Max and Numbers

Basic operators

Number Types
All max operations require that you decide ahead of time if the rules of integer or floating point math will apply. In most cases you put an object into float mode by entering a float as the argument. Most of the time, we send simple number messages to the operators, but you can use a list of two values. When you do that, the second member of the list replaces the argument as the operand. There are Lobjects to perform these operations on all members of a list.

Addition
The [+] function adds whatever has been received in the right to whatever comes in the left. In the following notes I'm going to call the right input (or written argument) the stored value, and the left input the input value.

Subtraction
The [-] function subtracts the stored value from the input. When you write an argument, be certain there is a space between the minus and the number. If you leave the space out, you won't get an error, you will get an integer object (exactly like[int -1]) and input will be passed through unchanged.

Complement
The [!-] function subtracts the input from the stored value. This is often called taking the "2s compliment" (if subtracting from 2). The 12s compliment of a music interval gives the interval's inversion. !- is not a standard notation, only Max refers to it this way. The Lcomp object does the same on lists.

Multiplication
The [*] function multiplies the input by the stored value.

Division
The [/] function divides the input by the stored value. Dividing integers gives truncated results, with any fractional part discarded. If you divide by 0, an error message appears in the Max window. It's always faster to multiply than divide, so [* 0.5] is preferred to [/ 2], at least when the computer is working hard.

Inversion
The [!/] function divides the stored value by the input. Linvert does this with lists.

Modulo
The [%] function divides the input by the stored value and returns the remainder. This works only on integers. Lrem does this on lists.

Integer and Float 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 4.823×10^4.

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_m.bbbbbbbbbbbbbbbbbbbbbb
where each b is either 1 or 0.

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

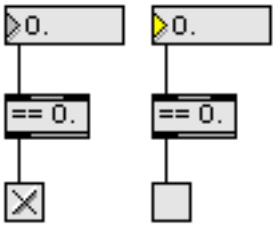
There is further massaging to save bits or processing time where possible. For instance, the exponent can be positive or negative, but instead of having a sign bit in the exponent, 127 is subtracted from the exponent when the number is converted. With 8 bits, the exponent may range from –127 to 128. In addition b_m is unnecessary 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 difference 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.

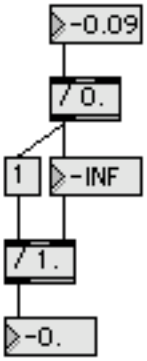
Here's another one. Tack a == 0.0 on that float box and watch the output with a toggle. If you type 0 into the box, the toggle shows that 0 in fact equals 0.0. However, if you scrub the box away from zero and back, you won't see an equality again. The number is probably 0.00000000002 or something.



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!

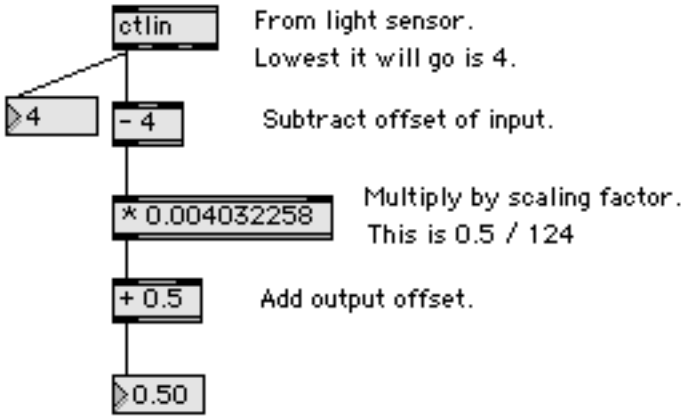


Another peculiar result you may see is "NaN". This stands for "not a number", and happens if you take the square root of –1, divide 0 by 0 or do anything else that is not defined.

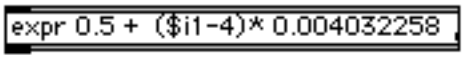
Range conversions

With the addition of MSP and even more so with Jitter, Max patchers often have to bridge the gulf between the MIDI world of 0-127 and floating point ranges of 0.0-1.0. To scale MIDI controls to the range of 0-1.0, multiply by 1/128 or 0.0078125. Why not just divide by 128? Multiplication is usually faster for the computer. The number you multiply by to get the equivalent of a division is called a scaling factor.[1]

In general, the scaling factor is found by dividing the desired range by the input range. This is made complicated when the input or output does not start at 0, a situation we call an offset. Then you must subtract the input offset, multiply by the scaling factor, then add the output offset.[2] Here's what it looks like to scale an input with the range 4 - 127 to an output of 0.5 to 1.0:



Of course the whole thing can be put in one expr object:



Exponential Scaling

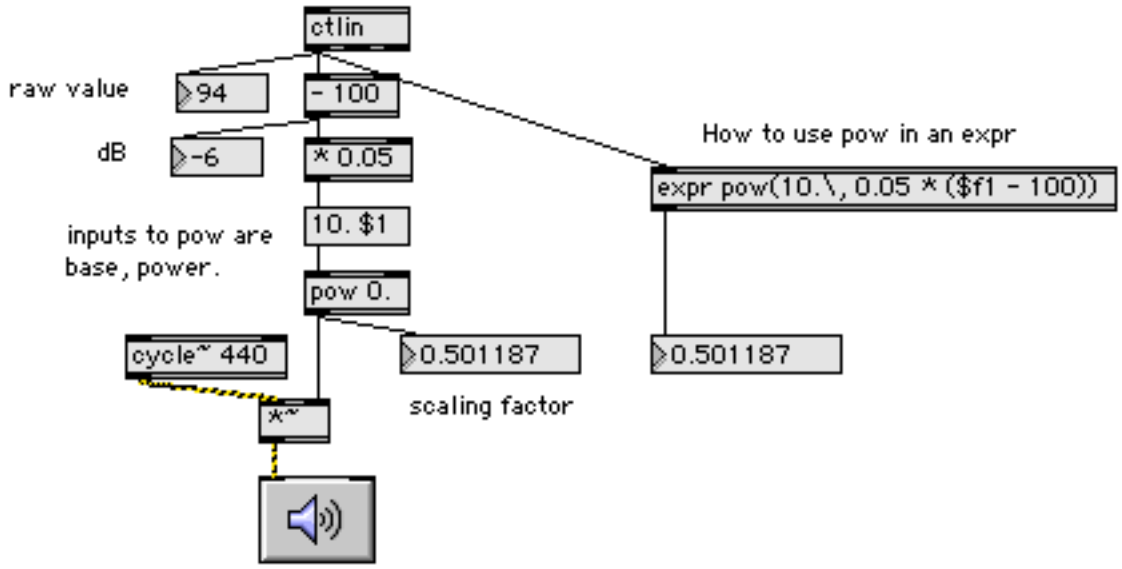
In many cases, such as controlling volume of an audio signal, the linear scaling described above doesn't work. Audio volume needs to be controlled by dB. You should remember the formula for decibels:

20 * log (signal/reference)

We need to reverse this to go from a slider that produces dB to an audio scaling factor. This is simplified a bit by the fact that the reference for MSP audio is 1.0. Thus the scaling factor is simply the value we want out with a full scale input. In digital systems, 0 db is maximum volume. Let's take a slider and call 100 the 0 dB point (so we can amplify weak inputs). We subtract 100 and divide by 20 (or multiply by0.05). Then we raise 10 to that power to get the scaling factor.

[1] Actually, Max division objects do multiply- they calculate scaling factors in the code.

[2] Unless the output offset equals the input offset * the scaling factor, of course.



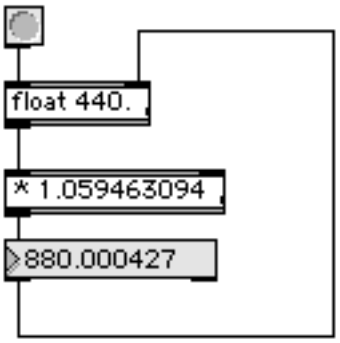
Frequency ratios

A similar process is buried in pitch to frequency conversion.

The frequency ratio between notes a semitone apart is the 12th root of 2, or 1.0594630943593 . If we call that R, the formula for the frequency of any note is

$$R^n * 440.0$$

Where n is the interval (positive or negative) between the desired pitch and A4.
This patch shows how to put this in an expr. Of course you can just as easily use mtof. The expr and mtof don't quite agree. Errors down in the bowels of the decimal place are common in floating point operations, but have no audible effect. Here's what happens if you just try to go up an octave by multiplying:



I just hit the button 12 times!

Zero based indexing

Why do most computer programmers prefer to count starting with 0? Because usually, we are not counting things, we are generating a series of numbers. For instance, to simulate the turning of a MIDI control knob we generate numbers from 0 to 127. You will note that 0-127 is a list of 128 numbers. To keep that clear in our minds, we usually count up to a target rather than up through a target. The C code often looks like this:

for(i = 0; i < 128; ++i){};
The actual operations involved in generating number series are less complex when starting at zero. To do what uzi does the actual code is visibly more complicated

For(i= 1; i <= 128; ++i){};

The "i<128" represents a test that must be done each time around to see if the process is finished. Admittedly "i<=128" is only slightly more complicated (two tests), but little things like this combine to slow code down.

