Name: Gareth Loy

Project: 1 Programmer: DGL

File Name: INTERM.SAI[SAM,MUS]

File Last Written: 21:28 16 Nov 1977

Time: 21:54 Date: 16 Nov 1977

Stanford University
Artificial Intelligence Laboratory
Computer Science Department
Stanford, California

```
COMMENT ⊗   VALID 00026 PAGES
C REC  PAGE   DESCRIPTION
C00001 00001
C00004 00002    entry PINIT,PFINISH,PRESET,SCLOCK,GETITM,RELITM,WAIT_UNTIL
C00008 00003    ! WAIT_UNTIL(sample number) - queues this process.
C00010 00004    ! MONITOR - low-priority scheduler
C00011 00005    ! GETSM, RELSM - Get and release sum memory.
C00012 00006    ! SCLOCK - sets clock rate and updates magic numbers
C00013 00007    ! PINIT, PFINISH - Set up world, clear world
C00019 00008    ! PRESET - Clear out all parameters of all units
C00021 00009    ! LSEG - Routine for stuffing SEG-type function into a linear function
C00028 00010    ! LSEGS - Routine for stuffing SEG-type function into a sampled function,
C00031 00011    ! LFUN - Routine for stuffing sampled-data function into a linear paramete
C00034 00012    ! LFUNS - Routine for stuffing sampled-data function into a parameter,
C00037 00013    ! COSC - Gets and claims oscillator.
C00039 00014    ! GOSC - Claims and starts up an oscillator
C00041 00015    ! FUNCT - Just reads a function into sum memory. Good for envelopes.
C00043 00016    ! GOSCP - Claims and starts up an oscillator, phase settable.
C00045 00017    ! GOSCVF - Claims and starts up an oscillator
C00048 00018    ! FACT - Multiply a signal by a constant factor between 0 and 16.
C00050 00019    ! MIX - Multiply two signals by a constant factors between 0 and 8.
C00053 00020    ! MPY - Multiply two signals together with scale factor<8.
C00055 00021    ! SHAPE - apply an envelope to a signal of amp up to 8.0
C00057 00022    ! RANDOM - Make random numbers scaledin a certain way
C00059 00023    ! FILTER - Does one or two pole or zero fixed character filtering.
C00062 00024    ! DMGET, DMREL - claim and release storage in delay memory
C00065 00025    ! REV - Reverberator. Can be all-pass or comb.
C00067 00026    ! End of the world
C00068 ENDMK
C⊗;
```

```
entry PINIT,PFINISH,PRESET,SCLOCK,GETITM,RELITM,WAIT_UNTIL;
begin "INTERM"
Require "SYS:PROCES.DEF" source_file;
Require "LOWER.DEF[SAM,MUS]" source_file;
Require "LOWER.REL[SAM,MUS]" load_module;
define crlf="'15&'12",!="comment",nitems="200",pi="3.14159265";
require nitems new_items;
safe itemvar array bag[1:nitems+1];
integer bagptr;
safe itemvar array queue[1:nitems+1];
safe integer array times[1:nitems+1];
item it;
integer qlen,nwaits,maxqlen,sumqlen;

Internal integer pak; ! Tested for packing mode by SCLOCK;

Internal real srate,mag,outflag;
Internal integer npticks,nuticks,Nchans;
Internal integer zero,  ! Is sum memory location that always has zeros in it;
  outA,outB,outC,outD,outE,outF,outG,outH;
                        ! Sum memory locations where output DAC channels go;
Internal integer outmA,outmB,outmC,outmD,outmE,outmF,outmG,outmH;
define maxchns="16";
Internal integer array outN[0:maxchns-1];! Has sum memory loc. to write into for dac;
integer array outNg[0:maxchns-1];       ! Has generator pe numbers for output chns;
Internal integer array outmN[0:maxchns-1];! Has mod sum mem. loc. to write for dac;
integer array outNm[0:maxchns-1];       ! Has modifier pe numbers for output chns;

! DM definitions . . . ;

define ndms="32",totdm="(48*1024)";     ! That's how much is available now;
integer ndmsegs,totleft;
integer array dmbase,dmlen[1:ndms+1];

Internal itemvar procedure GETITM;
if bagptr>1 then return(bag[bagptr←bagptr-1])
else usererr(0,0,"GETITM: No more items avaiable - Sorry!");

Internal procedure RTNITM(itemvar foo);
begin
        if bagptr≥nitems then
            usererr(0,0,"RTNITM: Attempt to return more items than there are?!?");
        bag[bagptr]←foo;
        bagptr←bagptr+1;
end;

record_class seg(integer type; record_pointer(any_class) next,last; string name;
        integer nsegs; real mintime,maxtime,minval,maxval; real array times,values);

record_class sseg(integer type; record_pointer(any_class) next,last; string name;
        integer nsegs; real mintime,maxtime,minval,maxval;
        real array times,values,svalues; integer npts);

record_class synth(integer type; record_pointer(any_class) next,last; string name;
        integer nh; integer array hnums; real array amps,phases;
        real maxamp,sumamp,dc);
```

```
! WAIT_UNTIL(sample number) - queues this process.

