makefile

```
# makefile for OFDM software
# using Microsoft Fortran v. 4.0 compiler
# Ed.Casas

COMP    = fl /Od /c /4Yd      # disable optimize, compile only
                              # strict declarations
ADD     = /4Yb /Ge            # /4Yb for runtime error trace,
                              # /Ge for stack check
LINK    = link                # link command
LINKOPT = /E ;                # EXEPACK option
D       = ..\bin              # binary destination directory
MOVE    = ( copy $*.exe $D ; \
            del $*.exe )      # install in binary directory

.SUFFIXES: .exe .obj .f .asm

.f.obj :
    ${COMP} ${ADD} /Tf $<

.asm.obj :
    masm $< ;

all : $D\fdint.exe $D\simrun.exe \
      $D\pint.exe $D\bkp.exe $D\ceval.c

clean :
    del *.obj
    del *.fX

# select code for for little- or big-endian CPUs

simfde.obj : simfde.f
   sed -f sim2ibm.sed <simfde.f >simfde.fX
   ${COMP} ${ADD} /4Ns /Tf simfde.fX

# IEEE DSP routines don't declare variables and subscripts are wrong

sifft.obj : sifft.f
   ${COMP} ${ADD} /4Ns /4Nd /4Nb /Tf sifft.f

pint.obj : pint.f simpdef.f simpget.f

$D\pint.exe : pint.obj simut.obj simsnr.obj
    ${LINK}      pint simut simsnr ${LINKOPT}
    ${MOVE}

fdint.obj : fdint.f simpdef.f simpget.f

$D\fdint.exe : fdint.obj simsnr.obj simfde.obj \
              simdiv.obj simut.obj simgen.obj
    ${LINK} fdint.obj simsnr.obj simfde.obj \
              simdiv simut simgen ${LINKOPT}
    ${MOVE}

simrun.obj : simrun.f simpdef.f simpget.f

io.obj : \hw\pcio\io.asm
    masm \hw\pcio\io.asm ;

$D\simrun.exe : simrun.obj simmod.obj simfec.obj \
              simfde.obj simgen.obj sifft.obj simsnr.obj simut.obj \
              simhw.obj simdiv.obj io.obj
    ${LINK}  simrun.obj simmod.obj simfec.obj \
              simfde.obj simgen.obj sifft.obj simsnr.obj simut.obj \
              simhw.obj simdiv.obj io.obj ${LINKOPT}
    ${MOVE}

$D\bkp.exe : bkp.c
    tc bkp.c
    ${MOVE}

$D\ceval.exe : ceval.c
    tc eval.c
    ${MOVE}
```

makefile (unix)

```
FC = f77
FFLAGS  = -f68881 -c -u -C
LD = f77 -f68881
LDOPT = -o $@
CVT = sed -f sim2unix.sed
OBJ = fdint pint simrun bkp ceval
BINDIR = ..\bin

all : ${OBJ}

install :
        mv ${OBJ} ../bin

sifft.o : sifft.f
        ${FC} -f68881 -c sifft.f

simfdeX.f : simfde.f
        ${CVT} simfde.f >simfdeX.f
```

```
fdintX.f : fdint.f simpdef.f simpget.f
        ${CVT} fdint.f >fdintX.f

fdint : fdintX.o simsnr.o simfdeX.o \
              simdiv.o simut.o simgen.o
        ${LD} fdintX.o simsnr.o simfdeX.o \
              simdiv.o simut.o simgen.o ${LDOPT}

pintX.f : pint.f simpdef.f simpget.f
        ${CVT} pint.f >pintX.f

pint : pintX.o simsnr.o simut.o
        ${LD} pintX.o simsnr.o simut.o ${LDOPT}

simrunX.f : simrun.f simpdef.f simpget.f
        ${CVT} simrun.f >simrunX.f

simrun : simrunX.o simmod.o simfec.o \
              simfdeX.o simgen.o sifft.o simsnr.o simut.o \
              simdum.o simdiv.o
        ${LD}  simrunX.o simmod.o simfec.o \
              simfdeX.o simgen.o sifft.o simsnr.o simut.o \
              simdum.o simdiv.o ${LDOPT}

pdf : pdf.o simut.o
    ${LD}  pdf.o simut.o ${LDOPT}

bkp : bkp.c
    gcc bkp.c -lm -o bkp

ceval : ceval.c
    gcc ceval.c -lm -o ceval
```

sim2ibm.sed

```
/^cIBM/s/cIBM/    /
```

sim2unix.sed

```
/^$LARGE/d
/^$INCLUDE: 'simpdef.f'/r simpdef.f
/^$INCLUDE: 'simpdef.f'/d
/^$INCLUDE: 'simpget.f'/r simpget.f
/^$INCLUDE: 'simpget.f'/d
/^cSUN/s/cSUN/    /
```

simrun.f

```
$LARGE
c ****************
c simrun.f - OFDM Simulation
c ****************
c Ed.Casas
c
c link with the following files :
c
c sifft.f  - FFTs (adapted from IEEE DSP library)
c simdiv.f - switching diversity
c simdum.f - dummy hardware interface (non-IBM PC)
c simfde.f - fading envelope generation
c simfec.f - FEC and bit/block error counting
c simgen.f - noise and data generators
c simhw.f  - hardware interface (IBM PC)
c simmod.f - OFDM (de)modulation
c simsnr.f - IF to AF signal/noise level conversion
c simut.f  - various utilities
c
c includes the following files:
c
c simpdef.f - declarations of simulation parameters
c simpget.f - input of simulation parameters
c
c Major Revisions:
c
c 85-  VAX/VMS FORTRAN and FPS AP-100 routines
c 86-  VAX/VMS FORTRAN
c 87-8 FORTRAN 77 (MicroSoft v3.3 compiler subset)
c

      program simrun

c ---------------------------------------------------------------------

$INCLUDE: 'simpdef.f'

c ---------------------------------------------------------------------

c local variables:

c i      - trial counter
c j      - simulation block counter
c k      - signal block size counter
c l      - snr counter
c n      - signal block size
c m      - index of OFDM block in simulation block
```

```
c sp     - signal power (calculated by modu)
c oldsnr - fading waveform average snr
c if1    - Index into array corresponding to value of f1
c if2    - Index into array corresponding to value of f2
c nf     - if2-if1+1 = number of frequencies between if1 and if2
c n0, n1 - number of non-error and errors (for runs test)
c nr     - number of runs (for runs test)
c r      - normalized number of runs statistic (for runs test)

      integer i, j, k, l, m, n
      integer n0, n1, nr
      integer if1, if2, nf
      real r
      real oldsnr

c statistics functions: mean, lower and upper .95 CI

      real stmn, stl95, stu95

c convert power to dB

      real dbp

c transmitted, received, and corrected bits and signal sample vectors:

      logical txdata(ns),             crdata(ns)
      real    txsig (ns), rxsig (ns), crsig (ns)

c the common block is a ns-real-long temporary work area
c used by interleaving and diversity routines

      real wrk(4*ns)
      common wrk

c noise and fading waveform sample vectors:

      real noise(ns), fade(ns)

c pre-emphasis and de-emphasis/channel-inversion vectors

      real prev(ns), dev(ns)

c error measurement variables:

c npass  - count of dfb correction passes
c nbe1  - bit error count on previous pass
c nbe2  - bit error count on current pass
c nwe2  - word error count on current pass
c nbe, nwe     - bit and word errors (in one trial)
c ber, bker    - sums of BERs and BkERs
c runs         - sums of number of runs statistics
c ber2, bker2  - sums of squares of BERs and BkERs
c runs2        - sums of squares of number of runs statistics
c nber, nbker  - number of BER and BkER trials
c nruns        - number of runs statistics
c nb, nw       - number of bits/words per trial
c nbx, nwx     - number of bits/words examined in a trial

      integer  nbe1, nbe2, nwe2, npass
      integer  nbx, nwx
      integer  nbe (mxblk,mxsnr), nwe  (mxblk,mxsnr)
      integer  nb  (mxblk,mxsnr), nw   (mxblk,mxsnr)
      real     ber (mxblk,mxsnr), bker (mxblk,mxsnr)
      real     runs (mxblk,mxsnr)
      real     ber2(mxblk,mxsnr), bker2(mxblk,mxsnr)
      real     runs2 (mxblk,mxsnr)
      integer  nber(mxblk,mxsnr), nbker(mxblk,mxsnr)
      integer  nruns (mxblk,mxsnr)

c test dibits or bits for runs test

      logical dibits
      parameter (dibits=.false.)


c initialize BER, BkER, and runs statistics variables

      data ber /mxblsn*0./, bker /mxblsn*0./, runs /mxblsn*0./
      data ber2/mxblsn*0./, bker2/mxblsn*0./, runs2/mxblsn*0./
      data nber/mxblsn*0 /, nbker/mxblsn*0 /, nruns/mxblsn*0 /

c *** program starts here ****

      write(*,*)'% OFDM Simulation - 88-6-4.'

c ------------------------------------------------------------------

$INCLUDE: 'simpget.f'

c ------------------------------------------------------------------

      if(fs.le.0)then
         write(*,*) ' simrun: bad fs '
         stop
      endif

      if((2*ne+n)*2.gt.4*ns)then
         write(*,*) ' simrun: ne too large '
         stop
      endif
```

```
c ------------------------------------------------------------------

      if(hw)then

c measure channel transfer function and generate pre/de-emphasis filter
c initialize for block of maximum lenght (ns), with sample work array
c enough for 4 seconds (4*8000=32000). txsig is temporary work vector.

         call hwinit(txdata,prev,dev,txsig,neqbl,
     +         wrk,ns,8*ns,fs,f1,f2,dbd,rms,peak,txemp,demp,
     +         empscl,nempsc,empfr,empsc)

      else

c initialize snr-to-signal and snr-to-noise tables

         call s2init(b,w,rms,peak,fm,fd,rfm,agclim,sqlim,
     +         nints,intsr,intss, nintn,intnr,intnn,
     +         fading, noisng )

      endif

c do ntr trials

      do 6 i=1,ntr

c reset bit, word, bit error and word error counts for this trial

         call vifill(nb ,mxblsn,0)
         call vifill(nw ,mxblsn,0)
         call vifill(nbe,mxblsn,0)
         call vifill(nwe,mxblsn,0)

c do "nblk" blocks per trial

         do 4 j=1,nblk

c generate data bits

            if(npat.eq.0)then
               call prbs(txdata,ns,sr)
            else
               call dwg(txdata,ns,pat,npat)
            endif

            if(.not.hw)then

c if noise enabled, generate a noise vector for proper noise density
c scaling is because measured noise powers only include frequencies
c in the data range.

               if(noisng)then
                  if(impnse)then
                     call vimp(noise,ns,prbimp,nseed)
                  else
                     call vgrand(noise,ns,nseed)
                  endif
                  call vsmul(noise,ns,sqrt( (fs/2.0)/(f2-f1) ))
               else
                  call vfill(noise,ns,0.)
               endif

c if fading enabled, generate snr values with 0 dB mean

               if(fading)then
                  call genfdb(fd/fs,fseed,fade,ns,
     +               ndbr,thr,ndw,nsw)
               else
                  call vfill(fade,ns,0.)
               endif
               oldsnr=0.

            endif

c do for all snrs

            do 3 l=1,nsnr

c set the fading waveform average snr

            if(.not.hw)then
               call setsnr(fade,ns,oldsnr,snr(l))
            endif

c do for all data block sizes

            do 3 k=1,nn

c set the block size

               n=na(k)

               if(n.le.0.or.n.gt.ns)then
                  write(*,*) ' simrun: bad N '
                  stop
               endif

c set lower/upper frequency limits and number of channels
```

```
               call setif(fs,n,f1,if1,f2,if2)
               nf=if2-if1+1

c do for all OFDM blocks in the ns-sample block

               do 3 m=1,ns,n

c encode the data into complex data values

               call encode(txdata(m),txsig,if1,if2,n)

c do pre-emphasis if sending over HW channel

               if(hw)then
                  call emp(txsig,prev,f1,f2,fs,n,ns)
               endif

c modulate the data into an ofdm signal (unit variance)

               call modu(txsig,n,nf,serial)

c send signal through noisy, fading channel

               call vquant(txsig,n,tmax,tquant)

               if(hw)then
                  call hwch(txsig,crsig,wrk,n,n+2*ne,rms,peak)
               else
                  call ch(txsig,crsig,noise(m),fade(m),n,rms,peak)
               endif

               call vquant(crsig,n,rmax,rquant)

c demodulate

               call demodu(crsig,n,nf,serial)

c do de-emphasis if sent over HW channel

               if(hw)then
                  call emp(crsig,dev,f1,f2,fs,n,ns)
               endif

               call decode(crdata,crsig,if1,if2,n)

c do [fec &] error checking

               call fec(crdata,txdata(m),nf,intlv,ecn,ect,
     +               nbe2,nbx,nwe2,nwx)

c continue correction passes until exceed iteration limit or no errors

               if(mxdfbp.gt.0)then

               call vcopy(crsig,rxsig,n)
               call modu(rxsig,n,nf,serial)

               npass=0
               nbe1=ns+1

1              continue
               if( (nbe2.eq.0) .or. (npass.gt.mxdfbp) )goto 2

c remodulate received data

                  call encode(crdata,crsig,if1,if2,n)
                  call modu(crsig,n,nf,serial)

c use original signal where received level > fade limit

                  call vsel(crsig,rxsig,fade(m),fdlim(l),n)

c demodulate the composite signal

                  call demodu(crsig,n,nf,serial)
                  call decode(crdata,crsig,if1,if2,n)

                  npass=npass+1
                  nbe1=nbe2

c do [fec &] error checking

                  call fec(crdata,txdata(m),nf,intlv,ecn,ect,
     +                  nbe2,nbx,nwe2,nwx)

                  goto 1
2              continue

                  if(dndfbp)then
                     write(*,'(1X,A2,I5,F4.0,I3,I5)')
     +                  '%D', n, snr(l), npass, nbe2
                  endif

               endif

c if required, display signal and noise powers in this block

                  if(dsn)then
                     call dvsn(crsig(if1),txdata(m),nf,
     +                  f1,fs,n,snr(l),ndsn,wrk(1),wrk(ns+1))
```

```
                  endif
                  if(dsnav)then
                     call davsn(crsig(if1),txdata(m),nf,
     +                  n,snr(l),wrk(1),wrk(ns+1))
                  endif

c if required, display transmitted and received signal values

                  if(ddat)then
                     call vddat(crsig(if1),txdata(m),nf,
     +                  f1,f2,n,snr(l))
                  endif

c if required, display number of errors in each fec block

               if(dnerr)then
                  call dne(crdata,txdata(m),nf,intlv,ecn,n,snr(l))
               endif

c update bit/word add bit/word error counts

               nbe(k,l)=nbe(k,l)+nbe2
               nb (k,l)=nb (k,l)+nbx
               nwe(k,l)=nwe(k,l)+nwe2
               nw (k,l)=nw (k,l)+nwx

c              write(*,*) ' blk, BER = ', j, float(nbe2)/nbx

c do runs test if required, only consider block with minrns+ runs

               if(rnstst)then
                  if(dibits)then
                     call vl2diff(crdata,txdata(m),nf,wrk,nf/2)
                     call runcnt(wrk,nf/2,n0,n1,nr,r)
                  else
                     call vxor(crdata,txdata(m),wrk,nf)
                     call runcnt(wrk,nf,n0,n1,nr,r)
                  endif
                  if(nr.ge.minrns)then
                     call stat(r,runs(k,l),runs2(k,l),nruns(k,l))
                  endif
               endif

3           continue

4     continue

c update BER and BkER statistics

      do 5 k=1,nn
      do 5 l=1,nsnr
         n=na(k)
         if(nb(k,l).ne.0)
     +   call stat(float(nbe(k,l))/nb(k,l),
     +               ber (k,l),ber2 (k,l),nber (k,l))
         if(nw(k,l).ne.0)
     +   call stat(float(nwe(k,l))/nw(k,l),
     +               bker(k,l),bker2(k,l),nbker(k,l))
5     continue

6     continue

c display results: runs tests

      if(rnstst)then
         do 7 k=1,nn
         do 7 l=1,nsnr
            if(nruns(k,l).ge.2)then
               write(*,'(4X,A,I6,F6.1,2(1PE11.2))')'%',na(k),snr(l),
     +            stl95(runs (k,l),runs2 (k,l),nruns (k,l)),
     +            stu95(runs (k,l),runs2 (k,l),nruns (k,l))
            else
               write(*,'(4X,A,I6,F6.1,A)')'%',na(k),snr(l),
     +            ' : too few runs.'
            endif
7        continue
      endif

c display results: mean BERs and BkERs (with .95 CI)

      write(*,*) '% bit error rates : '
      do 8 k=1,nn
      do 8 l=1,nsnr
         if(nber(k,l).ne.0)then
            write(*,'(1X,I6,F6.1,3(1PE11.2))')na(k),snr(l),
     +      stmn (ber (k,l),ber2 (k,l),nber (k,l)),
     +      stl95(ber (k,l),ber2 (k,l),nber (k,l)),
     +      stu95(ber (k,l),ber2 (k,l),nber (k,l))
         endif
8     continue

      write(*,*) '% FEC block error rates : '
      do 9 k=1,nn
      do 9 l=1,nsnr
         if(nbker(k,l).ne.0)then
            write(*,'(1X,I6,F6.1,3(1PE11.2))')na(k),snr(l),
     +      stmn (bker(k,l),bker2(k,l),nbker(k,l)),
     +      stl95(bker(k,l),bker2(k,l),nbker(k,l)),
     +      stu95(bker(k,l),bker2(k,l),nbker(k,l))
         endif
```

```
9     continue
      end
```

