USER MANUAL

FrequenC

Version 2.0

Hamilton Technical Services
195 Woodbury Street
Hamilton, MA 01982 USA

FILENAME: FUNCTIONS.DOC 04/23/03

FrequenC

A Library of C Functions for Frequency Stability Analysis

FrequenC is a library of C functions for the analysis of frequency stability that is distributed along with the Stable32 software package as a 32-bit Microsoft Windows dynamic link library (Freque32.dll). It includes many of the specialized functions needed to analyze time and frequency data. Included are functions for conversion between time and frequency data, conversion between time and frequency domains, and the calculation of various Allan, Hadamard and total variances from phase or frequency data. The library also includes functions for drift calculation and removal, common statistical functions, and several special functions for identifying noise types and determining confidence intervals. These functions use an array data format that defines analysis limits and can include gaps. The following tables list the FrequenC Library functions:

Allan Variance Functions	
CalcPhaseSigma	Calculate Allan deviation for phase data
CalcFreqSigma	Calculate Allan deviation for frequency data
CalcPhaseModSigma	Calculate modified Allan deviation for phase data
CalcFreqModSigma	Calculate modified Allan deviation for frequency data
CalcPhaseOverlapSigma	Calculate Allan deviation for phase data using overlapping samples
CalcFreqOverlapSigma	Calculate Allan deviation for frequency data using overlapping samples
CalcFastModSigma	Quickly calculate modified Allan deviation for gapless phase data
CalcGreenhallModSigma	Calculate mod sigma for gapless phase data using Greenhall method

Hadamard Variance Functions	
CalcHadamardDev	Calculate the 3-sample Hadamard deviation for phase data
CalcPhaseOverlapHadamardDev	Calculate the Hadamard deviation for phase data using overlapping samples
CalcFreqOverlapHadamardDev	Calculate the Hadamard deviation for freq data using overlapping samples

Total Variance Functions	
ModTotvarCalc	

Other Stability Functions	
TIErms	
CalcMTIE	
CalcFastMTIE	
Thêo1	

Drift Functions	
CalcQuadraticDrift	Calculate least-squares quadratic fit to phase data
CalcFirstDiff	
CalcSecondDiff	Calculate average of 2nd differences of phase data
CalcThreePointDrift	Calculate frequency drift using first, middle and last phase data points
CalcLinFreqDrift	Calculate least-squares linear fit for frequency data
CalcBisection Drift	Calculate freq drift using averages of first & last halves of freq data
CalcLogFreqDrift	Calculate least-squares log fit for frequency data
RemoveQuadraticDrift	Remove quadratic (frequency) drift from phase data
RemoveLinFreqDrift	Remove linear drift from frequency data
RemoveLogFreqDrift	Remove log drift from frequency data

Conversion Functions	
ConvFreqToPhase	Convert from frequency data to phase data
ConvPhaseToFreq	Convert from phase data to frequency data
CalcDomain	Calculate the time and frequency domain noise parameter h(a)
ConvDomain	Convert between time and frequency domains
DateToMJD	Convert calendar date to Modified Julian Date
MJDToDate	Convert Modified Julian Date to calendar date

Common Statistical Functions	
FindMinMax	Find min and max of phase or frequency data
FindMedian	
CalcMean	Calculate average of phase or frequency data
CalcPhaseStdDev	Calculate the standard deviation for phase data
CalcFreqStdDev	Calculate the standard deviation for frequency data

Data Modification	
ScaleData	Scale phase or frequency data by a+bx
NormalizeData	Normalize phase or freq data to zero mean
AveragePhaseData	Do averaging of phase data
AverageFreqData	Do averaging of frequency data
FillGaps	Fill gaps in phase or frequency data
FindFreqOutliers	Find, and optionally remove, outliers in frequency data

Special Functions	
CalcDegFree	Calculate # degrees of freedom for overlapping frequency data
CalcInvChiSqr	Calculate value of chi squared for certain # of degrees of freedom
CalcChiSqrProb	Calculate the area under the chi squared distribution
CalcNormalProb	Calculate the value of the normal deviate
CountGaps	Count # gaps in phase or frequency data
FindPlotScale	Find scale factor for plotting phase or freq data
CalcBias	Calculate the value of the B1 bias function for zero deadtime & integer mu
CalcBias1	Calculate the value of the B1 bias function
CalcBias2	Calculate the value of the B2 bias function
CalcBias3	Calculate the value of the B3 bias function
CalcRatio	Calculate the value of the R(n) function

• License

A license is hereby granted for the use of this software within the immediate user group that purchased it. The FrequenC Library source and object code may be installed on any number of computers within that group, and backup copies may be made, but only one copy of the Library may be used at any time. Programs compiled using the FrequenC Library may be distributed without any royality payment, but the Library source or object code may not be distributed unless a copy of the Library is purchased for every copy that is distributed.

The FrequenC functions have been extensively tested, but it is never possible to declare software completely bug-free. No warranty is made, nor is any liability assumed, in connection with the use of this software.

• Other Licenses

Portions of the FrequenC library utilize code adapted from other sources, and the user is required to obtain licenses from those vendors, as necessary, before utilizing it. A list of those source codes and their vendors is as follows:

The Hammer Library (Version 2.40 of 3/9/1990)
O.E.S. Systems
1906 Brushcliff Road
Pittsburgh, PA 15221

IPC-TC-006 Science and Engineering Tools (Revision 8.0 June 1992)
Quinn-Curtis, Inc.
18 Hearthstone Drive
Medfield, MA 02052

Numerical Recipes in C
Cambridge University Press
40 West 20th Street
New York, NY 10011

Specific references to these source codes are made in the text headers of the applicable FrequenC library files.

• Installation

The FrequenC Library is distributed in packed form. To install it, copy the INSTALL.EXE file to the desired directory on your hard disk, and run INSTALL to unpack the Library files. The INSTALL program may then be deleted.

• Files

The FrequenC package includes source files for all functions, and a header file that contains ANSI C prototypes for the functions. These source files may be compiled into a library using the complier and memory model of your choice. Also included is test data, and a driver programs that demonstrate the use of each function.

• Data Format

The FrequenC functions use an one-dimensional array format for phase or frequency data where the first three elements hold the number of data points, and the start and end analysis limits. The data is assumed to represent equally-spaced points, with gaps denoted by the value zero. All zeros are considered gaps in frequency data, while only embedded zeros are considered gaps in phase data. The sizes of the data arrays are determined by the calling program.

• Drift Analysis Functions

A complete set of drift analysis functions are provided for the calculation of linear and logarithmic drift for frequency data, and quadratic and second-difference drift for phase data. The "Calc" functions calculate the drift parameters, and the "Remove" functions drift-correct the data. These functions are summarized in the following table:

Drift Type	Phase Data	Frequency Data

Linear	CalcThreePointDrift()	CalcLinFreqDrift()
		RemoveLinFreqDrift()
		CalcBisectionDrift()

Log		CalcLogFreqDrift()
		RemoveLogFreqDrift()

2nd-Difference	CalcSecondDiff()	

Quadratic	CalcQuadraticDrift()
	RemoveQuadraticDrift()	

The arguments of all these functions include the name of the data array (x or y). The analysis functions have arguments for pointers to the relevant drift parameters (*p_slope, *p_intercept, etc.), and a pointer to the fit variance (*p_variance). Drift parameter calculations are direct except for the logarithmic case where an iterative algorithm is used. The removal functions have parameters for the values of the drift parameters (slope, intercept, etc.) to be subtracted. Drift removal does not include calculation of these parameters.

- **Variance Functions**

A complete set of Allan, Hadamard and total variance functions are provided for the calculation of those statistics for phase or frequency data. These functions are summarized in the following table:

Variance Type	Phase Data	Frequency Data
Standard	CalcPhaseStdDev()	CalcFreqStdDev()
Normal Allan (2-sample)	CalcPhaseSigma()	CalcFreqSigma()
Overlapping Allan	CalcPhaseOverlapSigma()	CalcFreqOverlapSigma()

Modified Allan	CalcPhaseModSigma()
	CalcFastModSigma() (no gaps)
	CalcGreenhallModSigma() (no gaps)
		CalcFreqModSigma()

Time
(multiply Mod s_y(t) by t/√3)
	CalcPhaseModSigma()
	CalcFastModSigma() (no gaps)
		CalcFreqModSigma()

Hadamard (3-sample)	CalcHadamardDev()	

The arguments of these functions include the name of the data array (x or y) and a pointer to the deviation (*p_dev) or sigma (*p_sig) value. The phase data functions also have an argument for the averaging time (tau). In addition, the overlapping and modified Allan variance functions include an argument for the averaging factor (m). All these functions except CalcFastModSigma() handle gaps in the data represented by zeros. Only embedded zeros are considered gaps in phase data. All zeros are considered gaps in frequency data. The CalcFastModSigma() and CalcGreenhallModSig() functions are available for gapless phase data only. They are faster than CalcPhaseModSigma(), and much faster than CalcFreqModSigma(), especially for a large data set at a large averaging factor. The modified Allan variance for a large data set at a large averaging factor should therefore be performed with gapless phase data if possible. The ConvFreqToPhase() and FillGaps() functions may be used to allow this to be done for frequency data having gaps.

- **Progress Indicator**

An example of a calculation function prototype that includes a progress indicator is as follows:

```
int WINAPI CalcPhaseModSigma(F_TYPE x[], F_TYPE *p_sig, F_TYPE tau,
      int m, BOOL *bAbort, BOOL bProgress,
      void (*ProgressFunction)(int nPercentDone, char *szMessage));
```

The last argument is a pointer to a void function having the signature:

```
void WriteProgressMessage(int nPercentDone, char *szMessage);
```

where the first parameter is the % of the calculation that is complete, and the second parameter is a string to be displayed by the progress indicator. This function is not included in the FrequenC library because it if specific to the particular application. Both the % done and message text are generated by the calculation function.

The calculation function is called as:

```
nModSig=NewCalcPhaseModSigma(fWorkData, &fModSig, fTau*nAF, nAF,
      &bAbort, !bAllTau, WriteProgressMessage);
```

where the last argument is the name of the progress function.

- **Abort Flag**

The variance calculation functions having a progress indicator also have an argument BOOL *bAbort, a pointer to an integer flag that indicates whether the calculation should be aborted. This variable must be defined and initialized to FALSE (0) before each calculation. If it is set to TRUE (1) during a calculation, the calculation will be aborted the next time the progress indicator is updated.

- **Source Code**

C source code for the FrequenC library is not included with the Stable32 software package, but is available for separate purchase. Please contact Hamilton Technical Services for further information.

• List of Source Files

The following is a list of all the source files that comprise the FrequenC Library. The source files names directly follow their corresponding function names. Source files are included for each individual function and combined into one master source file, FREQUENC.C. The FrequenC header file, FREQUENC.H, must be included in any source file that uses the FrequenC Library. Two test programs are incuded in both source and executable form. TEST_FC tests all the FrequenC functions. CALC_FC is an interactive calculator-mode program for some of the FrequenC functions. The source file listings in this document are for informational purposes only, and the individual source files should be used for compiling the FrequenC library.

- **Master Source File:**

```
FREQUENC.C  All FrequenC Functions
```

- **Function Source Files**

AFD.C	AverageFreqData()
APD.C	AveragePhaseData()
CB.C	CalcBias()
CB1.C	CalcBias1()
CB2.C	CalcBias2()
CB3.C	CalcBias3()
CBD.C	CalcBisectionDrift()
CCSP.C	CalcChiSqrProb()
CFD.C	CalcDegFree()
CD2.C	CalcDomain()
CFMS.C	CalcFreqModSigma()
CFMS2.C	CalcFastModSigma()
CFOS.C	CalcFreqOverlapSigma()
CFS.C	CalcFreqSigma()
CFSD.C	CalcFreqStd Dev()
CFTP.C	ConvFreqToPhase()
CG.C	CountGaps()
CGMS.C	CalcGreenhallModSigma()
CHD.C	CalcHadamardDev()
CICS.C	CalcInvChiSqr()
CLFD.C	CalcLinFreqDrift()
CLFD2.C	CalcLogFreqDrift()
CM.C	CalcMean()
CNP.C	CalcNormalProb()

CPMS.C	CalcPhaseModSigma()
CPOS.C	CalcPhaseOverlapSigma()
CPS.C	CalcPhaseSigma()
CPSD.C	CalcPhaseStdDev()
CPTF.C	ConvPhaseToFreq()
CQD.C	CalcQuadraticDrift()
CR.C	CalcRatio()
CSD.C	CalcSecondDiff()
CTPD.C	CalcThreePointDrift()
CD.C	ConvDomain()
DTM.C	DateToMJD()
FFO.C	FindFreqOutliers()
FG.C	FillGaps()
FMM.C	FindMinMax()
FPS.C	FindPlotScale()
MTD.C	MJDToDate()
ND.C	NormalizeData()
RLFD.C	RemoveLinFreqDrift()
RLFD2.C	RemoveLogFreqDrift()
RQD.C	RemoveQuadraticDrift()
SD.C	ScaleData()

- **Header File**

The FREQUENC.H header file invokes several Standard C Library header files, contains several macro definitions, and gives prototype information for the FrequenC Library functions. It should be included with each of the FrequenC Library source files when they are compiled. The F_TYPE definition determines the type of floating point variable that is used for the phase and frequency data arrays and other FrequenC variables. It can be defined as either float or double.

The PHASE and FREQ definitions are useful for determining the datype parameter. The PSD_Y, PSD_X, PSD_P, PSD_L, SIG_Y, MODSIG_Y and SIG_X definitions are useful for setting the domain conversion type parameter. The W_PM, F_PM, W_FM, F_FM and RW_FM definitions are useful for setting the domain conversion alpha parameter.

- **Test Program Files**

TEST_FC.C	Source File for Test Program for all FrequenC Functions
TEST_FC.EXE	Executable FrequenC Test Program
TEST.DAT	Test Data for TEST_FC Test Program
TEST_CD.C	Source File for Test Usage of Domain Conversion Functions
TEST_CD.EXE	Executable FrequenC Domain Conversion Program
CALC_FC.C	Source File for Calculator Mode for some FrequenC Functions
CALC_FC.EXE	Executable FrequenC Calculator Program

• Reference

An excellent reference for many of the FrequenC Library functions is *NIST Technical Note 1337*, "Characterization of Clocks and Oscillators", Edited by D.B. Sullivan, D.W. Allan, D.A. Howe and F.L. Walls. Other references are given in the Stable32 User Manual and the FrequenC Library source files.

