbles.

The most common sized nibble is 7 bits. This is the format you see in the 14 bit values for pitch bend and precision controllers, as well as many sysex messages. Here is a patch fragment that shows how these numbers break in two:

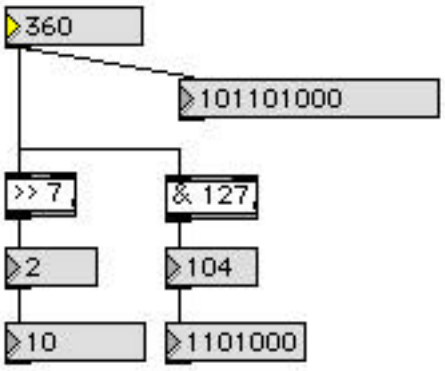


The number 357 requires 9 bits in binary representation. The / 128 operation discards the rightmost 7 bits and the % 128 turns all but those 7 bits to 0. How do these work? When you divide using integer division, you discard the remainder. The rem or modulus operator (%) gives you the remainder only If you are dividing by a power of two, the remainder is the same as the rightmost bits, where the number of bits is the same as the power. 128 is 2 to 7th, so you either discard or save the right 7 bits.

(Max doesn't display leading zeros, so you have to imagine the resulting two bytes are 00000010 and 01101000.)

Shift and Mask

Here's another way to do the same thing:



These use binary operators. >> 7 is right shift 7 steps, which directly carves off 7 bits. The bitwise AND (&) is also known as masking. When a bitwise AND is performed on two numbers, the result will have 1s only where both of the operands have 1s. As an example,

```
    10111
AND 00100
    00100
```

When you mask with a power of two, only a single bit will remain. If you mask with (2**n)- 1 only the rightmost n bits will remain. In the previous example, & 127 masks all but 7 bits.

We mask bits so often, it's worth committing this chart to memory:

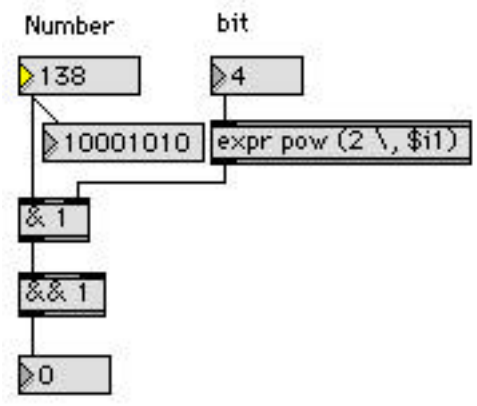
Bitmasks

bit	dec	hex	bit	dec	hex
0	1	0x01	4	16	0x0F
1	2	0x02	5	32	0x10
2	4	0x04	6	64	0x20
3	8	0x08	7	128	0x40

A little nomenclature explanation: The positions in a binary number are numbered 0 to whatever. The rightmost or least significant bit is bit 0 because it represents two to the zero power. The leftmost is (number of bits - 1). You also see these referred to as lsb and msb.

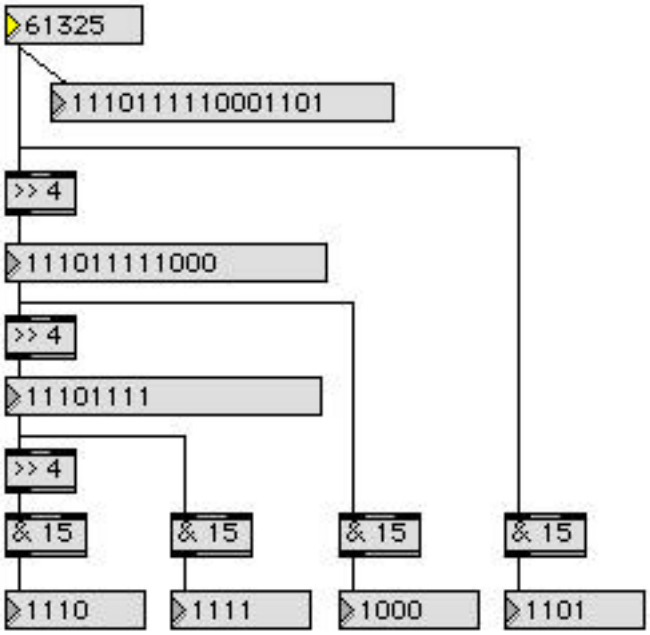
You can find assorted bits by adding their masks together. For instance, to isolate bits 1-3 add 2 + 4 + 8, getting 14 or 0x0E.

The bitwise AND must not be confused with the logical AND (&&). Logical AND returns either 1 or 0, yielding 1 only if neither operand is 0. We can use both kinds of AND together to find the value of an arbitrary bit within a number:



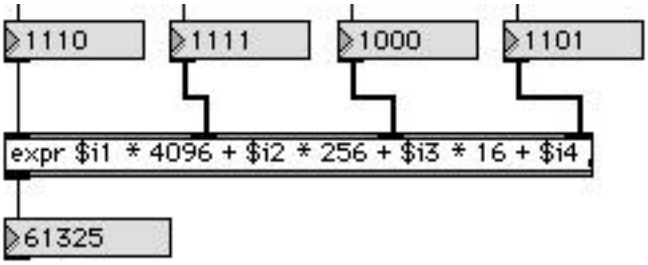
In this example we find that bit 4 of 138 (10001010 in binary) is 0.

Sysex messages from some synthesizers use 4 bit nibbles, and a big number will be spread over as many as 4 bytes. To create these bytes (or 7 bit nibbles for numbers with more than 14 bits) you need to cascade the nibblizers:



Again, you have to imagine that each of the resulting bytes starts with four zeros.

Numbers that have been broken apart like this will eventually need to be put back together. Usually an expr object will do the job:



Notice the magic numbers for 4 bit nibbles are 4096, 256, and 16. These are the divisors you would use if you prefer the division method of nibblizing. (The binary operators are a bit faster, but unless you are doing thousands of conversions, the speed difference is not noticeable.)

To rebuild a 14 bit number out of 7 bit nibbles,

expr $i1 * 128 + $i2

does the trick. The $i1 *128 could be replaced by the left shift operator

expr ($i1 << 7) + $i2

The parentheses are required to make sure the shift operation occurs first.