simpdef.f

```
c ************************
c simpdef.f - define simulation parameters
c ************************

c -------------------------------------------------------------------

c array sizes:

c ns     - maximum number of samples in each "simulation block"
c mxpat  - maximum number of elements in npat
c mxblk  - maximum number of elements in na
c mxsnr  - maximum number of elements in snr
c mxblsn - mxblk * mxsnr
c srlen  - number of elements in sr (must be 23 for CCITT v.29)
c mxint  - number of elements in intnr, intnn, intsr, intsn
c mxthr  - maximum number of elements in thrsh

      integer ns, mxblk, mxsnr, mxblsn, mxpat, srlen, mxint, mxemp
      integer mxthr
      parameter (ns=4096 ,mxblk=10, mxsnr=10, mxblsn=100, mxpat=20)
      parameter (srlen=23, mxint=100, mxemp=50, mxthr=30)

c agclim  - agc gain limit (SNR dB)
c agcvar  - true to test effct of varying agc threshold (pint only)
c b       - IF bandwidth (kHz)
c dbd     - dB per decade of pre-emphasis
c demp    - true to display equalization characteristics
c dndfbp  - true to display number of correction passes for each block
c dnerr   - true to display number of errors per fec work
c dsn     - true to display received data value signal and noise power
c dsnav   - true to display average of received signal and noise powers
c ddat    - true to display transmitted and received data values
c ecn     - word size
c ect     - correctable bit errors per word (0=no FEC)
c empfr   - upper frequency limit to use for scaling values in empsc
c empsc   - scaling values (in dB) for scaling pre-emphasis vectors
c empscl  - true to do scaling of pre-emphasis vector
c f1      - lowest  frequency to use on channel (<= 0    -> use minimum)
c f2      - highest frequency to use on channel (>= fs/2 -> use maximum)
c fading  - true to apply fading
c fd      - doppler rate (Hz)
c fdlim   - "faded" decision levels for DFB correction (SNR dB), per SNR
c fm      - true for FM, false for SSB
c fs      - sample rate (bps)
c fseed   - RNG seed to initialize fading generator (limits as above)
c hw      - true to send samples over A/D/A board (through hardware f.s.)
c impnse  - true to use IMPulse (cf. gaussian) NoiSE
c intlv   - true to do interleaving before checking for word errors
c intnn   - "n" part of points to use to change r-n table
c intnr   - SNR part of points to use to change r-n table
c intsr   - SNR part of points to use to change r-s table
c intss   - "s" part of points to use to change r-s table
c minrns  - minimum number of runs in block to use it for runs testing
c mxdfbp  - maximum number of dfb correction passes (0=no DFB)
c na      - array of OFDM block sizes, each a power of 2 in [8,ns]
c nblk    - number of simulation blocks (ns samples) per trial
c ndbr    - Number of Diversity BRanches
c ndsn    - number of channels averaged when display signal/noise powers
c ndw     - Number of samples to wait (DWell) after threshold crossed
c ne      - number of guard samples for hardware channel
c nempsc  - number of values in empsc and empfr
c neqbl   - number of blocks to average in measuring channel equalization
c nintn   - number of elements in intnr and intnn
c nints   - number of elements in intsr and intss
c nn      - number of OFDM block sizes to be tested
c noisng  - true to add noise
c npat    - Number of elements in data PATtern (0 for random data)
c nseed   - RNG seed for noise generators (0 <= nseed < 67108864)
c nsnr    - number of SNRs to test
c nsw     - Number of samples to blank output while SWitching branches
c nthrsh  - number of squelch or agc thresholds to test (pint only)
c ntr     - number of trials done each time subroutine called
c pat     - description of data PATtern
c pfile   - name of simulation parameter file
c rms     - rms voltage of modulating signal
c peak    - peak voltage of modulating signal
c prbimp  - probability of a sample having an impulsive noise "hit"
c rfm     - true to add Random FM
c rmax    - Receiver A/D MAXimum (clipping) value
c rnstst  - true to compute runs tests
c rquant  - bits of Receiver    A/D QUANTization (0=no quantization)
c serial  - true for "serial" (no OFDM) simulation
c snr     - IF SNRs (dB)
c sqlim   - squelch  limit (SNR dB)
c sqval   - true to test effect of varying squelch threshold (pint only)
c sr      - shift register logical values to generate PRBS data stream
c thr     - switching threshold (dB rel. to mean)
c thrsh   - agc or squelch thresholds to test
c tmax    - Transmitter D/A MAXimum (clipping) value
c tquant  - bits of Transmitter D/A QUANTization (0=no quantization)
c txemp   - true to amp./phase correction at transmitter (not receiver)
c vname   - variable name
c w       - baseband bandwidth (kHz)

      integer na(mxblk), nn, ntr, nblk
```

```
      integer ecn, ect
      integer ndbr, npat, pat(mxpat), nsnr
      integer tquant, rquant
      integer nsw, ndw
      integer ne, mxdfbp, neqbl
      integer nintn, nints, minrns
      integer ndsn, nempsc, nthrsh
      real fs, fd, snr(mxsnr), fdlim(mxsnr)
      real agclim, sqlim, b, w, rms, peak
      real prbimp
      real thr
      real f1, f2, tmax, rmax
      real dbd
      real intnr(mxint), intnn(mxint), intsr(mxint), intss(mxint)
      real empsc(mxemp), empfr(mxemp)
      real thrsh(mxthr)
      double precision nseed, fseed
      logical fm, fading, noisng, rfm, intlv, impnse, hw, serial
      logical dndfbp, dnerr, dsn, dsnav, demp, ddat, rnstst, txemp
      logical empscl, agcvar, sqvar
      logical sr(srlen)
      character*70 vname, pfile

c intrinsic functions

      integer min

c ------------------------------------------------------------------
```

simpget.f

```
c *********************************
c simpget.f - read simulation parameters
c *********************************

c ------------------------------------------------------------------

      write(*,*) '% simulation parameter file ? '
      read(*,'(A70)') pfile
      write(*,*) '% ', pfile

      open(10,file=pfile)

      read(10,*) vname
      write(*,*) '% ', vname

      read(10,*) vname, nn, (na(i),i=1,min(nn,mxblk))
      if(nn.gt.mxblk)then
         write(*,*) ' simpget : too many blocks sizes '
         stop
      endif
      write(*,*) '% na(i) =', (na(i),i=1,nn)

      read(10,*) vname, ntr, nblk
      write(*,*) '% ntr, nblk =', ntr, nblk

      read(10,*) vname, fs, fd
      write(*,*) '% fs, fd =', fs, fd

      read(10,*) vname, nsnr, (snr(i),i=1,min(nsnr,mxsnr))
      if(nsnr.gt.mxsnr)then
         write(*,*) ' simpget : too many SNRs '
         stop
      endif
      write(*,*) '% snr(i) =', (snr(i),i=1,nsnr)

      read(10,*) vname, serial
      write(*,*) '% serial =', serial

      read(10,*) vname, fm, b, w, rms, peak
      write(*,*) '% fm, b, w, rms, peak =', fm, b, w, rms, peak

      read(10,*) vname, agclim, sqlim
      write(*,*) '% agc, sqlim =', agclim, sqlim

      read(10,*) vname, fading, noisng
      write(*,*) '% fading, noisng =', fading, noisng

      read(10,*) vname, fseed, nseed
      write(*,*) '% fseed, nseed =', fseed, nseed

      read(10,*) vname, rnstst, minrns
      write(*,*) '% rnstst, minrns =', rnstst, minrns

      read(10,*) vname, mxdfbp, (fdlim(i),i=1,nsnr)
      write(*,*) '% mxdfbp, fdlim(i) =', mxdfbp, (fdlim(i),i=1,nsnr)

      read(10,*) vname, dndfbp, dnerr, demp, ddat
      write(*,*) '% dndfbp, dnerr, demp, ddat =',
     +   dndfbp, dnerr, demp, ddat

      read(10,*) vname, dsn, ndsn, dsnav
      write(*,*) '% dsn, ndsn, dsnav =', dsn, ndsn, dsnav

      read(10,*) vname, intlv
      write(*,*) '% intlv =', intlv

      read(10,*) vname, ecn, ect
      write(*,*) '% ecn, ect =',ecn,ect
```

```
      read(10,*) vname, ndbr, thr, ndw, nsw
      write(*,*) '% ndbr, thr, ndw, nsw =', ndbr, thr, ndw, nsw

      read(10,*) vname, npat, (pat(i),i=1,min(npat,mxpat))
      if(npat.gt.mxpat)then
         write(*,*) ' simpget : too many pattern elements '
         stop
      endif
      write(*,*) '% pat(i) =', (pat(i),i=1,npat)

      read(10,*) vname, (sr(i),i=1,srlen)
      write(*,*) '% sr(i) =', (sr(i),i=1,srlen)

      read(10,*) vname, rfm
      write(*,*) '% rfm =', rfm

      read(10,*) vname, impnse, prbimp
      write(*,*) '% impnse, prbimp =', impnse, prbimp

      read(10,*) vname, hw, ne, dbd, neqbl, txemp
      write(*,*) '% hw, ne, dbd, neqbl, txemp =',
     +      hw, ne, dbd, neqbl, txemp

      read(10,*) vname, f1, f2
      write(*,*) '% f1, f2 =', f1, f2
      if(f2.le.f1)then
         write(*,*) ' simpget : f2 <= f1 '
         stop
      endif

      read(10,*) vname, tquant, tmax, rquant, rmax
      write(*,*) '% tquant, tmax, rquant, rmax =',
     +   tquant, tmax, rquant, rmax

      read(10,*) vname, nints,
     +   (intsr(i), intss(i), i=1,min(nints,mxint))
      if(nints.gt.mxint)then
         write(*,*) ' simpget : too many signal interpolation values '
         stop
      endif
      write(*,'(a,1000(/'' %'',2f10.3))')
     +   ' % intsr(i), intss(i) =',
     +   (intsr(i), intss(i), i=1,nints)

      read(10,*) vname, nintn,
     +   (intnr(i), intnn(i), i=1,min(nintn,mxint))
      if(nintn.gt.mxint)then
         write(*,*) ' simpget : too many noise interpolation values '
         stop
      endif
      write(*,'(a,1000(/'' %'',2f10.3))')
     +   ' % intnr(i), intnn(i) =',
     +   (intnr(i), intnn(i), i=1,nintn)

      read(10,*) vname, agcvar, sqvar, nthrsh,
     +   (thrsh(i), i=1,min(nthrsh,mxthr))
      if(nthrsh.gt.mxthr)then
         write(*,*) ' simpget : too many agc/squelch test thresholds '
         stop
      endif
      write(*,*) '% agcvar, sqvar =', agcvar, sqvar
      write(*,'(a,1000(/'' %'',f6.2))')
     +   ' % thrsh(i) =', (thrsh(i), i=1,nthrsh)

      read(10,*) vname, empscl
      write(*,*) '% empscl =', empscl

      read(10,*) vname, nempsc,
     +   (empfr(i), empsc(i), i=1,min(nempsc,mxemp))
      if(nempsc.gt.mxemp)then
         write(*,*) ' simpget : too many pre-emphasis scaling values '
         stop
      endif
      write(*,'(a,1000(/'' %'',2f10.3))')
     +   ' % empfr(i), empsc(i) =',
     +   (empfr(i), empsc(i), i=1,nempsc)

      close(10)

c -------------------------------------------------------------------
```

simfde.f

```
c **********************
c simfde.f - fading envelope generator (Jake's method)
c **********************
c 87-7-9

c - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      subroutine genfd(fdbyfs,seed,x,ns,b)

c generate samples of the fading envelope
c method from W. C. Jakes, 1974, p. 65

c output vector (x) is in dB

      real fdbyfs
```

```
      integer ns
      real x(ns)
      double precision seed
      logical init
      integer b, m

c constant to scale envelope to unity rms

      real k

c max number of diversity branches

      integer nb
      parameter (nb=10)

c storage for phase counters for diversity branches

      integer*4 phs(9,nb)

c word and full-word length phase counter

      integer*4 pf(9)
      integer*4 p1, p2, p3, p4, p5, p6, p7, p8, p9
      integer*2 pw(18)
      integer*2 i1, i2, i3, i4, i5, i6, i7, i8, i9

      equivalence (pf,pw)
      equivalence (pf(1), p1), (pf(2), p2), (pf(3), p3),
     +            (pf(4), p4), (pf(5), p5), (pf(6), p6),
     +            (pf(7), p7), (pf(8), p8), (pf(9), p9)

c only one of the following two tables can be used
c using the wrong table will produce out-of-range subscripts

c this table for computers that store MS INTEGER*2 value first
c e.g. MOTOROLA 68000 (SUN)

cSUN  equivalence (pw( 1), i1), (pw( 3), i2), (pw( 5), i3),
cSUN +            (pw( 7), i4), (pw( 9), i5), (pw(11), i6),
cSUN +            (pw(13), i7), (pw(15), i8), (pw(17), i9)

c this table for computers that store LS INTEGER*2 value first
c e.g. INTEL 8088 (IBM PC), DEC VAX-11

cIBM  equivalence (pw( 2), i1), (pw( 4), i2), (pw( 6), i3),
cIBM +            (pw( 8), i4), (pw(10), i5), (pw(12), i6),
cIBM +            (pw(14), i7), (pw(16), i8), (pw(18), i9)

c itl is the table length for each sine/cosine lookup table
c *** values of itl must match ***

      integer itl
      parameter (itl=2048)

c scaled sine and cosine lookup tables

      integer*2 c1(itl), c2(itl), c3(itl), c4(itl), c5(itl),
     +          c6(itl), c7(itl), c8(itl), c9(itl)

      integer*2 s1(itl), s2(itl), s3(itl), s4(itl), s5(itl),
     +          s6(itl), s7(itl), s8(itl), s9(itl)

c I and Q sums

      integer*2 sumi, sumq


c phase increments per sample

      integer*4 dp1, dp2, dp3, dp4, dp5, dp6, dp7, dp8, dp9

      common /fdtab1/ c1, c2, c3, c4, c5, c6, c7, c8, c9,
     +                s1, s2, s3, s4, s5, s6, s7, s8, s9

      common /fdtab2/ dp1, dp2, dp3, dp4, dp5, dp6, dp7, dp8, dp9

      real tpi, oldfds
      integer i, j

      real uni
      real float

      data oldfds/0./, init/.false./

      data tpi/6.28318/

c compute look-up tables if have not been initialized

      if(.not.init)then
         call initos(k)
         init=.true.
      endif

c compute phase increments and initialize oscillator phases
c if doppler or sampling rate have changed

      if(fdbyfs.ne.oldfds)then

         call initph(fdbyfs)
         oldfds=fdbyfs
```

```
c initialize phases using uniform RNG

         m=1
         do 1 i=1,nb
            do 1 j=1,9
               phs(j,i)=itl*65536*uni(seed)
1        continue

      endif

c range-check branch number

      if((b.lt.1).or.(b.gt.nb))then
         write(*,*)'genfd:diversity branch out of range.'
         stop
      endif

c copy branch phases to phase counters

      do 2 i=1,9
         pf(i)=phs(i,b)
2     continue

c generate ns samples

      do 3 i=1,ns

c increment oscillator phases

      p1=p1+dp1
      p2=p2+dp2
      p3=p3+dp3
      p4=p4+dp4
      p5=p5+dp5
      p6=p6+dp6
      p7=p7+dp7
      p8=p8+dp8
      p9=p9+dp9

c keep phase modulo two pi

      if(i1.gt.itl)i1=i1-itl
      if(i2.gt.itl)i2=i2-itl
      if(i3.gt.itl)i3=i3-itl
      if(i4.gt.itl)i4=i4-itl
      if(i5.gt.itl)i5=i5-itl
      if(i6.gt.itl)i6=i6-itl
      if(i7.gt.itl)i7=i7-itl
      if(i8.gt.itl)i8=i8-itl
      if(i9.gt.itl)i9=i9-itl

c add up two sets of scaled oscillator outputs and find mag. squared

      sumi=c1(i1)+c2(i2)+c3(i3)+c4(i4)+
     1      c5(i5)+c6(i6)+c7(i7)+c8(i8)+c9(i9)

      sumq=s1(i1)+s2(i2)+s3(i3)+s4(i4)+
     1      s5(i5)+s6(i6)+s7(i7)+s8(i8)+s9(i9)

      x(i) =  k * ( float(sumi)**2 + float(sumq)**2 )

3     continue

c convert to dB (power) (x is magnitude squared of envelope)

      call vdbp(x,ns)

c copy phase counters back

      do 4 i=1,9
         phs(i,b)=pf(i)
4     continue

      return
      end

c - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      subroutine initos(k)

c initialize oscillator lookup tables

      real pi, tpi, sqrt2
      real N, N0
      real stk, ctk, k
      integer i, j
      real float, cos, sin

c itl is the table length for each sine/cosine lookup table
c *** values of itl must match ***

      integer itl
      parameter (itl=2048)

c scaled sine and cosine lookup tables

      integer*2 cw(itl,9), sw(itl,9)

      real rc(itl)
```

```
      common /fdtab1/ cw, sw

      data pi/3.14159/, tpi/6.28318/, sqrt2/1.414213/

      N0=8.
      N=4.*N0+2.

c compute un-scaled oscillator (cosine) look-up table

      do 1 j=1,itl
         rc(j)=cos(float(j-1)/itl*tpi)
1     continue

c generate scaled oscillator tables

      k=0.

      do 3 i=1,8
         ctk=1750.*2.*cos(pi*float(i)/N0)
         stk=1750.*2.*sin(pi*float(i)/N0)
         k=k+ctk**2+stk**2
         do 2 j=1,itl
            cw(j,i)=rc(j)*ctk
            sw(j,i)=rc(j)*stk
2        continue
3     continue

c last oscillator has different amplitude

      ctk=1750.*1./sqrt2*2.*cos(pi/4.)
      stk=1750.*1./sqrt2*2.*sin(pi/4.)
      k=k+ctk**2+stk**2
      do 4 j=1,itl
         cw(j,9)=rc(j)*ctk
         sw(j,9)=rc(j)*stk
4     continue

c compute normalization constant

      k=2./k

      write(*,*)'% oscillator tables (re)initialized '

      return
      end

c - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      subroutine initph(fdbyfs)

c initialize phase increment table

      real fdbyfs
      real tpi
      real N, N0

      integer i
      real cos, float

c itl is the table length for each sine/cosine lookup table
c *** values of itl must match ***

      integer itl
      parameter (itl=2048)

      integer*4 dpf(9)

      common /fdtab2/ dpf

      data tpi/6.28318/

      if((fdbyfs.lt.0.).or.(fdbyfs.ge.0.5))then
         write(*,*)'genfd:doppler too high or negative.'
         stop
      endif

c compute phase increments

      N0=8.
      N=4.*N0+2.

      do 1 i=1,8
         dpf(i)=cos(tpi*float(i)/N)*fdbyfs*itl*65536.
1     continue
      dpf(9)=                       fdbyfs*itl*65536.

      write(*,*) '% phase increments (re)initialized '
      write(*,*) '% fdbyfs = ',fdbyfs

      return
      end

c - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      subroutine setsnr(fade,n,oldsnr,newsnr)

      integer n
      real fade(n), oldsnr, newsnr
      real delta
```

```
      delta=newsnr-oldsnr

      call vadd(fade,n,delta)
      oldsnr=newsnr

      return
      end
```