- **FrequenC.DLL Contents**

```
Dump of file FrequenC.dll

File Type: DLL

         Section contains the following Exports for FrequenC.dll

                    0 characteristics
             3E7C7282 time date stamp Sat Mar 22 09:26:10 2003
                 0.00 version
                    1 ordinal base
                   93 number of functions
                   93 number of names

             ordinal hint   name

                   1    0   _AddPSD@8  (00001190)
                   2    1   _AddSigma@4  (000013A0)
                   3    2   _AverageFreqData@8  (00001000)
                   4    3   _AveragePhaseData@8  (000012B0)
                   5    4   _BasScale@28  (00001440)
                   6    5   _BreakDate@16  (00001400)
                   7    6   _CalcBias1@12  (00001800)
                   8    7   _CalcBias2@8  (00001B00)
                   9    8   _CalcBias3@12  (00001D20)
                  10    9   _CalcBias@8  (000016F0)
                  11    A   _CalcBisectionDrift@4  (000021D0)
                  12    B   _CalcChiSqrProb@8  (000022A0)
                  13    C   _CalcDegFree@12  (00003150)
                  14    D   _CalcDiffusionFreqDrift@20  (00003440)
                  15    E   _CalcDomain@36  (00002930)
                  16    F   _CalcFastMTIE@16  (00003A20)
                  17   10   _CalcFastModSigma@20  (00003FA0)
                  18   11   _CalcFirstDiff@8  (000037A0)
                  19   12   _CalcFreqHadamardDev@8  (000038F0)
                  20   13   _CalcFreqModSigma@12  (00003DC0)
                  21   14   _CalcFreqOffset@16  (000041A0)
                  22   15   _CalcFreqOverlapHadamardDev@12  (000043B0)
                  23   16   _CalcFreqOverlapSigma@12  (00004570)
                  24   17   _CalcFreqSigma@8  (000046F0)
                  25   18   _CalcFreqStdDev@8  (000047E0)
                  26   19   _CalcGreenhallModSigma@20  (00004C10)
                  27   1A   _CalcHadamardB1@8  (00004DE0)
                  28   1B   _CalcHadamardDev@16  (000051E0)
                  29   1C   _CalcInvChiSqr@8  (00005420)
                  30   1D   _CalcLinFreqDrift@16  (00005B20)
                  31   1E   _CalcLogFreqDrift@20  (00005D10)
                  32   1F   _CalcMTIE@24  (00010170)
                  33   20   _CalcMean@12  (00006030)
                  34   21   _CalcNormalProb@4  (00006110)
                  35   22   _CalcPhaseHadamardDev@16  (000062C0)
                  36   23   _CalcPhaseModSigma@32  (00006500)
                  37   24   _CalcPhaseOverlapHadamardDev@20  (00006840)
                  38   25   _CalcPhaseOverlapSigma@20  (00006B00)
                  39   26   _CalcPhaseSigma@16  (00006D50)
                  40   27   _CalcPhaseStdDev@16  (00006F30)
                  41   28   _CalcQuadraticDrift@28  (00007600)
                  42   29   _CalcRatio@12  (000079A0)
                  43   2A   _CalcSecondDiff@8  (00007D40)
                  44   2B   _CalcStarB1@20  (00007C50)
                  45   2C   _CalcThreePointDrift@4  (00007EC0)
                  46   2D   _CheckFrequenC@0  (00003790)
                  47   2E   _ConvDomain@36  (00002510)
                  48   2F   _ConvFreqToPhase@20  (00004940)
                  49   30   _ConvPhaseToFreq@20  (000071C0)
                  50   31   _ConvPhaseToFreqUsingTimetags@28  (000073B0)
                  51   32   _CountGaps@8  (00004B90)
                  52   33   _DateConv@24  (00007F80)
                  53   34   _DateToJulian@4  (00008180)
                  54   35   _DateToMJD@12  (00008250)
                  55   36   _DayOfWeek@4  (00008090)
                  56   37   _EDF@12  (000082B0)
                  57   38   _FillFloatGaps@8  (00008310)
                  58   39   _FillGaps@8  (00008670)
                  59   3A   _FindFreqOutliers@12  (00008580)
                  60   3B   _FindMedian@12  (000088E0)
                  61   3C   _FindMinMax@16  (00009010)
                  62   3D   _FindPlotScale@24  (00009170)
                  63   3E   _GenNoise@24  (00009580)
                  64   3F   _HadTotvarBias@4  (0000B180)
                  65   40   _HadTotvarCalc@24  (0000B210)
                  66   41   _HadTotvarEDF@8  (0000B7F0)
                  67   42   _HadamardEDF@12  (0000A1A0)
                  68   43   _HistoCalc@24  (0000A070)
                  69   44   _JulianToDate@4  (0000B8F0)
                  70   45   _MJDToDate@16  (0000FF20)
                  71   46   _MJDtoDOY@4  (0000FFE0)
                  72   47   _MJDtoGPS@4  (00010140)
                  73   48   _MakeDate@12  (0000BCD0)
                  74   49   _MedDev@8  (0000BA30)
                  75   4A   _ModTotvarBias@4  (0000F9F0)
                  76   4B   _ModTotvarCalc@32  (0000FA80)
                  77   4C   _ModTotvarEDF@8  (00010060)
                  78   4D   _MultiTaperSpectrumCalc@52  (0000BDA4)
                  79   4E   _NoiseID@20  (000103D0)
                  80   4F   _NormalizeData@16  (00010320)
                  81   50   _RemoveDiffusionFreqDrift@28  (00010B20)
                  82   51   _RemoveFreqOffset@24  (00010BD0)
                  83   52   _RemoveLinFreqDrift@20  (00010CC0)
                  84   53   _RemoveLogFreqDrift@28  (00010D50)
                  85   54   _RemoveQuadraticDrift@28  (00010E00)
                  86   55   _RoundAxes@12  (00010770)
                  87   56   _ScaleData@24  (00011FE0)
                  88   57   _SpectrumCalc@44  (00010ED0)
                  89   58   _TIErms@12  (000125D0)
                  90   59   _TotalvarBias@20  (000120B0)
                  91   5A   _TotvarBias@8  (00012170)
                  92   5B   _TotvarCalc@24  (00012220)
                  93   5C   _TotvarEDF@8  (00012540)

      Summary

         6000 .data
         1000 .idata
         3000 .rdata
         3000 .reloc
        1D000 .text
```

The FrequenC Library

NAME: AverageFreqData	**FUNCTION:** Average frequency data by an integer factor
SYNOPSIS: `int AverageFreqData(F_TYPE y[], int af)`	

F_TYPE y[]	Frequency data array:
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

int af	Averaging factor
RETURN: int	The # of data points in averaged data, or -1 if error

REMARKS:

All data points are processed regardless of analysis limits.
Averaging factor must be >0 and ≤ # data points.
Averaging Factor=1 does nothing.
All zeros are treated as gaps in frequency data.

EXAMPLE:

```
#include "frequenc.h"                           /* FrequenC header file */
F_TYPE y[512];                                  /* frequency data array */
int af;                                             /* averaging factor */
int num;                                   /* # points in averaged data */
.
.
af=2;                              /* assign value to averaging factor */
num=AverageFreqData(y, af);                       /* average freq data */
if(num==-1)                                         /* check for error */
{
    printf("\nError");                                /* error message */
}
else
{
    printf("\n# Averaged Data Points = %d", num);   /* display num */
}
```

SEE ALSO: AveragePhaseData()

REFERENCE: None

```
/****************************************************************************/
/*                                                                          */
/*                             AverageFreqData()                            */
/*                                                                          */
/*      Function to do averaging of frequency data                          */
/*                                                                          */
/*      Parameters:     F_TYPE y[] = frequency data                         */
/*                                   y[0] = # data points                   */
/*                                   y[1] = analysis start                  */
/*                                   y[2] = analysis end                    */
/*                      int    af  = averaging factor                       */
/*                                 = # points to combine into each average  */
/*                                                                          */
/*      Return:         int        = # data points in averaged data,        */
/*                                   or -1 if error                         */
/*                                                                          */
/*      Note:           All data points processed regardless of analysis    */
/*                      limits.  Zeros are treated as gaps in freq data.    */
/*                      Error if af<1 or af> # original data points.        */
/*                      af=1 is OK, and results in no change in data.       */
/*                                                                          */
/*      Revision record:                                                    */
/*          04/11/97    Cloned from AverageFrequencyData() of FrequenC Lib  */
/*                      Changed to use analysis limits                      */
/*          01/01/98    Changes for MS VC++ compatibility & warnings        */
/*          04/21/03    Adapted for Version 2.0 source code documentation   */
/*                                                                          */
/* (c) Copyright 1992-2003 Hamilton Technical Services All Rights Reserved  */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL AverageFreqData(F_TYPE y[], int af)
{
    int i;
    int j;
    int k=3;
    int n=(int)(y[2]-y[1]+1);                     /* # original data points */
    int num;
    double sum;

    if( (af<1) || (af>n) )
    {
        return(-1);                                               /* error */
    }

    for(i=(int)(y[1]+2); i+af-1<=(int)(y[2]+2); i+=af)
    {
        sum=0;
        num=af;
        for(j=0; j<af; j++)
        {
            if(*(y+i+j))                             /* don't average in a gap */
            {
                sum+=*(y+i+j);
            }                                                        /* end if */
            else
            {
                num--;
            }                                                      /* end else */
        }                                                           /* end for */
        if(num)                                          /* check if all gaps */
        {
            *(y+k++)=sum/num;
        }                                                            /* end if */
        else
        {
            *(y+k++)=0;
        }                                                          /* end else */
    }                                                               /* end for */

    *y=floor(n/af);
    *(y+1)=1.0;
    *(y+2)=*y;

    return (int)*y;                      /* # data points in averaged data */
}                                                  /* end AverageFreqData() */
```

The FrequenC Library	
NAME: AddPSD	**FUNCTION:** Add freq domain noise terms
SYNOPSIS: `float AddPSD(float PSD[], int PSDtype)`	
`float PSD[]`	Array of PSD terms
`int PSDtype`	PSD type macro (e.g. PSD_L)
RETURN: `float`	Sum of PSD noise terms
REMARKS: Used with CalcDomain() and ConvDomain(). See domain conversion type macros in frequenc.h	

EXAMPLE:

```
#include "frequenc.h"                         /* FrequenC header file */
float PSD[5];                                           /* PSD array */
float sum;                                         /* sum of PSD terms */
.
.
sum=AddPSD(PSD, PSD_L);                               /* call function */
printf("\nPSD Sum = %e", sum);                       /* display result */
```

SEE ALSO: CalcDomain(), ConvDomain(), AddSigma()

REFERENCE:

```
/******************************************************************************/
/*                                                                            */
/*                           AddPSD()                                         */
/*                                                                            */
/*      Function to add freq domain noise terms                               */
/*                                                                            */
/*      Revision record:                                                      */
/*          05/03/97    Adapted from AddPSD() of STABLE_A.C                   */
/*          01/01/98    Changes for MS VC++ compatibility & warnings          */
/*          04/21/03     Adapted for Version 2.0 source code documentation       */
/*                                                                            */
/* (c) Copyright 1992-2003 Hamilton Technical Services All Rights Reserved   */
/*                                                                            */
/******************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL AddPSD(float fPSD[], int nPSDType)
{
    // Local variables
    int i;                      // Index
    float fNoise[5];            // So(f) values
    float fAll;                 // Total So(f)

    if(nPSDType==PSD_L)
    {
        // Convert L(f) log values to So(f) numerical ones before adding
        // So{f)=2*10^[L(f)/10]
        for(i=0; i<5; i++)
        {
            if(fPSD[i])
            {
                fNoise[i]=(float)(2*(pow(10,fPSD[i]/10)));
            }
            else
            {
                fNoise[i]=0.0;
            }
        }

        // Sum noise terms
        fAll=fNoise[0]+fNoise[1]+fNoise[2]+fNoise[3]+fNoise[4];

        // Convert sum to L(f)
        if(fAll)
        {
            return (float)(10*log10(fAll/2));
        }
        else
        {
            return 0.0;
        }
    }
    else
    {
        return (float)(sqrt(fPSD[0]*fPSD[0]+fPSD[1]*fPSD[1]+fPSD[2]*fPSD[2]+
            fPSD[3]*fPSD[3]+fPSD[4]*fPSD[4]));
    }
}
```

The FrequenC Library

NAME: AveragePhaseData	**FUNCTION:** Average phase data by an integer factor

SYNOPSIS: int AveragePhaseData(F_TYPE x[], int af)

F_TYPE x[]	Phase data array:
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

int af	Averaging factor
RETURN: int	The # of data points in averaged data, or -1 if error

REMARKS:
All data points are processed regardless of analysis limits.
Averaging factor must be >0 and ≤ # data points.
Averaging factor = 1 does nothing.
Embedded zeros are treated as gaps in phase data.
Averaging of phase data simply omits the intermediate points (i.e decimates the data).

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE x[512];                                   /* phase data array */
int af;                                          /* averaging factor */
int num;                                /* # points in averaged data */
.
.
af=2;                           /* assign value to averaging factor */
num=AveragePhaseData(x, af);                   /* average phase data */
(if num==-1)                                      /* check for error */
{
    printf("\nError");                              /* error message */
}
else
{
    printf("\n# Averaged Data Points = %d", num);   /* display num */
}
```

SEE ALSO: AverageFreqData()

REFERENCE: None

```
/****************************************************************************/
/*                                                                          */
/*                           AveragePhaseData()                             */
/*                                                                          */
/*      Function to average phase data by an integer factor                 */
/*                                                                          */
/*      Parameters:     float x[] = phase data array                        */
/*                                  x[0] = # data points                    */
/*                                  x[1] = analysis start                   */
/*                                  x[2] = analysis end                     */
/*                      int af    = averaging factor                        */
/*                                                                          */
/*      Return:         int       = # points in averaged data,              */
/*                                  or -1 if error                          */
/*                                                                          */
/*      Note:           All data points are processed regardless of         */
/*                      analysis limits.  Averaging phase data by           */
/*                      factor=af is done by simply using only every        */
/*                      (af)th data point.  A gap is retained if it is      */
/*                      one of the averaged points.  Error if af<1 or       */
/*                      af> # original data points.  af=1 is OK, and        */
/*                      results in no change in the data.                   */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/30/91    Created                                             */
/*          12/31/91    Renamed                                             */
/*          01/16/92    Edited per FrequenC documentation                   */
/*          01/17/92    Edited title block                                  */
/*                      Changed return to int=# points processed            */
/*          01/18/92    Added error trap.                                   */
/*          02/03/96    Modified for use as Win 3.1 DLL                     */
/*          04/21/03    Adapted for Version 2.0 source code documentation  */
/*                                                                          */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL AveragePhaseData(F_TYPE x[], int af)

{
    int num=(int)x[0];      /* # original phase data points to be averaged */
    int i;                                 /* index for averaged phase data */
             /* i goes from 1 to ((num-1)/af)+1 averaged phase data points */
    int j;                                 /* index for original phase data */
                                   /* j starts at the analysis start offset */
          /* which is 1 for original phase data starting at 1st data point */
                             /* j is incremented by the averaging factor af */

    if( (af<1) || (af>num) )
    {
        return(-1);                                              /* error */
    }

    j=1;
    for(i=1; i<=((num-1)/af)+1; i++)
    {
        *(x+i+2) = *(x+j+2);                /* 2 is the data array offset */
        j += af;
    }                                                           /* end for */

    x[0]=((num-1)/af)+1;
    x[1]=1;
    x[2]=x[0];

    return( (int) x[0]);

}                                                 /* end AveragePhaseData() */
```

The FrequenC Library	
NAME: AddSigma	**FUNCTION:** Add time domain noise terms
SYNOPSIS: float AddSigma(float sigma[])	
Float sigma[]	
RETURN: float	Sum of sigma noise terms
REMARKS: Used with CalcDomain() and ConvDomain().	

EXAMPLE:

```
#include "frequenc.h"                           /* FrequenC header file */
float sigma[5];                                          /* sigma array */
float sum;                                        /* sum of sigma terms */
.
.
sum=AddSigma(sigma);                                   /* call function */
printf("\nSigma Sum = %e", sum);                      /* display result */
```

SEE ALSO: CalcDomain(), ConvDomain(), AddPSD()

REFERENCE:

```
/****************************************************************************/
/*                                                                          */
/*                            AddSigma()                                    */
/*                                                                          */
/*      Function to add time domain noise terms                             */
/*                                                                          */
/*      Revision record:                                                    */
/*          05/03/97    Adapted from AddSigma() of STABLE_A.C               */
/*          01/01/98    Changes for MS VC++ compatibility & warnings        */
/*          04/21/03     Adapted for Version 2.0 source code documentation     */
/*                                                                          */
/* (c) Copyright 1992-2003 Hamilton Technical Services All Rights Reserved  */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL AddSigma(float fSigma[])
{
    return (float)(sqrt(fSigma[0]*fSigma[0]+fSigma[1]*fSigma[1]+fSigma[2]*fSigma[2]+
        fSigma[3]*fSigma[3]+fSigma[4]*fSigma[4]));
}
```

The FrequenC Library

NAME: BreakDate	**FUNCTION:** Break down Gregorian date into month, day and year
SYNOPSIS: void BreakDate(long gdate, int *month, int *day, int *year)	
long gdate	Gregorian date (yyyymmdd)
int *month	Pointer to month (1-12)
int *day	Pointer to day (1-31)
int *year	Pointer to year (4-digit, e.g. 2003)
RETURN: void	

REMARKS:

EXAMPLE:

```
 #include "frequenc.h"                           /* FrequenC header file */
 long gdate;                                           /* Gregorian date */
 int month;                                                     /* month */
 int day;                                                         /* day */
 int year;                                                       /* year */
 .
 .
 gdate=20030101;                                             /* set 1date */
 BreakDate(gdate, &month, &day, &year);                 /* call function */
 printf("\nMonth = %d, Day = %d, Year = %d", month, day, year);
                                                      /* display results
*/
```

SEE ALSO: MakeDate()

REFERENCE:

```
/****************************************************************************/
/*                                                                          */
/*                              BreakDate()                                 */
/*                                                                          */
/*      Function to break down a Gregorian date into month, day and year    */
/*                                                                          */
/*      Parameters: long    gdate   Gregorian date (yyyymmdd)               */
/*                  int *   mo      month (1-12)                            */
/*                  int *   da      day   (1-31)                            */
/*                  int *   yr      year  (4-digit)                         */
/*                                                                          */
/*      Return:     void            None                                    */
/*                                                                          */
/*      Revision record:                                                    */
/*          07/17/97    Cloned from brkdate() of Hammer Library             */
/*                      Modified code so full 4-digit year is returned      */
/*          04/21/03     Adapted for Version 2.0 source code documentation     */
/*                                                                          */
/* (c) Copyright 1997-2003 Hamilton Technical Services All Rights Reserved   */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"                         /* FrequenC Library header file */

void __declspec(dllexport) FAR PASCAL BreakDate(LONG gdate, int *mo, int *da,
    int *yr)
{
    *yr=(int) (gdate/10000L);
    *mo=(int) ((gdate/100L) % 100L);
    *da=(int) (gdate%100L);
}
```

The FrequenC Library

NAME: BasScale	**FUNCTION:** Find neat plot scale
SYNOPSIS: int BasScale (double min, double max, int n, double* smin, double* step)	
double min	minimum data value
double max	maximum data value
int n	maximum # of divisions in plot scale
double *smin	pointer to scale minimum
double *step	pointer to step size
RETURN: int	1 if OK, or -1 if error

REMARKS:
This function is recommended for a small relative data range, even though its scale choice is often sub-optimum, and the # of minor tics is not always 5.
Written by Antonio Gómiz Bas based on subroutine SCALE by J.A. Nelder (1976) and W. Douglas Stirling (1981).

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE y[512];                                      /* freq data array */
int n=10;                                               /* # plot divs */
double min, max;                                     /* data min & max */
double smin, step;                             /* scale min & step size */
.
.
FindMinMax(y, &min, &max, FREQ_DATA);            /* get data min & max */
BasScale(min, max, n, &smin, &step);                 /* get plot scale */
printf("\nMin = %e, Step = %e", smin, step);      /* display results */
```

