INTERMEDIATE LEVEL

I propose that we do the entire intermediate level using the SAIL process and co-routine primitives. The advantage is versatility, clairity, and modifiability; the disadvantage is efficiency.

We need to make two procedures for scheduling:

```
        WAIT_UNTIL(time);
        DO_IT_ALL;
```

For instance, the "main program" for a six voice piece might look like this:

```
<declare procedures VOICE1, VOICE2, . . . VOICE6 and items IV1, IV2,
. . . IV6, etc etc etc>
        . . .
<do things here like initialize the box, set the number of process
ticks and update ticks (sampling rate), where the commands are to go,
and all that stuff>
        . . .
        CH1_GEN←GET(GENERATOR);            ⊃ Get an output generator;
        OUTA←GET(GEN_SUM_MEMORY);          ⊃ And a place for all to go;
        SET(CH1_GEN,MODE,'200);            ⊃ Set mode to feed DAC;
        SET(CH1_GEN,FM_ADDRESS,OUTA);      ⊃ Get its input from OUTA;
        SET(CH1_GEN,NUMBER,1);             ⊃ Send it to DAC channel 1;
        SPROUT(iv1,voice1,SUSPME);
        SPROUT(iv2,voice2,SUSPME);
        SPROUT(iv3,voice3,SUSPME);
        SPROUT(iv4,voice4,SUSPME);
        SPROUT(iv5,voice5,SUSPME);
        SPROUT(iv6,voice6,SUSPME);
        DO_IT_ALL;
END "WORLD";
```

A typical voice might look like this:

```
PROCEDURE VOICE1;
BEGIN
        TIME←0;
        BLAT(TIME,T←QUARTER*TEMPO,A,2047);
        TIME←TIME+T;
        BLAT(TIME,T←HALF*TEMPO,AS,2047);
        TIME←TIME+T;
        BLAT(TIME,T←QUARTER*TEMPO,G,2047);
        TIME←TIME+T;
        BLAT(TIME,T←QUARTER*TEMPO,FS,2047);
END;
```

This was a verbose definition and you will surely want to streamline this a bit. It was written out like this so it would be clear what is going on. We should go further and see what the "instrument" BLAT does. A typical definition might be this:

```
PROCEDURE BLAT(REAL TIME,DURATION,PITCH,AMPLITUDE);
BEGIN
        INTEGER MOD_OSC,CARRIER_OSC,CONNECT_LOC;
        ITEMVAR AMP_PI,INDEX_PI;
        WAIT_UNTIL(TIME);        ⊃ Make sure we are at start time;
        MOD_OSC←GET(GENERATOR); ⊃ Do a simple FM instrument;
```

```
          CARRIER_OSC←GET(GENERATOR);
          CONNECT_LOC←GET(GEN_SUM_MEMORY);
          SET(MOD_OSC,DECAY_EXPONENT,0);  ⊃ Set amplitude to zero first;
          SET(CARRIER_OSC,DECAY_EXPONENT,0); ⊃ Before turning on;
          SET(MOD_OSC,DECAY_RATE,0);       ⊃ Ditto;
          SET(CARRIER_OSC,DECAY_RATE,0);
          SET(MOD_OSC,FREQUENCY_SWEEP,0); ⊃ Ditto;
          SET(CARRIER_OSC,FREQUENCY_SWEEP,0);
          SET(CARRIER_OSC,FM_ADDRESS,CONNECT_LOC);
          SET(MOD_OSC,FM_ADDRESS,ZERO);    ⊃ Hmm. Sum memory ZERO must be a
                                             "null", or unused location;
          SET(CARRIER_OSC,WRITE_ADDRESS,OUTA);
          SET(MOD_OSC,WRITE_ADDRESS,CONNECT_LOC);
          SET(MOD_OSC,FREQUENCY,MAG*PITCH);
          SET(CARRIER_OSC,FREQUENCY,MAG*PITCH);    ⊃ We'll never get rid of MAG!;
          SET(MOD_OSC,MODE,'1704);         ⊃ Standard SIN(k+Fm);
          SET(CARRIER_OSC,MODE,'1704);     ⊃ Also standard SIN(k+Fm);
                    ⊃ I think that's all that has to be done here,
                        though I wouldn't swear on it!;

               ⊃ Here, we are going to feed an amplitude function and an index
               function into the oscillators. We will assume that there is a MRG
               file open on channel CHAN with functions called AMP and INDEX that
               are the amplitude and index functions.;

          AMP_PI←NEW;       ⊃ We need an item for each process we sprout;
          INDEX_PI←NEW;
          SPROUT(INDEX_PI,
            STUFF(MOD_OSC,DECAY_RATE,PITCH*MAG*8,CHAN,"INDEX",DURATION),
              SUSPME);
                    ⊃ This is all hypothetical, meaning stuff the
                    function called "INDEX" into the DECAY_RATE
                    parameter of generator MOD_OSC.  I think there
                    will have to be an additional scale factor in
                    the index, but I haven't thought it out;
          SPROUT(AMP_PI,
            STUFF(CARRIER_OSC,DECAY_RATE,AMPLITUDE,CHAN,"AMP",DURATION),
              SUSPME);
          WAIT_UNTIL(TIME+DURATION);       ⊃ Go until note is done;
          TERMINATE(AMP_PI);               ⊃ Should already be terminated;
          TERMINATE(INDEX_PI);
                    ⊃ I think these last three statements can be replaced
                    by a JOIN({AMP_PI,INDEX_PI});
          DELETE(AMP_PI);                  ⊃ Return the process items;
          DELETE(INDEX_PI);
          SET(MOD_OSC,MODE,0);             ⊃ Stop the presses;
          SET(CARRIER_OSC,MODE,0);
          UTAB[MOD_OSC]←FALSE;
          UTAB[CARRIER_OSC]←FALSE;         ⊃ Then release all the stuff;
          UTAB[CONNECT_LOC]←FALSE;
END;
```