This should eventually be recoded to be either a binary
search or a balanced binary tree.
;

Internal recursive procedure WAIT_UNTIL(integer sample);
begin
        integer i,j;

        nwaits←nwaits+1;
        if qlen>0 then
        begin "MVQ"
          for i←1 step 1 until qlen do
            if sample<times[i] then done;
          for j←qlen step -1 until i do
          begin "INSER"
            times[j+1]←times[j];
            queue[j+1]←queue[j];
          end "INSER";
          times[i]←sample;
          queue[i]←myproc;
          qlen←qlen+1;
          maxqlen←maxqlen max qlen;
          sumqlen←sumqlen+qlen;
        end "MVQ"
        else begin "FST"
          times[1]←sample;
          queue[1]←myproc;
          qlen←1;
        end "FST";

        suspend(myproc);
end;
```

```
! MONITOR - low-priority scheduler;

procedure MONITOR;
while true do
begin "MON"
        itemvar next;
        if qlen=0 then usererr(0,0,"MONITOR: Nobody to run!");
        next←queue[1];
        if (qlen←qlen-1)>0 then
        begin "MVQ"
            arrblt(queue[1],queue[2],qlen);
            arrblt(times[1],times[2],qlen);
        end "MVQ";
        resume(next,it,readyme);
end "MON";
```

```
! GETSM, RELSM - Get and release sum memory.
If it is for modifier rather than generator, then GENERATOR should
be false.
Returns -1 if there are no more left.

This should eventually be recoded to be a linked list, since order
doesn't really matter except for mod_this_pass stuff, which should
probably be allocated separately.
;

Internal integer procedure GETSM(boolean generator);
begin
        integer loc;
        loc ← Get(id_sum_memory+
                (if generator then last_gen_pass else last_mod_pass));
        return(loc);
end;

Internal procedure RELSM(integer sum);
Give(sum);
```

```
! SCLOCK - sets clock rate and updates magic numbers;

Internal procedure SCLOCK(integer nptix(96),nutix(32));
begin
        set_field(processing_ticks,nptix);
        set_field(total_ticks,nptix+nutix);
        srate←1/(0.000000195*(nptix+nutix+8)); ! dgl- spec. sheet changed to 8;
        mag←(1 lsh (if pak ≠ full_word then 20 else 28))/srate;
        npticks←nptix;
        nuticks←nutix;
end;
```

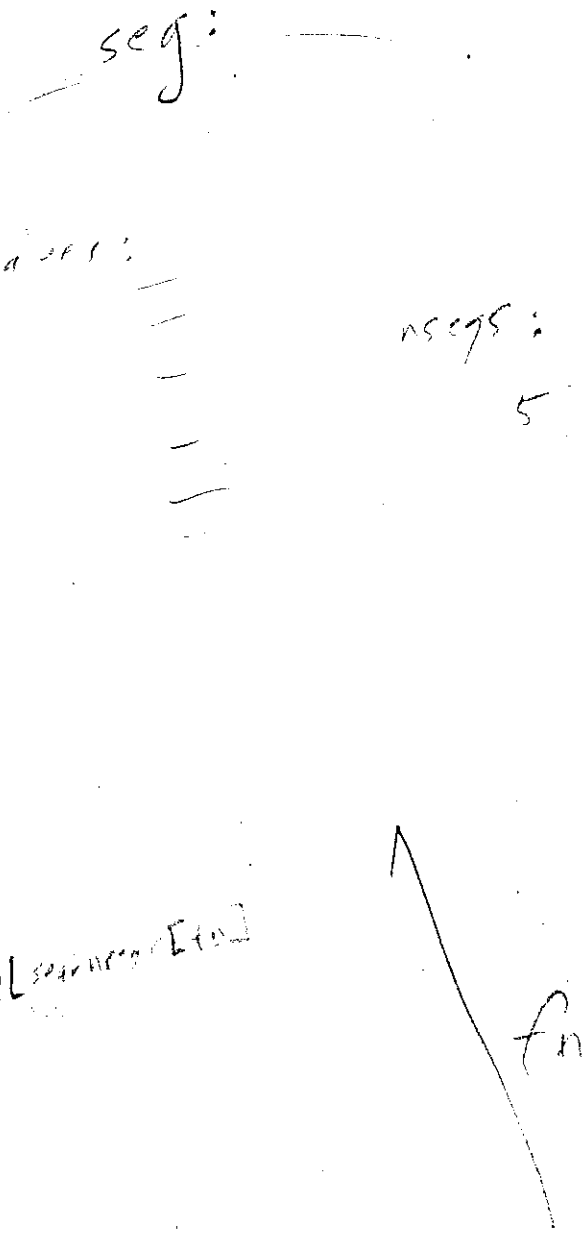
```
! PINIT, PFINISH - Set up world, clear world;

itemvar mon;

Internal procedure PINIT(integer noutchns(4),whichside(-1),filter(3),
        numptix(96), numutix(32));
begin
        integer gen,mod,gsm,msm,i;

        procedure CHSET(integer i,gsm,msm);
        begin "CHS"
            if 0<i≤8 then
              case i of
            begin "CMS"
                [1] begin outA←gsm; outmA←msm; end;
                [2] begin outB←gsm; outmB←msm; end;
                [3] begin outC←gsm; outmC←msm; end;
                [4] begin outD←gsm; outmD←msm; end;
                [5] begin outE←gsm; outmE←msm; end;
                [6] begin outF←gsm; outmF←msm; end;
                [7] begin outG←gsm; outmG←msm; end;
                [8] begin outH←gsm; outmH←msm; end
            end "CMS";
        end "CHS";

        initialize;             ! Init MWK's low-level code;

        SClock(Numptix,Numutix); ! set hi tix;

        set_field(ttl_load,case filter of(f0,f1,f2,f3)); ! Set analog filters;
        set_field(set_passes,0);

        outflag←whichside;
        ndmsegs←0;
        totleft←totdm;          ! Initialize DM allocation scheme;

        defpss←6*32;    ! Give processes a nice big stack;
        for bagptr←1 step 1 until nitems do
            bag[bagptr]←new;
        bagptr←nitems+1;
        maxqlen←nwaits←qlen←0;
        mon←getitm;
        sprout(mon,monitor,runme+priority(15));

        zero←getsm(false);      ! Get one for all zeros;

        if noutchns>maxchns then
            usererr(0,0,"INIT_ALL: Sorry, only "&cvs(maxchns)&" channels now");
        Nchans←noutchns;
        mod←-1;
        for i←0 step 1 until noutchns-1 do
        begin "NOUTC"
            gen←get(id_generator);
            if whichside≤0 then ! Gen only, or both mod and gen sum locs requested;
               gsm←msm←getsm(true)  ! Get generator side sum memory location;
            else gsm←msm←getsm(false); ! Only mod sum locs requested;