Negative Numbers

When a number is brought in as part of a sysex message. Max treats it as a positive integer. Negative numbers need another stage of conversion.

Negative numbers in offset format are easy- you just subtract the midpoint value (or add it going out). The most common is the pan value, defined as -64 to +63. The value you see will be from 0 to 127. Subtract 64 to get the proper display.

Larger negative numbers are usually encoded as 2s complements. This takes some explaining:

There are two limitations in the circuitry computers use to do math. First, it can add, but not subtract, and second, only a certain number of bits can be handled at a time. Luckily, we can use the second shortcoming to overcome the first. To understand this, imagine an adder that can only handle 4 bits. That means it can hold binary values from 0000 to 1111, representing 0 to 15 in decimal. As you count up from say 1101, you'd see:

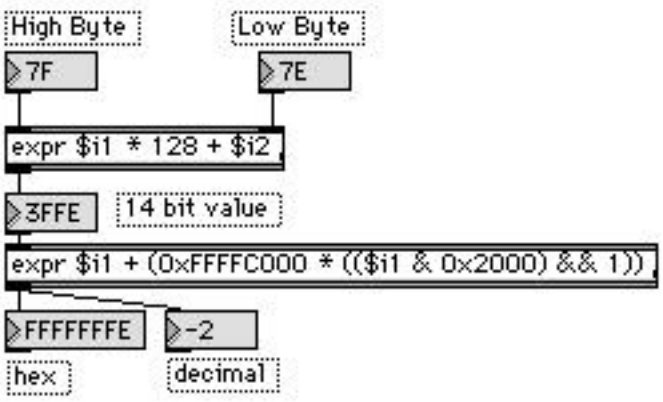
1101
1110
1111
0000
0001

and so on. The phenomenon of jumping from the highest to the lowest number is called overflow or wraparound. If this happens when you are adding numbers it's an error and can cause problems. (Try using Max to add 2,147,483,647 and 2) However, since the counting system has this accidental circularity, you can do the equivalent of any subtraction by adding the appropriate very large number. In the case of the four bit system, to subtract 1 we'd add 15.

To create the number that fakes subtraction, you start with the binary number you want to subtract, change all the 1s to 0 and all the 0s to 1 (bitwise complement), and add 1. This number is called the twos complement. In computers, negative numbers are stored as twos complements, and converted only for display. Max does this, as you can see by entering -1 in a number box and then changing the box to binary mode. You can always

spot a negative binary number because the most significant bit will be 1. That's why the msb is sometimes called the sign bit.

When decoding sysex messages, we occasionally encounter 14 bit two's complement numbers. Since Max uses 32 bit integers, it will display binary 1111111111 as 16383 instead of -1. The missing 18 bits were assumed to be 0. This patch will fix it up:



What we are doing is looking at the most significant bit of the incoming number with the mask 0x2000. If it's a one, the number is negative and we add the magic value 0xFFFFC000. That immense number will fill the leading bits of the incoming number with 1s to create a 32 bit negative.

If you should ever need to convert an 8 bit 2s complement, use 0x80 as the mask and 0xFFFFFF00 for the magic value.

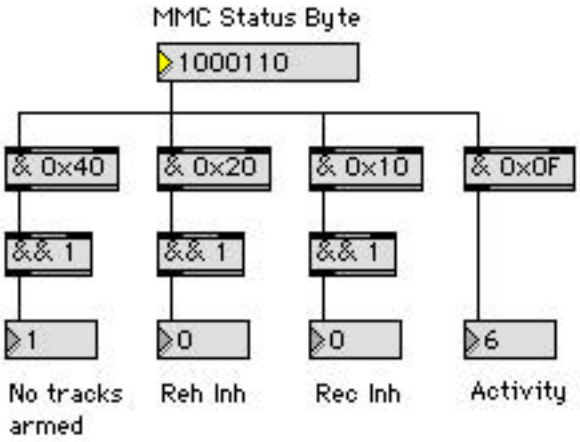
Packed Data

Once in a while, programmers try to cram a lot of little numbers in a single byte. For instance, in the Midi Machine Control specification, the record status byte contains a 4 bit number for current activity, and single bits to indicate record inhibit, rehearse inhibit, and No Tracks Active. (A single bit indicator is called a flag.) The data sheet reads like this:

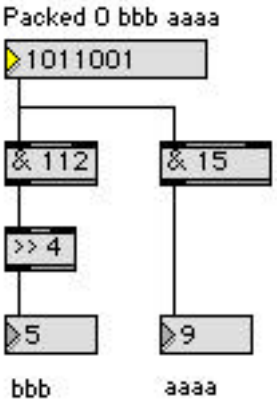
Status 0 d c b aaaa

aaaa = current Record/Rehearse Activity
(Six possibilities are listed)
b = Local Record Inhibit flag
c = Local Rehearse Inhibit flag
d + No Tracks Active (i.e. recording or rehearsing)

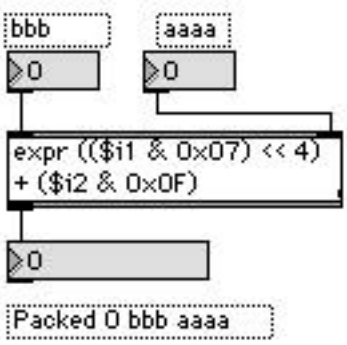
Here's a decoder:



These are familiar operations, masking and logical AND to extract the state of individual bits.
If the data in the upper bits were a single number, it could be found with a right shift:



The masking value for the upper bits (4-6) is found by adding the power of two each bit represents. In this case, 16+32+64 = 112. Building such numbers is easily done with expr:

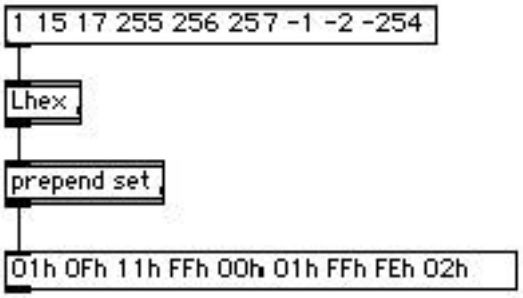


I mask the input values to keep them within range. If a user entered a 5 bit value for aaaa, it would also change bbb, which might be very hard to track down. The parentheses are required to make sure the steps are performed in the correct order.