simdiv.f

```
c *********************
c simdiv.f - diversity
c *********************
c 87-11-29

c - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      subroutine genfdb(fdbyfs,seed,x,ns,
     +      nb,thr,ndw,nsw)

c fading envelope generator (in dB)

c parameters:

c fdbyfs - Doppler rate / Sampling rate
c seed   - RNG seed for initializing fading generator
c x      - output vector containing fading sequence (dB)
c ns     - Number of valueS in x
c nb     - Number of diversity Branches
c           nb>1 means diversity
c thr    - switching diversity THReshold (dB relative to mean)
c ndw    - minimum Number of samples to DWell on branch
c           before switching
c nsw    - Number of samples lost while SWitching branches

      real fdbyfs
      double precision seed
      integer ns, nb
      real x(ns)
      real thr
      integer ndw, nsw

c local variables:

c i         - index into output vector x
c j         - branch counter
c k         - temporary variable for indexing
c nleft     - Number of samples LEFT to generate
c m         - number of saMples to generate in one pass
c thr       - switching THReshold (fraction of mean=1)
c maxbuf    - size of local buffer

      integer i, j, k, nleft, m
      integer bufsiz, maxbuf
      parameter (maxbuf=1000)

c buffer array (in common work area)

      real buf(maxbuf)
      common buf

c intrinsic functions

      integer min, int

c range-check number of diversity branches

      if(nb.lt.1)then
         write(*,*) ' genfdb: nb < 0 '
         stop
      endif

c initialize for piece-wise generation of ns samples

      i=1
      nleft=ns
      bufsiz=int(maxbuf/nb)

c generate (and combine) "nb" sections of "bufsize" samples per pass

1     continue

      m=min(nleft,bufsiz)

c generate branch signals

      do 2 j=1,nb
         k=(j-1)*m+1
         call genfd(fdbyfs,seed,buf(k),m,j)
2     continue

c combine branches

      if(nb.eq.1)then
         call vcopy(buf,x(i),m)
      else
         call divsw(buf,nb,m,x(i),nsw,ndw,thr)
      endif
```

```
      i=i+m
      nleft=nleft-m

c repeat until done

      if(nleft.gt.0)goto 1

      return
      end

c - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      subroutine divsw(buf,nb,m,x,nsw,ndw,thr)

c switching diversity routine

c NOTE: the switching and dwell counts are static so this
c     routine cannot be used to generate independent
c     diversity-switched signals.

c parameters:

c buf  - input buffer (m by nb) with m samples from each of
c           nb branches
c nb   - number of diversity branches
c m    - number of samples per branch
c x    - output vector containing the resulting fading sequence
c ndw  - minimum Number of samples to DWell on one branch (>=1)
c           (after switching is complete)
c nsw  - Number of samples blanked out while SWitching branches
c thr  - switching THReshold (dB relative to mean)

      integer nb, m, ndw, nsw
      real buf(m,nb), x(m), thr

c local (static!) variables:

c br    - current BRanch
c idw   - count for timer that indicates that must DWell on
c           current branch
c isw   - count for timer for SWitching time
c zval  - VALue value (dB) to indicate samples that should be
c         Zero'ed.  (anything > 1E6 can be used). used by "ch"
c         routine.
c maxdw - maximum dwell count (to avoid overflows)

      integer br, idw, isw, i
      real zval
      parameter(zval=2.0e6)

      data br/1/, idw/0/, isw/0/

c      write(*,*) ' thr = ',thr

c range checks

      if(nsw.lt.0)then
         write(*,*)' divsw: nsw < 0 '
         stop
      endif

      if(br.gt.nb)then
         write(*,*)' divsw: branch  > number of branches '
         stop
      endif

      if(ndw.lt.0)then
         write(*,*)' divsw: ndw < 0 '
         stop
      endif

c do for all samples ...

      do 1 i=1,m

c if below threshold not switching and and dwell timer expired,
c go to next antenna and reset timers

         if(buf(i,br).lt.thr .and. isw.eq.0 .and. idw.eq.0)then
            br=br+1
            if(br.gt.nb)br=1
            isw=nsw
            idw=ndw+nsw
         endif

c if switching (blanking) timer has expired, select current branch's
c level, else output is blanked

         if(isw.eq.0)then
            x(i)=buf(i,br)
         else
            x(i)=zval
         endif

c if still on, decrement switching timer

         if(isw.gt.0)then
            isw=isw-1
         endif
```

```
c if still on, decrement channel dwell timer

         if(idw.gt.0)then
            idw=idw-1
         endif

c        write(*,*) (buf(i,l),l=1,nb), '=> ', br, x(i)

1     continue


      return
      end
```

simgen.f

```
c ********************
c simgen.f - data and noise generators
c ********************

      subroutine prbs(x,n,sr)

c PRBS generator

c The generator's shift register is implemented as a logical array.
c For efficiency, instead of shifting the array to the right (up)
c pointers to the exclusive-or gate connections and the LS bit are
c shifted left (down the array).

c The tap connection (18,23) are taken from the data scrabler
c described in Appendix 2 of the CCITT standard (V.29) for a
c 9600 bps modem.  The pattern period is 2**23-1 = 8 388 607.

      integer len
      parameter (len=23)

c i - counts output values
c i0, i1, i2 - point to least significant bit and XOR connections
c              in shift register array
c n - number of values to put in x
c x - output logical array
c len - length of shift register
c sr - a logical array of len elements corresponding to the
c       elements of a prbs generator shift register.

      integer i, i0, i1, i2, n
      logical sr(len)
      logical x(n)

      i0=1
      i1=18
      i2=23

      do 1 i=1,n

      i0=i0-1
      if(i0.eq.0)i0=len

      if(sr(i1))then
         if(sr(i2))then
            sr(i0)=.false.
            x(i)=.false.
         else
            sr(i0)=.true.
            x(i)=.true.
         endif
      else
         if(sr(i2))then
            sr(i0)=.true.
            x(i)=.true.
         else
            sr(i0)=.false.
            x(i)=.false.
         endif
      endif

      i1=i1-1
      if(i1.eq.0)i1=len

      i2=i2-1
      if(i2.eq.0)i2=len

1     continue

      call rotdwn(sr,len,i0-1)

      return
      end

      subroutine rotdwn(sr,n,i)

c rotate (circular shift) the first "n" elements of logical array
c "sr" down the array by "i" places.

      integer n, i
      logical sr(n)

      integer j, k
```

```
      logical t
      do 2 j=1,i
         t=sr(1)
         do 1 k=1,n-1
            sr(k)=sr(k+1)
1        continue
         sr(n)=t
2     continue

      return
      end


      subroutine vimp(x,n,p,seed)

c generates impulse noise vector
c samples are uncorrelated with equal probability of an impulse = p
c zero mean, unit average power
c calls uniform RNG "uni"

      integer i, n
      real x(n), p, u
      real sqrt, uni
      double precision seed

      if(p.lt.0 .or. p.gt.1)then
         write(*,*)' vimp: p out of range. '
         stop
      endif

      u=sqrt(1./p)

      do 1 i=1,n
         if(uni(seed).lt.p)then
            if(uni(seed).gt.0.5)then
               x(i)=u
            else
               x(i)=-u
            endif
         else
            x(i)=0
         endif
1     continue

      return
      end

      subroutine dwg(x,n,y,m)

c digital waveform generator

c x - output logical vector
c n - number of values in x
c y - pattern definition vector (number false, number true)...
c m - number of elements in m (>=1)

c if pattern counts in y are insufficient to fill x, the pattern is
c repeated starting at y(3)

c i - index into x
c j - index into y
c k - counts up to y(j)

      integer i, j, k, n, m
      logical x(n), tf
      integer y(m)

      if(m.lt.3)then
         write(*,*)'dwg:pattern definition < 3 elements.'
         stop
      endif

      do 1 i=1,y(1)
         x(i)=.false.
1     continue

      do 2 i=y(1)+1,y(1)+y(2)
         x(i)=.true.
2     continue

      tf=.false.
      j=3
      k=0

      do 3 i=y(1)+y(2)+1,n
         if(k.ge.y(j))then
            tf=.not.tf
            j=j+1
            k=0
            if(j.gt.m)then
               j=3
            endif
         endif
         x(i)=tf
         k=k+1
3     continue

      return
      end
```

```
      subroutine vgrand(xo,n,seed)
      integer n
      real xo(n)

      integer i
      real r, rx, ry

      double precision seed, dmod
      real sqrt, alog

c function that returns a Gaussian distributed random number
c of zero mean, unity variance.
c adapted from IEEE DSP program library (in MXFFT.FOR)

      do 1 i=1,n,2

10    continue

      seed=dmod(67081293.0d0*seed+14181771.0d0,67108864.0d0)
      RX=(seed/67108864.0d0*2.0) - 1.0

      seed=dmod(67081293.0d0*seed+14181771.0d0,67108864.0d0)
      RY=(seed/67108864.0d0*2.0) - 1.0

C
C-----------------------------------------------------------------------
C SUBROUTINE:  NORMAL
C GENERATES AN INDEPENDENT PAIR OF RANDOM NORMAL DEVIATES
C METHOD DUE TO G. MARSAGLIA AND T.A. BRAY,
C SIAM REVIEW, VOL. 6, NO. 3, JULY 1964. 260-264
C-----------------------------------------------------------------------
C
C OUTPUT:  X,Y = INDEPENDENT PAIR OF RANDOM NORMAL DEVIATES
C FUNCTION UNI GENERATES PSEUDO-RANDOM NUMBER BETWEEN 0.0 AND 1.0
C
      R = RX**2 + RY**2
      IF (R.GE.1.0) GO TO 10
      R = SQRT(-2.0*ALOG(R)/R)

      xo(i)=RX*R
      if(i+1.le.n)then
         xo(i+1)=RY*R
      endif

1     continue

      return
      end

      real function uni(seed)

c modulo congruential uniform RNG on [0,1)
c parameters are from FPS AP library routine VRAND

      double precision seed
      double precision dmod

      seed=dmod(67081293.0d0*seed+14181771.0d0,67108864.0d0)
      uni=seed/67108864.

      return
      end

      double precision function iuni(seed)

c integer version of uni: RNG on [0,67108864)
c to check integer operations using d.p. f.p. numbers
c values taken from FPS AP library routine VRAND

      double precision seed
      double precision dmod

      seed=dmod(67081293.0d0*seed+14181771.0d0,67108864.0d0)
      iuni=seed

      return
      end
```

simmod.f

```
c ******************
c simmod.f - OFDM [de]modulation
c ******************

      subroutine encode(d,sig,if1,if2,n)

c encode logical data into data values

c d    - logical data
c sig  - generated data values, unit variance
c if1  - index of lowest frequency
c if2  - index of highest frequency
c n    - total number of values

      integer if1, if2, i, j, n
      logical d(n)
      real sig(n)

c zero out low frequency terms
```

```
      do 1 i=1,if1-1
         sig(i)=0.
1     continue

c encode data into QAM (complex) format

      j=1
      do 2 i=if1,if2
         if(d(j))then
            sig(i)=1.
         else
            sig(i)=-1.
         endif
         j=j+1
2     continue

c zero out high-frequency terms

      do 3 i=if2+1,n
         sig(i)=0.
3     continue

      return
      end

      subroutine decode(d,sig,if1,if2,n)

c recover logical data from signal values
c data not in [if1,if2] is unchanged

      integer n, if1, if2, i, j
      logical d(n)
      real sig(n)

      j=1
      do 1 i=if1,if2
         if(sig(i).gt.0)then
            d(j)=.true.
         else
            d(j)=.false.
         endif
         j=j+1
1     continue

      return
      end

      subroutine modu(sig,n,nf,serial)

c OFDM modulation

c sig    - generated signal samples, unit variance
c n      - OFDM block size = number of samples generated
c nf     - number of data values = +/- 1 (rest assumed equal 0)
c serial - true for serial modulation (no FFTs)

      integer n, nf
      logical serial
      real sig(n)
      real k
      real sqrt, float
c     real rms

c if OFDM (not serial) modulate the data values into an OFDM signal
c with unity power (variance = std. dev. = 1 V**2)

      if(.not.serial)then
         call ffsn(sig,n)
         k=sqrt( float(n)/nf ) * sqrt( float(n)/2.)
         call vsmul(sig,n,k)
c         write(*,*) ' modu: output rms signal level = ',rmsv(sig,n)
      endif

      return
      end

      subroutine demodu(sig,n,nf,serial)

c OFDM demodulation

c sig    - generated signal samples, unit variance
c n      - OFDM block size = number of samples generated
c nf     - number of data values = +/- 1 (rest assumed equal 0)
c serial - true for serial modulation (no FFTs)

      integer n, nf
      logical serial
      real sig(n)
      real k
      real sqrt, float
c     real rms

c if OFDM, un-scale values and demodulate the ofdm signal samples
c back into data values.

      if(.not.serial)then
c         write(*,*) ' demodu: input rms signal level = ',rmsv(sig,n)
         k=sqrt( float(n)/nf ) * sqrt( float(n)/2.)
         call vsmul(sig,n,1./k)
```

```
      call ffan(sig,n)
   endif

   return
   end
```