SEE ALSO: FindPlotScale(), RoundAxes()

REFERENCE: A. Bas, "Finding Neat Scales for Plotting", C Users Journal, March 2000

```
/*****************************************************************************/
/*                                                                           */
/*                              BasScale()                                   */
/*                                                                           */
/*      Function to find neat plot scale                                     */
/*      Ref: A. Bas, "Finding Neat Scales for Plotting", CUJ, March 2000     */
/*                                                                           */
/*      Parameters:    double XMin          = minimum data value             */
/*                     double XMax          = maximum data value             */
/*                     int N                = max # divs in plot scale       */
/*                     double *SMin         = pointer to scale minimum       */
/*                     double *Step         = pointer to step size           */
/*                                                                           */
/*      Return:        int                  = 1 if OK or 0 if error          */
/*                                                                           */
/*      Written by Antonio Gómiz Bas, 1999 based on subroutine SCALE         */
/*      by W. Douglas Stirling                                               */
/*                                                                           */
/*      Revision record:                                                     */
/*          03/09/02   Downloaded and adapted                                */
/*                     Runs AOK and seems to produce nice tight scales       */
/*          03/19/03   Adapted for FrequenC DLL                              */
/*          04/26/03   Adapted for Version 2.0 source code documentation     */
/*                                                                        */
/* (c) Copyright 2002-3 Hamilton Technical Services All Rights Reserved   */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

int  __declspec(dllexport) FAR PASCAL BasScale (double XMin, double XMax,
    int N, double* SMin, double* Step)
{
  int    iNegScl;       //  Negative scale flag
  int    iNm1;          //  Number of scale subintervals
  double lfIniStep;     //  Initial step
  double lfSclStep;     //  Scaled step
  double lfTmp;         //  Temporary value
  double lfSclFct;      //  Scaling back factor
  double lfSMax;        //  Scale maximum value
  int    it;            //  Iteration counter
  int    i;             //  Neat step counter

  //  Neat steps

  //  Original version had these steps
  int Steps [] = {10, 12, 15, 16, 20, 25, 30, 40, 50, 60, 75, 80, 100, 120, 150};

  //  Try reduced choices for use in case of narrow range
  //  int Steps [] = {10, 20, 50, 100};

  int iNS = sizeof (Steps) / sizeof (Steps [0]);

  //  Some checks
  if (XMin >  XMax)
    {
      lfTmp = XMin;
      XMin  = XMax;
      XMax  = lfTmp;
    }
  if (XMin == XMax) XMax = XMin == 0.0 ? 1.0 : XMin + fabs (XMin) / 10.0;

  //  Reduce to positive scale case if possible
  if (XMax <= 0)
    {
      iNegScl = 1;
      lfTmp   = XMin;
      XMin    = -XMax;
      XMax    = -lfTmp;
    }
  else
    iNegScl = 0;

  if (N < 2) N = 2;
  iNm1 = N - 1;

  for (it = 0; it < 3; it++)
    {
      //  Compute initial and scaled steps
      lfIniStep = (XMax - XMin) / iNm1;
      lfSclStep = lfIniStep;

      for (; lfSclStep <  10.0; lfSclStep *= 10.0);
      for (; lfSclStep > 100.0; lfSclStep /= 10.0);

      //  Find a suitable neat step
      for (i = 0; i < iNS && lfSclStep > Steps [i]; i++);
      lfSclFct = lfIniStep / lfSclStep;

      //  Compute step and scale minimum value
      do
        {
          *Step  = lfSclFct * Steps [i];
          *SMin  = floor (XMin / *Step) * *Step;
          lfSMax = *SMin + iNm1 * *Step;
          if (XMax <= lfSMax)           //  Function maximum is in the
                                        //  range: the work is done.
            {
              if (iNegScl) *SMin = -lfSMax;
              *Step *= iNm1 / (N - 1);
              return 1;
            }
          i++;
        }
      while (i < iNS);

      //  Double number of intervals
      iNm1 *= 2;
    }

  //  Could not solve the problem
  return 0;
}
```

The FrequenC Library

NAME: CalcBias	**FUNCTION:** Calculate the B1 bias function for zero dead time and integer power law noise slope

SYNOPSIS: `float CalcBias(int num, int mu)`

int num	The # of data points
int mu	Log-log slope, μ, of Allan variance, $s^2_y(t)$, for a power law noise process:

Noise Type	μ
Flicker Walk FM	2
Random Walk FM	1
Flicker FM	0
White FM	-1
Flicker PM	-2
White PM	-2

RETURN: float	The value of the $B_1(N, r, \mu)$ bias function, the ratio of the standard variance to the Allan variance for zero dead time (r=1) and integer μ power law $s^2_y(t)$ noise process, or -1.0 if error.

REMARKS:
The value of mu must be an integer between -2 and +2.

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
float b1;                                 /* value of B1 bias function */
int num;                                               /* # data points */
int mu;                                         /* power law noise slope */
.
.
num=(int) x[0];                               /* set num = # data points */
mu=0;                                         /* set mu for noise type */
b1=CalcBias(num, mu);                                     /* calc B1 */
if(b1==-1.0)                                        /* check for error */
{
    printf("\nError");                                  /* error message */
}
else
{
    printf("\nB1 = %e", b1);                               /* display b1 */
}
```

SEE ALSO: CalcBias1()

REFERENCE: NIST Technical Note 1337

```
/****************************************************************************/
/*                                                                          */
/*                             CalcBias()                                   */
/*                                                                          */
/*      Function to calculate bias function B1(N, r=1, æ), the ratio       */
/*      of the standard variance to the Allan variance for N samples,      */
/*      zero dead-time, and integer log sigma vs log tau slope æ.          */
/*                                                                          */
/*      Parameters:    int num   = # samples                               */
/*                     int mu    = sigma-tau slope (-2 to +2)               */
/*                                                                          */
/*      Return:        float     = B1 bias function value,                 */
/*                                 or -1 if error                           */
/*                                                                          */
/*      Revision Record:                                                    */
/*          11/16/90   Created                                              */
/*          12/31/91   Renamed                                              */
/*          01/05/92   Changed error code to -1                             */
/*                     Added error traps                                    */
/*          01/18/92   Edited title block                                   */
/*          02/03/96   Modified for use as Win 3.1 DLL                      */
/*          01/01/98   Changes for MS VC++ compatibility & warnings         */
/*          04/26/03   Adapted for Version 2.0 source code documentation   */
/*                                                                          */
/* (c) Copyright 1990-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL CalcBias(int num, int mu)
{
    float n=(float)num;
    float b;

    if(n<=1)
    {
        return(-1);                                         /* error code */
    }

    switch(mu)
    {
        case -2:                                  /* white or flicker PM */
        {
            b=((n*n)-1)/((float)(1.5)*n*(n-1));
            break;
        }
        case -1:                                            /* white FM */
        {
            b=1;
            break;
        }
        case 0:                                            /* flicker FM */
        {
            b=(n*(float)log(n))/(2*(n-1)*(float)log(2));
            break;
        }
        case 1:                                          /* random walk FM */
        {
            b=n/2;
            break;

        }
        case 2:                      /* frequency drift or flicker walk FM */
        {
            b=(n*(n+1))/6;
            break;
        }
        default:                                          /* unallowed mu */
        {
            b=-1;                                           /* error code */
            break;
        }
    }                                                       /* end switch */

    return(b);
}                                                  /* end CalcBias() */
```

The FrequenC Library

NAME: CalcBias1	**FUNCTION:** Calculate the B1 bias function
SYNOPSIS: float CalcBias1(int num, float r, float mu)	
int num	The # of data points
float r	Dead time ratio; r=T/t
float mu	Log-log slope, μ, of Allan variance, $s^2_y(t)$, for a power law noise process:
RETURN: float	The value of the $B_1(N, r, μ)$ bias function, the ratio of the standard variance to the Allan variance for dead time ratio r and power law $s^2_y(t)$ noise exponent μ, or -1.0 if error.

Noise Type	μ
Flicker Walk FM	2
Random Walk FM	1
Flicker FM	0
White FM	-1
Flicker PM	-2
White PM	-2

REMARKS:
The value of mu must be between -2 and +2.

EXAMPLE:
```
#include "frequenc.h"                        /* FrequenC header file */
float b1;                                  /* value of B1 bias function */
float r;                                         /* dead time ratio */
float mu;                                 /* power law noise exponent */
int num;                                          /* # data points */
.
.
num=(int) x[0];                            /* set num = # data points */
r=1.1;                                     /* set r = dead time ratio */
mu=0.0;                                      /* set mu for noise type */
b1=CalcBias(num, r, mu);                                  /* calc B1 */
if(b1==-1.0)                                      /* check for error */
{
    printf("\nError");                              /* error message */
}
else
{
    printf("\nB1 = %e", b1);                           /* display b1 */
}
```

SEE ALSO: CalcBias()

REFERENCE: NIST Technical Note 1337

```
/***************************************************************************/
/*                                                                         */
/*                              CalcBias1()                                */
/*                                                                         */
/*      Function to calculate bias function B1(N, r, æ) = ratio of         */
/*      standard to Allan variance as a function of N = # samples,  r =    */
/*      T/ç = dead time ratio and æ = power law noise exponent of åy(ç).   */
/*      r = 1 for no dead time.  B1 = 1 for N = 2.                         */
/*                                                                         */
/*      References:    (1) NBS Technical Note 375, J.A. Barnes, "Tables    */
/*                         of Bias Functions B1 and B2, for Variances      */
/*                         Based on Finite Samples of Processes with       */
/*                         Power Law Spectral Densities", January, 1969.   */
/*                     (2) NIST Technical Note 1337, TN-296 to TN-335,     */
/*                         J.A. Barnes and D.W. Allan, "Variances Based    */
/*                         on Data with Dead Time Between the              */
/*                         Measurements".                                  */
/*                     (3) J.A. Barnes, "The Analysis of Time and          */
/*                         Frequency Data", Austron, Inc, December, 1991.  */
/*                                                                         */
/*      Note:          Faster direct solution CalcBias() function          */
/*                     available for r=1 zero dead time, int æ case.       */
/*                                                                         */
/*      Parameters:    int num = # samples (tested for 4 to 1024)          */
/*                               OK for 2 to INT_MAX, large # takes longer */
/*                     float r = dead time ratio (tested for.001 to 8192)  */
/*                               Bounded to 1e-4 < r < 1e6 to avoid O/F    */
/*                     float mu= noise type (tested for -2 to +2)          */
/*                               Get -1 error code for mu < -2 or mu > 2   */
/*                                                                         */
/*      Return:        float   = B1 bias function value,                   */
/*                               or -1 if error                            */
/*                                                                         */
/*      Revision record:                                                   */
/*          12/22/91   Drafted & debugged                                  */
/*          12/23/91   Added more bounds traps                             */
/*          12/24/91   Added traps for special cases                       */
/*          12/31/91   Renamed                                             */
/*          01/05/92   More error trapping                                 */
/*          01/16/92   Edited per FrequenC documentation                   */
/*          01/18/92   Edited title block                                  */
/*          02/03/96   Modified for use as Win 3.1 DLL                     */
/*          01/01/98   Changes for MS VC++ compatibility & warnings        */
/*          04/26/03   Adapted for Version 2.0 source code documentation   */
/*                                                                         */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                         */
/***************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL CalcBias1(int num, float r, float mu)
{
    double f;                                                   /* term f */
    double p;                                             /* product term */
    double s;                                           /* summation term */
    int i;                                             /* summation index */
```

```
if(num<=1)                                         /* NG if # samples < 2 */
{
    return(-1.0);                                          /* error code */

}
r=(float)fabs(r);                                   /* avoid negative ratio */

if((mu<-2)||(mu>2))                                   /* avoid æ<-2 or æ>2 */
{
    return(-1.0);                                          /* error code */
}                                                              /* end if */

if(num==2)                                        /* check for special case */
{
    return(1.0);
}                                                              /* end if */

if(mu==2)                                         /* check for special case */
{
    p=num;                    /* done in two steps to avoid int overflow */
    return(float)((p*(num+1))/6);
}                                                              /* end if */

if((r==1)&&(mu==1))                               /* check for special case */
{
    return(float)(num/2);
}                                                              /* end if */

if((mu==-1)&&(r>=1))                              /* check for special case */
{
    return(1.0);
}                                                              /* end if */

if((mu==-2)&&(r!=0)&&(r!=1))                      /* check for special case */
{
    return(1.0);
}                                                              /* end if */

if((mu==0)||(mu==-2))                                /* check for æ=0 or -2 */
{
    mu+=(float)1e-6;                                /* to avoid singularity */
}                                                              /* end if */

if(r<.0001)                                          /* check for r near 0 */
{
    r=(float)1e-4;                                  /* to avoid singularity */
}                                                              /* end if */

if(r>1e6)                                        /* check for very large r */
{
    r=1e6;                                             /* to avoid overflow */
}                                                              /* end if */

p=num;                                                    /* initialize p */
p*=(num-1);                   /* done in two steps to avoid int overflow */
s=0;                                              /* initialize sum to zero */
```

```
    for(i=1; i<num; i++)                                   /* do summation */
    {
        f=2*pow(i*r, mu+2) - pow((i*r)+1, mu+2) - pow(fabs((i*r)-1), mu+2);
        s+=(f*(num-i))/p;
    }                                                           /* end for */

    f=2*pow(r, mu+2) - pow(r+1, mu+2) - pow(fabs(r-1), mu+2);

    return(float)((1+s)/(1+(f/2)));
}                                                      /* end CalcBias1() */
```

The FrequenC Library	
NAME: CalcBias2	**FUNCTION:** Calculate the B2 bias function
SYNOPSIS: `float CalcBias2(float r, float mu)`	
float r	Dead time ratio; r=T/t
float mu	Log-log slope, µ, of Allan variance $s^2_y(t)$ for a power law noise process: Noise Type / µ Flicker Walk FM 2 Random Walk FM 1 Flicker FM 0 White FM -1 Flicker PM -2 White PM -2
RETURN: float	The value of the $B_2(r, \mu)$ bias function, the ratio of the Allan variance with dead time to the Allan variance without dead time for dead time ratio r and power law $s^2_y(t)$ noise exponent µ, or -1.0 if error.

REMARKS:
The value of mu must be between -2 and +2.

EXAMPLE:
```
#include "frequenc.h"                        /* FrequenC header file */
float b2;                                /* value of B2 bias function */
float r;                                          /* dead time ratio */
float mu;                                /* power law noise exponent */
.
.
r=1.1;                                     /* set r = dead time ratio */
mu=0.0;                                      /* set mu for noise type */
b2=CalcBias(r, mu);                                      /* calc B2 */
if(b2==-1.0)                                        /* check if error */
{
    printf("\nError");                              /* error message */
}
else
{
    printf("\nB2 = %e", b2);                           /* display b2 */
}
```

SEE ALSO: CalcBias3()

REFERENCE: NIST Technical Note 1337