Now we need to look at what STUFF does, because this is where a lot of the magic occurs. To do this, I am going to have to use some MRG file reading routines.

```
PROCEDURE STUFF(INTEGER UNIT,PARAM; REAL SCALE; INTEGER CHANNEL;
          STRING FUNCTION; REAL DURATION);
BEGIN
          DEFINE NWB="128";        ⊃ We will buffer 128 words at a time;
```

```
        REAL ARRAY BUF[1:NWB+1];
        INTEGER CLOCK,COMPRS,WD1,WD2,NWDS,POS,WD;
        DIRGET(CHANNEL,FUNCTION,CLOCK,COMPRS,WD1,WD2,NWDS);
            ⊃ We ignore all the data but the number of words;
        FOR POS←0 STEP NWB UNTIL NWDS-2 DO
        BEGIN "PLP"
            PREAD(CHANNEL,FUNCTION,POS,BUF[1],NWB+1);
                ⊃ Reads NWB+1 words from that function;
            FOR WD←1 STEP 1 UNTIL (NWB MIN (NWDS-POS)) DO
            BEGIN "WLP"
                SET(UNIT,PARAM,(BUF[WD+1]-BUF[WD])*SCALE);
                WAIT_UNTIL(TIME_NOW+DURATION/NWDS);
            END "WLP";
        END "PLP";
        SET(UNIT,PARAM,0);      ⊃ Don't change it any more;
END;
```

Presumably, TIME_NOW is a global variable that has the current "time".

At the time DO_IT_ALL is called, we have a bunch of processes, one for each function, all of which are suspended in a WAIT_UNTIL. WAIT_UNTIL must have built a list (or whatever) that has all the processes on it in time order, and every time you call WAIT_UNTIL, it suspends you and puts you in this list in wake-up time order. There are some clever ways to do this described in Knuth, volume 3, under the name "balanced trees". All DO_IT_ALL does is find the first guy in time, remove him from the list, and RESUME him (a SAIL call). When he gives a WAIT_UNTIL, he will be suspended again and DO_IT_ALL will be RESUMEd. DO_IT_ALL will continue until the list is empty.

Well, it looks a bit complex, but that's only because it is, plus I've showed you all the gory details, which we can presumably hide under some more layers of subroutines and macros.

These ideas are not entirely debugged yet, plus I don't entirely understand SAILs process and co-routine stuff yet, but I think this is the way it all works.

There are several problems to be resolved. Like first off, there has to be some way to make sure that these processes get run in the right order. For instance, we have to run the main program which starts all the voices, then the main program has to wait while all the voices start their first notes, and the voices have to wait while all the STUFF routines get their first function points out. There is a problem such that if another voice gets run before all the STUFFs for a particular voice get started, then we'll get non-monotonic time. I think this can be done with the priority feature in SAIL. The other problem is that if the voices proceeded straight through, we would get a zillion processes started, because all of the notes for a voice would get SPROUTed. This is sort of a disaster, especially for a long piece. Ideally, each voice would start a note, each note would start all its STUFFs, each STUFF would issue one WAIT_UNTIL, and the instrument would proceed until its final WAIT_UNTIL(TIME+DURATION), at which point the voice would remain suspended and the main program should run to start the next voice. A bit tricky.

arbitrarily because it is the size of a disk track) and a pointer to the next record (if any).

We can thus "play" a pre-computed command list just by calling a routine like this:

```
        PLAY_FILE(channel);
        PLAY_CORE(core_pointer);
```

This will forward a string of commands to the box from a file (the channel must be already open for reading in mode '17) or from a record list like the one that was made in the above command.

For sample-data, we can imagine that it should either come from a file, or the ADC. There is a third case where it should come from both, but I don't want to think about having to interleave the data streams right now, so I'll ignore that possibility (although it could be added, if enough processing time were available).

```
        INPUT_DATA(channel);
```

This routine is enough. You either have a file opened for reading on that channel (in mode '17) which points to a sound file already in the proper format, or you have the ADC open and all the modes set (clock rate, paking, number of channels, etc etc etc).

Likewise:

```
        OUTPUT_DATA(channel);
```

I think this covers all the things the low level is supposed to do, so let's move on to the higher level.

There is one other routine that is needed:

```
        GIVE_BACK(core_pointer);
```

This releases the core list of commands.