Lobjects for Hex and Sysex

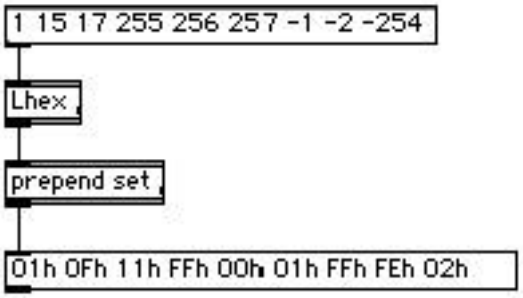
I do so much work with sysex that is documented in hex that I have written some Lobjects to let me use hex in lists and keep it on screen when I save patchers. (By now you have discovered that if you write 0x17 in an object box, it will say 23 when the patcher is reopened.) I have some objects that recognize hex numbers in the format 00h. (these are symbols to Max, so they are stored and retreived intact.)

Lhex



Lhex converts a number from 0 to 255 into the hex equivalent. If you give it a number bigger than 255, it is masked to 8 bits. If you give it negative numbers, the masking will create two's complements of the number. Note that since these are symbols, you need prepend set and a message box to see them.

Llong

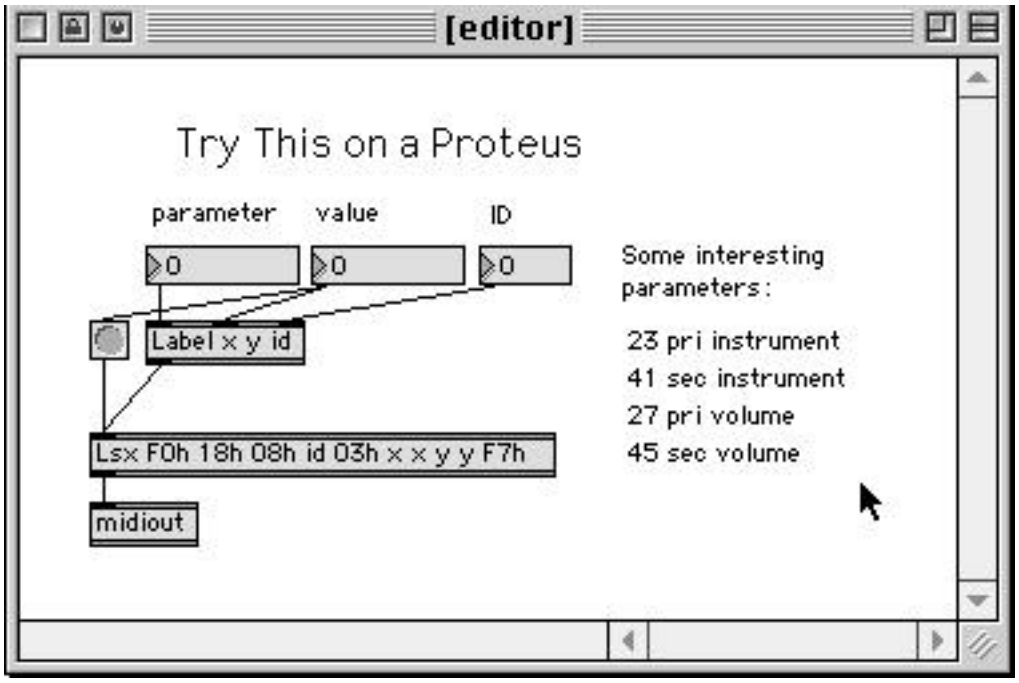


Llong turns the hex symbols back into numbers. If the orignal number was masked by Lhex, long can't know it and returns the modified value. Note that one digit symbols must be written 01h. Llong can deal with hex numbers of any sizeup to FFFFFFFFh (which is –1 in a 32bit int).

Lsx

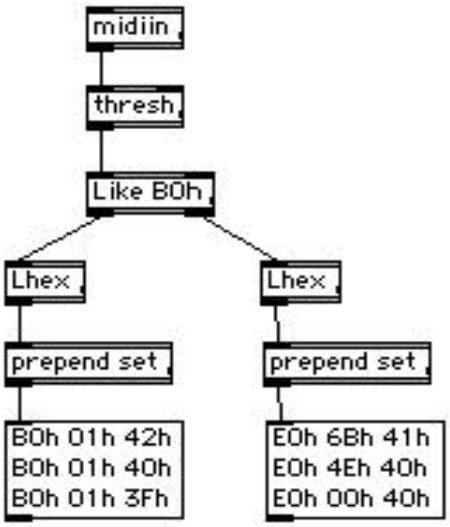
Lsx is my version of sxformat. You can copy hex values into the object as arguments, and when a bang is received, the values will be sent out (one at time). If the string is a valid

sysex message, midiout will send it on its way. Furthermore, other symbols in Lsx are wild cards. Their initial value is 0, but they are replaced by values input that are preceeded by the symbol. For instance, if the symbol "id" appears in Lsx and you give it id 4 as an input messge, the value 4 will appear instead of id. If the same symbol appears twice in Lsx, the value is nibblized across the two (up to four) locations. The default nibblizing is 7bit with low byte first, but all that can be changed- see the help file for details.



Here's a typical use for Lsx. Label is another Lobject that sticks labels on the front of messages.