            if whichside<0 then msm←getsm(false); ! Both sides of sum mem requested;
            bind(gen,fm,msm); ! Set up gen in dac write mode;
            bind(gen,sweep,i);   ! DAC number is in the sweep slot;
            bind(gen,mode,dac_write);       ! Start up generator feeding DAC;
            outmN[i]←outN[i]←gsm; ! These now have the adr. of the sum memory loc;
```

```
            outNg[i]←gen;          ! OutNg now has the p.e. number writing to dac;
            outNm[i]←-1; ! No mod number yet so set to fail until allocated;
            if whichside<0 then ! Set up both sides;
            begin "SHUTL"
                mod←get(id_modifier);
                outmN[i]←msm; ! Now set to mod sum mem loc;
                outNm[i]←mod; ! And the mod p.e.;
                bind(mod,coeff0,'1000000);
                bind(mod,coeff1,0);
                bind(mod,A_in,gsm);
                bind(mod,B_in,zero);
                bind(mod,sum_memory,msm);
                bind(mod,A_scale,1);
                bind(mod,B_scale,0);
                bind(mod,mode,mixing);
            end "SHUTL";
            chset(i,gsm,msm);
        end "NOUTC";

        for i←Nchans+1 step 1 until maxchns-1 do
        begin "EXTC"
            outN[i]←gsm;          ! Pad with last real channel;
            outNg[i]←gen;
            outmN[i]←msm;
            outNm[i]←mod;
            chset(i,gsm,msm);
        end "EXTC";

end;

Internal procedure PFINISH;
begin
        integer i;

        relsm(zero);
        for i←1 step 1 until Nchans do
        begin "RLCH"
            give(outNg[i]);
            relsm(outN[i]);
            if outflag<0 then
            begin "RLSH"
                give(outNm[i]);
                relsm(outmN[i]);
            end "RLSH";
        end "RLCH";
        flush;
        terminate(mon);
        outstr("PFINISH: Scheduling statistics:"&crlf&
            "Number of waits = "&cvs(nwaits)&
            ", Max Q len = "&cvs(maxqlen)&crlf&
            "Average Q len = "&cvs(sumqlen/nwaits)&crlf);
end;
```

```
! PRESET - Clear out all parameters of all units;

Internal procedure PRESET;
begin
        integer i,gen,mod,dly;

        for i←0 step 1 until 255 do
        begin "CLG"
            gen←bias_generator+i;
            bind(gen,mode,0);
            bind(gen,fm,0);
            bind(gen,sum_memory,0);
            bind(gen,asymptote,0);
            bind(gen,exponent,0);
            bind(gen,rate,0);
            bind(gen,scale,0);
            bind(gen,ncosines,0);
            bind(gen,angle,0);
            bind(gen,frequency,0);
            bind(gen,sum_memory,0);
            bind(gen,sweep,0);
        end "CLG";

        for i←0 step 1 until 127 do
        begin "CLM"
            mod←bias_modifier+i;
            bind(mod,mode,0);
            bind(mod,coeff0,0);
            bind(mod,coeff1,0);
            bind(mod,term_0,0);
            bind(mod,term_1,0);
            bind(mod,A_in,0);
            bind(mod,B_in,0);
            bind(mod,sum_memory,0);
        end "CLM";

        for i←0 step 1 until 31 do
        begin "CLD"
            dly←bias_delay+i;
            bind(dly,mode,0);
            bind(dly,delay_length,0);
            bind(dly,index,0);
            bind(dly,base_address,0);
        end "CLD";
end;
```

```
! LSEG - Routine for stuffing SEG-type function into a linear function;

Internal recursive procedure LSEG(integer stsamp,nsamps;
        real scl,offset;
        integer gen; boolean freq; record_pointer(seg) fn);
begin
        integer sg,cval,crate,np,csamp,nxtp,nxtval,cdiff,lstsamp;
        integer qfield,dfield;
        real abscl,aboff,lmag;  ! Scale factor and abcissa to convert
                                  to sample number;
        procedure SET_DIFF(integer ind);
        begin "SDIFF"
            nxtp←abscl*seg:times[fn][ind+1]+aboff;
            np←nxtp-pass;
            nxtval←lmag*(scl*seg:values[fn][ind+1]+offset);
            cdiff←(nxtval-cval)/np;
            lstsamp←pass;
            bind(gen,dfield,cdiff);
        end "SDIFF";

        if freq then
        begin "SFR"
            qfield←frequency;
            dfield←sweep;
            lmag←'4000000/srate;
        end "SFR"
        else begin "SAMPL"