```
/*****************************************************************************/
/*                                                                           */
/*                              CalcBias2()                                  */
/*                                                                           */
/*      Function to calculate the bias function B2(r, u) = ratio of          */
/*      Allan variance with dead time to Allan variance without dead time    */
/*      as a function of r and u, where r = T/tau = dead time ratio and      */
/*      u = power law noise exponent of sigma-tau.                           */
/*      r = 1 for no dead time and B2(1, u) = 1.                             */
/*                                                                           */
/*      References:    (1) NBS Technical Note 375, J.A. Barnes, "Tables      */
/*                         of Bias Functions B1 and B2 ...", Jan. 1969.      */
/*                     (2) NIST Technical Note 1337, TN-296 to TN-335,       */
/*                         J.A Barnes and D.W. Allan, "Variances Based on    */
/*                         Data with Dead Time Between the Measurements".    */
/*                     (3) J.A. Barnes, "The Analysis of Time and            */
/*                         Frequency Data", Austron, Inc, December, 1991.    */
/*                                                                           */
/*      Parameters:    float r =  dead time ratio                            */
/*                                Tested for.0001 to 4096)                   */
/*                                Bounded to 1e-4 < r < 1e6 to avoid O/F     */
/*                     float mu=  noise type (mu = -2 to +2)                 */
/*                                Get -1 error code for mu < -2 or mu > 2    */
/*                                                                           */
/*      Return:        float   =  B2 bias function value,                    */
/*                                or -1 if error                             */
/*                                                                           */
/*      Revision record:                                                     */
/*          09/21/02   Copied from CalcBias2() of FrequenC library           */
/*          09/22/02   Fixed error in F PM case                              */
/*          03/08/03   Brought into Ver 2.0 of FrequenC lib                  */
/*          04/26/03   Adapted for Version 2.0 source code documentation     */
/*                                                                           */
/* (c) Copyright 1991-2003  Hamilton Technical Services  All Rights Reserved */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL CalcBias2(float r, float mu)
{
    double f;                                                   /* term f */
    double d;                                         /* denominator term */

    r=(float)fabs(r);                               /* avoid negative ratio */

    if((mu<-2)||(mu>2))                                /* avoid æ<-2 or æ>2 */
    {
        return(-1.0);                                       /* error code */
    }                                                           /* end if */

    if(r==0)                                     /* no need to calc for r=0 */
    {
        return(0.0);
    }                                                           /* end if */

    if(r==1)                                     /* no need to calc for r=1 */
    {
        return(1.0);
    }                                                           /* end if */
    if(mu==2)                                        /* special case for æ=2 */
    {
        return(r*r);
    }                                                            /* end if */

    if((mu==1)&&(r>=1))                      /* special case for æ=2 and rò1 */
    {
        return(((3*r)-1)/2);
    }                                                            /* end if */

    if((mu==-1)&&(r>=0))                    /* special case for æ=-1 and rò0 */
    {
        if(r<=1)
        {
            return(r);
        }                                                        /* end if */
        else
        {
            return(1.0);
        }                                                      /* end else */
    }                                                            /* end if */

    if(mu==-2)                       /* special case for æ=-2 and r!=0 or 1 */
    {
        return((float)(2.0/3.0));
    }                                                            /* end if */

    if(mu==0)                                              /* check for æ=0 */
    {
        mu+=(float)1e-6;                             /* to avoid singularity */
    }                                                            /* end if */

    if(r<.0001)                                        /* check for r near 0 */
    {
        r=(float)1e-4;                               /* to avoid singularity */
    }                                                            /* end if */

    if(r>1e6)                                      /* check for very large r */
    {
        r=1e6;                                          /* to avoid overflow */
    }                                                            /* end if */


    f=2*pow(r, mu+2) - pow(r+1, mu+2) - pow(fabs(r-1), mu+2);
    d=2*(1-pow(2, mu));

    return(float)((1+(f/2))/d);
}                                                       /* end CalcBias2() */
```

The FrequenC Library

NAME: CalcBias3	**FUNCTION:** Calculate the B3 bias function
SYNOPSIS: float CalcBias3(int m, float r, float mu)	
int m	The # of averages
float r	Dead time ratio; r=T/t
float mu	Log-log slope, μ, of Allan variance, $s^2_y(t)$ for a power law noise process: Noise Type — μ Flicker Walk FM — 2 Random Walk FM — 1 Flicker FM — 0 White FM — -1 Flicker PM — -2 White PM — -2
RETURN: float	The value of the B_3(m, r, μ) bias function, the ratio of the Allan variance with distributed dead times to the Allan variance with all dead time at end as a function of # averages m, dead time ratio r and power law $s^2_y(t)$ noise exponent μ, or -1.0 if error.

REMARKS:
The value of mu must be between -2 and +2.

EXAMPLE:
```
#include "frequenc.h"                           /* FrequenC header file */
float b3;                                    /* value of B3 bias function */
float r;                                                /* dead time ratio */
float mu;                                    /* power law noise exponent */
int m;                                                     /* # averages */
.
.
m=2;                                                /* set m = # averages */
r=1.1;                                         /* set r = dead time ratio */
mu=0.0;                                          /* set mu for noise type */
b3=CalcBias(m, r, mu);                                        /* calc B3 */
if(b3==-1.0)                                          /* check for error */
{
    printf("\nError");                                  /* error message */
}
else
{
    printf("\nB3 = %e", b3);                              /* display b3 */
}
```

SEE ALSO: CalcBias2()

REFERENCE: NIST Technical Note 1337

```
/*****************************************************************************/
/*                                                                           */
/*                              CalcBias3()                                  */
/*                                                                           */
/*      Function to calculate bias function B3(2, M, r, u) =  ratio of       */
/*      Allan variance with distributed dead times to Allan variance         */
/*      with all dead time at end as a function of M=# averages,             */
/*      r=T/tau=dead time ratio and u=power law noise exponent of sigma-tau. */
/*      r=1 for no dead time.  B1=1 for N=2.                                 */
/*      Use product of B2 and B3 for averaged data with dead time.        */
/*                                                                        */
/*      References:    (1) NBS Technical Note 375, J.A. Barnes, "Tables      */
/*                         of Bias Functions B1 and B2 ...", Jan. 1969.      */
/*                     (2) NIST Technical Note 1337, TN-296 to TN-335,       */
/*                         J.A. Barnes and D.W. Allan, "Variances Based      */
/*                         on Data with Dead Time Between the                */
/*                         Measurements".                                    */
/*                     (3) J.A. Barnes, "The Analysis of Time and            */
/*                         Frequency Data", Austron, Inc, Dec. 1991.         */
/*                                                                           */
/*      Parameters:    int m   = # averages (tested for 1 to 1024)           */
/*                               OK for 1 to INT_MAX, large n takes longer   */
/*                     float r = dead time ratio (tested for.01 to 8192)     */
/*                               Bounded to 1e-4 < r < 1e6 to avoid O/F      */
/*                     float mu= noise type (tested for -2 to +2)            */
/*                               Get -1 error code for mu < -2 or mu > 2     */
/*                                                                           */
/*      Return:        float   = B3 bias function value,                     */
/*                               or -1 if error                              */
/*                                                                           */
/*      Revision record:                                                     */
/*          09/21/02   Copied from CalcBias2() of FrequenC library           */
/*          03/08/03   Brought into Ver 2.0 of FrequenC lib                  */
/*          04/26/03   Adapted for Version 2.0 source code documentation     */
/*                                                                           */
/* (c) Copyright 1991-2003  Hamilton Technical Services  All Rights Reserved */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL CalcBias3(int m, float r, float mu)
{
    double f;                                                     /* term f */
    double s;                                             /* summation term */
    int i;                                               /* summation index */

    if(m==0)
    {
        return(-1.0);                                         /* error code */
    }
    m=abs(m);                                      /* avoid negative # points */
    r=(float)fabs(r);                                 /* avoid negative ratio */

    if((mu<-2)||(mu>2))                                  /* avoid æ<-2 or æ>2 */
    {
        return(-1.0);                                         /* error code */
    }                                                             /* end if */
    if((r==1)||(m==1)||(mu==2))     /* no need to calc if r=1 or m=1 or mu=2 */

    {
        return(1.0);
    }                                                               /* end if */

    if(mu==-2)                                   /* no need to calc if mu=-2 */
    {
        return(float)(m);
    }                                                               /* end if */

    if(mu==0)                                                /* check for æ=0 */
    {
        mu+=(float)1e-6;                               /* to avoid singularity */
    }                                                               /* end if */

    if(r<.0001)                                         /* check for r near 0 */
    {
        r=(float)1e-4;                                 /* to avoid singularity */
    }                                                               /* end if */

    if(r>1e6)                                       /* check for very large r */
    {
        r=1e6;                                            /* to avoid overflow */
    }                                                               /* end if */

    s=0;                                            /* initialize sum to zero */

    for(i=1; i<m; i++)                                         /* do summation */
    {
        f=2*pow(i*r, mu+2) - pow((i*r)+1, mu+2) - pow(fabs((i*r)-1), mu+2);
        f*=2;
        f-=2*pow((m+i)*r, mu+2) - pow(((m+i)*r)+1, mu+2)
          - pow(fabs(((m+i)*r)-1), mu+2);
        f-=2*pow((m-i)*r, mu+2) - pow(((m-i)*r)+1, mu+2)
          - pow(fabs(((m-i)*r)-1), mu+2);
        f*=(m-i);
        s-=f;
    }                                                              /* end for */

    s+=m*(2*pow(m*r, mu+2) - pow((m*r)+1, mu+2) - pow(fabs((m*r)-1), mu+2));
    s+=2*m;
    f=2*pow(r, mu+2) - pow(r+1, mu+2) - pow(fabs(r-1), mu+2);
    f+=2;

    return(float)(s/(f*pow(m, mu+2)));
}                                                       /* end CalcBias3() */
```

The FrequenC Library

NAME:	**FUNCTION:**
CalcBisectionDrift	Calc freq drift using the average freq over the first and last halves of the data points

SYNOPSIS:

```
float CalcBisectionDrift(F_TYPE y[])
```

F_TYPE y[]	Frequency data array
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

RETURN: float	Drift per unity data interval

REMARKS:

Only data points between start and end analysis limits are modified.
All zeros are treated as gaps in frequency data.
This is the best estimator of drift for the combination of white and random walk FM noise (see reference.
Tau=1 is assumed. Drift is per tau interval.
This function calls FrequenC function CalcMean().

EXAMPLE:

```
#include "frequenc.h"                           /* FrequenC header file */
F_TYPE y[512];                                           /* data array */
float drift;                                                  /* drift */
.
.
drift=CalcBisectionDrift(y);                          /* call function */
printf("\nDrift = %e", drift);                       /* display result */
```

SEE ALSO: CalcMean()

REFERENCE: M. Weiss, D. Allan, and D. Howe, "Confidence on the Second Difference Estimation of Frequency Drift", 1992 IEEE Frequency Control Symposium, pp.300-305, June, 1992.

```
/***************************************************************************/
/*                                                                         */
/*                       CalcBisectionDrift()                              */
/*                                                                         */
/*      Function to calculate frequency drift using the average frequency  */
/*      over the first and last halves of the data points.                 */
/*                                                                         */
/*      Parameters:     y[]     frequency data array                       */
/*                                                                         */
/*      Return:         float   frequency drift                            */
/*                                                                         */
/*      Notes:  This is the best estimator of drift for the combination    */
/*              of white and random walk FM noise (see reference).         */
/*              Tau=1 is assumed.   Drift is per tau interval.             */
/*              This function calls FrequenC function CalcMean().          */
/*                                                                         */
/*      Reference:                                                         */
/*      M. Weiss, D. Allan, and D. Howe, "Confidence on the Second Diff-   */
/*      erence Estimation of Frequency Drift", 1992 IEEE Frequency Control */
/*      Symposium, pp.300-305, June, 1992.                                 */
/*                                                                         */
/*      Revision record:                                                   */
/*          11/01/92    Created                                            */
/*          02/03/96    Modified for use as Win 3.1 DLL                    */
/*          01/01/98    Changes for MS VC++ compatibility & warnings       */
/*          04/26/03    Adapted for Version 2.0 source code documentation     */
/*                                                                         */
/* (c) Copyright 1992-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                         */
/***************************************************************************/

#include "frequenc.h"                                /* FrequenC header file */

float __declspec(dllexport) FAR PASCAL CalcBisectionDrift(F_TYPE y[])
{
    F_TYPE start, end, mid;         /* original analysis limits & midpoint */
    F_TYPE a1, a2;                                  /* frequency averages */
    float d;                                           /* frequency drift */

    /* save original analysis limits */
    start=y[1];
    end=y[2];

    /* find midpoint of analysis data */
    mid = ((y[2]-y[1])+1)/2;                 /* not necessarily an integer */

    /* calc avg freq over 1st half */
    y[1]=1;
    y[2]=floor(mid);           /* rounded down to the next smaller integer */
    CalcMean(y, &a1, FREQ);                  /* calc avg freq over 1st half */

    /* calc avg freq over 2nd half */
    y[1]=y[2]+1;
    y[2]=end;
    CalcMean(y, &a2, FREQ);

    /* calc drift */
    d=(float)((a2-a1)/(mid));
    /* restore analysis limit */
    y[1]=start;

    return(d);                                          /* frequency drift */
}                                               /* end CalcBisectionDrift() */
```

The FrequenC Library

NAME: CalcChiSqrProb	**FUNCTION:** Calculate the chi-square probability function

SYNOPSIS: `float CalcChiSqrProb(float x, float df)`

float x	The chi-square value
float df	The # of chi square degrees of freedom
RETURN: float	The value of the chi-square probability, the the area under the chi-square curve for the chi-square value x and the # of degrees of freedom df, or -1.0 if error.

REMARKS: None

EXAMPLE:

```
#include "frequenc.h"                               /* FrequenC header file */
float p;                                 /* value of chi-square probability */
float x;                                                /* chi-square value */
float df;                                           /* # degrees of freedom */
.
.
x=3.94;                                         /* set r = dead time ratio */
df=10;                                            /* set mu for noise type */
p=CalcChiSqrProb(r, mu);                                   /* calc X² prob */
if(p==-1.0)                                             /* check for error */
{
    printf("\nError");                                    /* error message */
}
else
{
    printf("\np(X²,df) = %e", p);                             /* display p */
}
```

SEE ALSO: CalcInvChiSqr()

REFERENCE: Collected Algorithms from CACM, Vol. I, # 299

```
/****************************************************************************/
/*                                                                          */
/*                           CalcChiSqrProb()                               */
/*                                                                          */
/*      Function to calculate the chi-square probability, the area under   */
/*      the chi-square distribution.                                        */
/*                                                                          */
/*      Reference:     Collected Algorithms from CACM, Vol. I, #299,       */
/*                     I.D. Hill and M.C. Pike, 1965.                       */
/*                                                                          */
/*      Parameters:    float x  = chi-square value                          */
/*                     float df = # of degrees of freedom                   */
/*                                                                          */
/*      Return:        float    = chi-square probability value,             */
/*                                or -1 if error                            */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/30/91   Created                                              */
/*          12/31/91   Renamed                                              */
/*          01/18/92   Edited title block                                   */
/*          02/03/96   Modified for use as Win 3.1 DLL                      */
/*          01/01/98   Changes for MS VC++ compatibility & warnings         */
/*          04/26/03   Adapted for Version 2.0 source code documentation   */
/*                                                                          */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL CalcChiSqrProb(float x, float df)

{
    int f;
    BOOL even;
    BOOL bigx=FALSE;
    float chiprob;
    double a, c, e, y, z, s;

    if(x > 1416)        /* avoid O/F of exp(-0.5*x) < DBL_MIN ÷ 2.225e-308 */
    {
        bigx=TRUE;
    }                                                           /* end if */

    f = (int) df;
    if (x < 0.0 || f < 1.0)
    {
        return (-1.0);                                      /* error code */
    }                                                           /* end if */
    a = .5 * x;
    even = (f % 2);
    even = !even;
    if (even || f > 2 && ! bigx)
    {
        y = exp(-a);
    }                                                           /* end if */
    if (even)
    {
        s = y;
    }                                                           /* end if */
    else
    {
        s = 2.0 * CalcNormalProb((float)-sqrt(x));
    }                                                           /* end else */
    if (f > 2)
    {
        x = (float)(0.5 * (df - 1.0));
        if (even)
        {
            z = 1.0;
        }                                                         /* end if */
        else
        {
            z = .5;
        }                                                       /* end else */
        if (bigx)
        {
            e = (even) ? 0.0 : .572364942925;
            c = log(a);
            for ( ; z <= x; z += 1.0)
            {
                e = log(z) + e;
                s = exp(c * z - a - e) + s;
            }                                                    /* end for */
            chiprob = (float)s;
        }                                                         /* end if */
        else
        {
            e = (even) ? 1.0 : .564189583548 / sqrt(a);
            c = 0.0;
            for ( ; z <= x; z += 1.0)
            {
                e = e * a / z;
                c += e;
            }                                                    /* end for */
        }                                                       /* end else */
        chiprob = (float)(c * y + s);
    }                                                             /* end if */
    else
    {
        chiprob = (float)s;
    }                                                           /* end else */

    return (chiprob);
}                                                /* end CalcChiSqrProb() */
```

The FrequenC Library	
NAME: ConvDomain	**FUNCTION:** Perform time-frequency domain conversions
SYNOPSIS: int ConvDomain(float h, float *z, float tau, float f, float v, float fh, int type, int alpha, int n)	
float h	Noise parameter, h(α)
float *z	Pointer to output parameter: 1=S_y(f), 1/Hz, 2= S_x(f), sec²/Hz, 3=S_ϕ(f), rad²/Hz, 4=£(f), dBc/Hz, 5=$\sigma_y(\tau)$, 6= Mod $\sigma_y(\tau)$, 7=$\sigma_x(\tau)$, sec
float tau	Averaging time τ, sec
float f	Fourier frequency f, Hz
float v	Carrier frequency, Hz
float fh	System high frequency cut off, Hz
int type	Input parameter type: 1=S_y(f), 2=S_x(f), 3=S_ϕ(f), 4=£(f), 5=$\sigma_y(\tau)$, 6=Mod $\sigma_y(\tau)$, 7=$\sigma_x(\tau)$
int alpha	Power law noise type, α
int n	# samples averaged; sampling time=τ_0=τ/n
RETURN: int	Status: +1=OK, -1=Error

REMARKS: This function converts the noise parameter h(α)to an output parameter determined by type. Depending on the conversion desired, the parameters t, f, v, fh, alpha and n are used. Use the function CalcDomain() for the inverse conversion from one of the time or frequency noise parameters to h(α). A general domain conversion is performed by using both functions. Calls FrequenC Library function CalcRatio(). Original z unchanged if bad parameter (error).

EXAMPLE:

```
#include "frequenc.h"                          /* FrequenC header file */
float h, tau, f, v, fh, type;                      /* input parameters */
float z;                                           /* output parameter */
int alpha, n;                                      /* input parameters */
int status;                                             /* return code */
.
/* set parameters */
.
ConvDomain(h, &z, tau, f, fh, type, alpha, n);       /* call function */
printf("\nv = %e", v);                              /* display result */
```

SEE ALSO: CalcDomain()

REFERENCE: NIST Technical Note 1337

