On the MUSIC LANGUAGE

OK, here's my proposal for low-to-intermediate level music language. First off, I propose that all but a few lowest-level routines be coded in SAIL.

LOW LEVEL

First off, all "things" in the box have a number. This number is divided into two fields. The low order 8 bits specify the unit number and the high order 2 bits specify the unit type. For instance, units 0-255 would be generators, units 256-383 would be the modifiers (units 384-511 would be unused), units 512-767 would be sum memory locations, and units 768 to 799 would be delay memory ports. I propose there be a boolean array called UTAB (unit table?) that has UTAB[THING_NUMBER] as FALSE if the thing is free and TRUE if the thing is in use. Having this as an array allows the user to just claim these things any way he wants to. If he so desires, he can go through the system claiming routines as follows:
(reserved words are in caps, generics are in lower case)

```
        id←GET(GENERATOR);
        id←GET(MODIFIER);
        id←GET(SUM_MEMORY);
        id←GET(DELAY);
```

This allocation of sum memory has a bug in it, because you must specify which quadrant of sum memory you are talking about. There is the generator side or the modifier side, and there is this tick and last tick. So perhaps this:

```
        id←GET(GEN_SUM_MEMORY);
        id←GET(THIS_MOD_SUM_MEMORY);
        id←GET(LAST_MOD_SUM_MEMORY);
```

Anything else, like getting consecutive locations of sum memory, or specially spaced locations, you have to do yourself, like with this SAIL routine:

```
integer procedure GET_N_GENERATORS(integer N);
begin
        integer i,j;
        define ⊃="comment";
        boolean gotit;

        ⊃ This routine gets N consecutively numbered generators.
        It assumes a global integer called N_PROCESS_TIX which
        tells how many generators are available this run. The
        routine returns the number of the first generator in
        the sequence. If there are not N generators in a row,
        it returns -1;

        gotit←false;
        for i←0 step 1 until N_PROCESS_TIX-N do
        if ¬UTAB[i] then
        begin "CHKIT"
            for j←i+1 step 1 until i+N-1 do
                if UTAB[j] then
                begin "NOGOOD"
                    i←j;
                    continue "CHKIT";
                end "NOGOOD";
            gotit←true;
            done "CHKIT";
        end "CHKIT";
        if ¬gotit then return(-1);
        for j←i step 1 until i+N-1 do
            UTAB[j]←true;
        return(i);
end;
```

The idea is that this gives the user complete control on how he wants to allocate things. If he is happy with the way the system allocates things, he can just use the standard routines. There, of course, are the inverse routines for deallocation:

```
        GIVE(id);
```

It can figure out what kind of thing this is by its number, so this routine can release any kind of thing. All it does is do UTAB[id]←FALSE, so there isn't really any need to have a separate GIVE routine.

We then have the following routines for setting parameters:

```
        SET(id,parameter_name,value);
```

where parameter_name is like MODE, ANGLE, FREQUENCY, FM_ADDRESS, DECAY_RATE (for generators) or like A_ADDRESS, B_ADDRESS, WRITE_ADDRESS, M0, M1 (for modifiers). These are codes that specify what parameter in the named thing to change. Naturally, asking to change the frequency of a modifier will get you an error message.

We can also imagine several other "pseudo-things" that can be set via the SET command, like the time (might have to emit LINGER commands) or the number of update ticks, or whatever. Thus the SET command can be used in general to set any old thing and issue the proper command string.

By adding enough reserved words like parameter names, thing types, and pseudo-thing types, we can in fact change anything there is. There are three other things to be dealt with now:

Where do the commands go?
How does sample data get in and out of the box?
How do we "play" a pre-computed command list?

For where the commands go, there are three options which are not mutually exclusive: They go directly to the box, they go to a file, or they go to a block of core. The following routine should be enough to specify that:

```
        core_pointer←DIRECT(COMMANDS,channel+TO_CORE+TO_DISK+TO_BOX);
```

The way you specify which of the options you want is by inclusion of the appropriate term. If you say TO_DISK, you must include a channel number of a file which has been opened for writing in mode '17. If you say TO_CORE, you get back a record pointer (see SAIL record stuff) of the first record of a linked list of records, each one of which has a 2.2K buffer of commands in it (2.2K is chosen somewhat

The signal path for one analog output involves the following sections:
Channel selection logic (addressing)
Digital hold register
Digital to analog converter
Sample-and-hold
Program-controlled filter
Buffer amplifier.

Each section is specified at 25 degrees C as follows.

Channel selection logic: 4 bits (1 of 16)

Digital hold register: 14 bits

Digital to analog converter: 14 bits
Linearity: 0.005%

Sample-and-hold: full power bandwidth 40 kHz

Filter: two modes
Mode 0: 1-pole RC at 200 kHz
Mode 1: 6-pole Butterworth, 4 programmable frequencies subject to the relationships f0=A, f1=A+B, f2=A+C, f3=A+B+C; full power bandwidth 18.5 kHz max.

Buffer amplifier: output +/- 10 V max., unbalanced
Output current: 4 mA max.
Short circuit protection: to ground only
Full power bandwidth: 13 kHz for 20 V swing; greater in proportion for lower voltages
Output source impedance: 100 ohms
Output connector: BNC jack

The following are overall figures with Mode 0 filtering:

Gain error: 2.5%

Offset error: 20 mV

Noise at sampling rate and its harmonics: 1 mV max. (RMS)

Other noise 10 Hz to 50 kHz: 1 mV max. (RMS)