            qfield←exponent;
            dfield←rate;
            lmag←'77777777;! DGL  11/16 - changed from '3777777;
        end "SAMPL";

        if seg:type[fn]≠0 ∧ seg:type[fn]≠1 then
            usererr(0,0,"LSEG: function record not a SEG record");
        if seg:nsegs[fn]≤0 then return;
        abscl←nsamps/(seg:maxtime[fn]-seg:mintime[fn]);
        aboff←stsamp-abscl*seg:mintime[fn];

        wait_until(csamp←stsamp);       ! Get to beginning of note;
        if csamp>pass then set_field(dwell,csamp);
                ! Get box to beginning of note;

        cval←lmag*(scl*seg:values[fn][1]+offset);
        if seg:nsegs[fn]≤1 then bind(gen,dfield,0)
        else set_diff(1);

        lstsamp←pass;   ! Record sample when value went out;
        bind(gen,qfield,cval);

        for sg←2 step 1 until seg:nsegs[fn]-1 do
        begin "STFLP"
            wait_until(nxtp);
            if nxtp>pass then set_field(dwell,nxtp);
            cval←cval+cdiff*(pass-lstsamp);  ! Figure out where it is;
            set_diff(sg);
        end "STFLP";

        wait_until(nxtp);
        if nxtp>pass then set_field(dwell,nxtp);
        if seg:nsegs[fn]>1 then
        begin "FINAL"
```

```
            bind(gen,dfield,0);
            bind(gen,qfield,
                imag*(scl*seg:values[fn][seg:nsegs[fn]]+offset));
        end "FINAL";
        rtnitm(myproc);
end;
```



```
! LSEGS - Routine for stuffing SEG-type function into a sampled function,
I.E., one without the ability to ramp anywhere.
It will reset the value every SPACING samples.
Quantity is assumed to be 20-bit number;

Internal recursive procedure LSEGS(integer stsamp,nsamps,spacing;
        real scl,offset;
        integer unit,field; record_pointer(seg) fn);
begin
        integer sg,cval,crate,np,csamp,nxtp,nxtval,cdiff,lstsamp;
        integer qfield,lmag;
        real abscl,aboff;       ! Scale factor and abcissa to convert
                                  to sample number;
        lmag←'3777777;

        if seg:type[fn]≠0 ∧ seg:type[fn]≠1 then
            usererr(0,0,"LSEGS: function record not a SEG record");
        if seg:nsegs[fn]≤0 then return;
        abscl←nsamps/(seg:maxtime[fn]-seg:mintime[fn]);
        aboff←stsamp-abscl*seg:mintime[fn];

        sg←0;
        for nxtp←stsamp step spacing until stsamp+nsamps-1 do
        begin "STFLP"
            real slope,intercept;

            wait_until(nxtp);
            if nxtp>pass then set_field(dwell,nxtp);
            if sg=0 ∨ nxtp≥abscl*seg:times[fn][sg+1]+aboff then
            begin "CMPA"
                sg←sg+1;
                if sg≥seg:nsegs[fn] then
                begin "ND"
                    slope←0;
                    intercept←lmag*(scl*seg:values[fn][seg:nsegs[fn]]+offset);
                end "ND"
                else begin "MID"
                    real v2,v1,s2,s1;
                    v2←lmag*(scl*seg:values[fn][sg+1]+offset);
                    v1←lmag*(scl*seg:values[fn][sg]+offset);
                    s2←abscl*seg:times[fn][sg+1]+aboff;
                    s1←abscl*seg:times[fn][sg]+aboff;
                    slope←(v2-v1)/(s2-s1);
                    intercept←v1-s1*slope;
                end "MID";
            end "CMPA";
            cval←nxtp*slope+intercept;
            bind(unit,field,cval);
        end "STFLP";

        wait_until(nxtp←stsamp+nsamps-1);
        if nxtp>pass then set_field(dwell,nxtp);
        if seg:nsegs[fn]>1 then
            bind(unit,field,
                lmag*(scl*seg:values[fn][seg:nsegs[fn]]+offset));
        rtnitm(myproc);
end;
```

```
! LFUN - Routine for stuffing sampled-data function into a linear parameter;

Internal recursive procedure LFUN(integer stsamp,npts,spacing;
        real scl,offset;
        integer gen; boolean freq; integer locX);
begin
! LOCX is location of word before first data word;

        integer sg,cval,crate,np,csamp,nxtp,nxtval,cdiff,lstsamp;
        integer qfield,dfield;
        real lmag;

        procedure SET_DIFF(integer ind);
        begin
            nxtp←stsamp+ind*spacing;
            np←nxtp-pass;
            nxtval←lmag*(scl*memory[locX+ind+1,real]+offset);
            cdiff←(nxtval-cval)/np;
            lstsamp←pass;