simsnr.f

```
c ********************
c simsnr.f - IF/AF SNR conversion
c ********************
c modified 89-3-9 to do agc and squelch after corrections.
c

      subroutine s2init(b,w,rms,peak,fm,fd,rfm,agclim,sqlim,
     +         nints,intsr,intss, nintn,intnr,intnn,
     +         fading, noise )

c initialize IF SNR -to- AF SNR conversion tables

c input parameters:

c fm     true for an FM channel false for SSB
c b      IF bandwidth (Hz or kHz)
c w      baseband bandwidth (same units as "b")
c rms    rms level of the modulating signal
c peak   peak level of the modulating signal
c           peak-to-average ratio = (10 for voice, about 2-3 for data)
c pk2rms peak/rms
c fd     doppler rate (same units as "b" and "w")
c rfm    true to add random fm noise
c agclim SNR above which the AGC operates (dB)
c sqlim  SNR below which the receiver output is squelched (dB)
c        (applied after interpolation given by intXX)
c nintn  - number of elements in intnr and intnn
c nints  - number of elements in intsr and intss
c intnr  - SNR part of points to use to change r-n table
c intnn  - "n" part of points to use to change r-n table
c intsr  - SNR part of points to use to change r-s table
c intss  - "s" part of points to use to change r-s table
c fading - false to disable fading (constant signal output)
c noise  - false to disable noise (no noise added)
c mxint  - number of elements in intnr, intnn, intsr, intsn (not req'd)
c
c the arrays snr2s and snr2n contain the SNR-to-signal level
c and SNR-to-noise level lookup tables. the look-up function
c is: s2min+1 + ifix(SNR)*npdb
c
c npdb   is number of table entries per dB of SNR change
c s2min  is "npdb" times the minimum SNR value in the table
c s2max  is "npdb" times the maximum SNR value in the table

      integer mxint
      parameter (mxint=100)

      real b, w
      logical fm, rfm
      real rms, peak, pk2rms
      real agclim, sqlim
      integer nintn, nints
      real intnr(mxint), intnn(mxint), intsr(mxint), intss(mxint)

      integer i, j, i1, i2
      real pi, n, a, r, s2asum, e2mr, nk, sk, c, nrfm, fd

      integer npdb, s2min, s2max
      real snr2s(1101), snr2n(1101)
      integer ix
      logical fading, noise
      real alog, float, exp, sqrt
      real interp
      real dbtor, x

      common /s2com/ npdb, s2min, s2max, snr2s, snr2n

      data pi /3.14159/

c function to address 1-base arrays with negative indexes

      ix(i)=i-s2min+1

c function to convert db to linear (voltage) units

      dbtor(x)=10.**(x/20.)

      npdb=10
      s2min=-500
      s2max=600

      write(*,*)'%   FM, IF & AF bandwiths = ', fm, b, w
      write(*,*)'%     rms, peak deviation = ', rms, peak
      write(*,*)'% random FM, doppler rate = ', rfm, fd
      write(*,*)'%    AGC & squelch limits = ', agclim, sqlim
      write(*,*)'%           fading, noise = ', fading, noise

c check number points in interpolation tables

      if( (nints.lt.0) .or. (nints.eq.1) .or.
     +      (mod(nints,2).ne.0) ) then
```

```
         write(*,*) ' % s2init - warning : bad nints '
      endif

      if( (nintn.lt.0) .or. (nintn.eq.1) .or.
     +      (mod(nintn,2).ne.0) ) then
         write(*,*) ' % s2init - warning : bad nintn '
      endif

c check for random FM being in valid approximation region

      if(rfm.and.(fd.gt.0.1*w))then
         write(*,*) ' s2init : doppler > .1 AF B/W '
         stop
      endif

      if(rms.le.0.)then
         write(*,*) ' rms <= 0. '
         stop
      endif

      pk2rms = peak/rms

      if((pk2rms).lt.1.)then
         write(*,*) ' s2init : (pk/rms) < 1 '
         stop
      endif

      if(fm.and.(w.ge.b/2.0))then
         write(*,*) ' % s2init : warning : AF B/W >= 1/2 IF B/W '
      endif

c modified to give snr2s=1 maximum (for FM)

      if(fm)then
         a=s2asum(b,w)
         nk=8.*pi*b*w
         sk=pi**2/(pk2rms**2)*(b-2.*w)**2
         if(rfm)then
            nrfm=2.*pi**2*alog(10.)*fd**2
         else
            nrfm=0.
         endif
         do 1 i=s2min,s2max
               r=10.0**(float(i)/(npdb*10))
               e2mr=exp(-r)

               n=a*(1.0-e2mr)**2/r + nk*e2mr/sqrt(2*(r+2.35)) + nrfm

c ( sk factor scales snr2n to make snr2s independent of b & w )
               snr2n(ix(i))=sqrt(n/sk)
               snr2s(ix(i))=(1.-e2mr)

1        continue
      else
         do 2 i=s2min,s2max
            snr2s(ix(i))=10.**(float(i)/(npdb*20))
            snr2n(ix(i))=1.
2        continue
      endif

c apply corrections to conversion tables using linear interpolation

c algorithm -
c for each pair of points :
c    find indices in snr table for this pair
c    for all table entries between these points :
c       if the point is in the table :
c          interpolate and substitute new table value

c corrections to snr2s

      do 9 i=1,nints-1,2
c         write (*,'(a,2f10.3,a,2f10.3)') ' % snr2s corr''n : ',
c     +      intsr(i),intss(i),' to ',intsr(i+1),intss(i+1)
         i1=intsr(i)*npdb
         i2=intsr(i+1)*npdb
         do 8 j=i1,i2
            if( (j.ge.s2min) .and. (j.le.s2max) )then
               snr2s(ix(j))=dbtor(
     +            interp( intsr(i),intsr(i+1),intss(i),intss(i+1),
     +            float(j)/npdb ) )
            endif
8        continue
9     continue

c corrections to snr2n

      do 11 i=1,nintn-1,2
c         write (*,'(a,2f10.3,a,2f10.3)') ' % snr2n corr''n : ',
c     +      intnr(i),intnn(i),' to ',intnr(i+1),intnn(i+1)
         i1=intnr(i)*npdb
         i2=intnr(i+1)*npdb
         do 10 j=i1,i2
            if( (j.ge.s2min) .and. (j.le.s2max) )then
               snr2n(ix(j))=dbtor(
     +            interp( intnr(i),intnr(i+1),intnn(i),intnn(i+1),
     +            float(j)/npdb ) )
            endif
10       continue
```

```
11    continue
c apply AGC for snrs from agclim up

      j=agclim*npdb
      if(j.lt.s2min)j=s2min
      if(j.gt.s2max)j=s2max
      c=1./snr2s(ix(j))
      do 3 i=j,s2max
         snr2n(ix(i))=snr2n(ix(i))/snr2s(ix(i))
         snr2s(ix(i))=1.0
3     continue

c scale transfer curves below agc limit to make them continuous

      do 4 i=s2min,j-1
         snr2s(ix(i))=snr2s(ix(i))*c
         snr2n(ix(i))=snr2n(ix(i))*c
4     continue

c squelch for snrs below sqlim

      j=sqlim*npdb
      if(j.gt.s2max)j=s2max
      do 5 i=s2min,j
         snr2n(ix(i))=0.
         snr2s(ix(i))=0.
5     continue

c if no fading

      if(.not.fading)then
         call vfill(snr2s, s2max-s2min+1, 1.)
      endif

c if no noise

      if(.not.noise)then
         call vfill(snr2n, s2max-s2min+1 ,0.)
      endif

      return
      end

      real function s2asum(b,w)

c function to calculate a sum for calculating noise power
c for FM, see Jakes, Ch. 4.

      real w, b, s, pi, t, t1
      parameter (pi=3.14159)

      integer n, nmax
      parameter (nmax=12)

      t1=(-pi)*(w/b)**2
      t=1.
      s=0.

      do 1 n=0,nmax
         s=s+t/(n+n+3)
         t=t*t1/(n+1)
1     continue

      s2asum=s*4.*pi*pi*w*w*w/b

      return
      end

      subroutine ch(sin,sout,noisev,fadev,ns,rms,peak)

c channel simulation

c sin/sout  - input/output signal samples
c noisev    - additive noise samples, unit variance
c fadev     - signal snr levels (in dB snr)
c           - any value > 1E6 (dB) indicates a blanked sample
c rms    rms level of the modulating signal
c peak   peak level of the modulating signal

      integer ns
      real sin(ns), sout(ns), noisev(ns), fadev(ns), rms, peak

      integer i, j
      integer ifix

      integer npdb, s2min, s2max
      real snr2s(1101), snr2n(1101)
      real s, smax, smin
      real rmsv

      common /s2com/ npdb, s2min, s2max, snr2s, snr2n

c function to address 1-base arrays with negative indexes

      integer ix
      ix(i)=i-s2min+1

c      write(*,*) ' ch:sin:rms: ', rmsv(sin,ns)
c      write(*,*) ' ch:noisev:rms: ', rmsv(noisev,ns)
```

```
c compute maximum and minimum signal levels
      smax =  peak/rms
      smin = -peak/rms

c multiply by fading and add noise

      do 1 i=1,ns

c check for blanking

         if(fadev(i).ge.1.0e6)then

            sout(i)=0.0

         else

            j=ifix(fadev(i)*npdb)
            if(j.lt.s2min)j=s2min
            if(j.gt.s2max)j=s2max
            j=ix(j)

c clip signal (at transmitter - before fading)

            s = sin(i)
            if(s.gt.smax) s = smax
            if(s.lt.smin) s = smin

            sout(i) = s*snr2s(j) + noisev(i)*snr2n(j)

         endif

1     continue

      return
      end

      subroutine r2sns(r,s,n,ns,snr)

c convert signal envelope input vector to s and n vectors
c used for numerical integration routine

c r - received signal envelope level (dB) (0 dB mean)
c s - corresponding signal scale value (linear)
c n - corresponding noise scale value (linear)
c ns - number of samples in r, s, n
c snr - average snr

      integer ns
      real r(ns), s(ns), n(ns)
      real snr

      integer i, j
      integer ifix

      integer npdb, s2min, s2max
      real snr2s(1101), snr2n(1101)

      common /s2com/ npdb, s2min, s2max, snr2s, snr2n

c function to index 1-base arrays with negative indices

      integer ix
      ix(i)=i-s2min+1

      do 1 i=1,ns
         j=ifix( ( r(i)+snr )*npdb )
         if(j.lt.s2min)j=s2min
         if(j.gt.s2max)j=s2max
         j=ix(j)
         s(i)=snr2s(j)
         n(i)=snr2n(j)

c         write(*,*) ' r2j: ', j

1     continue

      return
      end

      real function interp(x1,x2,y1,y2,x)

c linear interpolation for value x using line between x1,y1 and x2,y2
c for x1=x2, returns y2

      real x1, x2, y1, y2, x

      if(x2.ne.x1)then
         interp = y1 + (y2-y1)/(x2-x1) * (x-x1)
      else
         interp = y2
      endif

      return
      end
```

simfec.f

```
c *****************
c simfec.f - test and/or correct a block of data
c *****************

      subroutine fec(data,okdata,n,intlv,ecn,ect,ne,nbx,nwe,nwx)

c number of remaining bit errors is returned in ne
c bits that don't fit into FEC blocks are ignored for BkER counts
c     and are not processed for FEC correction
c no bits corrected if number of errors in a block > ect
c all bits corrected if number of errors in block <= ect
c     (this is an idealized FEC code)
c for efficiency make ecn as large as possible

c data   - input/output data
c okdata - correct (transmitted) data
c n      - number of values in data, okdata
c intlv  - true to do interleaving
c ecn    - FEC block size
c ect    - maximum number of correctable errors (0 for no FEC)
c ne     - number of bit errors remaining
c nbx    - number of bits examined
c nwe    - number of word errors remaining
c nwx    - number of words examined

      integer n
      logical data(n), okdata(n)
      integer ne, nwe
      integer nbx, nwx
      logical intlv
      integer ecn, ect

      integer i, k
      integer nbe
      integer mod

      if(ecn.le.0)then
         write(*,*) ' % fec : ecn <= 0 '
         stop
      endif

      if(ect.lt.0 .or. ect.gt.ecn)then
         write(*,*) ' fec : correctable errors (ect) out of range '
         stop
      endif

c interleave the received data and the correct (transmitted) data

      if(intlv)then
         call scr(data  ,n,.true.)
         call scr(okdata,n,.true.)
      endif

c initialize number of bit errors and bits examined

      ne=0
      nbx=0

c count (and optionally correct) word errors

      nwe=0
      nwx=0

      i=1
1     continue
      if(i+ecn-1.gt.n)goto 2

c find number of errors in this word

         k=nbe(data(i),okdata(i),ecn)

         ne=ne+k
         nbx=nbx+ecn

c if any errors
c if they are correctable, correct them
c else increment word error count

         if(k.eq.0)then
         elseif(k.le.ect)then
            call vlcopy(okdata(i),data(i),ecn)
         else
            nwe=nwe+1
         endif
         nwx=nwx+1

         i=i+ecn
         goto 1
2     continue

c count remaining bit errors

      if(i.le.n)then
         ne=ne+nbe(data(i),okdata(i),n-i+1)
         nbx=nbx + n-i+1
      endif
```

```
c un-interleave data and correct data
      if(intlv)then
         call scr(data,  n,.false.)
         call scr(okdata,n,.false.)
      endif

      return
      end

      integer function nbe(data,okdata,n)

c return number of differences between data and okdata

c data   - input/output data
c okdata - correct data
c n      - number of values in data, okdata (FEC block size)
c ne     - local error counter

      integer i, n, ne
      logical data(n), okdata(n)

      ne=0

      do 1 i=1,n
         if(data(i).neqv.okdata(i))ne=ne+1
1     continue

      nbe=ne

      return
      end

      subroutine scr(in,n,fwd)

c interleave (scramble) n in(put) data bits
c interleaving factor is sqrt(n)
c in direction fwd

      integer i, j, k, l, n
      logical in(n)
      logical fwd

      integer int
      real sqrt, float

c common work vector

      logical out(4096)
      common out
      k=int(sqrt(float(n))+0.5)

      l=1
      do 1 i=1,k
         do 1 j=i,n,k
            if(fwd)then
               out(l)=in(j)
            else
               out(j)=in(l)
            endif
            l=l+1
1        continue

      do 2 i=1,n
2        in(i)=out(i)

      return
      end

      subroutine dne(data,okdata,n,intlv,ecn,nofdm,snr)

c display number of remaining bit errors

c data   - input/output data
c okdata - correct (transmitted) data
c n      - number of values in data, okdata
c intlv  - true to do interleaving
c ecn    - FEC block size
c nofdm  - OFDM block size (for printing only)
c snr    - RF SNR (for printing only)

      integer n
      logical data(n), okdata(n)
      logical intlv
      integer ecn
      integer nofdm
      real snr

      integer i, j, k
      integer nwoerr
      integer nbe
      integer mod

      if(ecn.le.0)then
         write(*,*) ’ % fec : ecn <= 0 ’
         stop
      endif

c interleave the received data and the correct (transmitted) data
```

```
      if(intlv)then
         call scr(data  ,n,.true.)
         call scr(okdata,n,.true.)
      endif

      i=1
1     continue
      if(i+ecn-1.gt.n)goto 2

c find number of errors in this word

         k=nbe(data(i),okdata(i),ecn)

c print a flag, block size, snr, number of errors in word

         write(*,'(1X,A2,I5,F4.0,I5)') '%N', nofdm, snr, k

         i=i+ecn
         goto 1
2     continue

c compute and print error-free run lengths for the block

         nwoerr=0
         do 3 i=1,n
            if(data(i).neqv.okdata(i))then
            call dnew(nwoerr)
            nwoerr=0
         else
            nwoerr=nwoerr+1
         endif
3        continue
         call dnew(nwoerr)

c terminate the block with a -1

         call dnew(-1)

c un-interleave data and correct data

      if(intlv)then
         call scr(data,  n,.false.)
         call scr(okdata,n,.false.)
      endif

      return
      end

      subroutine dnew(nwoerr)
      integer nwoerr
      write(*,'(1X,A2,I5)') '%R', nwoerr
      return
      end
```