The topic is confused by items that are represented in code as 0-127 but numbered in text and panel displays as 1-128. For instance MIDI channel and program numbers. The MIDI Manufacturers Association felt that having channel 0 or program 0 was just too geeky.

Bits and Bytes

The characters in a telephone number are called digits, a word derived from counting on the fingers. They are also decimal digits, with decimal meaning ten from the Greek word Decca. Each digit can have one of ten values in a decimal system. Likewise, in an octal counting system there are eight possibilities and in hexadecimal, sixteen. If the options are merely on or off, you have a binary digit, which can be 0 or 1. Binary digit was contracted to bit by a Bell labs engineer named John Tukey in 1947[3].

[3] At least according to Wikipedia -- there may be earlier uses. In any case the entry on bit is very enlightening.

The contribution of a bit to the value of the number is dependent on its position, with the right most being the least significant, adding either 0 or 1 to the total. As bits are added to the left, each increases by a power of two thus:

8s	4s	2s	1	total
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5

A byte is a data word of 8 bits. The data word of a computer is based on the size of the registers used to hold numbers for math operations or the number of wires in the connections to the registers. Early computers had bytes of 6 to 12 bits, but when microprocessors started to take over with the introduction of the Intel 8008, 8 bits became the standard byte. Later systems boast multi-byte data busses, but 8 is still a popular data size for dealing with smallish numbers.

The bits in a byte (or longer data word) are numbered. The least significant bit (lsb) of a number is bit 0. Thus the msb of a byte is bit 7.

If all of the bits in a byte are 1, the value is 255.

A kilobyte is not 1000 bytes as you might think. It is 1024 (2^{10}) bytes. A megabyte is 1024 Kbytes and a gigabyte is 1024 Mbytes.[4]

MIDI bytes

If a byte can represent numbers up to 255, why does the byte based midi protocol restrict data to 0-127? The most significant bit of a MIDI byte is used to indicate whether the byte represents data or a midi command. So a data byte can only have 7 bits. In some circumstances, two data bytes are combined to give a single value. Then the range is 0 to (2^{14} –1) or 16383. The midpoint of that range is 8192, a number you will see a lot in MIDI documentation.

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 bit nibblized values, 2s complement and constant offset signed numbers, packed bytes, and other wonders.

[4] So a MB is 1048576 bytes and a GB is 1073741824.

Hexadeciamal 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:

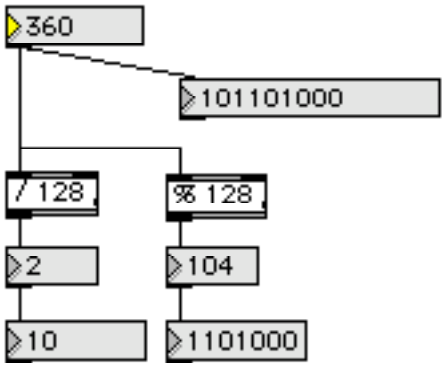
Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

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 largest value that can be directly represented in MIDI 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:

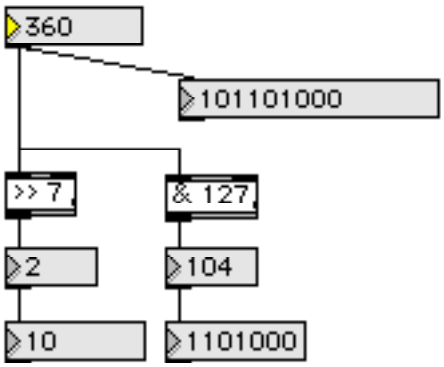


The number 357 requires 9 bits in binary representation. The / 128 operation discards the rightmost 7 bits and the % 128 turns all but those 7 bits to 0. How do these work? When you divide using integer division, you discard the remainder. The rem or modulus operator (%) gives you the remainder only 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:

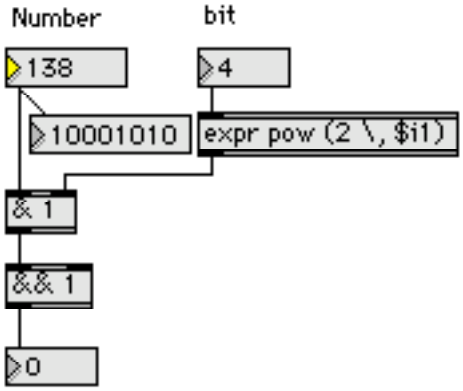
Bitmasks

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.