```
/****************************************************************************/
/*                                                                          */
/*                         ConvDomain()                                     */
/*                                                                          */
/*       Function to do time-frequency domain conversions                   */
/*                                                                          */
/*       Parameters:      float h    = Noise parameter h(à)                 */
/*                        float *p_z = Pointer to output parameter z:       */
/*                                   = Sy(f), 1/Hz    if type=1             */
/*                                   = Sx(f), secý/Hz if type=2             */
/*                                   = Sí(f), radý/Hz if type=3             */
/*                                   = œ(f), dBc/Hz   if type=4             */
/*                                   = åy(ç)          if type=5             */
/*                                   = Mod åy(ç)      if type=6             */
/*                                   = åx(ç), sec     if type=7             */
/*                        float tau  = Averaging time ç, sec                */
/*                        float f    = Fourier frequency f, Hz              */
/*                        float v    = Carrier frequency, Hz                */
/*                        float fh   = System high frequency cut off, Hz    */
/*                        int type   = Input parameter type:                */
/*                                     1=Sy(f), 2=Sx(f), 3=Sí(f), 4=œ(f)    */
/*                                     5=åy(ç), 6=Mod åy(ç), 7=åx(ç)        */
/*                        int alpha  = Power law noise type, à              */
/*                                   = Log=log slope of Sy(f)=h(à)f^à       */
/*                                     White PM à=+2, Flicker PM à=+1       */
/*                                     White FM à= 0, Flicker FM à=-1       */
/*                                     Random Walk FM à=-2                  */
/*                        int n      = # samples averaged                   */
/*                                     sampling time = tau_0 = tau/n        */
/*                                                                          */
/*       Return:          int        = Status: +1=OK, -1=Error              */
/*                                                                          */
/*       Notes:           This function converts the noise parameter h(à)   */
/*                        to an output parameter determined by type.        */
/*                        Depending on the conversion desired, the          */
/*                        parameters t, f, v, fh, alpha and n are used.     */
/*                        Use the function CalcDomain() for the inverse     */
/*                        conversion from one of the time or frequency      */
/*                        noise parameters to h(à).  A general domain       */
/*                        conversion is performed by using both functions.  */
/*                        Calls FrequenC Library function CalcRatio().      */
/*                        Original z unchanged if bad parameter (error).    */
/*                                                                          */
/*       Revision record:                                                   */
/*           01/25/92     Adapted from conversion() of DOMAIN_F.C           */
/*           01/26/92     Changed type #s to agree with CalcDomain()        */
/*                        Fixed errors in Sp & L calculations               */
/*           02/03/96     Modified for use as Win 3.1 DLL                   */
/*           04/26/03     Adapted for Version 2.0 source code documentation    */
/*                                                                          */
/* (c) Copyright 1992-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"                        /* FrequenC Library header file */

int __declspec(dllexport) FAR PASCAL ConvDomain(float h, float *p_z,
    float tau, float f, float v, float fh, int type, int alpha, int n)
{
    double pi;                                                       /* ã */
    double a;
    double b;
    double c;
    double d;
    double e;
    double Sx;                                                  /* Sx(f) */
    double Sp;                                                  /* Sí(f) */
    double L;                                                    /* œ(f) */
    double Sy;                                                  /* Sy(f) */
    double sy;                                                  /* åy(ç) */
    double sx;                                                  /* åx(ç) */
    double mod_sy;                                          /* Mod åy(ç) */

    /* check parameters */

    if( (h==0)   || (tau<=0) || (f<=0)     || (v<=0)    || (fh<=0) ||
        (type<1) || (type>7) || (alpha<-2) || (alpha>2) || (n<1)      )
    {
        return(-1);
    }

    /* initialize constants */

    pi=4*atan(1);
    a=4*pi*pi/6;
    b=2*log(2);
    c=0.5;
    d=(1.038+(3*log(2*pi*fh*tau)))/(4*pi*pi);
    e=(3*fh)/(4*pi*pi);

    /* perform basic frequency domain conversion */

    Sy=h*pow(f, alpha);

    /* perform basic time domain conversion */

    switch(alpha)
    {
        case  2:    /* W í */
        {
            sy=sqrt(e*Sy/(f*f*tau*tau));
            break;
        }
        case  1:    /* F í */
        {
            sy=sqrt(d*Sy/(f*tau*tau));
            break;
        }
        case  0:    /* W f */
        {
            sy=sqrt(c*Sy/tau);
            break;
        }
        case -1:    /* F f */
        {
            sy=sqrt(b*f*Sy);
            break;
        }
```

```
        case -2:    /* RW f */
        {
            sy=sqrt(a*f*f*Sy*tau);

            break;
        }
        default:
        {
            return(-1);                                                /* error */
        }
    }                                                          /* end switch */

    /* complete specific conversion */

    switch(type)
    {
        case 1:   /* Sy(f) */
        {
            *p_z=Sy;
            break;
        }
        case 2:   /* Sx(f) */
        {
            Sx=Sy/(4*pi*pi*f*f);
            *p_z=Sx;
            break;
        }
        case 3:   /* Sí(f) */
        {
            Sp=Sy*v*v/(f*f);
            *p_z=Sp;
            break;
        }
        case 4:   /* œ(f) */
        {
            Sp=Sy*v*v/(f*f);
            L=10*log10(Sp/2);
            *p_z=L;
            break;
        }
        case 5:   /* sy(ç) */
        {
            *p_z=sy;
            break;
        }
        case 6:   /* Mod sy(ç) */
        {
            if( (tau/n)<1 )
            {
                return(-1);
            }
            mod_sy=sqrt(sy*sy*CalcRatio(alpha, n, 2*pi*fh*tau/n));
            *p_z=mod_sy;
            break;
        }
        case 7:   /* sx(ç) */
        {
            if( (tau/n)<1 )
            {
                return(-1);
            }
            mod_sy=sqrt(sy*sy*CalcRatio(alpha, n, 2*pi*fh*tau/n));
            sx=tau*mod_sy/sqrt(3);
            *p_z=sx;

            break;
        }
        default:
        {
            return(-1);                                         /* error */
        }
    }                                                      /* end switch */
        return(1);
}                                                   /* end ConvDomain() */
```

The FrequenC Library

NAME: CalcDomain	**FUNCTION:** Calc time-frequency domain noise parameters

SYNOPSIS:

```
int CalcDomain(float *h, float z, float tau, float tau, float v,
float fh, int type, int alpha, int n)
```

float *h	Pointer to noise parameter output, h(α)
float z	Input parameter: 1=S_y(f), 1/Hz, 2=S_x(f), sec^2/Hz, 3=S_ϕ(f), rad^2/Hz, 4=£(f), dBc/Hz, 5=$\sigma_y(\tau)$, 6= Mod $\sigma_y(\tau)$, 7=$\sigma_x(\tau)$, sec
float tau	Averaging time τ, sec
float v	Carrier frequency, Hz
float fh	System high frequency cut off, Hz
int type	Input parameter type: 1=S_y(f), 2=S_x(f), 3=S_ϕ(f), 4=£(f), 5=$\sigma_y(\tau)$, 6=Mod $\sigma_y(\tau)$, 7=$\sigma_x(\tau)$
int alpha	Power law noise type, α
int n	# samples averaged; sampling time=τ_0=τ/n
RETURN: int	Status: +1=OK, -1=Error

REMARKS:
This function calculates the power law noise parameter h(α) for an input parameter type. Depending on the conversion desired, the parameters t, f, v, fh, alpha and n are used. Use the function ConvDomain() for the inverse conversion from h(α) to one of the time or frequency noise parameters . A general domain conversion is performed by using both functions. Calls FrequenC Library function CalcRatio(). Original h unchanged if bad parameter (error).

EXAMPLE:

```
#include "frequenc.h"                      /* FrequenC header file */
float z, tau, f, v, fh, type;                  /* input parameters */
float h;                                       /* output parameter */
int alpha, n;                                  /* input parameters */
int status;                                         /* return code */
.
/* set parameters */
.
CalcDomain(&h, z, tau, v. fh, type, alpha, n);    /* call function */
printf("\nh = %e", h);                           /* display result */
```

SEE ALSO: ConvDomain()

REFERENCE: NIST Technical Note 1337

```
/************************************************************************/
/*                                                                      */
/*                           CalcDomain()                               */
/*                                                                      */
/*      Function to calc time-frequency domain noise parameters        */
/*                                                                      */
/*      Parameters:     float *p_h = Pointer to noise parameter h(à)    */
/*                      float  z   = input parameter:                   */
/*                                 = Sy(f), 1/Hz    if type=1           */
/*                                 = Sx(f), secý/Hz if type=2           */
/*                                 = Sí(f), radý/Hz if type=3           */
/*                                 = œ(f), dBc/Hz   if type=4           */
/*                                 = åy(ç)          if type=5           */
/*                                 = Mod åy(ç)      if type=6           */
/*                                 = åx(ç), sec     if type=7           */
/*                      float tau  = Averaging time ç, sec              */
/*                      float f    = Fourier frequency f, Hz            */
/*                      float v    = Carrier frequency, Hz              */
/*                      float fh   = System high frequency cut off, Hz  */
/*                      int type   = Input parameter type:              */
/*                                   1=Sy(f), 2=Sx(f), 3=Sí(f), 4=œ(f)  */
/*                                   5=åy(ç), 6=Mod åy(ç), 7=åx(ç)      */
/*                      int alpha  = Power law noise type, à            */
/*                                 = Log=log slope of Sy(f)=h(à)f^à     */
/*                                   White PM à=+2, Flicker PM à=+1     */
/*                                   White FM à= 0, Flicker FM à=-1     */
/*                                   Random Walk FM à=-2                */
/*                      int n      = # samples averaged                 */
/*                                   sampling time = tau_0 = tau/n      */
/*                                                                      */
/*      Return:         int        = Status: +1=OK, -1=Error            */
/*                                                                      */
/*      Notes:          This function calculates the power law noise    */
/*                      parameter h(à) for an input parameter type.     */
/*                      Depending on the conversion desired, the        */
/*                      parameters t, f, v, fh, alpha and n are used.   */
/*                      Use the function ConvDomain() for the inverse   */
/*                      conversion from h(à) to one of the time or      */
/*                      frequency noise parameters .  A general domain  */
/*                      conversion is performed by using both functions.*/
/*                      Calls FrequenC Library function CalcRatio().    */
/*                      Original h unchanged if bad parameter (error).  */
/*                                                                      */
/*      Revision record:                                                */
/*          01/25/92    Adapted from table_calc() of DOMAIN_C.C         */
/*          01/26/92    Changed title block type #s                     */
/*          02/03/96    Modified for use as Win 3.1 DLL                 */
/*          04/26/03     Adapted for Version 2.0 source code documentation   */
/*                                                                      */
/* (c) Copyright 1992-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                      */
/************************************************************************/

#include "frequenc.h"                        /* FrequenC Library header file */

int __declspec(dllexport) FAR PASCAL CalcDomain(float *p_h, float z,
    float tau, float f, float v, float fh, int type, int alpha, int n)
{
    double pi;                                                  /* ã */
    double a;
    double b;
    double c;
    double d;
    int index;                                              /* case index */

    /* check parameters */

    if( (z==0)   || (tau<=0) || (f<=0)     || (v<=0)    || (fh<=0) ||
        (type<1) || (type>7) || (alpha<-2) || (alpha>2) || (n<1)      )
    {
        return(-1);
    }

    /* initialize constants */

    pi=4*atan(1);
    a=4*pi*pi;
    b=2*log(2);
    c=2*pi*fh*tau;
    d=1.038+(3*log(c));

    index=((alpha+2)*7)+type;

    /* process noise data entry */

    switch(index)
    {
        case 1:     /* Sy(f) for RW f à=-2 */
                    {
                        *p_h=z*f*f;
                        break;
                    }
        case 2:     /* Sx(f) for RW f à=-2 */
                    {
                        *p_h=a*z*f*f*f*f;
                        break;
                    }
        case 3:     /* Sí(f) for RW f à=-2 */
                    {
                        *p_h=z*f*f*f*f/(v*v);
                        break;
                    }
        case 4:     /* œ(f) for RW f à=-2 */
                    {
                        *p_h=2*pow(10,(z/10))/(v*v);
                        *p_h*=f*f*f*f;
                        break;
                    }
        case 5:     /* åy(ç) for RW f à=-2 */
                    {
                        *p_h=6*z*z/(a*tau);
                        break;
                    }
        case 6:     /* Mod åy(ç) for RW f à=-2 */
                    {
                        *p_h=6*z*z;
                        *p_h/=CalcRatio(2, n, c/n);
                        *p_h/=a*tau;
                        break;
                    }
```

```
case 7:     /* åx(ç) for RW f à=-2 */

            {
                *p_h=3*6*z*z;
                *p_h/=CalcRatio(2, n, c/n);
                *p_h/=a*tau*tau*tau;
                break;
            }
case 8:     /* Sy(f) for F f à=-1 */
            {
                *p_h=z*f;
                break;
            }
case 9:     /* Sx(f) for F f à=-1 */
            {
                *p_h=a*f*f*f*z;
                break;
            }
case 10:    /* Sí(f) for F f à=-1 */
            {
                *p_h=f*f*f*z/(v*v);
                break;
            }
case 11:    /* œ(f) for F f à=-1 */
            {
                *p_h=2*pow(10,(z/10))/(v*v);
                *p_h*=f*f*f;
                break;
            }
case 12:    /* åy(ç) for F f à=-1 */
            {
                *p_h=z*z/(b);
                break;
            }
case 13:    /* Mod åy(ç) for F f à=-1 */
            {
                *p_h=z*z;
                *p_h/=b;
                *p_h/=CalcRatio(1, n, c/n);
                break;
            }
case 14:    /* åx(ç) for F f à=-1 */
            {
                *p_h=3*z*z/(b);
                *p_h/=CalcRatio(1, n, c/n);
                *p_h/=tau*tau;
                break;
            }
case 15:    /* Sy(f) for W f à=0 */
            {
                *p_h=z;
                break;
            }
case 16:    /* Sx(f) for W f à=0 */
            {
                *p_h=a*f*f*z;
                break;
            }
```

```
case 17:    /* Sí(f) for W f à=0 */
            {
                *p_h=f*f*z/(v*v);
                break;

            }
case 18:    /* œ(f) for W f à=0 */
            {
                *p_h=f*f*2*pow(10,(z/10))/(v*v);
                break;
            }
case 19:    /* åy(ç) for W f à=0 */
            {
                *p_h=2*z*z*tau;
                break;
            }
case 20:    /* Mod åy(ç) for W f à=0 */
            {
                *p_h=2*z*z*tau;
                *p_h/=CalcRatio(0, n, c/n);
                break;
            }
case 21:    /* åx(ç) for W f à=0 */
            {
                *p_h=3*2*z*z*tau;
                *p_h/=CalcRatio(0, n, c/n);
                *p_h/=tau*tau;
                break;
            }
case 22:    /* Sy(f) for F í à=1 */
            {
                *p_h=z/f;
                break;
            }
case 23:    /* Sx(f) for F í à=1 */
            {
                *p_h=a*f*z;
                break;
            }
case 24:    /* Sí(f) for F í à=1 */
            {
                *p_h=f*z/(v*v);
                break;
            }
case 25:    /* œ(f) for F í à=1 */
            {
                *p_h=f*2*pow(10,(z/10));
                *p_h/=v*v;
                break;
            }
case 26 :   /* åy(ç) for F í à=1 */
            {
                *p_h=a*z*z*tau*tau;
                *p_h/=(d);
                break;
            }
```

```
case 27:    /* Mod åy(ç) for F í à=1 */
            {
                *p_h=a*z*z;
                *p_h/=CalcRatio(-1, n, c/n);
                *p_h*=tau*tau;
                *p_h/=(d);
                break;
            }
case 28:    /* åx(ç) for F í à=1 */
            {
                *p_h=3*a*z*z*tau*tau;
                *p_h/=CalcRatio(-1, n, c/n);
                *p_h/=tau*tau;
                *p_h/=(d);
                break;
            }
case 29:    /* Sy(f) for W í à=2 */
            {
                *p_h=z/(f*f);
                break;
            }
case 30:    /* Sx(f) for W í à=2 */
            {
                *p_h=a*z;
                break;
            }
case 31:    /* Sí(f) for W í à=2 */
            {
                *p_h=z/(v*v);
                break;
            }
case 32:    /* œ(f) for W í à=2 */
            {
                *p_h=2*pow(10,(z/10));
                *p_h/=v*v;
                break;
            }
case 33:    /* åy(ç) for W í à=2 */
            {
                *p_h=a*z*z;
                *p_h*=tau*tau/(3*fh);
                break;
            }
case 34:    /* Mod åy(ç) for W í à=2 */
            {
                *p_h=a*z*z;
                *p_h*=tau*tau;
                *p_h/=CalcRatio(-2, n, c/n);
                *p_h/=3*fh;
                break;
            }
case 35:    /* åx(ç) for W í à=2 */
            {
                *p_h=3*a*z*z;
                *p_h*=tau*tau;
                *p_h/=CalcRatio(-2, n, c/n);
                *p_h/=3*tau*tau*fh;
                break;
            }
```

```
        default:    break;
    }                                                       /* end switch */
        return(1);
}                                                     /* end CalcDomain() */
```

The FrequenC Library	
NAME: CalcDegFree	**FUNCTION:** Calculate the # of degrees of freedom for a set of phase or frequency data
SYNOPSIS: float CalcDegFree(int alpha, int num, int m)	
int alpha	The log-log slope, a, of $S_y(f)$ for a power law noise process:
int num	The # of phase data points = # frequency data points + 1
int m	Averaging factor
RETURN: float	The # of chi-squared degrees of freedom for num fully-overlapping phase data points with an averaging factor of m, or -1.0 if error.
REMARKS: None	

Noise Type	a
Flicker Walk FM	-3
Random Walk FM	-2
Flicker FM	-1
White FM	0
Flicker PM	1
White PM	2

EXAMPLE:

```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE x[513];                                     /* phase data array */
float df;                                   /* # of degrees of freedom */
int num;                                       /* # phase data points */
int m;                                            /* averaging factor */
int alpha;                                     /* noise exponent alpha */
.
.
m=2;                                            /* set m = # averages */
num=(int)x[0];                                   /* set # data points */
alpha=0;                                     /* set noise type alpha */
df=CalcDegFree(alpha, num, m);                          /* calc df */
if(df==-1.0)                                       /* check for error */
{
    printf("\nError");                              /* error message */
}
else
{
    printf("\ndf = %e", df);                           /* display df */
}
```

SEE ALSO: None

REFERENCE: NIST Technical Note 1337

```
/****************************************************************************/
/*                                                                          */
/*                          CalcDegFree()                                   */
/*                                                                          */
/*      Function to calculate the # of chi-square degrees of freedom       */
/*      for phase or frequency data.                                        */
/*                                                                          */
/*      Reference:     NIST Technical Note 1337, p.TN-85, Table 12-4       */
/*                     (as corrected).                                      */
/*                                                                          */
/*      Parameters:    int alpha = power-law exponent of Sy(f) noise type  */
/*                     int n     = # phase data points                     */
/*                               = # frequency data points + 1             */
/*                     int m     = fully-overlapping averaging factor      */
/*                                                                          */
/*      Return:        float     = # chi-square degrees of freedom,        */
/*                                 or -1 if error                          */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/30/91   Created                                              */
/*          12/31/91   Renamed                                              */
/*          01/04/92   Changed switch from (2-a) to a                       */
/*                     Added default case for NG alpha                      */
/*                     Changed n and m parameters to ints                   */
/*                     Do calcs with double versions dn and dm              */
/*                     Added error traps                                    */
/*          01/18/92   Edited title block                                   */
/*          02/03/96   Modified for use as Win 3.1 DLL                      */
/*          01/01/98   Changes for MS VC++ compatibility & warnings         */
/*          04/26/03   Adapted for Version 2.0 source code documentation   */
/*                                                                          */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL CalcDegFree(int alpha, int n, int m)

{
    double dn;                                          /* double version of n */
    double dm;                                          /* double version of m */
    double df;                                         /* # degrees of freedom */
    double df1,df2,df3;                           /* terms of df calculation */

    if( (n==0) || (m==0) )
    {
        return(-1.0);                                           /* error code */
    }

    dn=(double) abs(n);                           /* make sure n is positive */
    dm=(double) abs(m);                           /* make sure m is positive */

    switch(alpha)          /* alpha is exponent of power-law noise process */
    {                                     /* slope of Sy(f) on log-log plot */
            case 2:
            {
```

```
        if(dn==dm)
            {
                return(-1.0);                              /* error code */
            }

            df=((dn+1)*(dn-(2*dm)))/(2*(dn-dm));  /* white phase noise */
            break;                                            /* alpha=2 */
        }
        case 1:
        {
            if(dn==1)
            {
                return(-1.0);                             /* error code */
            }
            df1=log((((2*dm)+1)*(dn-1))/(4));   /* flicker phase noise */
            df2=log((dn-1)/(2*dm));                         /* alpha=1) */
            if(df1*df2<0)
            {
                return(-1.0);                              /* error code */
            }
            df=exp(sqrt(df1*df2));
            break;
        }
        case 0:
        {
            df1=(4*dm*dm)/((4*dm*dm)+5);      /* white frequency noise */
            df2=(3*(dn-1))/(2*dm);                            /* alpha=0 */
            df3=(2*(dn-2))/(dn);
            df=(df2-df3)*df1;
            break;
        }
        case -1:
        {
            if(dm==1.0)                     /* flicker frequency noise */
            {                                                /* alpha=-1 */
                df=(2*(dn-2)*(dn-2))/((2.3*dn)-4.9);
                break;
            }                                                  /* end if */
            else
            {
                df=(5*dn*dn)/((4*dm)*(dn+(3*dm)));
                break;
            }                                                /* end else */
        }
        case -2:
        {
            if(dn==3)
            {
                return(-1.0);                              /* error code */
            }
            df1=(dn-1)*(dn-1);          /* random-walk frequency noise */
            df2=(dn-2)/((dm)*(dn-3)*(dn-3));                /* alpha=-2 */
            df3=(3*dm)*(1-dn)+(4*dm*dm);
            df=df2*(df1+df3);
            break;
        }
```

```
            default:                                       /* NG alpha */
            {
                df=-1.0;                                 /* error code */
            }
    }                                                    /* end switch */

    return(float)df;                     /* chi-squared degrees of freedom */
}                                                   /* end CalcDegFree() */
```

The FrequenC Library

NAME:	**FUNCTION:**
CalcDiffusionFreqDrift()	Calculate diffusion frequency drift. Fit fractional frequency data to time-shifted diffusion model: y(t) = a + b*sqrt(t+c)

SYNOPSIS:

```
int CalcDiffusionFreqDrift(F_TYPE y[], F_TYPE *a, F_TYPE *b,
    F_TYPE *c, F_TYPE *v)
```

F_TYPE y[]	Frequency data array
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

F_TYPE *a	Pointer to offset coefficient
F_TYPE *b	Square root of time multiplier coefficient
F_TYPE *c	Time shift coefficient
F_TYPE *v	Pointer to fit variance
RETURN: int	# non-gap data points processed, or –1 if error

REMARKS:

Only data points between start and end analysis limits are used.
All zeros are treated as gaps in frequency data.
A value of c is chosen, the diffusion fit is found, and c is varied iteratively until the rms error is below a limit criterion.
Need at least 2 non-zero data points.

EXAMPLE:

```
#include "frequenc.h"                       /* FrequenC header file */
F_TYPE y[512];                              /* frequency data array */
F_TYPE a;                                        /* fit parameter a */
F_TYPE b;                                        /* fit parameter b */
F_TYPE c;                                        /* fit parameter c */
F_TYPE v;                                           /* fit variance */
int num;                                 /* # data points processed */
.
.
num= CalcLogFreqDrift (y, &a, &b, &c, &v);         /* call function */
printf("\na = %e", b = %e, c = %e, a, b, c);     /* display results */
```

SEE ALSO: CalcLogFreqDrift()

REFERENCE: None

```
/*****************************************************************************/
/*                                                                           */
/*                       CalcDiffusionFreqDrift()                            */
/*                                                                           */
/*       Function to calculate diffusion frequency drift                     */
/*       Fit fractional frequency data to time-shifted diffusion model:      */
/*                       y(t) = a + b*sqrt(t+c)                              */
/*                       where: y(t) = fractional frequency                  */
/*                              t    = time                                  */
/*                              a    = offset term                           */
/*                              b    = sqrt time multiplier term             */
/*                              c    = time shift term                       */
/*                                                                           */
/*       Parameters:     F_TYPE y[]  = frequency data                        */
/*                                     y[0] = # data points                  */
/*                                     y[1] = analysis start                 */
/*                                     y[2] = analysis end                   */
/*                       F_TYPE *p_a  = pointer to offset term               */
/*                       F_TYPE *p_b  = pointer to sqrt time multiplier term  */
/*                       F_TYPE *p_c  = pointer to time shift term           */
/*                       F_TYPE *p_v  = pointer to fit variance              */
/*                                                                           */
/*       Return:         int          = # non-zero anlaysis points,          */
/*                                      or -1 if error                       */
/*                                                                           */
/*       Notes:          Zeros treated as gaps in frequency data.            */
/*                       A value of c is chosen, the diffusion fit parameters */
/*                       are found, and c is varied iteratively until the    */
/*                       rms error is below a limit criterion.               */
/*                       Need at least 2 non-zero data points.               */
/*                                                                           */
/*       Revision record:                                                    */
/*           06/09/97    Created from CalcLogFreqDrift() of FrequenC Lib and  */
/*                       CalcDiffusionFreqDrift() of Stable/Win              */
/*           06/10/97    Running - converges OK                              */
/*           12/09/97    Added to FrequenC Library                           */
/*           01/01/98    Changes for MS VC++ compatibility & warnings        */
/*           04/26/03     Adapted for Version 2.0 source code documentation     */
/*                                                                           */
/* (c) Copyright 1997-2003 Hamilton Technical Services All Rights Reserved   */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcDiffusionFreqDrift(F_TYPE y[],
    F_TYPE *p_a, F_TYPE *p_b, F_TYPE *p_c, F_TYPE *p_v)
{
    int i;                                              /* freq data index */
    int num = (int)(*(y+2) - *(y+1) +1);          /* # frequency data points */

    #if(0) // Original code of Release 1.01 - Doesn't always converge
    double error_limit=0.00000001;    /* fractional error limit for variance */
    #endif

    #if(1) // Trial code 06/20/97
    // This seems better - converges for all known test cases & results OK
    double error_limit=0.000001;      /* fractional error limit for variance */
    #endif
```

```
double err=1.0;                                  /* sum of squared error */

double prev_err=0;    /* initialized so (err-prev_err)/err > error_limit */
double sum_x;                                      /* sum of x=sqrt(t+c) */
double sum_y;                                              /* sum of y=t */
double sum_xx;                                             /* sum of x^2 */
double sum_xy;                                              /* sum of xy */
double d=100.0;                                  /* c iteration factor */

*p_c=0.0;                                         /* initial value for c */

while(fabs(((err-prev_err)/err))>error_limit)                /* iterate */
{
    prev_err=err;                                 /* save previous error */
    sum_x = 0;                                     /* sum of x=sqrt(t+c) */
    sum_y = 0;                                               /* sum of y */
    sum_xx = 0;                                            /* sum of x^2 */
    sum_xy = 0;                                             /* sum of xy */
    num = (int)(*(y+2) - *(y+1) +1);          /* # frequency data points */

    for( i=(int)(*(y+1)+2); i<=(int)(*(y+2)+2); i++)  /* accumulate sums */
    {
        if(*(y+i))                                      /* gap in data? */
        {
            sum_x  += sqrt(((double)(i-2))+ *p_c);
            sum_y  += *(y+i);
            sum_xx += ((double)(i-2))+ *p_c;
            sum_xy += sqrt(((double)(i-2))+ *p_c) * *(y+i);
        }                                                     /* end if */
        else
        {
            num--;
        }                                                   /* end else */
    }                                                        /* end for */

    if(num<2)                          /* must have at least 2 data points */
    {
        return(-1);
    }                                                         /* end if */

    *p_b = sum_xy - (sum_x * sum_y / num);            /* find parameters */
    *p_b /= sum_xx- (sum_x * sum_x / num);
    *p_a = (sum_y - (*p_b * sum_x ))/ num;

    err=0;                                  /* calc sum of squared error */

    for( i=(int)(*(y+1)+2); i<=(int)(*(y+2)+2); i++)
    {
        if(*(y+i))
        {
            // error = SQR [ fit - actual = a + b*sqrt(t+c) - y(t) ]
            err += SQR(*p_a + *p_b * sqrt((double)(i-2)+ *p_c) - *(y+i));
        }
    }

    if(err>prev_err)                                       /* wrong way? */
    {
        d *= -0.1;
    }

    *p_c+= d;                              /* modify c for next iteration */
```

```
        if(*p_c<0.0)                              /* avoid sqrt domain error */
        {
            d *= -0.1;
            *p_c=0.0;
        }

    }

    *p_v = err;

    return(num);
}                                         /* end NewCalcDiffusionFreqDrift() */
```

The FrequenC Library

NAME: `CalcFirstDiff`	**FUNCTION:** Calculate the average of the 1st differences (frequency offset) for phase data

SYNOPSIS:
```
int CalcFirstDiff(F_TYPE x[], F_TYPE *d)
```

F_TYPE x[]	Phase data array
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE *d	Pointer to average of 1st diffs of phase data
RETURN: int	# non-gap data points processed or -1 if error

REMARKS:
Only data points between start and end analysis limits are analyzed.
Embedded zeros are treated as gaps in phase data.
There must be at least 2 adjacent non-gap phase data points.

EXAMPLE:
```
 #include "frequenc.h"                          /* FrequenC header file */
 F_TYPE x[512];                                     /* phase data array */
 F_TYPE d;                                                  /* 1st diff
*/
 int num;                                    /* # data points processed */
 .
 .
 num=CalcFirstDiff(x, &d);                             /* call function */
 printf("\nFreq Offset = %e", d);                     /* display result */
```

SEE ALSO: CalcFreqOffset()

REFERENCE: NIST Technical Note 1337

```
/****************************************************************************/
/*                                                                          */
/*                          CalcFirstDiff()                                 */
/*                                                                          */
/*      Function to calculate average of 1st differences of phase data     */
/*      Can also be used to check phase data for zero frequency due to      */
/*      zero first differences (equal adjacent phase points)                */
/*                                                                          */
/*      Parameters:     F_TYPE x[]  = array of phase data                   */
/*                                    x[0] = # phase data points            */
/*                                    x[1] = analysis start                 */
/*                                    x[2] = analysis end                   */
/*                      F_TYPE *p_d = pointer to average of first           */
/*                                    differences of phase data             */
/*                                  = frequency offset of phase data        */
/*                                  = 0 if zero frequency point exists      */
/*                                                                          */
/*      Return:         int         = # analysis points,                    */
/*                                    or -1 if error                        */
/*                                                                          */
/*      Note:           Embedded gaps are treated as gaps in phase data.    */
/*                      There must be at least 1 analysis point,            */
/*                      which requires at least 2 adjacent non-gap          */
/*                      phase data points.                                  */
/*                                                                          */
/*      Revision record:                                                    */
/*          03/23/98    Created from CSD.C                                  */
/*          04/26/03    Adapted for Version 2.0 source code documentation     */
/*                                                                          */
/* (c) Copyright 1998-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcFirstDiff(F_TYPE x[], F_TYPE *p_d)
{
    float num;
    double d1;
    double sum;
    int i;

    num = 0;
    sum = 0;

    for(i=(int)(x[1]+2); i<(int)(x[2]+1); i++)
    {
        if( (x[i] && x[i+1]) || (i==3 && x[4]) || ((i==x[0]+1) && x[i]) )
        /* no phase points are gaps, or 1st point with non-zero 2nd point, */
                         /* or last point with non-zero next-to-last point */
        {
            if(d1  =  x[i+1] - x[i])
            {
                sum += d1;
                num++;
            }
            else
            {
                *p_d = 0.0;
                return(-1);
            }
        }                                                   /* end if */
    }                                                      /* end for */
    if(num>0)
    {
        *p_d = sum / num;
        return(int)num;
    }
    else
    {
        return(-1);                                          /* error */
    }
}                                             /* end CalcFirstDiff() */
```

The FrequenC Library

NAME: CalcFreqHadamardDev	**FUNCTION:** Calc Hadamard deviation from frequency data

SYNOPSIS:
```
int CalcFreqHadamardDev(F_TYPE y[], F_TYPE *dev)
```

F_TYPE y[]	Frequency data array
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

F_TYPE *dev	Pointer to Hadamard deviation
RETURN: int	# non-gap data points processed or -1 if error

REMARKS:
Only data points between start and end analysis limits are analyzed.
All zeros are treated as gaps in frequency data.
Must have at least 3 adjacent non-gap freq data points.

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE y[512];                                      /* freq data array */
F_TYPE dev;                                      /* Hadamard deviation */
int num;                                   /* # data points processed */
.
.
num=CalcFreqHadamardDev(y, &dev);                     /* call function */
printf("\nHadamard Dev = %e", dev);                 /* display result */
```

SEE ALSO: CalcPhaseHadamardDev

REFERENCE: S. Hutsell, Proc. 27th PTTI Meeting, Dec. 1995

```
/*****************************************************************************/
/*                                                                           */
/*                          CalcFreqHadamardDev()                            */
/*                                                                           */
/*      Function to calculate Hadamard variance from frequency data          */
/*                                                                           */
/*                       1    M-2                                            */
/*      H åýy(ç) = ÄÄÄÄÄÄ  ä [y(i+2) -2y(i+1) -y(i)]ý                        */
/*                     6(M-2) i=1                                            */
/*                                                                           */
/*                                                                           */
/*      Parameters:     F_TYPE y[]    = frequency data                       */
/*                                      y[0] = # data points                 */
/*                                      y[1] = analysis start                */
/*                                      y[2] = analysis end                  */
/*                      float *p_dev = pointer to Hadamard deviation         */
/*                                                                           */
/*      Return:         int           = # analysis points,                   */
/*                                      or -1 if error                       */
/*                                                                           */
/*      Note:           Embedded zeros treated as gaps in freq data.         */
/*                      Must have at least 1 analysis point; this            */
/*                      requires at least 3 adjacent non-gap freq data       */
/*                      points.                                              */
/*                                                                           */
/*      Reference:      S. Hutsell, Proc. 27th PTTI Meeting, Dec. 1995.      */
/*                                                                           */
/*      Revision record:                                                     */
/*          05/03/96    Adapted from hadamardev_f() of STABLE_A.C            */
/*          01/01/98    Changes for MS VC++ compatibility & warnings         */
/*          04/26/03    Adapted for Version 2.0 source code documentation    */
/*                                                                           */
/* (c) Copyright 1996-2003 Hamilton Technical Services All Rights Reserved   */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcFreqHadamardDev(F_TYPE y[],
    F_TYPE *p_dev)
{
    double sum;                           /* summing variable to calc sigma */
    int i;                                               /* index for summing */
    int num = 0;                                         /* # analysis points */

    for( i = (int)(*(y+1)+2), sum = 0; i < (int)(*(y+2)+1); i++)
                                               /* index goes from 1 to n-2 */
                       /* for n data points from 1 to n; data starts at y[3] */
    {
        if( *(y+i) && *(y+i+1) && *(y+i+2) )
                                     /* skip if any freq data value is zero */
        {
            sum += SQR( *(y+i+2) - 2 * *(y+i+1) + *(y+i) );
            num++;
        }
    }
    if(num>0)
    {
        *p_dev = ( sqrt( sum / (6.0 * num )));
        return(num);                                    /* # analysis points */
    }
    else                                               /* no analysis points */
    {
        return(-1);                                                 /* error */
    }
}
```

The FrequenC Library

NAME: CalcFastMTIE	**FUNCTION:** Quickly calculate maximum time interval error for phase data

SYNOPSIS:
```
int CalcFastMTIE(F_TYPE fX[], struct sigmadata *results, BOOL
*abort, void *ProgressFunc(int percent, char *msg))
```

`F_TYPE x[]`	Phase data array
	`x[0] = # data points`
	`x[1] = analysis start point`
	`x[2] = analysis end point`

`struct sigmadata *results`	Pointer to sigmadata structure for results
`BOOL *abort`	Pointer to abort flag (FALSE = no abort)
`void *ProgressFunc( int percent, char *msg)`	Pointer to function to display progress, with int arg set to % calc complete, and char* arg set to text message to display. Can be NULL.
RETURN: int	# MTIE points, or –1 if error

REMARKS:
Only data points between start and end analysis limits are used.
Embedded zeros are treated as gaps in phase data.
Sigmadata structure is defined in frequenc.h
This function calculates a complete set of MTIE points.