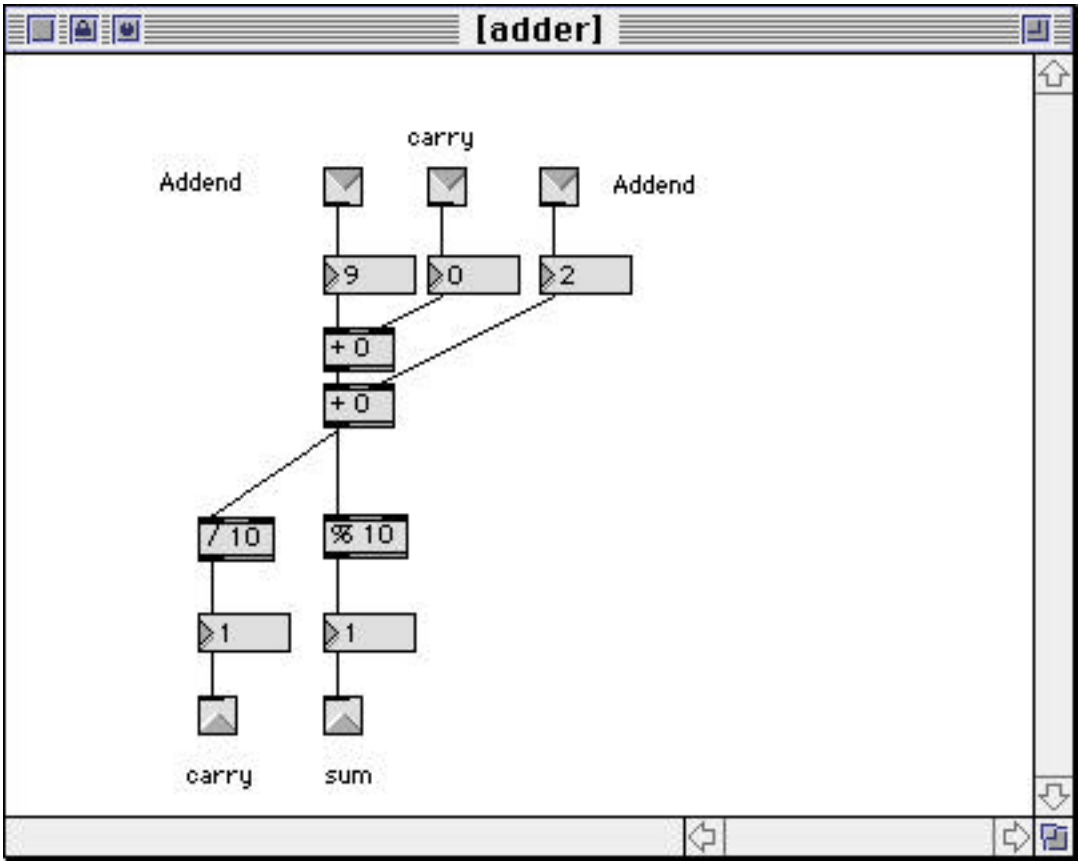
Like



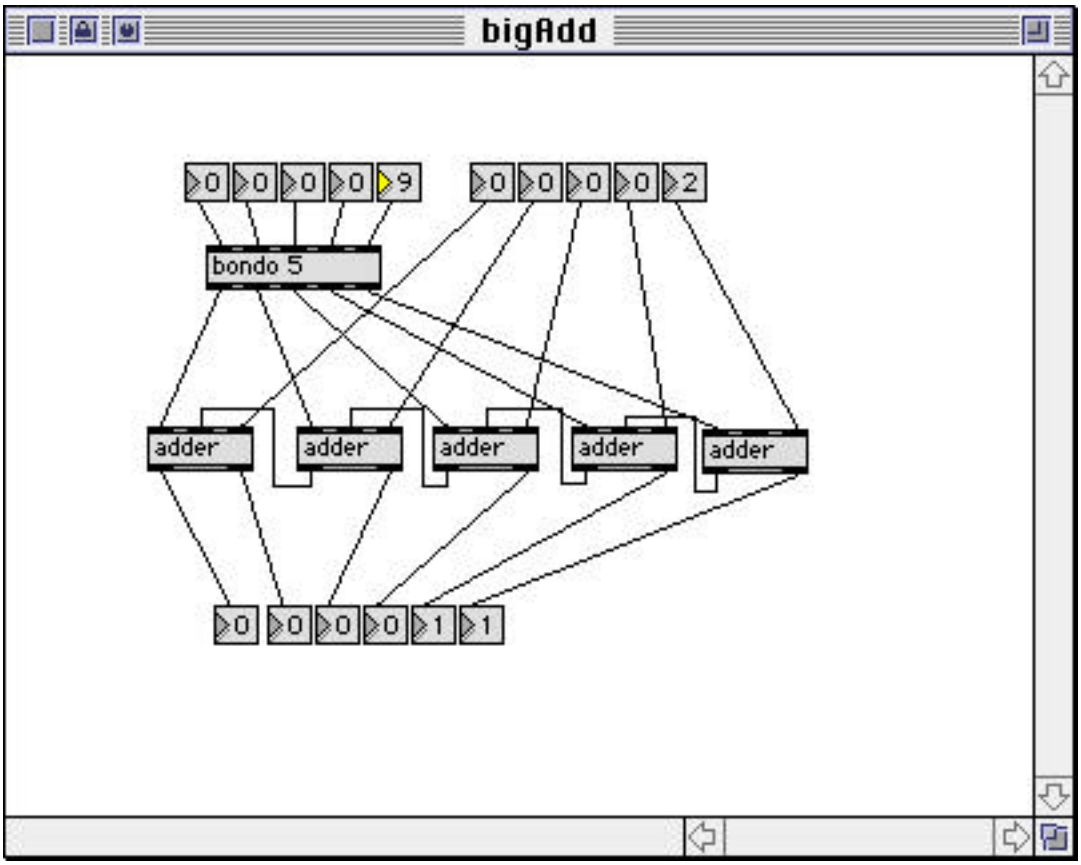
A problem when dealing with sysex is detecting particular messages when they come into the computer. Usually, a message is identified by a "header", the first few bytes of the data. Like is a object that tests the incoming list against its arguments, and passes the list out the left if they match. If no match, the list goes out the right. Here it's looking for control messages on channel 0, passing pitch bends on for another Like to deal with.