            bind(gen,dfield,cdiff);
        end;

        if freq then
        begin "SFRR"
            qfield←frequency;
            dfield←sweep;
            lmag←'4000000/srate;
        end "SFRR"
        else begin "SMPL"
            qfield←exponent;
            dfield←rate;
            lmag←'3777777;
        end "SMPL";

        if npts≤0 then return;

        wait_until(csamp←stsamp);       ! Get to beginning of note;
        if csamp>pass then set_field(dwell,csamp);
                ! Get box to beginning of note;

        cval←lmag*(scl*memory[locX+1,real]+offset);
        if npts≤1 then bind(gen,dfield,0)
        else set_diff(1);

        lstsamp←pass;   ! Record sample when value went out;
        bind(gen,qfield,cval);

        for sg←2 step 1 until npts-1 do
        begin "STFIT"
            wait_until(nxtp);
            if nxtp>pass then set_field(dwell,nxtp);
            cval←cval+cdiff*(pass-lstsamp);  ! Figure out where it is;
            set_diff(sg);
        end "STFIT";

        wait_until(nxtp);
        if nxtp>pass then set_field(dwell,nxtp);
        if npts>1 then
        begin "FINPT"
            bind(gen,dfield,0);
            bind(gen,qfield,
```

```
                imag*(scl*memory[locX+npts,real]+offset));
        end "FINPT";
        rtnitm(myproc);
end;
```

```
! LFUNS - Routine for stuffing sampled-data function into a parameter,
I.E., one without the ability to ramp anywhere.
It will reset the next point in the array every SPACING samples.
Again, locX is address of word before first word of data.
It will deliver exactly NPTS points to the parameter then quit.
Quantity is assumed to be 20-bit number;

Internal recursive procedure LFUNS(integer stsamp,npts,spacing;
        real scl,offset;
        integer unit,field,locX);
begin
        integer sg,cval,crate,np,csamp,nxtp,nxtval,cdiff,lstsamp;
        integer qfield,lmag;
        real abscl,aboff;       ! Scale factor and abcissa to convert
                                  to sample number;
        lmag←'3777777;

        if npts≤0 then return;

        for sg←1 step 1 until npts do
        begin "STFLP"
            real slope,intercept;

            nxtp←stsamp+(sg-1)*spacing;
            wait_until(nxtp);
            if nxtp>pass then set_field(dwell,nxtp);
            bind(unit,field,lmag*memory[locX+sg,real]);
        end "STFLP";
        rtnitm(myproc);
end;
```

```
! COSC - Gets and claims oscillator.
Returns oscillator number.
If mode is SUM_OF_COSINES, then NCOSINES is interpreted and set accordingly.
Else NCOSINES is ignored and actual oscillator parameter is cleared.
Just clears things out. Doesn't sprout anything.
Don't forget - calling routine is responsible for returning generator.
;

Internal integer procedure COSC(real freq,angl; ! Frq in Hz, ang in rads;
        integer mod,ncs,        ! Generator mode, number of cosines;
            fmsum,outsum        ! FM input, output location;);
begin
        integer gen,i;

        gen←get(id_generator);
        bind(gen,mode,0);
        bind(gen,sum_memory,outsum);
        bind(gen,sweep,0);
        bind(gen,frequency,freq*mag);
        bind(gen,fm,fmsum);
        bind(gen,asymptote,0);
        bind(gen,exponent,0);           ! Everything starts at zero;
        bind(gen,rate,0);
        bind(gen,angle,'4000000*angl/(2*pi));
        if mod=sum_of_cosines then
        begin "SOC"
            for i←1 step 1 until 10 do
              if (1 lsh i)≥ncs then done;
            bind(gen,scale,i);  ! Get power of two greater than or equal to;
            bind(gen,ncosines,ncs);
        end "SOC"
        else begin "ELS"
            bind(gen,scale,0);
            bind(gen,ncosines,0);
        end "ELS";
        bind(gen,mode,mod);
        return(gen);
end;
```

```
! GOSC - Claims and starts up an oscillator
Returns oscillator number.
Returns process item of amplitude function.
If mode is SUM_OF_COSINES, then NCOSINES is interpreted and set accordingly.
Else NCOSINES is ignored and actual oscillator parameter is cleared.
Note that calling routine is responsible for returning process item.
Don't forget - calling routine is responsible for returning generator.
Sets initial phase to zero.