simut.f

```
c ****************
c simut.f - Simulation Utility Routines
c ****************

      real function vsum(x,n)

      integer n
      real x(n)
      integer i
      real sum

      sum=0.

      do 1 i=1,n
         sum=sum+x(i)
1     continue

      vsum=sum

      return
      end

      subroutine vmul(a,b,c,n)

      integer n
      real a(n), b(n), c(n)
      integer i

      do 1 i=1,n
         c(i)=a(i)*b(i)
1     continue

      return
      end

      subroutine vlcopy(in,out,n)

c copy in into out

      integer i, n
      logical in(n), out(n)
```

```
      do 1 i=1,n
         out(i)=in(i)
1     continue

      return
      end

      subroutine vxor(in1,in2,out,n)

c exclusive-or of two logical vectors

      integer i, n
      logical in1(n), in2(n), out(n)

      do 1 i=1,n
         out(i)=in1(i).neqv.in2(i)
1     continue

      return
      end

      subroutine vdbp(x,n)

c convert vector x to dB (power)

      integer i, n
      real x(n)
      real alog10
      do 1 i=1,n
         if(x(i).le.0.)then
            write(*,*) ' vdbp: argument <= 0 result set to -100 '
            x(i)=-100.
         else
            x(i)=10.*alog10(x(i))
         endif
1     continue
      return
      end

      real function rmsv(x,n)

c root mean square of a vector

      integer n
      real x(n)
      real sqrt, ssq

      if(n.le.0)then
         write(*,*) ' rms : n <= 0 '
         stop
      endif

      rmsv=sqrt(ssq(x,n)/n)
      return
      end

      real function ssq(x,n)

c sum of squares of elements of a vector

      integer n
      real x(n), p
      integer i
      p=0.
      do 1 i=1,n
      p=p+x(i)**2
1     continue
      ssq=p
      return
      end

      subroutine vsq(x,y,n)

c vector square y=x**2 for all n elements

      integer i, n
      real x(n), y(n)

      do 1 i=1,n
1     y(i)=x(i)**2

      return
      end

      subroutine vquant(x,n,max,k)

c quantize all n elements of x to k bits.
c if k = 0 no quantization is done
c values are assumed to lie between +/- max
c the range +/- max is divided into 2**k equal regions
c all values within a region are converted to the mean of the region

      integer n, k
      real x(n), max
      real c1, c2
      integer i
      integer nint

      if(k.ne.0)then
```

```
        if(k.lt.0)then
           write(*,*) ' vquant : number of bits < 0 '
           stop
        endif

        if(max.le.0)then
           write(*,*) ' vquant : max < or = 0 '
           stop
        endif

        c1=2**(k-1)/max
        c2=1./c1

        do 1 i=1,n
           if(x(i).gt.max)then
              x(i)=max
           elseif(x(i).lt.-max)then
              x(i)=-max
           else
              x(i) = c2 * ( nint( x(i)*c1 +0.5 ) - 0.5)
           endif
1       continue

      endif

      return
      end

      subroutine vifill(k,n,v)

c fill all n elements of integer vector k with value v

      integer i, n, k(n), v
      do 1 i=1,n
1     k(i)=v
      return
      end

      subroutine vfill(x,n,v)

c fill all n elements of real vector k with value v

      integer i, n
      real x(n), v
      do 1 i=1,n
1     x(i)=v
      return
      end
```

```
      subroutine vcopy(x1,x2,ns)

      integer ns
      real x1(ns),x2(ns)
      integer i

      do 1 i=1,ns
         x2(i)=x1(i)
1     continue

      return
      end

      subroutine vadd(x,n,a)

c add a to each element of x

      integer n
      real x(n), a
      integer i

      do 1 i=1,n
         x(i)=x(i)+a
1     continue

      return
      end

      subroutine vsmul(x,n,a)

c multiply each element of x by a

      integer n
      real x(n), a
      integer i

      do 1 i=1,n
         x(i)=x(i)*a
1     continue

      return
      end

c statistics routines:

c initialized statistics variables

      subroutine stinit(x,x2,n)

c x   - sum of a's
```

```
c x2  - sum of a**2's
c n   - number of observations

      real x, x2
      integer n

      x=0.
      x2=0.
      n=0

      return
      end

c update statistics variables

      subroutine stat(a,x,x2,n)

c a   - observed value
c x   - sum of a's
c x2  - sum of a**2's
c n   - number of observations

      real a, x, x2
      integer n

      x=x+a
      x2=x2+a**2
      n=n+1

      return
      end

c mean

      real function stmn(x,x2,n)

      real x, x2
      integer n

      if(n.le.0)then
         write(*,*) ' stmn : n <= 0 '
         stop
      endif

c add 0 * x2 to avoid compiler warnings

      stmn=x/n + 0.0*x2

      return
      end

c sample variance

      real function stvr(x,x2,n)

      real x, x2
      integer n

      if(n.le.1)then
         stvr=0.
      else
         stvr=x2/(n-1)-x**2/(n*(n-1))
      endif

      return
      end

c lower .95 CI

      real function stl95(x,x2,n)

      real stmn, stvr, t95, sqrt
      real x, x2, t
      integer n

      if(n.le.0)then
         write(*,*) ' stl95 : n <= 0 '
         stop
      endif

      t=stvr(x,x2,n)/n
      if(t.lt.0)then
         write(*,*)' stl95: negative variance, set to 0.'
         t=0.
      endif
      stl95=stmn(x,x2,n)-t95(n)*sqrt(t)

      return
      end

c upper .95 CI

      real function stu95(x,x2,n)

      real stmn, stvr, t95, sqrt
      real x, x2, t
      integer n

      if(n.le.0)then
         write(*,*) ' stu95 : n <= 0 '
```

```
         stop
      endif

      t=stvr(x,x2,n)/n
      if(t.lt.0)then
         write(*,*)' stu95: negative variance, set to 0.'
         t=0.
      endif
      stu95=stmn(x,x2,n)+t95(n)*sqrt(t)

      return
      end

      real function t95(n)

c t-table for 0.95 confidence interval
c n is number of trials (degrees of freedom plus 1)
c alpha = 0.025
c (rounded to 3 sig. digits)

      integer n
      real ttab(30)
      data ttab/ 0., 12.70, 4.30, 3.18, 2.78,
     1 2.57, 2.45, 2.37, 2.31, 2.26,
     2 2.23, 2.20, 2.18, 2.16, 2.15,
     3 2.13, 2.12, 2.11, 2.10, 2.09,
     4 2.08, 2.07, 2.07, 2.06, 2.06,
     5 2.06, 2.05, 2.05, 2.05, 2.04 /

      if(n.lt.1)then
         write(*,*)'t95: too few trials : ',n
      elseif(n.le.30)then
         t95=ttab(n)
      elseif(n.le.40)then
         t95=2.03
      elseif(n.le.60)then
         t95=2.01
      else
         t95=2.0
      endif
      return
      end

c single precision complementary error function

c Ed Casas - UBC Electrical Engineering
c Feb. 21, 1986

c ref.: W. J. Cody, "Rational Chebyshev Approximation for the
c Error Function," Mathematics of Computation, 23(107), pp. 631-638,
c 1969.

c these functions should be accurate to the limit of single
c precision operations.

      real function erfc(x0)

      logical neg
      real x0, x, x2, x3, x4, y
      real abs, erf, exp

      if(x0.lt.0.)then
         neg=.true.
      else
         neg=.false.
      endif

      x=abs(x0)

      if(x.le.0.5)then
c evaluate indirectly

      y=1.0 - erf(x)

      elseif(x.ge.4.)then
c approximation 3

      x2=1.0/(x*x)
      x4=x2*x2
      y = exp(-x*x)/x * ( 0.5641896 + x2 *
     1  ( -4.257996e-2   -1.960690e-1*x2   -5.168823e-2*x4 ) /
     2  (  1.509421e-1 +  9.214524e-1*x2 +  1.000000e00*x4 ) )

      else
c approximation 2

      x2=x *x
      x3=x2*x
      x4=x3*x

      y = exp(-x2) *
     1 ( 7.373888e00    + 6.865018e00*x  + 3.031799e00*x2
     2                  + 5.631696e-1*x3 + 4.318779e-5*x4 ) /
     3 ( 7.373961e00    + 1.518491e01*x  + 1.279553e01*x2
     4                  + 5.354217e00*x3 + 1.000000e00*x4 )

      endif

      if(neg)then
         erfc=2.0-y
```

```
      else
         erfc=y
      endif

      return
      end

c single precision error function

      real function erf(x0)

      logical neg
      real x0, x, x2, x4, y
      real abs, erfc

      if(x0.lt.0.)then
         neg=.true.
      else
         neg=.false.
      endif

      x=abs(x0)

      if(x.le.0.5)then
c approximation 1

      x2=x*x
      x4=x2*x2
      y = x * ( 2.138533e1 + 1.722276e0*x2 + 3.166529e-1*x4 ) /
     1        ( 1.895226e1 + 7.843746e0*x2 + 1.000000e00*x4 )

      else
c evaluate indirectly

      y=1.0-erfc(x)

      endif

      if(neg)then
         erf=-y
      else
         erf=y
      endif

      return
      end

      subroutine vsel(a,b,x,z,n)

c substitute b into a for x > z

      integer n
      real a(n), b(n), x(n), z
      integer i

      do 1 i=1,n
         if(x(i).gt.z)then
            a(i)=b(i)
         endif
1     continue

      return
      end

      real function berblk(a,b,c)

c theoretical BER of an OFDM block

      real a, b, c, d
      real erfc, sqrt

      d=2.0*(b-a**2+c)

      if(d.lt.-0.0001)then
         write(*,*)'berblk: b-a**2+c < 0 : ',d/2.0
         stop
      endif

c the case where a and d are close to zero is unstable

      if(d.le.0.)then
         if (a.gt.0) then
            berblk=0.0
         else
            berblk=0.5
         endif
      else
         berblk=0.5*erfc( a/sqrt( d ) )
      endif

      return
      end

      double precision function dray(x,n)

c Rayleigh CPDF : prob. that signal is x dB below mean
c with n ideal selection diversity branches

      integer n
      real x
```

```
      double precision dexp
      dray = ( 1.0d0 - dexp( -1.0d0*10.0**(x/10.0) )  )**n

      return
      end

      subroutine setif(fs,n,f1,if1,f2,if2)

c (two real values per frequency, first pair has DC and fs components)

      integer n
      real fs, f1, f2
      integer if1, if2
      integer mod
      real float

      if(fs.le.0.)then
         write(*,*) ' setif : fs <= 0 '
         stop
      endif

c find array index corresponding to frequency f1

      if1=2.*f1/fs*float(n)+1.

c round if1 up to an odd number so that it points to the
c real element of the first complex number
c e.g. 1->1, 2->3

      if1=if1-mod(if1,2)+1

c find array index corresponding to frequency f2

      if2=2.*f2/fs*float(n)+1.

c round if2 down to an even number so that it points to the
c imaginary element of the last complex number
c e.g. 2048->2048, 2047->2046

      if2=if2-mod(if2,2)

      if(if1.le.0)then
         if1=1
      endif

      if(if2.ge.n)then
         if2=n
      endif

      if(if1.gt.n)then
         write(*,*) ' setif: f1 > N '
         stop
      endif

      if(if2.lt.0)then
         write(*,*) ' setif: f2 < 0 '
         stop
      endif

      if(if1.gt.if2)then
         write(*,*) ' setif: f1 > f2 '
         stop
      endif

      return
      end

c count number of 0's, 1's and runs

      subroutine runcnt(data,n,n0,n1,nr,r)

      integer n, n0, n1, nr
      integer i
      real tn0n1, avgr, varr, r
      logical data(n), prev

c test for n > 1

      if(n.le.1)then
         write(*,*) ' runcnt: n < 2 . '
         stop
      endif

c initialize

      if(data(1))then
         n0=0
         n1=1
      else
         n0=1
         n1=0
      endif
      nr=1
      prev=data(1)

c go through data and count up true, false, and changes

      do 1 i=2,n
         if(data(i))then
            n1=n1+1
```

```
            if(.not.prev)then
               nr=nr+1
               prev=.true.
            endif
         else
            n0=n0+1
            if(prev)then
               nr=nr+1
               prev=.false.
            endif
         endif
1     continue

c compute normalized r.v. (hopefully distributed n(0,1))

      if(n0.eq.0 .or. n1.eq.0)then
         r=0.
      else
         tn0n1 = 2.0 * n0 * n1
         avgr = 1.0 + tn0n1/float(n)
         varr = tn0n1*(tn0n1-float(n))/(float(n)*float(n)*float(n-1))
         r= (float(nr) - avgr) / sqrt(varr)
      endif

      return
      end

      real function norct(data,n)

c generate normalized run count (easier to call than runcnt)

      integer n, i1, i2, i3
      logical data
      real r

      call runcnt(data,n,i1,i2,i3,r)
      norct=r

      return
      end

      subroutine vl2diff(in1,in2,nin,out,nout)

c compare dibits (2-bit sequences) and generate a logical vector
c map of dibit differences (.true.=difference)

      integer nin, nout
      logical in1(nin), in2(nin), out(nout)
      integer i, j

      if(2*nout.ne.nin)then
         write(*,*) ' vl2diff : nin <> 2*nout. '
         stop
      endif

      j=1
      do 1 i=1,nin,2
         out(j) = ( in1(i) .eqv. in2(i) )
     +      .and. ( in1(i+1) .eqv. in2(i+1) )
      j=j+1
1     continue

      return
      end

      real function dbp(x)

c convert a power to dB

      real x

      if(x.gt.0.)then
         dbp=10.*alog10(x)
      else
         write(*,*) ' dbp : db(x) for x<0 set to -99 '
         dbp=-99.
      endif

      return
      end

      subroutine vddat(crsig,txdata,nf,f1,f2,n,snr)

      integer nf, n
      real crsig(nf)
      logical txdata(nf)
      real f1, f2, snr
      integer i
      real f, txmag, rxmag, txan, rxan
      real atan2
      integer mod
      real a, b

      if(mod(nf,2).ne.0)then
         write(*,*) ' vddat : nf not even. '
         stop
      endif

      do 1 i=1,nf,2
```

```
         txmag=1.4142
         if(txdata(i))then
            if(txdata(i+1))then
               txan=45.
            else
               txan=-45.
            endif
         else
            if(txdata(i+1))then
               txan=135.
            else
               txan=-135.
            endif
         endif

         rxmag=sqrt(crsig(i)**2+crsig(i+1)**2)
         rxan=atan2(crsig(i+1),crsig(i))*57.3

         f=f1+i*(f2-f1)/float(nf)

         if (txdata(i)) then
            a=1.0
         else
            a=-1.0
         endif

         if (txdata(i+1)) then
            b=1.0
         else
            b=-1.0
         endif

         write(*,'(1X,A2,2F10.4)') '%D', a, crsig(i)
         write(*,'(1X,A2,2F10.4)') '%D', b, crsig(i+1)

         write(*,'(1X,A2,I5,F4.0,F6.0,2(F4.1,F6.0))')
     +      '%A', n, snr, f, txmag, txan, rxmag, rxan

1     continue

      return
      end

      subroutine dvsn(x,data,nx,f1,fs,n,snr,nav,svec,nvec)

c x    - vector of received data values
c data - transmitted logical data values
c nx   - number of elements in x and data
c nav  - number of snr measurements (of x) to average
c n    - OFDM block size (to print)
c snr  - RF SNR (to print)

      integer n, nx, nav
      logical data(nx)
      real x(nx), svec(nx), nvec(nx)
      real fs, f1, snr

      real f, df
      integer i, nsn
      real dbp, float

      call vsnv(x,data,nx,svec,nvec,nav,nsn)

      f=f1
      df=float(nav)/float(n)*(fs/2.0)

      do 1 i=1,nsn
         write(*,'(1X,A2,I5,F4.0,2F6.0,3F6.1)')
     +      '%S', n, snr, f, f+df, dbp(svec(i)), dbp(nvec(i)),
     +      dbp(svec(i))-dbp(nvec(i))
         f=f+df
1     continue

      return
      end

      subroutine davsn(x,data,nx,n,snr,svec,nvec)

c same as dvsn but displays the average signal and noise power
c of several signal vectors over all frequencies.

c x    - vector of received data values
c data - transmitted logical data values
c nx   - number of elements in x and data
c n    - OFDM block size (to print)
c snr  - RF SNR (to print)

c nav  - number of snr measurements (of x) averaged so far
c n2av - number of snr measurements to average

      integer n, nx
      logical data(nx)
      real x(nx), svec(nx), nvec(nx)
      real snr

      integer nsn, n2av
      real sums, sums2, sumn, sumn2
      integer nsums, nsumn
      real dbp, stmn
      logical first
```

```
      data first /.true./

      if(first)then
         first=.false.
         call stinit(sums,sums2,nsums)
         call stinit(sumn,sumn2,nsumn)
         write(*,*) ' how many blocks to average ? '
         read(*,*) n2av
      endif

      call vsnv(x,data,nx,svec,nvec,nx,nsn)

      call stat(svec(1),sums,sums2,nsums)
      call stat(nvec(1),sumn,sumn2,nsumn)

      n2av=n2av-1

      if(n2av.le.0)then
         write(*,'(1X,A2,I5,F4.0,3F6.1)')
     +      '%X', n, snr,
     +      dbp(stmn(sums,sums2,nsums)),
     +      dbp(stmn(sumn,sumn2,nsumn)),
     +      dbp(stmn(sums,sums2,nsums)) -
     +      dbp(stmn(sumn,sumn2,nsumn))
         call stinit(sums,sums2,nsums)
         call stinit(sumn,sumn2,nsumn)
         write(*,*) ' how many blocks to average ? ', char(7)
         read(*,*) n2av
      endif

      return
      end

      subroutine vsnv(x,data,nx,s,n,nav,nsn)

c computes vectors of mean square and variance of received data
c values averaged over over several values vector x, (to get signal
c and noise powers as a function of frequency).

c x    - vector of received data values
c data - transmitted logical data values
c nx   - number of elements in tx and rx
c s    - square of mean of received data values
c n    - variance of received data values
c nav  - (maximum) number of elements of x to average
c nsn  - number of elements in s and n

      integer nx, nav, nsn
      logical data(nx)
      real x(nx)
      integer mxnsn
      parameter (mxnsn=100)
      real s(mxnsn), n(mxnsn)

      real sp, np
      integer i, k, nleft

      if(nx.le.0)then
         write(*,*) ' vsn : nx <= 0 .'
         stop
      endif

      i=1
      nsn=1
      nleft=nx
1     continue
         k=min(nleft,nav)
         call vsn(x(i),data(i),k,sp,np)
         s(nsn)=sp
         n(nsn)=np
         i=i+nav
         nsn=nsn+1
         nleft=nleft-nav
      if(nleft.gt.0)goto 1

      nsn=nsn-1

      return
      end

      subroutine vsn(x,data,nx,s,n)

c computes mean square and variance of received data values in
c a vector x, (to get signal and noise powers) by using negatives
c of values in x whose corresponding element in data are 'false'.

c x    - vector of received data values
c data - transmitted logical data values
c nx   - number of elements in tx and rx
c s    - square of mean of received data values
c n    - variance of received data values

      integer nx
      logical data(nx)
      real x(nx), s, n

      real ts, fs, ts2, fs2, tsp, tnp, fsp, fnp
      integer i, nt, nf
```

```
      if(nx.le.0)then
         write(*,*) ' vsn : nx <= 0 .'
         stop
      endif

      ts =0.
      ts2=0.
      fs =0.
      fs2=0.
      nt=0
      nf=0

      do 1 i=1,nx
         if(data(i))then
            ts =ts  + x(i)
            ts2=ts2 + x(i)**2
            nt=nt+1
         else
            fs =fs  + x(i)
            fs2=fs2 + x(i)**2
            nf=nf+1
         endif
1     continue

      tsp = ( ts / nt )**2
      fsp = ( fs / nf )**2

      tnp = ts2 / nt - tsp
      fnp = fs2 / nf - fsp

      s = ( tsp + fsp ) / 2.
      n = ( tnp + fnp ) / 2.

      return
      end
```