EXAMPLE:
```
 #include "frequenc.h"                         /* FrequenC header file */
 F_TYPE x[512];                                    /* phase data array */
 struct sigmadata results;                             /* MTIE results */
 int num;                                             /* # MTIE points */
 .
 .
 num=CalcFastMTIE(x, &results FALSE, NULL);          /* call function */
                                 /* No aborting or progress display */
 printf("\nMTIE = %e", results.sigma[0]);  /* display 1st MTIE point
*/
```

SEE ALSO: CalcMTIE()

REFERENCE: S. Bregni and S. Maccabruni, "Fast Computation of Maximum Time Interval Error by Binary Decomposition", IEEE Transactions on Instrumentation and Measurement, Vol. 49, No. 6, Dec. 2000, pp. 1240-1244.

```
/****************************************************************************/
/*                                                                          */
/*                              CalcFastMTIE()                              */
/*                                                                          */
/*      Function to calc MTIE (maximum time interval error) for phase data  */
/*      Parameters:                                                         */
/*          F_TYPE fX[]        = data array, where:                         */
/*                                       F_TYPE fX[0] = # data points       */
/*                                       F_TYPE fX[1] = analysis start      */
/*                                       F_TYPE fX[2] = analysis end        */
/*          plotdata *pResults = MTIE results sigmadata structure           */
/*                                                                          */
/*      Return:                                                             */
/*          k   = Outer index (# MTIE points), or -1 if error               */
/*                                                                          */
/*      sigmadata structure elements for sigma data:                        */
/*          int     points        number of sigma data points               */
/*          float   *tau          pointer to averaging time data array      */
/*          float   *num          pointer to number of data points array    */
/*          float   *sigma        pointer to sigma data array               */
/*          float   *ci           pointer to confidence interval            */
/*          float   *sig_min      pointer to lower sigma error bar array    */
/*          float   *sig_max      pointer to upper sigma error bar array    */
/*          float   *std_dev      pointer to standard deviation array       */
/*          int     *alpha        pointer to noise type exponent array      */
/*          int     *af           pointer to averaging factor               */
/*                                                                          */
/*      Note: Signature of this function is necessarily different from     */
/*      CalcMTIE() because it includes the outer averaging factor loop.     */
/*      It is used differently in main program.                             */
/*                                                                          */
/*      Basic Algorithm:                                                    */
/*          An n-point wide window is moved thru the phase data and the     */
/*          difference between the maximum and minimum phase (time) values  */
/*          is found at each window position.  MTIE is the overall maximum  */
/*          of this time interval error over the whole data set.            */
/*                                                                          */
/*      Fast Algorithm:                                                     */
/*          Calc MTIE in octave steps only.  Store max and min for each tau  */
/*          = AF*tau0 in two vectors.  For next AF, just search these       */
/*          vectors, not all data, for next maxs and mins.                  */
/*                                                                          */
/*      Notes:                                                              */
/*          (1) This function does not handle gaps.                         */
/*          (2) No abort option or progress message for this function       */
/*          (4) # time intervals is N-1, where N = # phase data points, so  */
/*              AF can go from 1 to N-1.                                    */
/*          (3) B&M algorithm uses AF sequence 1,3,7,15,...2^k -1           */
/*              where k is outer index that goes from 1 to <= log2(N)       */
/*                                                                          */
/*      Reference 1:                                                        */
/*          S. Bregni, "Clock Stability Characterization and Measurement    */
/*          in Telecommunications", IEEE Transactions on Instrumentation    */
/*          and Measurement, Vol. 46, No. 6, Dec. 1997, pp. 1284-1294,      */
/*          Eq. 14.                                                         */
/*      Reference 2:                                                        */
/*          S. Bregni and S. Maccabruni, "Fast Computation of Maximum Time  */
/*          Interval Error by Binary Decomposition", IEEE Transactions on   */
/*          Instrumentation and Measurement, Vol. 49, No. 6, Dec. 2000, pp.  */
/*          1240-1244.                                                      */
/*                                                                          */
/*      Revision record:                                                    */
/*          01/26/01    Drafted from CalcMTIE()                             */
/*          01/27/01    Running                                             */
/*          01/28/01    Added progress message                              */
/*                      Changed 2nd param to plotdata struct                */
/*                      Changed min & max arrays from F_TYPE to float       */
/*                      Tried various speed enhancements                    */
/*                      Moved here from stable_r.c to use compiler speed    */
/*                      optimizations - they work the best                  */
/*                      Copy in stable_r.c still has debug code if needed.  */
/*          01/30/01    Added results on-the-fly & progress display         */
/*          03/22/03    Adapted for FrequenC DLL                            */
/*                      Added abort flag & progress function arguments      */
/*                      Not yet fully-integrated into FrequenC DLL          */
/*                      Function prototype noy in frequenc.h                */
/*          04/26/03     Adapted for Version 2.0 source code documentation     */
/*                                                                          */
/* (c) Copyright 2001-3  Hamilton Technical Services  All Rights Reserved    */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

// Struct for sigma run data
struct sigmadata                        /* structure template for sigma data */
{
    int points;                               /* number of sigma data points */
    int rows;                            /* # rows of results = # tau values */
    float *tau;                        /* pointer to averaging time data array */
    float *num;                      /* pointer to number of data points array */
    float *sigma;                             /* pointer to sigma data array */
    float *ci;         /* pointer to confidence interval from SIGMA.TAU file */
    float *sig_min;                  /* pointer to lower sigma error bar array */
    float *sig_max;                  /* pointer to upper sigma error bar array */
    float *std_dev;                    /* pointer to standard deviation array */
    float *df;                         /* pointer to degrees of freedom array */
    int *alpha;            /* pointer to power-law noise type exponent array */
    int *af;                            /* pointer to averaging factor array */
};

int __declspec(dllexport) FAR PASCAL CalcFastMTIE(F_TYPE fX[],
    struct sigmadata *pResults, BOOL *bAbort,
    void *ProgressFunction(int nPercentDone, char *szMessage))
{
    // Local variables
    // Ints
    int i;                          // Inner and general inner index
    int j;                          // Middle index
    int last_j;                     // Previous middle index
    int k;                          // Outer index
    int p;                          // Span
    int nStart;                     // Analysis start
    int nEnd;                       // Analysis end
    int nNum;                       // # analysis points
    int nAF;                        // Averaging factor
    int nStop;                      // Max value of outer index
    int tic;                        // Progress interval
    // Floats
    float fMTIE;                    // Current MTIE value
    float *fSmall;                  // Min phase data value array
    float *fBig;                    // Max phase data value array

    // Initialize progress display
    ProgressFunction(0, "Calculating MTIE");

    // Set analysis limits
    nStart=(int)fX[1];
    nEnd=(int)fX[2];
    nNum=nEnd-nStart+1;

    // Allocate working memory for max and min vectors
    // First pass they must be same size as analysis data array (w/o header)
    // Together they hold all points in overlapping big/small pairs
    // No point in reallocing as they get smaller

    // Allocate maximum array
    if((fBig=(float*)malloc((nNum)*sizeof(float)))==NULL)
    {
        return -1; // Error
    }

    // Allocate minimum array
    if((fSmall=(float*)malloc((nNum)*sizeof(float)))==NULL)
    {
        free(fBig);
        return -1; // Error
    }

    // Initialize the max and min arrays by copying the phase data into them
    for(i=0; i<nNum; i++)
    {
        // Copy data
        fBig[i]=(float)fX[nStart+ARRAY_OFFSET+i-1];
        fSmall[i]=(float)fX[nStart+ARRAY_OFFSET+i-1];
    }

    // Calc MTIE
    // Outer k loop over octave AFs up to max nAF = nNum-1
    // nAF is 2^k -1 and max k is log2(nNum)

    // Find maximum outer index = log2(nNum) as truncated integer
    nStop=(int)(log(nNum)/log(2));

    // Calc MITE points at octave averaging factors up to largest possible
    for(k=1; k<=nStop; k++)
    {
        // Find the current averaging factor nAF = 2^k -1
        nAF=(int)(pow(2, k))-1;

        // Find the current span p = 2^(k-1)
        p=(int)(pow(2, k-1));

        // Initialize previous middle index
        last_j=0;
```

```
// Update the min & max vectors for this AF
// Loop through all nNum-AF points
for(j=0; j<nNum-nAF; j++)
{
    /* Set 1% interval for progress report */

    tic=(int)ceil((nNum-nAF)/100);

    // The max and min values are already initialized
    // Find max & min values for each step in this window of span p
    // Note: This is apparently the section of code that takes the
    // most time as nAF and k get larger, and p gets much larger
    // But its hard to see how one can do any better than the
    // max() and min() macros.
    // This code was tried with index arithmitic outside loop
    // and just incrementing inside - little difference in speed
    // Pointer arithmitic was also tried - ditto
    // What does help is to ask the complier to optimize for speed
    // That makes 5:1 improvement for any of the code versions
    for(i=1; i<=p; i++)
    {
        fBig[j]=max(fBig[j+i], fBig[j]);
        fSmall[j]=min(fSmall[j+i], fSmall[j]);
    }

    // Display progress
    // Note:  This code provides updates only if index has changed
    // by at least tic amount (1%)
    if((j-last_j)>=tic)
    {
        ProgressFunction((int)(100.0*j/(nNum-nAF)),
            "Calculating MTIE");
        last_j=j;
    }
}

// Initialize the MTIE value
// Note that the results array is zero-based
fMTIE=(float)(fBig[0]-fSmall[0]);

// Loop thru the rest of the nNum-nAF points in each vector
for(i=1; i<nNum-nAF; i++)
{
    // MTIE is the overall maximum p-p excursion
    fMTIE=(float)(max((fBig[i]-fSmall[i]), fMTIE));
}

// Store results
pResults->sigma[k-1]=fMTIE;
pResults->points=k;
pResults->af[k-1]=nAF;
pResults->num[k-1]=(float)(nNum-nAF);
// Note: tau calc external to this function
```

```
        // Check for abort
        if(bAbort)
        {
            // Bail out - program flow goes to code below, then to IDCANCEL
            // handler, and finally back to where this function was called
            break;
        }
    }

    // Free working memory
    free(fBig);
    free(fSmall);

    // Done
    // Clear progress display
    ProgressFunction(0, "");
    return k;
}
```

The FrequenC Library

NAME: CalcFreqModSigma	**FUNCTION:** Calculate modified Allan deviation for frequency data

SYNOPSIS: `int CalcFreqModSigma(F_TYPE y[], F_TYPE *p_sig, int m)`

F_TYPE y[]	Frequency data array:
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

F_TYPE *p_sig	Pointer to sig, the modified Allan deviation for the frequency data
int m	Averaging factor, m = tau/tau_0
RETURN: int	The # of non-gap data points processed, or -1 if error.

REMARKS:
All zeros are treated as gaps in frequency data.
Only data points between start and end analysis limits are analyzed.
There must be at least 2 non-gap analysis points.
Maximum m = (int) (# freq data points + 1) / 3.
This calculation of the modified Allan variance from frequency data that may contain gaps is slow. Use ConvFreqToPhase() and CalcPhaseModSigma(), or preferably ConvFreqToPhase(), FillGaps() and CalcFastModSigma() if the # of data points is large.

EXAMPLE:
```
#include "frequenc.h"                         /* FrequenC header file */
F_TYPE y[512];                                /* frequency data array */
F_TYPE sig;                               /* modified Allan deviation */
int m;                                            /* averaging factor */
int num;                                   /* # data points processed */
.
.
m=2;                                         /* assign value to m */
num=CalcFreqModSigma(y, &sig, m);               /* calc mod sigma */
if(num==-1)                                    /* check for error */
{
    printf("\nError");                           /* error message */
}
else
{
    printf("\nMod Sigma = %e", sig);         /* display mod sigma */
    printf("\n# Data Points Analyzed = %d", num);   /* display num */
}
```

SEE ALSO: CalcFreqSigma(), CalcPhaseModSigma()

REFERENCE: NIST Technical Note 1337

```
/****************************************************************************/
/*                                                                          */
/*                          CalcFreqModSigma()                              */
/*                                                                          */
/*      Function to calculate modified Allan deviation for freq dat        */
/*                                                                          */
/*      Parameters:     F_TYPE y[]    = frequency data                      */
/*                                      y[0] = # data points                */
/*                                      y[1]  = analysis start              */
/*                                      y[2] = analysis end                 */
/*                      F_TYPE *p_sig = pointer to mod Allan deviation      */
/*                      int m         = averaging factor = tau/tau_0        */
/*                                                                          */
/*      Return:         int           = # analysis points,                  */
/*                                      or -1 if error                      */
/*                                                                          */
/*      Notes:          Zeros treated as gaps in frequency data.            */
/*                      Frequency data is not averaged before calling       */
/*                      this function.  The data consists of num-1 freq     */
/*                      samples at interval tau_0.  m must be >0.           */
/*                      Denominator term (num-3m+1) must be >=1             */
/*                      This sets max m=(int)(num/3), where num=            */
/*                      (# frequency points +1) before averaging.           */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/16/91    Created                                             */
/*          12/31/91    Renamed                                             */
/*          01/16/92    Edited per FrequenC documentation                   */
/*          01/17/92    Edited title                                        */
/*          01/18/92    Edited title block                                  */
/*          01/19/92    Added error traps                                   */
/*          02/03/96    Modified for use as Win 3.1 DLL                     */
/*          01/01/98    Changes for MS VC++ compatibility & warnings        */
/*          04/26/03    Adapted for Version 2.0 source code documentation  */
/*                                                                          */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcFreqModSigma(F_TYPE y[],
    F_TYPE *p_sig, int m)
{
    double sum_i, sum_j, sum_k;     /* summing variables to calc mod sigma */
    float fm;                                          /* float version of m */
    int i;                                  /* index for inner summing loop */
    int j;                                  /* index for outer summing loop */
    int k;                                /* index for inner averaging loop */
    int n_avg;                                /* # points in averaging loop */
    int n_sum=0;                            /* # points in outer summing loop */
    int num = (int)(*(y+2)-*(y+1)+2);  /* # phase data points for analysis */
                                                     /* = # freq points +1 */
    if( (m<1) || (num<3*m) )
    {
        return(-1);                                              /* error */
    }

    fm=(float)m;
    sum_j=0;
    for(j=1; j<=num-(3*m)+1; j++)                            /* outer loop */

          /* j goes from 1 to N-3m+1 where N=# phase points & m=avg factor */
                                           /* regardless of analysis limits */
    {
        sum_i=0;
        for(i=j+(int)(*(y+1))-1; i<=m+j-1+(int)(*(y+1))-1; i++)   /* inner */
                   /* i is the actual freq data index;  it includes a term */
                    /* *(y+1)-1 to account for the analysis starting point */
                 /* the data array offset +2 is included in the code below */
        {
            sum_k=0;
            n_avg=0;                        /* # freq differences averaged */
            for(k=i; k<=i+m-1; k++)             /* find frequency averages */
            {
                if(*(y+k+2+m) && *(y+k+2))       /* skip if either is zero */
                {
                    sum_k += *(y+k+2+m) - *(y+k+2);
                    n_avg++;
                }                                             /* end if */
            }                                                /* end for */
            if(n_avg)                /* test to avoid div by 0 if all gaps */
            {
                sum_i+=sum_k/n_avg;
            }
        }                                                    /* end for */
        if(sum_i)
        {
            sum_j += SQR(sum_i);
            n_sum++;
        }                                                         /* if */
    }                                                        /* end for */
    if(n_sum>0)
    {
        *p_sig = sqrt((sum_j)/(2.0*fm*fm*n_sum));
        return(n_sum);                               /* # analysis points */
    }
    else
    {
        return(-1);                                            /* error */
    }
}                                             /* end CalcFreqModSigma() */
```

The FrequenC Library

NAME: `CalcFastModSigma`	**FUNCTION:** Quickly calculate modified Allan deviation for gapless phase data

SYNOPSIS:
```
int CalcFastModSigma(F_TYPE x[], F_TYPE *p_sig, F_TYPE tau, int m)
```

`F_TYPE x[]`	Phase data array:
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

`F_TYPE *p_sig`	Pointer to sig, the modified Allan deviation for the phase data
`F_TYPE tau`	The averaging time for the phase data
`int m`	Averaging factor, m = tau/tau_0
RETURN: `int`	The # of data points processed, or -1 if error

REMARKS:
No gaps are allowed in the phase data.
Only data points between start and end analysis limits are analyzed.
There must be at least 3 analysis points. Tau must be >0.
Maximum m = (int)(# phase data points/3, and m must be >0.
This calculation of the modified Allan variance from gapless phase data is faster than any other method. Use CalcPhaseModSigma() if there are gaps in the phase data, or, preferably, FillGaps() and CalcFastModSigma() if the # of phase data points is large.