;

Internal integer procedure GOSC(integer stsamp,nsamps; real freq;
        record_pointer(seg) fn;          ! This is the amplitude function;
        real ampscl,ampoff;              ! Scale factor and offset for amplitude;
        integer mod,ncs,                 ! Generator mode, number of cosines;
           fmsum,outsum                  ! FM input, output location;);
begin
        integer gen,i;

        gen←cosc(freq,0,mod,ncs,fmsum,outsum);
        sprout(getitm,lseg(stsamp,nsamps,ampscl,ampoff,gen,false,fn),runme);
        return(gen);
end;
```

```
! FUNCT - Just reads a function into sum memory. Good for envelopes.
Returns oscillator number.
Returns process item of amplitude function.
Note that calling routine is responsible for returning process item.
Don't forget - calling routine is responsible for returning generator.
;

Internal integer procedure FUNCT(integer stsamp,nsamps;
        record_pointer(seg) fn;         ! This is the amplitude function;
        real ampscl,ampoff;             ! Scale factor and offset for amplitude;
        integer outsum                  ! output location;);
begin
        integer gen,i;

        gen←cosc(0,pi/2,a_running+1plusq+sine,0,zero,outsum);
        sprout(getitm,lseg(stsamp,nsamps,ampscl,ampoff,gen,false,fn),runme);
        return(gen);
end;
```

```
! GOSCP - Claims and starts up an oscillator, phase settable.
Returns oscillator number.
Returns process item of amplitude function.
If mode is SUM_OF_COSINES, then NCOSINES is interpreted and set accordingly.
Else NCOSINES is ignored and actual oscillator parameter is cleared.
Note that calling routine is responsible for returning process item.
Don't forget - calling routine is responsible for returning generator.
;

Internal integer procedure GOSCP(integer stsamp,nsamps; real freq,ang;
        record_pointer(seg) fn;          ! This is the amplitude function;
        real ampscl,ampoff;              ! Scale factor and offset for amplitude;
        integer mod,ncs,                 ! Generator mode, number of cosines;
            fmsum,outsum                 ! FM input, output location;);
begin
        integer gen,i;

        gen←cosc(freq,ang,mod,ncs,fmsum,outsum);
        sprout(getitm,lseg(stsamp,nsamps,ampscl,ampoff,gen,false,fn),runme);
        return(gen);
end;
```

```
! GOSCVF - Claims and starts up an oscillator
Same as GOSC but with function controlling frequency as well as amplitude.
Returns oscillator number.
Returns process item of amplitude function and frequency function
If mode is SUM_OF_COSINES, then NCOSINES is interpreted and set accordingly.
Else NCOSINES is ignored and actual oscillator parameter is cleared.
Note that calling routine is responsible for returning process item.
Don't forget - calling routine is responsible for returning generator.
Sets initial phase to zero.
;

Internal integer procedure GOSCVF(integer stsamp,nsamps;
        record_pointer(seg) frfn;       ! This is the frequency function;
        real frscl,froff;               ! Scale factor and offset for frequency;
        record_pointer(seg) fn;         ! This is the amplitude function;
        real ampscl,ampoff;             ! Scale factor and offset for amplitude;
        integer mod,ncs,                ! Generator mode, number of cosines;
            fmsum,outsum                ! FM input, output location;);
begin
        integer gen,i;

        gen←cosc(0,0,mod,ncs,fmsum,outsum);
        sprout(getitm,lseg(stsamp,nsamps,ampscl,ampoff,gen,false,fn),runme);
        sprout(getitm,lseg(stsamp,nsamps,frscl,froff,gen,true,frfn),runme);
        return(gen);
end;
```

```
! FACT - Multiply a signal by a constant factor between 0 and 16.
Returns number of modifier it got.
You have to take care of returning said modifier when you are done!;

Internal integer procedure FACT(integer outloc,inloc; real factor);
begin
        integer mod,scal,cval;

        if factor>16.0 v factor ≤-16.0 then
            usererr(0,0,"FACT: Factor out of bounds  = "&cvg(factor));
        mod←get(id_modifier);
        bind(mod,mode,0);
        if abs(factor)≥8.0 then scal←3
        else if abs(factor)≥4.0 then scal←2
        else if abs(factor)≥2.0 then scal←1
        else scal←0;
        cval←(1 lsh (19+1-scal))*factor+0.5;    ! Round it. 19 because signed;
        cval←(cval min '1777777) max (-'2000000);
        mod←get(id_modifier);
        bind(mod,coeff0,cval);
        bind(mod,coeff1,cval);
        bind(mod,A_in,inloc);
        bind(mod,B_in,inloc);
        bind(mod,sum_memory,outloc);
        bind(mod,A_scale,scal);
        bind(mod,B_scale,scal);
        bind(mod,mode,mixing);

        return(mod);
end;
```

```
! MIX - Multiply two signals by a constant factors between 0 and 8.
Returns number of modifier it got.