simhw.f

```
$LARGE
c ************************************
c simhw.f - harware channel routines
c ************************************
c 87-11-19

      subroutine hwinit(data,prev,dev,tmp,neqbl,
     +      ia,nmax,ns,fs,f1,f2,dbd,rms,peak,txemp,demp,
     +      empscl,nempsc,empfr,empsc)

c generate pre- and de-emphasis and channel correction vectors
c by measuring channel responce

c data   - temporary logical vector, length ns
c prev   - generated pre-emphasis vector, length ns
c dev    - generated de-emphasis vector, length ns
c tmp    - temporary real vector, length ns
c neqbl  - number of blocks to average in generating equalization
c          vector
c ia     - integer*2 sample vector, length nmax
c nmax   - (OFDM) block size of channel probe signal and size of
c          prev/dev
c ns     - duration of channel probe signal (samples)
c fs     - sampling rate
c f1     - lower frequency limit of channel to use
c f2     - upper frequency limit of channel to use
c dbd    - dB/decade of pre-emphasis to use
c rms    - rms voltage of output signal
c peak   - peak voltage of output signal
c txemp  - true to do phase/amplitude correction at transmitter
c devrms - rms value of de-emphasis vector (dev)
c demp   - true to display equalization
c empscl -
c nempsc -
c empfr  -
c empsc  -

      integer ns, nmax, nempsc
      logical data(nmax), txemp, demp, empscl
      real prev(nmax), dev(nmax), tmp(nmax)
      real empfr(nempsc), empsc(nempsc)
      integer*2 ia(ns)
      integer i, neqbl, if1, if2, nf
      real fs, f1, f2, dbd, rms, peak
      real dbp, atan2

      integer j
      logical sr(23)
      real devrms
      real rmsv

      data sr/
     1 .false., .true., .true., .false., .false.,
     1 .false., .true., .true., .true., .true.,
     1 .false., .true., .false., .true., .false.,
     1 .true., .false., .true., .false., .true.,
     1 .false., .true., .false. /

c      write(*,*) ' fs,nmax,f1,f2 = ', fs,nmax,f1,f2

c find indices for frequency limits
```

```
      call setif(fs,nmax,f1,if1,f2,if2)
      nf=if2-if1+1

      call vfill(tmp,nmax,0.)

c prompt operator to turn off noise and fading

      pause ' turn noise and fading * OFF * '

c run prbs a few times to get rid of possible transients

         call prbs(data,nmax,sr)
         call prbs(data,nmax,sr)

c compute and average "neqbl" equalization vectors

      do 1 j=1,neqbl

c generate vector for pre-emphasis ("dbd" dB/decade)

         call empgen(prev(if1),nf,f2/f1,dbd)

c generate random +/- 1 values in frequency skipping DC and fs/2 terms

         call prbs(data,nmax,sr)

         call encode(data,dev,if1,if2,nmax)
         call emp(dev,prev,f1,f2,fs,nmax,nmax)
         call vcopy(dev,prev,nmax)

c modulate to time-domain

         call modu(prev,nmax,nf,.false.)

c send it through the channel

         call hwch(prev,dev,ia,nmax,ns,rms,peak)

c recover the (frequency-domain) channel output

         call demodu(dev ,nmax,nf,.false.)

c regenerate the (frequency-domain) input (before pre-emphasis)

         call encode(data,prev,if1,if2,nmax)

c complex divide input by output (in frequency domain) to generate
c correction (de-emphasis) vector

         call cdiv(prev(if1),dev(if1),dev(if1),nf)

c accumulate this equalization vector

         call vvadd(dev,tmp,tmp,nmax)

1     continue

c prompt operator to turn noise and fading back on

      pause ' turn noise and fading * ON * '

c scale by number of equalization vectors averaged

      call vsmul(tmp,nmax,1./float(neqbl))

c re-generate pre-emphasis vector (no dynamic storage, *%!&$%)

      call empgen(prev(if1),nf,f2/f1,dbd)

c if desired, print initial de-emphasis vector

c *** debug ***

c      write(*,*) ' if1, if2 = ', if1, if2

      if(demp)then
         do 2 i=if1,if2,50
            write(*,'(1X,A3,F6.0,F5.1,F8.0)') '%E ',
     +         float(i)/nmax*(fs/2),
     +         dbp(dev(i)**2+dev(i+1)**2),
     +         atan2(dev(i),dev(i+1))*57.3
2        continue
      endif

c if phase/magnitude correction is done at transmitter,
c swap emphasis vectors

      if(txemp)then

c normalize the old de-emphasis vector to (complex) magnitude of 1 and
c copy to pre-emphasis vector

         devrms=rmsv(dev(if1),nf)*sqrt(2.0)
         call vsmul(dev(if1),nf,1.0/devrms)
         call vcopy(dev,prev,nmax)

c generate a de-emphasis vector with appropriate magnitude

         call empgen(dev(if1),nf,f2/f1,dbd)
```

```
         call vsmul(dev,nmax,devrms)
      endif

      if(empscl)then
         call empscf(prev(if1),nf,f1,(f2-f1)/float(nf/2),
     +      empsc,empfr,nempsc)
      endif

      return
      end

      subroutine hwch(x,y,ia,n,ns,rms,peak)

c do modem i/o (with guard samples)

c x      - samples to be sent (variance = 1)
c y      - received samples (nominal variance = 1)
c ia     - integer*2 sample work vector
c n      - number of elements in x and in y
c ns     - total number of samples to generate
c rms    - rms voltage of output signal
c peak   - peak voltage of output signal
c          peak must be less than 2.5 (Volts) (maximum DAC output)

      integer i, j, k, n, nov, over, istart, iend
      real peak, rms
      real x(n), y(n), z, k1, k2
      integer*2 ia(ns)
      integer int
      integer io
      real float

c voltage-to-DAC and ADC-to-voltage conversion factors.
c values are for a 10-bit ADC and 12-bit DAC

      real dacscl, adcscl
      parameter ( dacscl  = 4096./5. )
      parameter ( adcscl  = 5./1024. )

c gain required to compensate for any (measured) filter loss

      real gfilt
      parameter ( gfilt = 1.11 )

c ( skip i/o for debugging )
c      call vcopy(x,y(2),n-1)
c      y(1)=x(n)
c      return

c test peak value

      if(peak.le.0.)then
         write(*,*) ' hwch : peak <= 0. '
         stop
      endif

      if(peak.gt.(2.5/gfilt))then
         write(*,*) ' hwch : peak level too large. '
         stop
      endif

c test block size and number of samples

      if(n.lt.0)then
         write(*,*) ' hwch: n < 0 '
         stop
      endif

      if(ns.lt.n)then
         write(*,*) ' hwch: ns < n '
         stop
      endif

c factor to give unit-rms samples the rms value

      k1=rms

c center the n OFDM samples in the ns-sample output

      istart=ns/2-n/2+1
      iend=ns/2+n/2

c combine dacscl and gfilt

      k2=dacscl*gfilt

c convert the f.p. samples to DAC levels
c limit peak level (limits peak deviation)

c       zmin= 1.e30
c       zmax=-1.e30
c       zrms= 0.

      nov=0

      j=istart
      do 1 i=1,n
         z=x(i)*k1
         if(z.gt.peak)then
            z=peak
```

```
            nov=nov+1
         else if(z.lt.-peak)then
            z=-peak
            nov=nov+1
         endif

         ia(j)=int( z * k2 + 0.5 )

c        if(z.lt.zmin)zmin=z
c        if(z.gt.zmax)zmax=z
c        zrms=zrms+z**2

         j=j+1
1     continue

c      write(*,*) ' % TX signal rms = ',sqrt(zrms/n)
c      write(*,*) ' % TX signal min = ',zmin
c      write(*,*) ' % TX signal max = ',zmax

      if(nov.ne.0)then
c         write (*,*) ' % ', 100.*float(nov)/n, ' % overflow. '
      endif

c add guard band before data

      j=istart-1
      i=iend
2     continue
      if(j.lt.1)goto 3
         ia(j)=ia(i)
         i=i-1
         j=j-1
         goto 2
3     continue

c add guard band after data

      j=iend+1
      i=istart
4     continue
      if(j.gt.ns)goto 5
         ia(j)=ia(i)
         j=j+1
         i=i+1
         goto 4
5     continue

c do io and stop if overrun

      over=io(ia,ns)

      if ( over.ne.0 ) then
         write(*,*) ' hwch: A/D or D/A overrun '
         stop
      endif

c scale A/D samples back to FP

      j=istart
      do 6 i=1,n
         y(i)=float(ia(j)) * adcscl
         j=j+1
6     continue

      return
      end


      subroutine emp(x,y,f1,f2,fs,n,ns)

c multiply vector x by vector y over the indices if1 to if2 to do
c pre-emphasis or de-emphasis and correction for channel gain/phase
c transfer function

c x - vector to be corrected
c y - correction vector
c if1 - first element of x to correct
c if2 - last element of x to correct
c n - number of values in x
c ns - number of values in y (multiple of n)

      integer i, j, j2, n, ns, skip
      integer if1, if2
      real f1, f2, fs
      real a, b, c, d
      real x(n)
      real y(ns)
      integer mod

      if(mod(ns,n).ne.0)then
         write(*,*) ' emp: ns not multiple of n '
         stop
      endif

      call setif(fs,n,f1,if1,f2,if2)
      call setif(fs,ns,f1,j,f2,j2)
      skip=2*ns/n

      do 1 i=if1,if2,2
            a=y(j)
```

```
          b=y(j+1)
          c=x(i)
          d=x(i+1)
          x(i  ) = a*c - b*d
          x(i+1) = a*d + b*c
          j=j+skip
1     continue

c ( debugging )
c      write(*,*) ' emphasis results: '
c      write(*,*) ' n, ns = ', n, ns
c      write(*,*) ' if1, if2 = ', if1, if2
c      write(*,*) ' j, skip = ', j, skip
c      write(*,*) ' results: i  j  a,b  c,d  result '
c            write(*,900) i, j, a, b, c, d, x(i), x(i+1)
c900   format(1x,2i5,6f7.3)

      return
      end

      subroutine cdiv(x,y,z,n)

c divide two complex vectors (z=x/y)

      integer i, n
      real x(n), y(n), z(n)
      real a, b, c, d, r
      integer mod

      if(mod(n,2).ne.0)then
         write(*,*) ' cdiv: n not even '
         stop
      endif

      do 1 i=1,n,2
c find mag. squared of y
         c=y(i)
         d=y(i+1)
         r=c*c+d*d
         if(r.ne.0.)then
c set c,d = 1/y
            c= c/r
            d=-d/r
c multiply by x=a,b
            a=x(i)
            b=x(i+1)
            z(i)  =a*c-b*d
            z(i+1)=a*d+b*c
         else
            write(*,*) ' % cdiv: complex divide by zero at i = ',i
            z(i)  =0.
            z(i+1)=0.
         endif
1     continue

      return
      end

      subroutine empgen(x,n,f2byf1,dbd)

c generate (complex) pre-emphasis vector

c x      - emphasis vector to be generated
c n      - number of elements (even)
c f2byf1 - ratio of highest to lowest frequency
c dbd    - number of dB per decade emphasis

      integer n, i
      real x(n), dbd, f2byf1
      real decs, xn, k, y, ss
      real alog10, sqrt, float
      integer mod

      if(mod(n,2).ne.0)then
         write(*,*) ' empgen : n not even '
         stop
      endif

c number of decades between frequency limits

      decs = alog10(f2byf1)

c total increase (linear factor)

      xn = 10. ** ( dbd * decs / 20. )

c constant factor to obtain required increase

      k = 10. ** ( alog10(xn) / (n/2-1) )

c generate scaled vector and find total power

      y=1.
      ss=0.
      do 1 i=1,n,2
         x(i  )=y
         x(i+1)=0.
         ss=ss+y**2
         y=y*k
1     continue
```

```
      ss=sqrt(ss/float(n/2))

c scale to unity power

      call vsmul(x,n,1./ss)

c ( debugging )
c      write(*,*) ' f2byf1, n, dbd, k = ', f2byf1, n, dbd, k
c      write(*,*) ' ss = ', ss
c      write(*,*) ' empgen power = ',rmsv(x,n)*sqrt(2.)

      return
      end

      subroutine waitfor(prompt)
      character*(*) prompt
      character c
      close(0)
      write(0,*)prompt
      read(0,'(A1)')c
      return
      end

      subroutine vvadd(a,b,c,n)

c c(i)=a(i)+b(i) for i=1 to n

      integer i, n
      real a(n), b(n), c(n)

      do 1 i=1,n
         c(i)=a(i)+b(i)
1     continue

      return
      end

      subroutine empscf(x,nx,f1,df,sc,fr,nsc)

c subroutine to scale a pre- or de-emphasis vector using a
c measured channel power transfer function or baseband SNR
c characteristics.

c x   - emphasis vector to be scaled
c nx  - number of real elements in x (assumed as real/imag. pairs)
c f1  - starting frequency of values in x
c df  - frequency increment between values in x
c sc  - the responce of the channel to be used to scale x
c fr  - the upper frequency limits for each scaling value in sc
c nsc - number of values in sc and fr

      integer nx, nsc
      real x(nx), sc(nsc), fr(nsc)
      real f1, df

      integer i, j
      real k, f, ss

      if(nsc.lt.1)then
         write(*,*) ' empscf : nsc < 1 .'
         stop
      endif

c initialize

      j=1
      f=f1
      ss=0.0
      k=10.0**(-1.0*sc(1)/20.0)

c scale the real part of every emphasis vector pair

      do 3 i=1,nx,2

c go on to next scaling value if necessary for current frequency

1     continue
      if((f.le.fr(j)) .or. (j.ge.nsc))goto 2
         j=j+1
         k=10.0**(-1.0*sc(j)/20.0)
         goto 1
2     continue

c scale and sum squares

      x(i)=x(i)*k
      ss=ss+x(i)**2

      f=f+df

3     continue

      ss=sqrt(ss/float(nx/2))
      call vsmul(x,nx,1.0/ss)

      return
      end
```

simdum.f

```
c ************************************
c simdum.f - dummy hardware channel routines for non-PC systems
c ************************************

      subroutine hwinit(data,prev,dev,tmp,neqbl,
     +      ia,ni,ns,fs,f1,f2,dbd,rms,peak,txemp,demp)

      integer neqbl, ni, ns
      logical data(ni), txemp, demp
      real prev(ni), dev(ni), tmp(ni)
      integer*2 ia(ns)
      real fs, f1, f2, dbd, rms, peak

      write(*,*) ' hwinit - dummy routine called '
      stop

      end

      subroutine hwch(x,y,ia,n,ns,rms,peak)

      integer n, ns
      real x(n), y(n), rms, peak
      integer*2 ia(n)

      write(*,*) ' hwch - dummy routine called '
      stop

      end

      subroutine emp(x,y,f1,f2,fs,n,ns)

      integer n, ns
      real f1, f2, fs
      real x(n), y(ns)

      write(*,*) ' emp - dummy routine called '
      stop

      end
```

sifft.dif (diffs from FAST.FOR)

```
0a1,5
> c warning !!!! : these routines have been modified to work
> c on arrays of size N instead of N+2
> c modified 87-8-14 to send all output to default output
> c
> C
1a7,9
> C SUBROUTINE:  FFA
> C FAST FOURIER ANALYSIS SUBROUTINE
> C-----------------------------------------------------------------
3c11
<       SUBROUTINE FFA(B, NFFT)
---
>       SUBROUTINE FFAn(B, NFFT)
37c45
<       WRITE (IW,9999)
---
>       WRITE (*,9999)
72,75c80,83
<       T = B(2)
<       B(2) = 0.
<       B(NFFT+1) = T
<       B(NFFT+2) = 0.
---
> c      T = B(2)
> c      B(2) = 0.
> c      B(NFFT+1) = T
> c      B(NFFT+2) = 0.
88c96
<       SUBROUTINE FFS(B, NFFT)
---
>       SUBROUTINE FFSn(B, NFFT)
119c127
<       WRITE (IW,9999)
---
>       WRITE (*,9999)
123c131
<       B(2) = B(NFFT+1)
---
> c      B(2) = B(NFFT+1)
710c718
<       WRITE (IW,9999)
---
>       WRITE (*,9999)
941a950,1062
>
> C
```

```
> C-----------------------------------------------------------------------
> C  FUNCTION:  I1MACH
> C  THIS ROUTINE IS FROM THE PORT MATHEMATICAL SUBROUTINE LIBRARY
> C  IT IS DESCRIBED IN THE BELL LABORATORIES COMPUTING SCIENCE
> C  TECHNICAL REPORT #47 BY P.A. FOX, A.D. HALL AND N.L. SCHRYER
> C-----------------------------------------------------------------------
> C
>       INTEGER FUNCTION I1MACH(I)
> C
> C  I/O UNIT NUMBERS.
> C
> C    I1MACH( 1) = THE STANDARD INPUT UNIT.
> C
> C    I1MACH( 2) = THE STANDARD OUTPUT UNIT.
> C
> C    I1MACH( 3) = THE STANDARD PUNCH UNIT.
> C
> C    I1MACH( 4) = THE STANDARD ERROR MESSAGE UNIT.
> C
> C  WORDS.
> C
> C    I1MACH( 5) = THE NUMBER OF BITS PER INTEGER STORAGE UNIT.
> C
> C    I1MACH( 6) = THE NUMBER OF CHARACTERS PER INTEGER STORAGE UNIT.
> C
> C  INTEGERS.
> C
> C    ASSUME INTEGERS ARE REPRESENTED IN THE S-DIGIT, BASE-A FORM
> C
> C               SIGN ( X(S-1)*A**(S-1) + ... + X(1)*A + X(0) )
> C
> C               WHERE 0 .LE. X(I) .LT. A FOR I=0,...,S-1.
> C
> C    I1MACH( 7) = A, THE BASE.
> C
> C    I1MACH( 8) = S, THE NUMBER OF BASE-A DIGITS.
> C
> C    I1MACH( 9) = A**S - 1, THE LARGEST MAGNITUDE.
> C
> C  FLOATING-POINT NUMBERS.
> C
> C    ASSUME FLOATING-POINT NUMBERS ARE REPRESENTED IN THE T-DIGIT,
> C    BASE-B FORM
> C
> C               SIGN (B**E)*( (X(1)/B) + ... + (X(T)/B**T) )
> C
> C               WHERE 0 .LE. X(I) .LT. B FOR I=1,...,T,
> C               0 .LT. X(1), AND EMIN .LE. E .LE. EMAX.
> C
> C    I1MACH(10) = B, THE BASE.
> C
> C  SINGLE-PRECISION
> C
> C    I1MACH(11) = T, THE NUMBER OF BASE-B DIGITS.
> C
> C    I1MACH(12) = EMIN, THE SMALLEST EXPONENT E.
> C
> C    I1MACH(13) = EMAX, THE LARGEST EXPONENT E.
> C
> C  DOUBLE-PRECISION
> C
> C    I1MACH(14) = T, THE NUMBER OF BASE-B DIGITS.
> C
> C    I1MACH(15) = EMIN, THE SMALLEST EXPONENT E.
> C
> C    I1MACH(16) = EMAX, THE LARGEST EXPONENT E.
> C
> C  TO ALTER THIS FUNCTION FOR A PARTICULAR ENVIRONMENT,
> C  THE DESIRED SET OF DATA STATEMENTS SHOULD BE ACTIVATED BY
> C  REMOVING THE C FROM COLUMN 1.  ALSO, THE VALUES OF
> C  I1MACH(1) - I1MACH(4) SHOULD BE CHECKED FOR CONSISTENCY
> C  WITH THE LOCAL OPERATING SYSTEM.
> C
>       INTEGER IMACH(16),OUTPUT
> C
>       EQUIVALENCE (IMACH(4),OUTPUT)
> C
> C     MACHINE CONSTANTS FOR THE VAX-11 WITH
> C     FORTRAN IV-PLUS COMPILER
> C
>       DATA IMACH( 1) /    5 /
>       DATA IMACH( 2) /    6 /
>       DATA IMACH( 3) /    5 /
>       DATA IMACH( 4) /    6 /
>       DATA IMACH( 5) /   32 /
>       DATA IMACH( 6) /    4 /
>       DATA IMACH( 7) /    2 /
>       DATA IMACH( 8) /   31 /
> c      DATA IMACH( 9) / 2147483647 /
> c imach 9 not used by fftsn
>       data imach( 9) /    0 /
>       DATA IMACH(10) /    2 /
>       DATA IMACH(11) /   24 /
>       DATA IMACH(12) / -127 /
>       DATA IMACH(13) /  127 /
>       DATA IMACH(14) /   56 /
>       DATA IMACH(15) / -127 /
>       DATA IMACH(16) /  127 /
> C
>       IF (I .LT. 1  .OR.  I .GT. 16) GO TO 10
```

```
> C
>       I1MACH=IMACH(I)
>       RETURN
> C
>  10   WRITE(*,9000)
>  9000 FORMAT(39H1ERROR    1 IN I1MACH - I OUT OF BOUNDS)
> C
>       STOP
> C
>       END
>
```