Really Big Numbers

As I mentioned earlier, Max uses 32 bit integers. This means the range of expressible numbers is -2,147,483,648 to 2,147,483,647. If you want to work with numbers bigger than this, you have to write patches that process the numbers digit by digit. For instance, for addition, you'd first make a subpatcher called an adder:

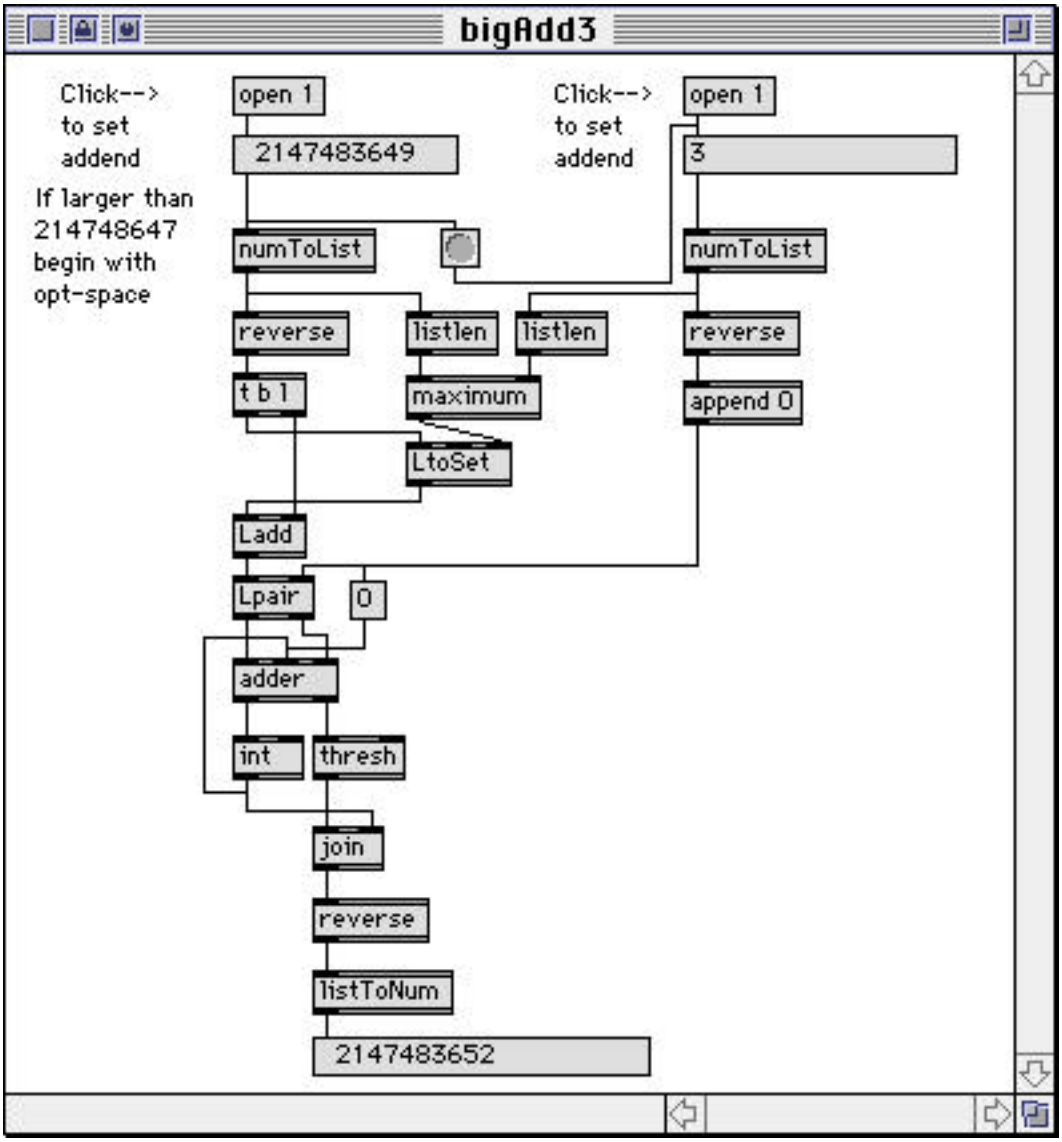


The number boxes for the addends are restricted to 0-9, the carry input is restricted to 0 or 1. This accepts two addends and a carry, and produces a one digit sum and a carry of 1 or 0. You'd put it in an addition patch like this:

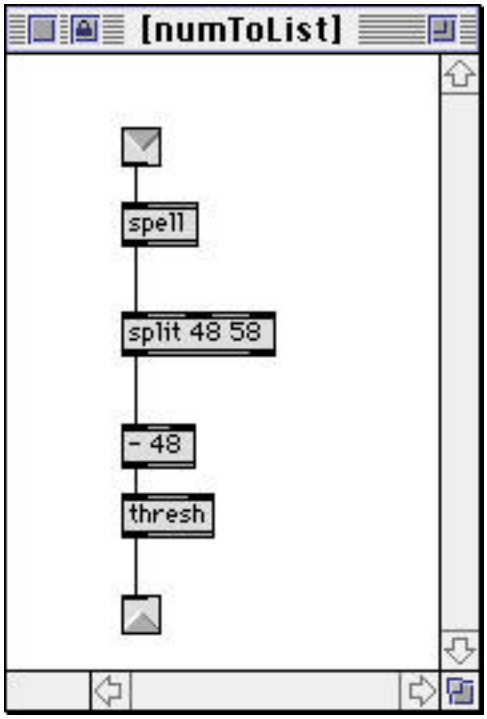


Again the digits are restricted. You can make the numbers as big as you want by adding adders. (You used to be able to get little mechanical calculators that worked this way.)

Of course, this would get wide for really big numbers, and entering each digit separately is tedious, so it may be rewarding to explore a second approach. In this version the numbers are converted into a list of digits, and the digits are processed serially:

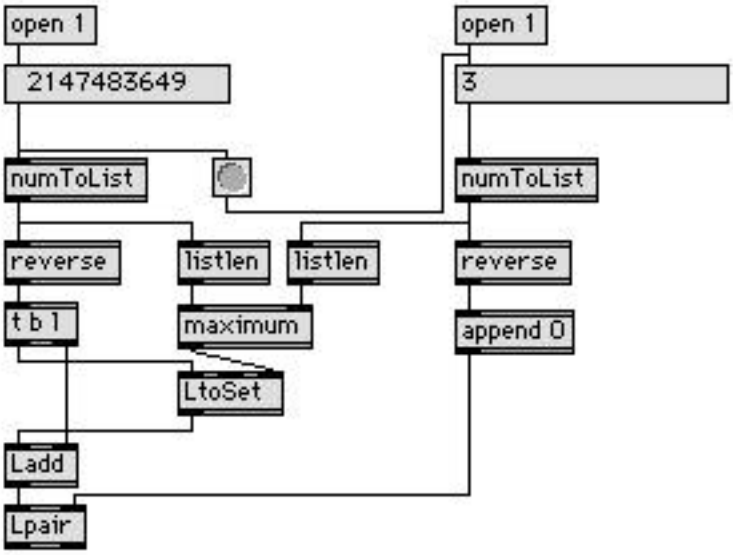


The numToList subpatcher converts whatever is entered into the message box into a list of digits. The message box will try to make a number out of any entry that is all digits and spaces, so for large numbers you must include some other character. The non breaking space will display nicely. NumToList strips these out:



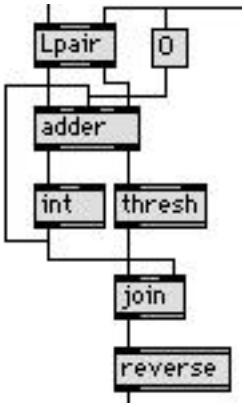
It's pretty simple, taking advantage of the fact that the ACSII values for the digits start with 0 = 48.

Next comes a section that prepares the two lists for application to the adder:



The adder will need to receive the lists least significant digit first, so the lists are reversed.

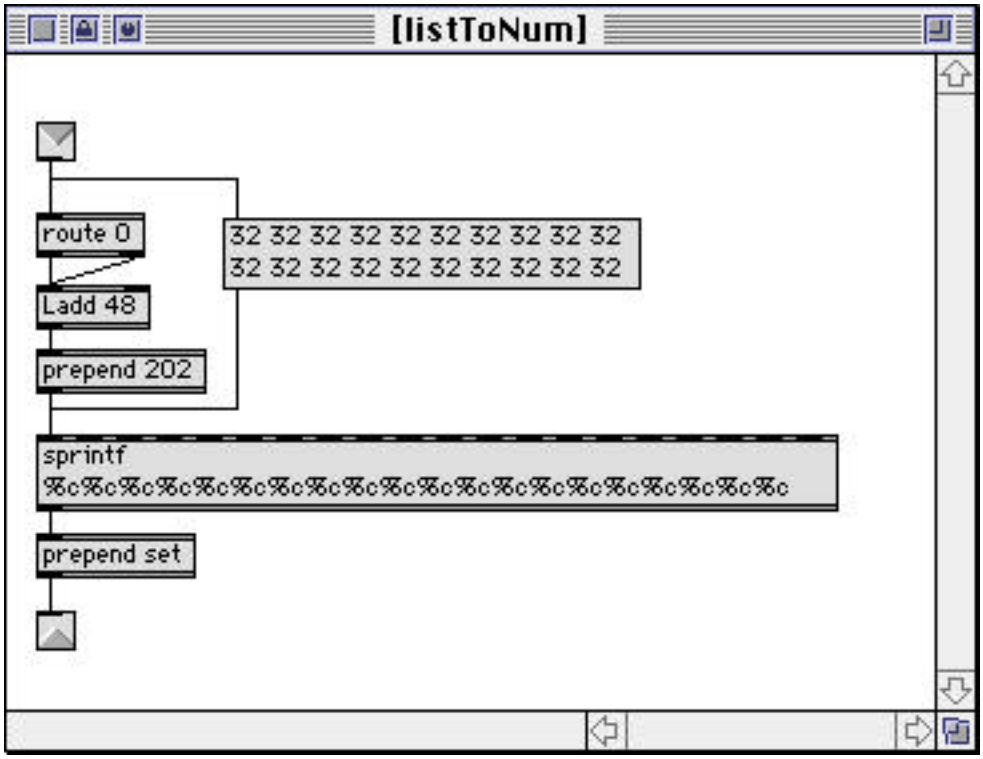
Lpair[1] accepts two lists and iters them as synchronized pairs of int. The left list determines the number of outputs, so it must be as long as or longer than the right list. To do this we find the longer of the two lengths, produce a list of 0s (LtoSet creates a list of the desired length,) and add the reversed left list in. If the right input list is short, the final value is repeated as needed, so we make that a 0.



The adder is the same one used above. It will work sequentially if we just feed the carry back in. Of course we have to clear the carry before the addition starts, hence the 0 in a box. The carry also becomes the first digit of the number, even if it is zero. Thresh collects the sum into another list, which is still backwards. Join tacks the carry on and the list is reversed.

ListToNum is another simple subpatcher:

[1]I'm embarassed to report that I discoverd a bug in Lpair while writing this tutorial. To make this patch work, you need the fixed version, which reports a copyright date of 1997. It's available from http://arts.ucsc.edu/pub/ems/Lupdates/

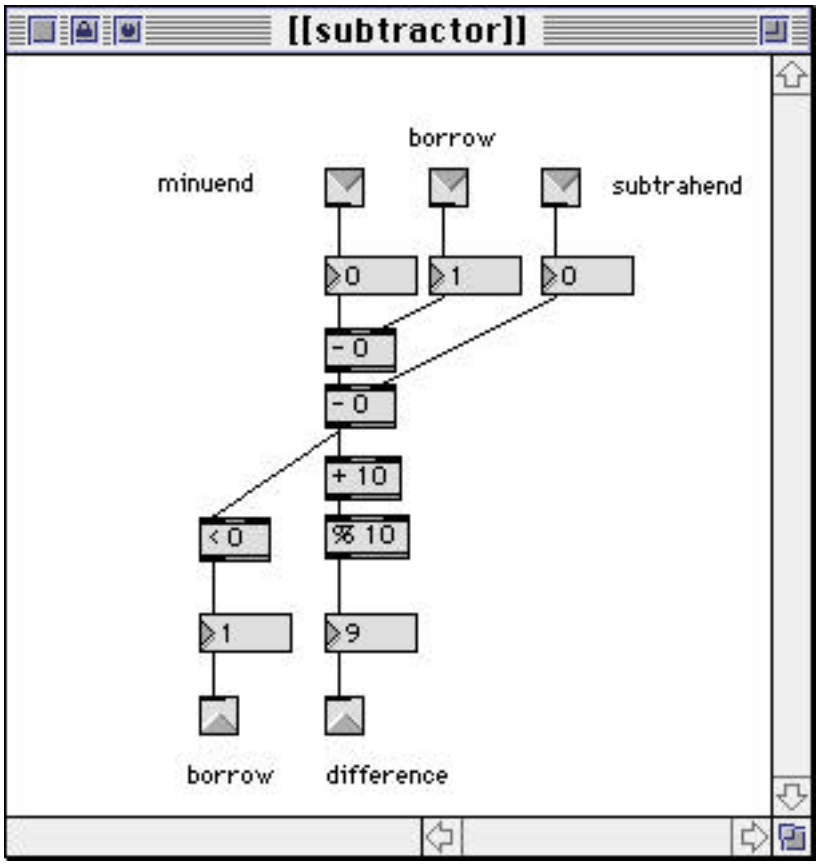


Route 0 is used to strip off the leading 0, if any. The prepend 202 starts the list with a non-breaking space. The list of 32's clears sprintf, in case the number coming in is shorter than the last one.