You have to take care of returning said modifier when you are done!;

Internal integer procedure MIX(integer outloc,inlloc; real factor1;
        integer in2loc; real factor2);
begin
        integer mod,scale1,cval1,scale2,cval2;

        if factor1>8.0 v factor1 ≤-8.0 then
            usererr(0,0,"MIX: First factor out of bounds  = "&cvg(factor1));
        if factor2>8.0 v factor2 ≤-8.0 then
            usererr(0,0,"MIX: Second factor out of bounds  = "&cvg(factor2));

        mod←get(id_modifier);
        bind(mod,mode,0);

        if abs(factor1)≥4.0 then scale1←3
        else if abs(factor1)≥2.0 then scale1←2
        else if abs(factor1)≥1.0 then scale1←1
        else scale1←0;
        cval1←(1 lsh (19+1-scale1))*factor1+0.5;
        cval1←(cval1 min '1777777) max (-'2000000);

        if abs(factor2)≥4.0 then scale2←3
        else if abs(factor2)≥2.0 then scale2←2
        else if abs(factor2)≥1.0 then scale2←1
        else scale2←0;
        cval2←(1 lsh (19+1-scale2))*factor2+0.5;
        cval2←(cval2 min '1777777) max (-'2000000);

        bind(mod,coeff0,cval1);
        bind(mod,coeff1,cval2);
        bind(mod,A_in,inlloc);
        bind(mod,B_in,in2loc);
        bind(mod,sum_memory,outloc);
        bind(mod,A_scale,scale2);
        bind(mod,B_scale,scale1);       ! A_scale goes with coeff0;
        bind(mod,mode,mixing);

        return(mod);
end;
```

```
! MPY - Multiply two signals together with scale factor<8.
Returns modifier number.
Don't forget to return modifier when you are done.
;

Internal integer procedure MPY(integer outloc,in1loc,in2loc; real factor);
begin
        integer mod,scal,cval;

        if factor>8.0 v factor ≤-8.0 then
            usererr(0,0,"MPY: Factor out of bounds  = "&cvg(factor));

        mod←get(id_modifier);
        bind(mod,mode,0); ! m_inactive;

        if abs(factor)≥4.0 then scal←3
        else if abs(factor)≥2.0 then scal←2
        else if abs(factor)≥1.0 then scal←1
        else scal←0;
        cval←(1 lsh (19+1-scal))*factor+0.5;
        cval←(cval min '1777777) max (-'2000000);

        bind(mod,coeff1,cval);
        bind(mod,term_1,0);
        bind(mod,A_in,in1loc);
        bind(mod,B_in,in2loc);
        bind(mod,sum_memory,outloc);
        bind(mod,A_scale,scal);
        bind(mod,mode,four_quad_multiply);

        return(mod);
end;
```

```
! SHAPE - apply an envelope to a signal of amp up to 8.0
Claims a generator, a modifier, and a sum memory location.
It returns the generator as the value, but the sum memory
location and the modifier are returned through reference parameters.
You have to give them all back.
It gives you the process item for the amplitude function also.
;

Internal integer procedure SHAPE(integer stsamp,nsamps;
        record_pointer(seg) fn;          ! This is the amplitude function;
        real ampscl,ampoff;              ! Scale factor and offset for amplitude;
        integer outloc,inloc;            ! Sum memory locs of output and input;
        reference integer mod,sum;       ! Place to put modifier and sum mem;
        real factor                      ! Multiply whole thing by this;);
begin
        integer gen;

        sum←getsm(true);
        mod←mpy(outloc,inloc,sum,factor);
        gen←funct(stsamp,nsamps,fn,ampscl,ampoff,sum);
        sprout(getitm,lseg(stsamp,nsamps,ampscl,ampoff,gen,false,fn),runme);
        return(gen);
end;
```

```
! RANDOM - Make random numbers scaledin a certain way
TRIGGER is a sum-memory location. If it is non-zero, then the triggered
white noise will be used with that as the input. This parameter is
defaulted to zero (no-trigger) normally. The returned numbers will be
between -FACTOR and +FACTOR (which should be of magnitude
less than one). This mess uses two modifiers and a sum memory locations.
If SEED is zero, a new and unique random number will be used to start
the generator. If SEED is non-zero, then that number will specify the
starting point and the entire sequence will thus be determined from
this number. If you want to repeat the sequence ever, you ought to
set this to something non-zero
;

Internal procedure RANDOM(integer outloc; real factor;
        reference integer mod1,mod2,sum; integer trigger(0),seed(0));
begin
        mod1←get(id_modifier);
        bind(mod1,mode,0);
        sum←getsm(false);
        mod2←fact(outloc,sum,factor);
        bind(mod1,B_in,trigger);
        bind(mod1,sum_memory,sum);
        bind(mod1,B_scale,0);
        bind(mod1,coeff0,'46445);
        bind(mod1,coeff1,'1000000);
        bind(mod1,A_scale,1);
        bind(mod1,term_0,0);
        bind(mod1,term_1,'3777777*ran(seed));
        if trigger=0 then bind(mod1,mode,U_noise)
        else bind(mod1,mode,tr_U_noise);
end;
```

```
! FILTER - Does one or two pole or zero fixed character filtering.
Sets coefficients for you. R should be less than 1.0.
In the case of one pole, the frequency is ignored.