fdint.f

```
$LARGE
c ***************
c fdint.f - MC Integration
c ***************
c Ed. Casas 86-2-19

c Monte-Carlo integration program to obtain OFDM BER.
c revised for variable r-sn characteristics 87-7-3
c added block error rate calculations 88-11-11

c ------------------------------------------------------------------
$INCLUDE: 'simpdef.f'
c ------------------------------------------------------------------

c local variables:

c counters

      integer i, j, k, l

c workspace to generate fading waveforms

       real fade(ns)

c vectors to store BER in one trial, sums of BERs, squares of BERs
c and number of trials

      double precision berout, bkrout
      real bert (mxblk,mxsnr), ber  (mxblk,mxsnr), bers  (mxblk,mxsnr)
      real bkert(mxblk,mxsnr), bker (mxblk,mxsnr), bkers (mxblk,mxsnr)
      real temp
      integer ntrial(mxblk,mxsnr), nbktr(mxblk,mxsnr)

c vectors for received signal and noise values and their squares

      real rs(ns), rn(ns), rs2(ns), rn2(ns)

c external utilities

      real stmn, stu95, stl95

c intialize variance and BER sums (and sums of squares) to zero

      data ber /mxblsn*0.0/, bers /mxblsn*0.0/

c ------------------------------------------------------------------
$INCLUDE: 'simpget.f'
c ------------------------------------------------------------------

c initialize snr tables

      call s2init(b,w,rms,peak,fm,fd,rfm,agclim,sqlim,
     +         nints,intsr,intss, nintn,intnr,intnn,
     +         fading, noisng )

c do for each of ntr trials

      do 50 l=1,ntr

c clear sum of BERs for this trial

         do 60 j=1,nn
         do 60 k=1,nsnr
            bert (j,k)=0.d0
            bkert(j,k)=0.d0
60       continue

c do appropriate number of passes

      do 30 i=1,nblk

c generate a fading envelope with 0 dB mean

      if(fading)then
               call genfdb(fd/fs,fseed,fade,ns,
     +            ndbr,thr,ndw,nsw)
      else
         call vfill(fade,ns,0.)
      endif

c do for each snr

      do 30 k=1,nsnr

c convert received envelope level to r and s values and squares
```

```
        call r2sns(fade,rs,rn,ns,snr(k))
        call vsq(rs,rs2,ns)
        call vsq(rn,rn2,ns)

c do for each block size

        do 30 j=1,nn

c update variance and BER sums

              call sum2(rs,rs2,rn2,ns,na(j),ecn,berout,bkrout,
     +           snr(k),dnerr)
              bert (j,k)=bert (j,k)+berout
              bkert(j,k)=bkert(j,k)+bkrout

c              write(*,*) ' blk, BkER = ',i, bkrout

30    continue

c update statistics counts at end of a trial

      do 70 j=1,nn
      do 70 k=1,nsnr

            temp=bert(j,k)/nblk
            call stat(temp, ber(j,k), bers(j,k), ntrial(j,k))

            temp=bkert(j,k)/nblk
            call stat(temp, bker(j,k), bkers(j,k), nbktr(j,k))

c            write(*,*) 'j,k,ber,ntrial=',j,k,ber(j,k),ntrial(j,k)

70     continue

50    continue

c print results

      write(*,*)' % BER, -/+ .95 conf limits '
      do 100 j=1,nn
         do 100 k=1,nsnr
            write(*,'(1X,I6,F6.1,3(1PE11.2))') na(j), snr(k),
     +         stmn (ber(j,k), bers(j,k), ntrial(j,k)),
     +         stl95(ber(j,k), bers(j,k), ntrial(j,k)),
     +         stu95(ber(j,k), bers(j,k), ntrial(j,k))
100    continue

      write(*,*)' % BkER, -/+ .95 conf limits '
      do 200 j=1,nn
         do 200 k=1,nsnr
            write(*,'(1X,I6,F6.1,3(1PE11.2))') na(j), snr(k),
     +         stmn (bker(j,k), bkers(j,k), nbktr(j,k)),
     +         stl95(bker(j,k), bkers(j,k), nbktr(j,k)),
     +         stu95(bker(j,k), bkers(j,k), nbktr(j,k))
200    continue

       end

c------------------- Sum for one snr, len -------------------

       subroutine sum2(rs,rs2,rn2,ns,len,ecn,berout,bkrout,snr,dnerr)

c subroutine to calculate fade and snr statistics of a block and
c correspoding BER

c variables:

c counters

      integer i, j, k

c length of received signal and noise vectors

      integer ns

c vectors for received signal and noise values and squares

      real rs(ns), rs2(ns), rn2(ns)

c OFDM block length, FEC block (word) length

      integer len, ecn

c added to display block snr: snr and true to display block BER

      real snr
      logical dnerr

c inverse of block length

      real invlen

c number of blocks tested

      integer ntst, n

c sums of BERs

      double precision ber, berout, bker, bkrout, dtemp
```

```
c sums of s, s^2, n^2
      real a, b, c, d

c intrinsics

      integer mod

c external utilities

      real berblk

c find number of OFDM blocks included in the vector fade

      if(len.eq.0 .or. mod(ns,len).ne.0)then
         write(*,*)'sum2: bad OFDM block size.'
         stop
      endif

      n=ns/len

c find inverse of block length

      invlen=1./len

c start at the first sample of the input block

      i=1
      ntst=0
      ber= 0.0d0
      bker=0.0d0

c do for each block

      do 10 j=1,n

         a=0.
         b=0.
         c=0.

c get statistics for this block

         do 20 k=1,len
            a=a+rs (i)
            b=b+rs2(i)
            c=c+rn2(i)
            i=i+1
20       continue

         a=a*invlen
         b=b*invlen
         c=c*invlen

c         write(*,*) ' a, b, b-a**2, c = ', a, b, b-a**2, c

c add resulting block BER to the appropriate BER averaging sum
c and compute BkER assuming independent errors

         dtemp=berblk(a,b,c)
         ber =ber +dtemp

         if ( dnerr ) write(*,'(1X,A2,I5,F4.0,E16.7)') '%B ',
     +      len, snr, dtemp

         bker=bker+(1.0d0-(1.0d0-dtemp)**ecn)
         ntst=ntst+1


c         write(*,*) ' BER  = ', dtemp
c         write(*,*) ' BkER = ', (1.0d0-(1.0d0-dtemp)**ecn)

10    continue

      berout=ber /ntst
      bkrout=bker/ntst

      return
      end
```

pint.f

```
c ***************
c Integration for large/small N
c ***************
c pint.f
c ed.casas 87-9-16

c ----------------------------------------------------------
$INCLUDE: 'simpdef.f'
c ----------------------------------------------------------

c local variables

      real r(2000), p(2000), s(2000), s2(2000), n(2000), wrk(2000)

      integer i, j, k
      real al, be, ga
      real sber, lber
```

```
      real vsum, berblk

      real snrmin, snrinc
      integer nst
      parameter (nst=1100)

      snrmin=-50.
      snrinc=0.1

c ------------------------------------------------------------
$INCLUDE: 'simpget.f'
c ------------------------------------------------------------

c can only vary one threshold (agc OR squelch) at a time

      if ( agcvar .and. sqvar ) then
         write(*,*) ' pint : agcvar and sqvar true. '
         stop
      endif

c do at least one loop

      if ( nthrsh .le. 0 ) then
         nthrsh=1
         thrsh(1)=0.
      endif

      do 1 k=1,nthrsh

      if ( .not. (agcvar .or. sqvar) ) call
     +   s2init(b,w,rms,peak,fm,fd,rfm,agclim,sqlim,
     +         nints,intsr,intss, nintn,intnr,intnn,
     +         fading, noisng )

      do 1 j=1,nsnr

c initialize snr-to-signal and snr-to-noise tables

c if testing effect of varying the agc limit

      if ( agcvar ) call
     +   s2init(b,w,rms,peak,fm,fd,rfm,snr(j)+thrsh(k),sqlim,
     +         nints,intsr,intss, nintn,intnr,intnn,
     +         fading, noisng )

c if testing effect of varying the squelch limit

      if ( sqvar ) call
     +   s2init(b,w,rms,peak,fm,fd,rfm,agclim,snr(j)+thrsh(k),
     +         nints,intsr,intss, nintn,intnr,intnn,
     +         fading, noisng )

c initialize probability (Rayleigh) table

         call pgen(snrmin,snrinc,snr(j),ndbr,p,r,nst)

c find signal and noise scaling signals from snr signal

         call r2sns(r,s,n,nst,0.)

c compute s**2 and n**2

         call vmul(s,s,s2,nst)
         call vmul(n,n,n,nst)

c integrate to find averages of al(pha), be(ta), and ga(mma)

         call vmul(p,s,wrk,nst)
         al=vsum(wrk,nst)

         call vmul(p,s2,wrk,nst)
         be=vsum(wrk,nst)

         call vmul(p,n,wrk,nst)
         ga=vsum(wrk,nst)

c use block BER equation to estimate long-block BER

         lber=berblk(al,be,ga)

c find single-sample-block BERS and scale by the sample probability

         do 2 i=1,nst
            wrk(i)=berblk(s(i),s2(i),n(i))*p(i)
c            write(*,'(1X,I10,F10.2,5E15.3)')i,r(i),
c     &         berblk(s(i),s2(i),n(i)),p(i),wrk(i)
2        continue

c add up (average) to find short-block BER

         sber=vsum(wrk,nst)

         if ( agcvar .or. sqvar ) then
            write(*,*) snr(j), thrsh(k), sber, lber
         else
            write(*,*) snr(j), sber, lber
         endif
```

```
1     continue
      end

      subroutine pgen(snrmin,snrinc,snravg,ndbr,p,r,n)

c generate tables of snrs and probabilities

c input:
c snrmin - minimum snr
c snrinc - snr increment per step
c snravg - average snr
c ndbr   - number of diversity branches
c r      - signal levels (snr)
c p      - probability of a given step
c n      - number of values in p

      integer n, ndbr
      real snrmin, snrinc, snravg
      real p(n), r(n)

c local variables:

c i      - counter into p and r
c sump   - sum of probabilities (should add to 1)

      integer i
      real sump
      real snrk1, snrk2, snr

c rayleigh CPDF

      double precision dray

c calculate signal level points and probabilities

      sump=0.
      snr=snrmin
      snrk1=snrinc/2.0-snravg
      snrk2=snrinc/2.0+snravg

      do 1 i=1,n

c save the snr and probability between the two signal levels

         r(i) = snr
         p(i) = dray(snr+snrk1,ndbr) - dray(snr-snrk2,ndbr)

c sum probabilities to check

         sump=sump+p(i)

c         write(*,'(1X,I10,F10.2,2E15.3)') i, r(i), p(i), sump

         snr=snr+snrinc

1     continue

      if(abs(sump-1.0).gt.0.001)then
         write(*,*) 'pgen:total probability <>1; = ',sump
         stop
      endif

      return
      end
```

io.asm

```
COMMENT $

This Microsoft FORTRAN-callable function reads/writes a block of
samples from/to the analog interface board.  Ed. Casas 87-10-19.

The FORTRAN use is:

    NR=IO(IA,N)

where:

IA - INTEGER*2 (16-bit) array containing the D/A samples on entry
       and containing the A/D samples on return.  If "convert" is
       not zero the samples are left justified in binary (unsigned)
       format.  In this case the samples should be pre-/post-
       converted to two's-complement.

N  - INTEGER*4 number of values to be input and output.

NR - INTEGER*4 number of samples *NOT* read/written.  If this number
       is not zero, an over-run occurred.

$

IO_TEXT  segment byte public 'CODE'
         assume cs:IO_TEXT

; timing constants

MHZ    equ     2       ; 8253 clock input frequency (MHz)
PERIOD equ     125     ; sampling period, (us) (125 minimum)
LEN    equ     4       ; S/H sampling time (us) (4 maximum)
```

```
; non-zero to convert between offset-binary and 2's complement

convert equ  1          ; set to 0 if get overrun errors on a slow PC

; hardware

IBMIO  equ     300H    ; I/O base address for IBM prototyping card

PIA    equ     IBMIO   ; 8255 par. port: MS bit=overrun, LS=sampling
PIA0   equ     PIA+0   ;       PIA port A
PIA1   equ     PIA+1   ;       PIA port B
PIA2   equ     PIA+2   ;       PIA port C
PIA3   equ     PIA+3   ;       PIA control port

ADC    equ     PIA+4   ; NEC uPD7004 A/D converter
ADC0   equ     ADC+0   ;       channel select & LS 2 bits
ADC1   equ     ADC+1   ;       "initialize" & MS 8 bits

DAC    equ     PIA+8   ; National DAC1208 D/A converter
DAC0   equ     DAC+0   ;       LS 4 bits (load second)
DAC1   equ     DAC+1   ;       MS 8 *AND* LS 4 bits (load first)

CLK    equ     PIA+12  ; 8253 timer/counter
CLK0   equ     CLK+0   ;       counter 0
CLK1   equ     CLK+1   ;       counter 1
CLK2   equ     CLK+2   ;       counter 2
CLK3   equ     CLK+3   ;       mode register

IO     proc    far
       public  IO

; entry

       push    bp                      ; save bp
       mov     bp,sp
       push    si                      ; save si
       pushf                           ; save flags (& interrupt status)

; disable interrupts

       cli

; set up sample count

       les     bx, dword ptr [bp+6]
       mov     si, es:[bx]             ; SI has LS word of sample count
       mov     di, es:[bx+2]           ; DI has MS word of sample count

; negate sample count so can count up to zero

       not     si                            ; complement di:si
       not     di
       add     si, 1                         ; add 1
       adc     di, 0

; es:bx points into sample array

       les     bx, dword ptr [bp+10]

; dh retains high byte of I/O board address

       mov     dh, high PIA

; hardware initialization

; set up PIA

PIAr   record  modeset:1=1, Amod:2,Adir:1,Aptc:1, Bmod:1,Bdir:1,Bptc:1

       mov     dl, low PIA3
       mov     al, PIAr<1, 01,1,1, 0,1,1>
       out     dx, al

; set timers first to stop them and prevent overrun error on
; first (unused) conversion

CLKr   record  counter:2, readload:2=3, clkmode:3, bcd:1=0

; start by setting timer 0 mode (and so stopping it)

       mov     dl, low CLK3
       mov     al, CLKr<0,,2,>         ; 0 = MODE 2 (rate generator)
       out     dx, al

; set up timer 1

       mov     dl, low CLK3
       mov     al, CLKr<1,,1,>         ; 1 = MODE 1 (one-shot)
       out     dx, al

       mov     dl, low CLK1             ; set S/H sample time
       mov     al, low  (MHZ*LEN)
       out     dx, al
       mov     al, high (MHZ*LEN)
       out     dx, al

; set up ADC, start first (unused) conversion, and clear "sampling"
; and "overrun" latches
```

```
ADCr0  record  channel:3=0                ; CH 0 input
ADCr1  record  twoscomp:1=0, divider:2=1  ; binary, divide clock by 2

       mov     dl, low ADC1
       mov     al, ADCr1<>
       out     dx, al

       mov     dl, low ADC0
       mov     al, ADCr0<>
       out     dx, al

; finish setting up timer 0 :

; set sampling rate and start the timer, first S/H pulse is PERIOD us
; later.  the first (initialization) conversion will have completed by
; then.

       mov     dl, low  CLK0          ; set sampling rate
       mov     al, low  (MHZ*PERIOD)
       out     dx, al
       mov     al, high (MHZ*PERIOD)
       out     dx, al

; end of hardware initialization

; *** critical timing within loop: do not change code ***

loop:

; load sample into DAC (to be transferred by next S/H pulse)

       mov     ax, es:[bx]

       if      convert        ; if converting
       rept 4                 ; make sample left-aligned
       shl     ax, 1
       endm
       xor     ax, 8000h      ; convert to offset-binary
       endif

       xchg    al, ah          ; write MS byte, then LS
       mov     dl, low DAC1    ; write DAC1 and wrap around to DAC0
       out     dx, ax

; set up ADC control word in cl, and PIA address in dx

       mov     cl, ADCr0<>
       mov     dl, low PIA0

; wait for S/H to start sampling (implies conversion complete)

l1:    in      al, dx
       or      al, al          ; test for sampling or overrun
       jz      l1

; get A/D sample and start next conversion (sampling should be complete)

       mov     dl, low ADC0
       in      ax, dx          ; input A/D result
       xchg    ax, cx          ; sample to CX, get A/D control to AL
       out     dx, al          ; start A/D conversion

; exit if MS bit of PIA0 was set (overrun)

       jl      done

; replace output sample with input sample

       if      convert         ; if converting in loop

       mov     ax, cx
       xor     ax, 8000h       ; convert to 2's complement
       rept 6
       sar     ax, 1           ; make right-justified
       endm
       mov     es:[bx], ax

       else                     ; if not converting, save sample as read

       mov     es:[bx], cx

       endif

; point to next element in (possibly $LARGE) array

      add      bx,2             ; increment pointer to next sample
      jnc      l2               ; test for offset < 64k
      mov      ax, es           ; move segment up if not
      add      ax, 1000h
      mov      es, ax
l2:

; increment (negated) sample count

       inc     si
       jnz     l3
       inc     di
l3:

; loop 'til done
```

```
        jnz     loop

; return to caller

done:

        not     si                  ; negate unused count
        not     di
        add     si, 1
        adc     di, 0

        mov     ax, si              ; return unused count
        mov     dx, di

; return

        popf                        ; restore flags (& interrupt status)
        pop     si                  ; restore si
        mov     sp,bp               ; restore sp and bp
        pop     bp
        ret     08h

io      endp

IO_TEXT ends

        end
```