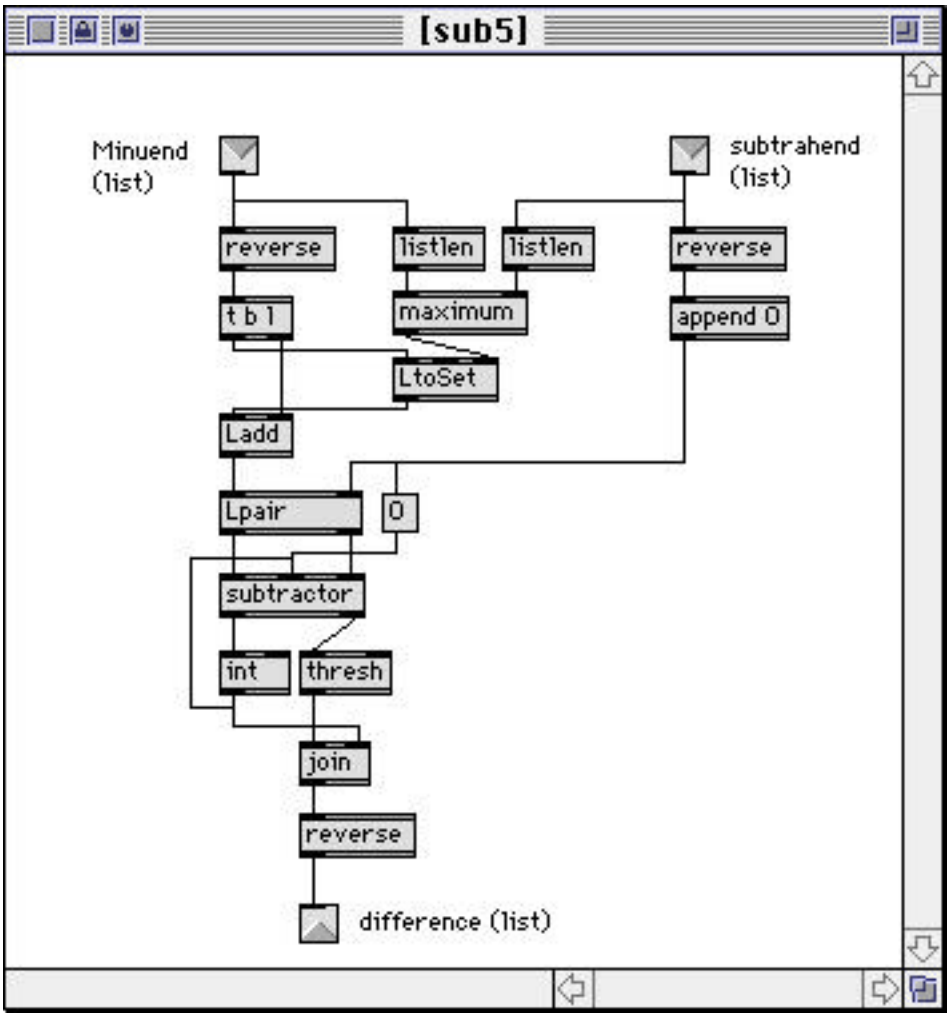
The number of %c's in the sprintf limits the size of the number that can be displayed; 20 digits ought to be enough, but if the input number exceeds this, strange things will happen. The 21st will produce 2,417,483,647, and any more will simply output 0.

Big Difference

For subtraction, we start with a subtractor subpatcher:

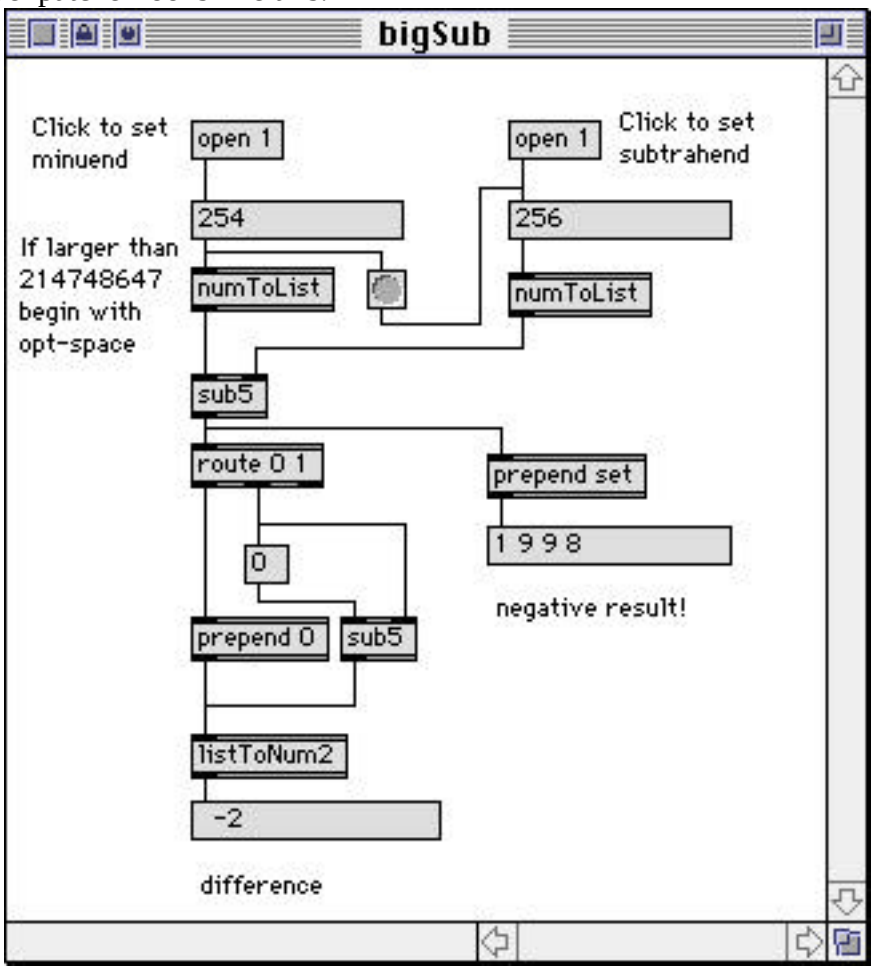


It's a lot like an adder, but it subtracts and borrows. This will fit into a subtraction patch:



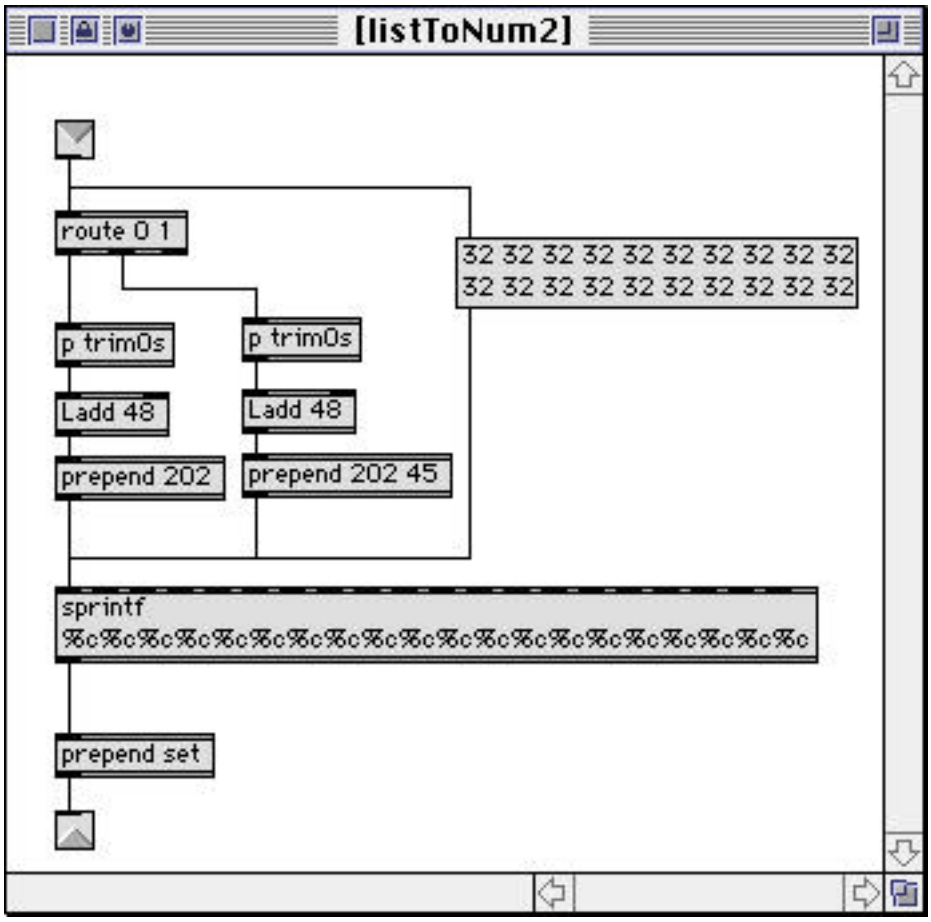
Actually, it's another subpatch. You'll see why in a moment. It is taken directly from the middle of the addition patch above, with a subtractor replacing the adder. It expects input in the form of digit lists as before.

The top level patcher looks like this:

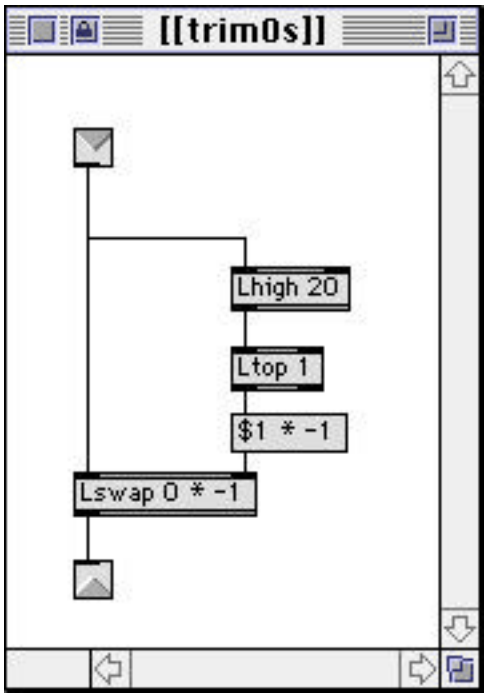


I'll bet there's a lot more at the bottom than you expected. The sub5 subpatcher works fine as long as the result is positive. When the minuend is smaller than the subtrahend, the result is strange, as shown on the extra message box to the right. This is the tens complement of the number. Each digit is subtracted from 10 and the leading 0s are converted to 9s. The final borrow is a 1, indicating a negative number. Luckily, it's easy to sort out. Just shed the leading 1 and subtract from 0. So there's another sub5 in there.

ListToNum needs some tweaking to provide a minus sign and remove extra 0s. There's still a 1 in front of negatives, which now looks something like 1 0 0 2. (We still need the leading 0 for positives, so we have to put it back.)



Here the route treats negative numbers differently from positive ones. Prepend 45 gives a minus sign. The trim0s subpatcher does a trick with Lobjects:



Lhigh changes all digits that aren't 0 to 1. Ltop reports the position of the first 1, and this sets the template for Lswap which allows the digits from there on through.

Beyond

You could make patchers to do more complex math using similar techniques, but they would be big and slow. It would probably be a better use of your time to learn to write external objects and create one that uses the PPC's 64 bit number crunching ability.