In the case of one zero, the frequency is taken to be the first
multiply (M1);

Internal integer procedure FILTER(integer outloc,inloc; real R,freq;
        boolean second_order(true),all_pole(true));
begin
        integer mod;
        integer scal,cval;
        real c1,c2;

        integer procedure GSCALE(integer factor);
        if abs(factor)≥4.0 then return(3)
        else if abs(factor)≥2.0 then return(2)
        else if abs(factor)≥1.0 then return(1)
        else return(0);

        mod←get(id_modifier);
        bind(mod,mode,0);
        bind(mod,A_in,inloc);
        bind(mod,sum_memory,outloc);

        if second_order then
        begin "SECO"
            c1←-2*R*cos(2*pi*freq/srate);
            c2←R↑2;
        end "SECO"
        else begin "FSTO"
            c1←R;
            c2←freq;
        end "FSTO";

        if all_pole then
        begin "RV"
            c1←-c1;
            if second_order then c2←-c2;
        end "RV";

        scal←gscale(c1);
        cval←(1 lsh (19+1-scal))*c1+0.5;
        cval←(cval min '1777777) max (-'2000000);
        if second_order then
        begin "SOC1"
            bind(mod,coeff0,cval);
            bind(mod,term_0,0);
            bind(mod,B_scale,scal);
        end "SOC1"
        else begin "FOC1"
            bind(mod,coeff1,cval);
            bind(mod,term_1,0);
            bind(mod,A_scale,scal);
        end "FOC1";

        scal←gscale(c2);
        cval←(1 lsh (19+1-scal))*c2+0.5;
        cval←(cval min '1777777) max (-'2000000);
        if second_order then
        begin "SOC2"
            bind(mod,coeff1,cval);
```

```
            bind(mod,term_1,0);
            bind(mod,A_scale,scal);
        end "SOC2"
        else begin "FOC2"
            if all_pole then bind(mod,term_0,cval)
            else bind(mod,coeff0,cval);
            bind(mod,B_scale,scal);
        end "FOC2";

        if second_order then
          if all_pole then bind(mod,mode,notwopoles)
          else bind(mod,mode,notwozeroes)
        else if all_pole then bind(mod,mode,one_pole)
        else bind(mod,mode,one_zero);
end;
```

```
! DMGET, DMREL - claim and release storage in delay memory
Returns base address of block upon success
Returns -1 if there isn't enough left
;

Internal integer procedure DMGET(integer length);
begin
        integer i,j,lstba;

        if length>totleft then return(-1);

        if ndmsegs=0 then
        begin "FST"
            ndmsegs←1;
            dmbase[1]←0;
            dmlen[1]←length;
            totleft←totleft-length;
            return(0);
        end "FST";

        lstba←0;
        for i←1 step 1 until ndmsegs do
        begin "TST"
            if (dmbase[i]-lstba)≥length then
            begin "INSR"
                for j←ndmsegs step -1 until i do
                begin "MVDN"
                    dmbase[j+1]←dmbase[j];
                    dmlen[j+1]←dmlen[j];
                end "MVDN";
                ndmsegs←ndmsegs+1;
                dmbase[i]←lstba;
                dmlen[i]←length;
                totleft←totleft-length;
                return(lstba);
            end "INSR";
            lstba←dmbase[i]+dmlen[i];
        end "TST";
        if (totdm-lstba)<length then return(-1);
        ndmsegs←ndmsegs+1;
        dmbase[ndmsegs]←lstba;
        dmlen[ndmsegs]←length;
        totleft←totleft-length;
        return(lstba);
end;

Internal procedure DMREL(integer baseaddr);
begin
        integer i,j;

        for i←1 step 1 until ndmsegs do
          if dmbase[i]=baseaddr then
        begin "CLRS"
            totleft←totleft+dmlen[i];
            arrblt(dmbase[i],dmbase[i+1],ndmsegs-i);
            arrblt(dmlen[i],dmlen[i+1],ndmsegs-i);
            ndmsegs←ndmsegs-1;
            return;
        end "CLRS";
        usererr(0,0,"DMREL: Releasing DM area that's not claimed = "&cvs(baseaddr));
end;
```

```
! REV - Reverberator. Can be all-pass or comb.
To make an allpass, set gl←-g0←G
Must release the delay and the memory separately.
The memory must be released with a DMREL(dlyadr)
;

Internal integer procedure REV(integer outloc,inloc;
        real g0,gl; integer length;
        reference integer dly,dlyadr);
begin
        integer mod;

        mod←get(id_modifier);
        dly←get(id_delay);
        bind(mod,mode,0);
        bind(dly,mode,0);

        bind(dly,delay_length,length-3);        ! Pipelining adds 3 samples;
!       bind(dly,index,0);
        dlyadr←dmget(length-3);
        bind(dly,base_address,dlyadr);

        bind(mod,coeff0,'1777777*g0);
        bind(mod,coeff1,'1777777*gl);
        bind(mod,term_0,0);
        bind(mod,term_1,0);
        bind(mod,A_in,inloc);
        bind(mod,B_in,dly);
        bind(mod,sum_memory,outloc);
        bind(mod,A_scale,0);
        bind(mod,B_scale,0);
        bind(dly,mode,delayline);
        bind(mod,mode,delay);

        return(mod);
end;
```

```
! End of the world;
end "INTERM";
```