sim.asm

```
; Fading simulator controller. Ed Casas and Ron Jeffery. July 24, 1987.

idac    equ      00H                ; address of I D/A
qdac    equ      20H                ; address of Q D/A
xfer    equ      40H                ; address of common D/A output strobe
switch  equ      60H                ; address of switch port (8255 port A)
Ncos    equ      2048               ; number of entries in cosine tables
ioffst  equ      2066 shl 4         ; measured I and Q DAC values for minimum
qoffst  equ      2046 shl 4         ;  RF output. 12 bits left justified
m0dB    equ      7215               ; DAC output for 0 dB (MS 12 bits used)
m20dB   equ      m0dB/10            ; DAC output for -20 dB

mem     segment  at 0               ; absolute memory references. (DEBUG
        org      0400h              ;  creates file for EPROM programmer).
phases  dd       9 dup (?)          ; phase counters
        org      8000h
start   label    far                ; code start address (init:)
mem     ends

sum     macro    opI, cosI, opQ, cosQ
        lodsw                       ;; get LS word of phase increment
        add      ax, [di]           ;; add it to the LS word of phase
        stosw                       ;; store the LS word of phase
        lodsw                       ;; repeat for MS word
        adc      ax, [di]           ;; (plus carry)
        stosw                       ;; si and di now point to next ones
        and      ax, (Ncos-1)*2     ;; MS word of phase mod Ncos *2
        mov      bx, ax             ;; is now cosine table offset in bx
        opI      cx, cosI[bx]       ;; add/subtact cosine value to I sum
        opQ      dx, cosQ[bx]       ;;  and to Q sum
        endm

rom     segment
        assume   cs:rom,ds:rom,es:rom,ss:rom
        org      8000h              ; EPROM starts at 32k
init:   cli                         ; interrupts off
        cld                         ; set direction flag = up
        mov      ax, 0              ; set segment registers = 0
        mov      ds, ax
        mov      es, ax
        mov      ss, ax
        mov      cx, 9*2            ; clear phase counters
        mov      di, offset phases
        rep      stosw
        mov      sp, 8000h          ; set stack (not used)
        mov      al,10011011b       ; set all 8255 ports as unlatched
        out      switch+3, al       ;   input and start in off mode ...
off:    mov      dx, 0              ; set Q for minimum o/p
        mov      cx, m0dB           ; and I for for 0dB
        test     al, 10000000b      ; -20 dB level switch on ?
        jz       l1                 ; if not, skip ahead
        mov      cx, m20dB          ; else set I for -20 dB
l1:     jmp      outpt              ; and set D/As
loop:   in       al, switch
        test     al, 00000001b      ; run switch on ?
        jz       off                ; if not, go set a fixed level
        and      ax, 01111110b      ; middle 6 bits of switch is pointer
        xchg     bx, ax             ; to pointer to phase increments
        mov      si, pntrs[bx]      ; si --> first of 9 phase increments
        mov      di, offset phases ; di --> first of 9 phase variables
        sum      mov,  cos3, mov, cos1   ; \
        sum      add,  cos2, add, cos2   ; |
        sum      add,  cos1, add, cos3   ; | increment phases and sum
        sum      <;>,      , add, cos4   ; |  cosine table values for
        sum      sub,  cos1, add, cos3   ; |  I (in cx) and Q (in dx)
        sum      sub,  cos2, add, cos2   ; |
        sum      sub,  cos3, add, cos1   ; |
        sum      sub,  cos4, <;>,        ; |
```

```
       sum      add,  cos5, add, cos5   ; /
outpt: add      cx, ioffst      ; convert I sum to offset-binary
       mov      al, ch          ; move to ax and swap bytes to
       mov      ah, cl          ;   output MS byte first
       out      idac+1, ax      ; idac+0 is also at idac+2
       add      dx, qoffst      ; \
       mov      al, dh          ;  |
       mov      ah, dl          ;  | repeat for Q sum
       out      qdac+1, ax      ; /
       out      xfer, al        ; change both outputs at same time
       jmp      loop            ; repeat forever

pntrs  label    word            ; pointers into phase increments
       x =      offset incrmnts ; table. 4 byte * 9 increments
       rept     128             ; per frequency = 36 bytes/entry
       dw       x
       x =      x+36
       endm
       include  tables          ; cos1 to cos5 and incrmnts
       org      0fff0h          ; reset vector
       jmp      start
rom    ends
       end
```

ctabgen.for

```
C Print scaled cosine tables in 8088 assembler format.

      integer i, j, k, N, No, tmp(8), tmsusd(5)
      real A, pwr, pi, sqrt, sin, cos, float
      data N/2048/, No/8/, pi/3.14159/, pwr/0./, tmsusd/2,2,2,1,1/

      do 3 i=1,5
         if(i.eq.5)then
            A=1750*sqrt(2.)*sin(pi/4.)
         else
            A=1750*2.*sin(pi*float(i)/No)
         endif
         pwr=pwr+tmsusd(i)*(A**2)/2.
         write(*,'('' cos'',i1,'' label word'')')i
         do 2 j=1,N,8
            do 1 k=1,8
               tmp(k)=A*cos(float(j+k-2)*2.*pi/N)
1           continue
            write(*,'(''  dw '',7(i5,'',''),i5)')(tmp(k),k=1,8)
2        continue
3     continue
      write(*, '('' ; 0 dB at '',f10.1)')sqrt(2.*pwr)
      end
```

ptabgen.for

```
C Print phase increment tables in 8088 assembler format.

      integer*4 i, j, M, tmp(9), No, N
      real fs, tpi, cos, float
      data No/8/, N/2048/, tpi/6.28318/, fs/2958./

      M=4*No+2
      write(*,'('' incrmnts label word'')')
      do 2 i=0,254,2
         do 1 j=1,8
            tmp(j)=cos(tpi*float(j)/M)*i/fs*N*65536*2
1        continue
         tmp(9)=i/fs*N*65536*2
         write(*,'(''  dd '',4(i11,'',''),i11)') (tmp(j), j=1,5)
         write(*,'(''  dd '',3(i11,'',''),i11)') (tmp(j), j=6,9)
2     continue
      end
```

ceval.c

```
/* ceval.c - evaluate BCH code performance from a run-length file */

#include <stdio.h>
#include <assert.h>
#include <math.h>

#define BUFFSIZE 4096

main(int argc, char **argv) {

int
    i ,        /* arg counter */
    buf[BUFFSIZE], /* bit buffer */
    *p ,       /* pointer to start of an FEC block in buf */
    e ,        /* number of errors in a block */
    n=0,
    argn = 0 ,    /* symbols per block */
    m, argm = 0 ,    /* bits per symbol */
    t, argt = 0 ,    /* correctable symbols per block */
    arge = 0 ,        /* display errors in each block */
    argi = 0 ,        /* do interleaving */
    argh = 0,        /* produce histogram of errors/block */
    argC = 0,     /* display cumulative pdf */
    argN = 0,    /* normalize pdf */
```

```
    args = 0,    /* BER, BKER summary */
    nm ,       /* n*m */
    bits ;     /* bits left to test in buf */

long
    sume, sumb,        /* sum of errors and bits tested */
    h_sum,            /* sum of histogram values */
    *phist,            /* pointer to hist */
    hist [BUFFSIZE+1] ; /* histogram */

FILE
    *infile=NULL ; /* input file */

    for (i=1 ; i<argc ; i++) {
         if ( !strcmp(argv[i],"-m") ) sscanf(argv[++i],"%d",&argm) ;
         if ( !strcmp(argv[i],"-n") ) sscanf(argv[++i],"%d",&argn) ;
         if ( !strcmp(argv[i],"-t") ) sscanf(argv[++i],"%d",&argt) ;
         if ( !strcmp(argv[i],"-h") ) argh=1 ;
         if ( !strcmp(argv[i],"-e") ) arge=1 ;
         if ( !strcmp(argv[i],"-i") ) argi=1 ;
         if ( !strcmp(argv[i],"-C") ) argC=1 ;
         if ( !strcmp(argv[i],"-N") ) argN=1 ;
         if ( !strcmp(argv[i],"-s") ) args=1 ;
         if ( !strcmp(argv[i],"-f") )
            if ( (infile=fopen(argv[++i],"r")) == NULL )
                perror (argv[i]) ;
         } ;

    /* initialize histogram and bit/error counters */

    for ( i=0 ; i<BUFFSIZE+1 ; i++ ) hist[i] = 0 ;
    sume = sumb = 0 ;

    /* ensure an input file */

    if ( infile == NULL ) infile=stdin ;

    /* do for all FEC blocks in one OFDM block :
        interleave if necessary
           if fewer than t symbol errors, bit errors = 0
           else count bit errors
        update histogram and bit/error counts
           maybe print number of bit errors */

    while ( (bits=get_buf(infile,buf)) > 0 ) {

        if ( n <= 0 ) {    /* initialize n,m,t,nm */
            if ( argn ) n=argn ; else n=bits ;
            if ( argm ) m=argm ; else m=1 ;
            if ( argt ) t=argt ; else t=0 ;
            nm=n*m ;
            printf("%% BCH (n=%d, m=%d, t=%d)\n",n, m, t) ;
            if (n <= 0 || m<=0 || t<0) err("bad parameter") ;
            if ( arge ) printf("%% errors per block :\n") ;
            } ;

              p=buf ;
              while ( bits >= nm ) {
                if ( argi ) intlv (p, nm, 1) ;
                if ( nserr(p,m,n) > t ) e=nerr(p,nm) ; else e=0 ;
                (hist[e])++ ;
                sume += e ;
                sumb += nm ;
                if ( arge ) printf("%d\n",e) ;
                bits-=nm ;
                p+=nm ;
                } ;
      } ;

    /* compute sum and maybe make cumulative */

    h_sum = 0 ;
    for ( i=0 ; i<=nm ; i++ ) {
        h_sum+=hist[i] ;
        if ( argC ) hist[i] = h_sum ;
        } ;

    /* error checks */

    if ( sumb <= 0 ) err("no input") ;
    assert( h_sum*n*m == sumb ) ;
    assert( h_sum > 0 ) ;
    assert( sumb > 0 ) ;

      /* maybe display [C] pdf, maybe normalized */

    if ( argh ) {
        printf("%% [C]PDF : \n") ;
        for ( i=0 ; i<= n ; i++ )
            if ( argN ) printf ("%d %g\n",
                i, ((float) ( hist[i] )) / h_sum ) ;
            else printf("%d %ld\n",i,hist[i]) ;
        } ;

    /* maybe display summary  */

    if ( args ) {
        printf("%% BER = %g\n",
                       (float) (sume) / (float) (sumb) ) ;
        printf("%% BKER = %g\n",
```

```
                        (float) (h_sum - hist[0]) / (float) h_sum ) ;
            } ;

    } ;

err (char *msg) {
    fputs(msg,stderr) ;
    fputs(".\n",stderr) ;
    exit(1) ;
    } ;

int nserr(int *p, int m, int n) {

/* count number of m-bit symbols with errors in a block of n symbols */

    int e=0 ;
    while ( n-- ) {
        if ( nerr(p,m) ) e++ ;
        p+=m ;
        } ;
    return ( e ) ;
    } ;

int nerr(int *p, int n) {

/* count number of bits in error in a block of n bits */

    int e=0 ;
    while ( n-- ) if ( *p++ ) e++ ;
    return ( e ) ;
    } ;

int get_buf(FILE *file, int *buf) {

/* Reads error-free run lengths from a file and unpacks the run
   lengths into a bit-error pattern (0=no error, 1=error).  Last
   (error-free) run length in the block should be followed by a -1.
   Returns the number of bits generated. */

    int i=0, k ;

    while ( fscanf(file,"%d",&k) == 1 && k >= 0 ) {
       if ( i+k+1 >= BUFFSIZE ) k = BUFFSIZE-i-2 ;
       while ( k-- ) { buf[i++] = 0 ; } ;
       buf[i++]=1 ;
       } ;
    return (i ? i-1 : 0) ;
    } ;

int put_buf(FILE *file, int *p, int n) {

#define put_cnt(x) fprintf(file,"%d\n",x)

/*  Packs a bit-error pattern (0=no error, 1=error) into
    corresponding error-free run lengths and writes it to a file.
    The last (error-free) run length in the block is followed by a
    -1.  */

    int k ;

    k=0;
    while ( n-- )
        if ( *p++ ) { put_cnt(k) ; k=0 ; }
        else k++ ;
    put_cnt(k) ;
    put_cnt(-1) ;
    } ;

int intlv(int *in, int n, int dir) { /* block interleaver */

int
    out [ BUFFSIZE ],
    i, j, k, l ;

    /* compute interleaving step size (round up to make sure
       interleave all) */

    k = (int) floor ( 1.0 + sqrt ( (float) n ) ) ;

    l=0 ;
    for ( i=0 ; i<k ; i++ ) { /* step through offsets within blocks */
        for ( j=i ; j<n ; j+=k ) { /* step through blocks */
            if ( dir ) out [l] = in [j] ;
            else out [j] = in [l] ;
            l++ ;
            } ;
        } ;

    for ( i=0 ; i<n ; i++ ) in[i] = out[i] ;

    } ;
```

bkp.c

```
/* bkp.c

   computes average distribution of the number of bit errors in blocks
   of size N assuming independent errors within each block but different
   BERs for each block.  The standard input contains the block BERs
   and the program takes one argument, the block size.  At input EOF
   the distribution is written to standard output.  The binomial
   distribution is computed using the 'bico' routine from _Numerical
   Recipes_.

   Ed.Casas  89-3-7  */

double lnbico(), factln(), gammln() ;
double ipow() ;
void nrerror () ;

#include <stdio.h>
#include <math.h>

#define NMAX 4096

main(int argc, char **argv) {

int i, n, nblk ;
double ber, nber ;
double prob [ NMAX+1 ] ;
double sump ;

    /* check arguments */

    if ( argc < 2 ) {
        fprintf(stderr,"Usage %s <bits/block>\n",argv[0]) ;
        exit(1) ;
        } ;

    if ( sscanf(argv[1],"%d",&n) != 1 || n > NMAX ) {
        fprintf(stderr,"N (%s) bad or too large.\n",argv[1]) ;
        exit(1) ;
        } ;

    /* test for log factorial

    { int x1, x2 ;
    printf ("enter x1 and x2 ");
    scanf("%d %d",&x1,&x2) ;
    printf("ln of factorial: = %f %f \n",factln(x1),factln(x2)) ;
    printf("x1!/x2! ?= %lg\n",
            exp( (double) (factln(x1)-factln(x2)) ) ) ;
    } ;

    */

    /* initialize distribution */

    for ( i=0 ; i<=n ; i++ ) prob[i]=0.0 ;
    nblk = 0 ;

    /* loop through input BERs, compute, and sum distributions */

    while ( scanf("%lg",&ber) == 1 ) {
        for ( i=0 ; i<=n ; i++) prob[i] += exp ( lnbico(n,i)
            + log ( 1.0-ber ) * ( n-i )
            + log ( ber ) * i ) ;
        nblk ++ ;
        } ;

    /* display cumulative results */

    sump=0. ;
    if ( nblk <= 0 ) {
       fprintf(stderr,"No input.\n") ;
       exit(1) ;
       }
    else for ( i=0 ; i<=n ; i++)
         printf("%d %lg\n", i, sump += (prob[i]/nblk) ) ;

    } ;

/* the following routines are adapted from _Numerical_Recipes_in_C_ */

double lnbico(n,k)      /* modified to return ln */
int n,k;
{
    double factln();
    return factln(n)-factln(k)-factln(n-k) ;
}

#define MAXN 2048
double factln(n)
int n;
{
    static double a[MAXN+1]; /* cache blocksizes up to MAXN bits */
    double gammln();
    void nrerror();

    if (n < 0) nrerror("Negative factorial in routine FACTLN");
    if (n <= 1) return 0.0;
```

```
    if (n <= MAXN) return a[n] ? a[n] : (a[n]=gammln(n+1.0));
    else return gammln(n+1.0);
}


double gammln(xx)
double xx;
{
    double x,tmp,ser;
    static double cof[6]={76.18009173,-86.50532033,24.01409822,
        -1.231739516,0.120858003e-2,-0.536382e-5};
    int j;

    x=xx-1.0;
    tmp=x+5.5;
    tmp -= (x+0.5)*log(tmp);
    ser=1.0;
    for (j=0;j<=5;j++) {
        x += 1.0;
        ser += cof[j]/x;
    }
    return -tmp+log(2.50662827465*ser);
}


void nrerror(error_text)
char error_text[];
{
    fprintf(stderr,"Numerical Recipes run-time error...\n");
    fprintf(stderr,"%s\n",error_text);
    fprintf(stderr,"...now exiting to system...\n");
    exit(1);
}
```

out2bers.csh

```
#!/bin/csh
#
# extracts block BERs from fdint output (out.bers)
# Ed.Casas 89-3-17
#
foreach s (10 15 20 25)
  foreach n (256 1024 4096)
    awk -f out2bers.awk n=$n snr=$s. out.bers >bers.$n.$s
  end
end
```

out2bers.awk

```
#
# bers.awk - extracts the block BERs for one set of N, SNR values
# ed.casas 89-3-17
#
/^\ %B/ { if ( $2 == n && $3 == snr ) print $4 }
```

out2runs.awk

```
#
# runs.awk - extracts the error-free run lengths for one set of
# N, SNR values
# ed.casas 89-3-2
#
/^\ %N/ { if ( $2 == n && $3 == snr ) on=1 ; else on=0 }
# un-comment next line for testing
#/^\ %N/ { if ( on ) print $0 }
/^\ %R/ { if ( on ) print $2 }
```