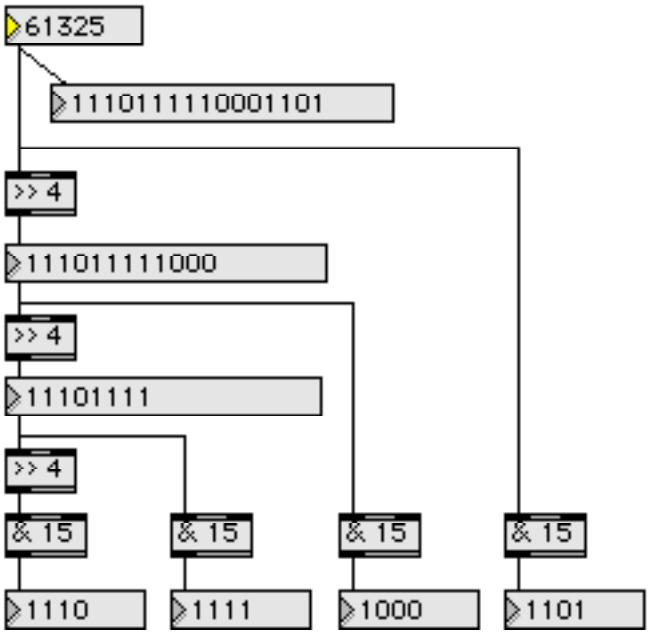
You can find assorted bits by adding their masks together. For instance, to isolate bits 1-3 add 2 + 4 + 8, getting 14 or 0x0E.

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:



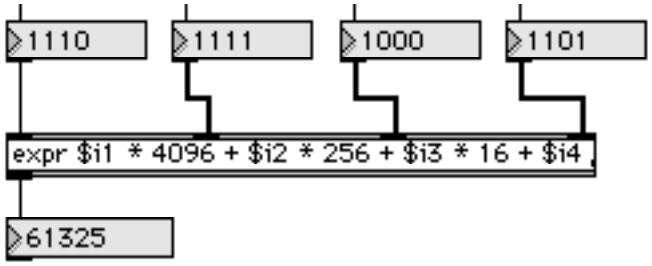
In this example we find that bit 4 of 138 (10001010 in binary) is 0.

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:

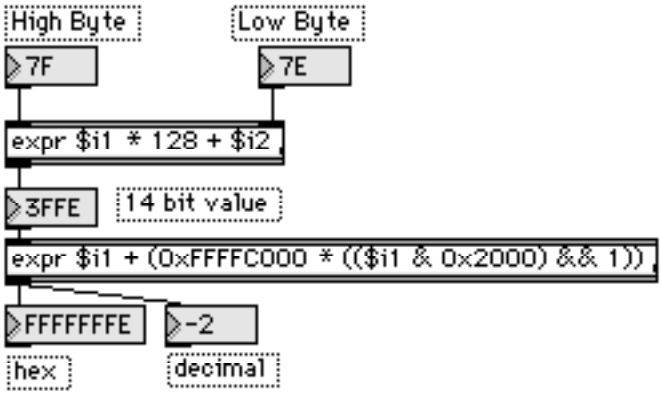
There are two limitations in the circuitry computers use to do math. First, it can add, but not subtract, and second, only a certain number of bits can be handled at a time. Luckily, we can use the second shortcoming to overcome the first. To understand this, imagine an adder that can only handle 4 bits. 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 11111111111111 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.

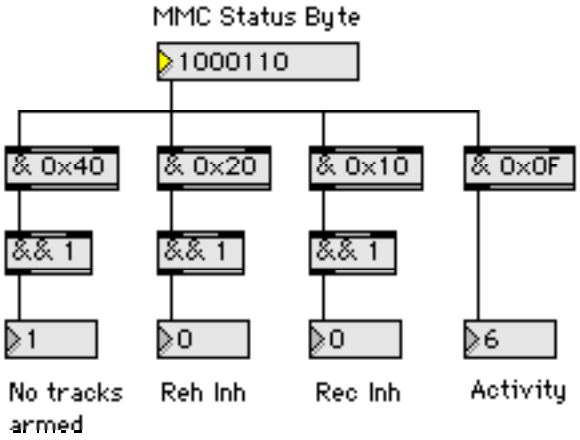
Packed Data

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:

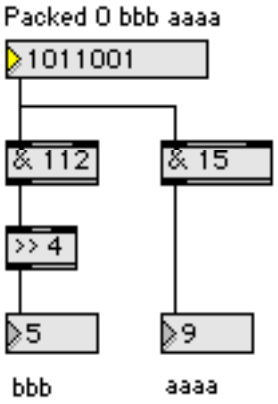
Status 0 d c b aaaa

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)

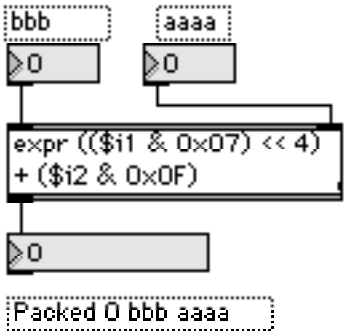
Here's a decoder:



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:

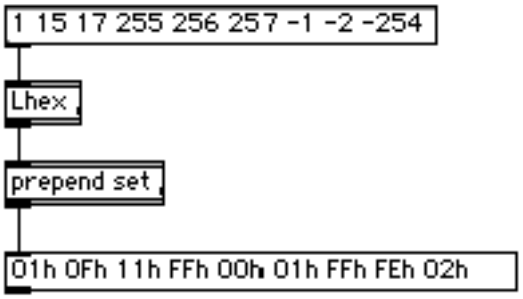


I mask the input values to keep them within range. If a user entered a 5 bit value for aaaa, it would also change bbb, which might be very hard to track down. The parentheses are required to make sure the steps are performed in the correct order.

Lobjects for Hex and Sysex

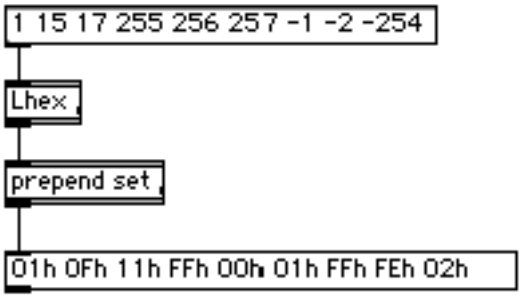
I do so much work with sysex that is documented in hex that I have written some Lobjects to let me use hex in lists and keep it on screen when I save patchers. (By now you have discovered that if you write 0x17 in an object box, it will say 23 when the patcher is reopened.) I have some objects that recognize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

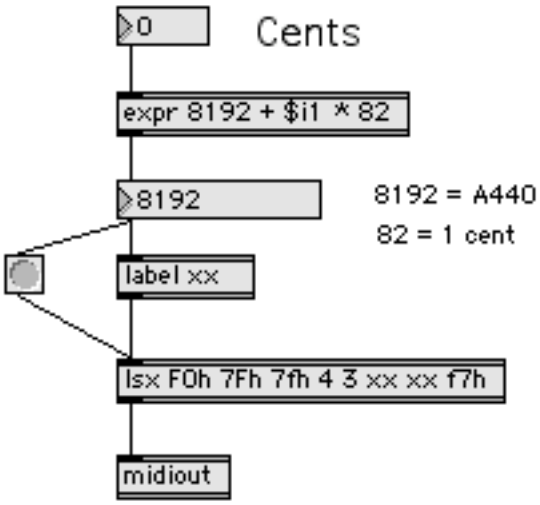
Llong



Llong turns the hex symbols back into numbers. If the original 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 size up to FFFFFFFFh (which is –1 in a 32 bit int).

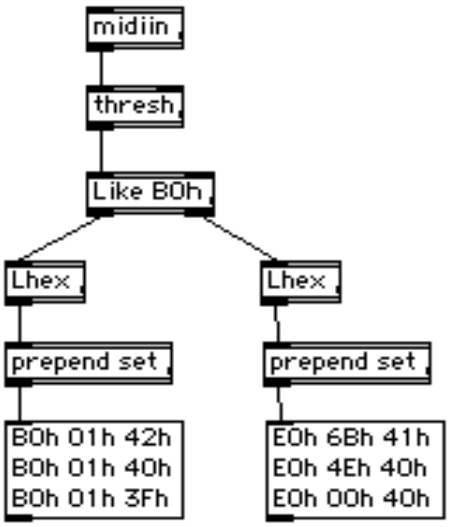
Lsx

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 preceded by the symbol. For instance, if the symbol "id" appears in Lsx and you give it id 4 as an input message, 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 7 bit 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.