EXAMPLE:
```
#include "frequenc.h"                      /* FrequenC header file */
F_TYPE x[512];                                /* phase data array */
F_TYPE sig;                                          /* mod sigma */
F_TYPE tau;                                     /* averaging time */
int m;                                        /* averaging factor */
int num;                              /* # data points processed */
.
.
tau=1.0; m=2;                         /* assign values to tau and m */
num=CalcFastModSigma(x, &sig, tau, m);          /* calc mod sigma */
if(num==-1)                                    /* check for error */
{
   printf("\nError");                            /* error message */
}
else
}
    printf("\nMod Sigma = %e", sig);         /* display mod sigma */
    printf("\n# Data Points Analyzed = %d", num);   /* display num */
}
```

SEE ALSO: `CalcPhaseModSigma()`

REFERENCE: NIST Technical Note 1337

```
/****************************************************************************/
/*                                                                          */
/*                        CalcFastModSigma()                                */
/*                                                                          */
/*      Function to calculate modified Allan variance for phase data        */
/*      Uses faster algorithm that avoids recalculation of inner sums       */
/*                                                                          */
/*      Reference:      D.W. Allan, "Time and Frequency Metrology:          */
/*                      Current Status and Future Considerations",          */
/*                      Proc. 5th EFTF, pp. 1-9, March, 1991                */
/*                                                                          */
/*      Parameters:     F_TYPE x[]    = phase data                          */
/*                                      x[0] = # data phase points          */
/*                                      x[1]  = analysis start              */
/*                                      x[2] = analysis end                 */
/*                      F_TYPE *p_sig = pointer to mod Allan deviation      */
/*                      F_TYPE tau    = analysis averaging time = mùtau_0   */
/*                      int m         = averaging factor = tau/tau_0        */
/*                                                                          */
/*      Return:         int           = # analysis points,                  */
/*                                      or -1 if error                      */
/*                                                                          */
/*      Notes:          No gaps allowed in data; check for embedded zeros   */
/*                      before using this function. If gaps, use (slower)   */
/*                      CalcPhaseModSigma(), or use FillGaps() first.       */
/*                      Phase data is not averaged before calling this      */
/*                      function.  The data consists of num phase samples   */
/*                      at interval tau_0.  Denominator term (num-3m+1)     */
/*                      must be >0.  This sets max m=(int)(num/3),          */
/*                      where num=# phase points before averaging.          */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/16/91    Created                                             */
/*          12/31/91    Renamed                                             */
/*          01/16/92    Edited per FrequenC documentation                   */
/*          01/17/92    Edited title                                        */
/*          01/18/92    Edited title block                                  */
/*          02/03/96    Modified for use as Win 3.1 DLL                     */
/*          01/01/98    Changes for MS VC++ compatibility & warnings        */
/*          04/26/03    Adapted for Version 2.0 source code documentation   */
/*                                                                          */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcFastModSigma(F_TYPE x[],
    F_TYPE *p_sig, F_TYPE tau, int m)
{
    double sum_i, sum_j;        /* summing variables to calc modified sigma */
    float fm;                                           /* float version of m */
    int i;                                     /* index for inner summing loop */
    int j;                                     /* index for outer summing loop */
    int s = (int)*(x+1);                                    /* analysis start */
    int e = (int)*(x+2);                                      /* analysis end */
    int n_sum=0;                             /* # points in outer summing loop */
    int num = e-s+1;                       /* # phase data points for analysis */

    fm=(float)m;
    sum_j=0;
    sum_i=0;

    if( (num<3*m) || (m<1)  || (tau<=0) )
    {
        return(-1);                                             /* error */
    }



/* 1st time find the average of all m phase points by summing them */
    for(i=s; i<m+s; i++)               /* i is the actual phase data index */
      /* it includes the term s to account for the analysis starting point */
                 /* the data array offset +2 is included in the code below */
    {
        sum_i += *(x+i+2+(2*m)) - (2*(*(x+i+2+m))) + *(x+i+2);
    }

    sum_j += SQR(sum_i);
    n_sum++;

/* for rest of points need only drop 1st sum and add next sum */
    for(j=2; j<=num-(3*m)+1; j++)             /* j goes from 2 to num-3m+1 */
                                     /* where num=# phase points & m=avg factor */
                                               /* regardless of analysis limits */
    {
        i=j+s-2;                       /* index of old sum_i being dropped */
        sum_i -= *(x+i+2+(2*m)) - (2*(*(x+i+2+m))) + *(x+i+2);

        i=m+j-1+s-1;                    /* index of next sum_i being added */
        sum_i += *(x+i+2+(2*m)) - (2*(*(x+i+2+m))) + *(x+i+2);

        sum_j += SQR(sum_i);
        n_sum++;
    }                                                          /* end if */

    if(n_sum>0)                                        /* avoid div by 0 */
    {
        *p_sig = sqrt((sum_j)/(2.0*tau*tau*fm*fm*n_sum));
        return(n_sum);                               /* # analysis points */
    }
    else
    {
        return(-1);                                             /* error */
    }
}                                                /* end CalcFastModSigma() */
```

The FrequenC Library	
NAME: CalcFreqOffset	**FUNCTION:** Calc freq for phase data as slope of least-squares linear fit to phase data: x(t)=a+bt
SYNOPSIS: int CalcFreqOffset(F_TYPE x[], F_TYPE *a, F_TYPE *b, F_TYPE *v)	

F_TYPE x[]	Phase data array
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE *a	parameter one, and description of what it is
F_TYPE *b	Pointer to slope = freq offset
F_TYPE *v	Pointer to fit variance
RETURN: int	# non-gap data points processed, or -1 if error

REMARKS:
Only data points between start and end analysis limits are analyzed.
Embedded zeros are treated as gaps in phase data.
# non-gap data points must be at least 2 and at least 3 to calc finite fit variance.

EXAMPLE:

```
#include "frequenc.h"                        /* FrequenC header file */
F_TYPE x[512];                                   /* phase data array */
F_TYPE a;                                            /* phase offset */
F_TYPE b;                                             /* phase slope */
F_TYPE c;                                            /* fit variance */
int num;                                /* # data points processed */
.
.
num=CalcFreqOffset(x, &a, &b, &c);                  /* call function */
printf("\nFreq Offset = %d", b);                   /* display result */
```

SEE ALSO: RemoveFreqOffset()

REFERENCE: NIST Technical Note 1337

```
/******************************************************************************/
/*                                                                            */
/*                         CalcFreqOffset()                                   */
/*                                                                            */
/*      Function to calculate freq for phase data: x(t)=a+bt                  */
/*                                                                            */
/*      Parameters:     F_TYPE x[]  = phase data                              */
/*                                    x[0] = # data points                    */
/*                                    x[1] = analysis start                   */
/*                                    x[2] = analysis end                     */
/*                      F_TYPE p_a  = pointer to y-intercept                  */
/*                      F_TYPE p_b  = pointer to slope = freq offset          */
/*                      F_TYPE p_v  = pointer to fit variance                 */
/*                                                                            */
/*      Return:         int         = # analysis points,                      */
/*                                    or -1 if error                          */
/*                                                                            */
/*      Notes:          Embedded zeros treated as gaps in phase data          */
/*                      # non-gap data points must be at least 2              */
/*                      and at least 3 to calc finite fit variance            */
/*                                                                            */
/*      Revision record:                                                      */
/*          02/22/97    Adapted from CalcLinFreqDrift() of FrequenC           */
/*          01/01/98    Changes for MS VC++ compatibility & warnings          */
/*          04/26/03   Adapted for Version 2.0 source code documentation      */
/*                                                                            */
/* (c) Copyright 1997-2003 Hamilton Technical Services All Rights Rserved    */
/*                                                                            */
/******************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcFreqOffset(F_TYPE x[], F_TYPE *p_a,
    F_TYPE *p_b, F_TYPE *p_v)
{
    int i;
    int num = (int)(*(x+2)-*(x+1)+1);
    double sum_x = 0;
    double sum_y = 0;
    double sum_xx = 0;
    double sum_yy = 0;
    double sum_xy = 0;

    for( i = (int)(*(x+1)+2); i <= (int)(*(x+2)+2); i++)
    {
        if( (*(x+i)) || (i==3) || (i==*(x+2)+2) )           /* check for gap */
        {
            sum_x  += i-2;
            sum_y  += *(x+i);
            sum_xx += SQR ( (float) (i-2) );            /* must cast to float */
            sum_yy += SQR ( *(x+i) );
            sum_xy += (i-2) * *(x+i);
        }
        else
        {
            num--;
        }
    }
    if(num<2)
    {
        return(-1);                                              /* error */
    }

    *p_b = (sum_xy - ((sum_x) * (sum_y) / num));
    *p_b = *p_b / (sum_xx- ((sum_x) * (sum_x)) / num);
    *p_a = ((sum_y) / num) - (*p_b * sum_x / num);
    *p_v = 0;                            /* fit variance 0 if only 2 points */

    if(num>=3)     /* need at least 3 analysis points to calc fit variance */
    {
        *p_v = sum_yy  - (sum_y * sum_y / num);
        *p_v -= *p_b * *p_b * (sum_xx - (sum_x * sum_x / num ));
        *p_v /= (num-2);
    }

    return(num);
}
```

The FrequenC Library	
NAME: CalcFreqOverlap HadamardDev	**FUNCTION:** Description of what function does
SYNOPSIS: `int FunctionName( F_TYPE z[], F_TYPE one, F_TYPE two)`	

F_TYPE z[]	Phase or frequency data array
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysios end point

F_TYPE one	parameter one, and description of what it is
F_TYPE two	parameter two, and description of what it is
RETURN: int	# non-gap data points processed
REMARKS: Only data points between start and end analysis limits are modified. Embedded zeros are treated as gaps in phase data. All zeros are treated as gaps in frequency data.	

EXAMPLE:

```
#include <frequenc.h>                          /* FrequenC header file */
F_TYPE z[512];                                          /* data array */
F_TYPE one;                                          /* parameter one */
F_TYPE two;                                          /* parameter two */
int num;                                   /* # data points processed */
.
.
one=1.0;                                   /* set 1st parameter value */
two=2.0;                                   /* set 2nd parameter value */
num=FunctionName(z, one, two);                       /* call function */
printf("\n# Data Points Scaled = %d", num);            /* display num */
```

SEE ALSO: RelatedFunctionName()

REFERENCE: NIST Technical Note 1337

```
/***************************************************************************/
/*                                                                         */
/*                    CalcFreqOverlapHadamardDev()                         */
/*                                                                         */
/*      Function to calc Hadamard deviation using overlapping samples      */
/*      of frequency data                                                  */
/*                                                                         */
/*      Parameters:     F_TYPE y[]    = frequency data array               */
/*                                      y[0] = # data points               */
/*                                      y[1]   = analysis start            */
/*                                      y[2] = analysis end                */
/*                      F_TYPE *p_sig = pointer to Hadamard deviation      */
/*                      int m         = averaging factor                   */
/*                                                                         */
/*      Return:         int           = # analysis points,                 */
/*                                      or -1 if error                     */
/*                                                                         */
/*      Note:           Zeros treated as gaps in frequency data.           */
/*                      Must have at least 1 analysis point; this          */
/*                      requires at least two adjacent non-gap frequency   */
/*                      data points.  m must be >0 and must have at least  */
/*                      3 non-gap adjacent frequency data points.          */
/*                                                                         */
/*      Revision record:                                                   */
/*          03/06/99    Created from CFOS.C of Frequenc32 Library          */
/*          04/26/03   Adapted for Version 2.0 source code documentation   */
/*                                                                         */
/* (c) Copyright 1999-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                         */
/***************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcFreqOverlapHadamardDev(F_TYPE y[],
    F_TYPE *p_sig, int m)
{
    double sum_j, sum_i;                /* summing variables to calc sigma */
    float fm=(float) m;                                /* float version of m */
    int i;                                      /* inner index for averaging */
    int j;                          /* outer index for summing 1st differences */
    int num=0;                                          /* # analysis points */
    int n;                                         /* # frequency data points */

    n=(int)(*(y+2)- *(y+1)+2);                  /* total # phase data points */
                                           /* N=M+1 where M is # freq points */
    if(m<1)
    {
        return(-1);                                                /* error */
    }

    sum_j=0;
    for( j=1; j<=n-(3*m); j++)                                 /* outer loop */
                                  /* j goes over basic range from 1 to N-3m */
    {
        sum_i=0;
        for(i=(int)(j+(*(y+1)-1)); i<=(int)(j+(*(y+1))-1+m-1); i++)
                                                       /* inner loop */
                        /* i index includes analysis start limit of *(y+1) */
                       /* data array offset of 2 is included in code below */
        {
            if(*(y+i+2*m+2) && *(y+1+m+2) && *(y+i+2))

            {
                sum_i += *(y+i+2*m+2) - *(y+i+m+2) - *(y+i+m+2) + *(y+i+2);
            }                                                   /* end if */
        }                                                      /* end for */
        if(sum_i)
        {
            sum_j += SQR(sum_i);
            num++;
        }                                                       /* end if */
    }                                                          /* end for */
    if(num>0)
    {
        *p_sig=sqrt(sum_j/(6.0*fm*fm*num));
        return(num);                                /* # analysis points */
    }
    else
    {
        return(-1);                                              /* error */
    }
}                                    /* end CalcFreqOverlapHadamardDev() */
```

The FrequenC Library	
NAME: CalcFreqOverlapSigma	**FUNCTION:** Calculate overlapping Allan deviation for frequency data
SYNOPSIS: int CalcFreqOverlapSigma(F_TYPE y[], F_TYPE *p_sig, int m)	

F_TYPE y[]	Frequency data array:
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

F_TYPE *p_sig	Pointer to sig, the overlapping Allan deviation for the frequency data
int m	Averaging factor, m = tau/tau_0
RETURN: int	The # of non-gap data points processed, or -1 if error

REMARKS:
Only data points between start and end analysis limits are modified.
There must be at least 2 non-gap analysis points.
Maximum m = (int)(# freq data points - 1)/2, and m must be >0.
All zeros are treated as gaps in frequency data.

EXAMPLE:

```
#include <stdio.h>                                 /* for printf() */
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE y[512];                                 /* frequency data array */
F_TYPE sig;                             /* overlapping Allan deviation */
int m;                                             /* averaging factor */
int num;                                     /* # data points processed */
.
.
m=2;                                              /* assign value to m */
num=CalcFreqSigma(y, &sig, m);              /* calc overlapping sigma */
if(num==-1)                                         /* check for error */
{
    printf("\nError");                                /* error message */
}
else
{
    printf("\nSigma = %e", sig);      /* display overlapping sigma */
    printf("\n# Data Points Analyzed = %d", num);   /* display num */
}
```

SEE ALSO: CalcFreqSigma(), CalcPhaseOverlapSigma()

REFERENCE: NIST Technical Note 1337

```
/***************************************************************************/
/*                                                                         */
/*                        CalcFreqOverlapSigma()                           */
/*                                                                         */
/*      Function to calculate Allan deviation using overlapping samples    */
/*      of frequency data                                                  */
/*                                                                         */
/*      Parameters:     F_TYPE y[]    = frequency data array               */
/*                                      y[0] = # data points               */
/*                                      y[1]   = analysis start            */
/*                                      y[2] = analysis end                */
/*                      F_TYPE *p_sig = pointer to Allan deviation         */
/*                      int m         = averaging factor                   */
/*                                                                         */
/*      Return:         int           = # analysis points,                 */
/*                                      or -1 if error                     */
/*                                                                         */
/*      Note:           Zeros treated as gaps in frequency data.           */
/*                      Must have at least 1 analysis point; this          */
/*                      requires at least two adjacent non-gap frequency   */
/*                      data points.  m must be >0 and the # non-gap       */
/*                      frequency data points M must be > 2m-1.            */
/*                                                                         */
/*      Revision record:                                                   */
/*          12/16/91    Created                                            */
/*          12/31/91    Renamed                                            */
/*          01/18/92    Edited title block                                 */
/*                      Added error traps                                  */
/*          02/03/96    Modified for use as Win 3.1 DLL                    */
/*          01/01/98    Changes for MS VC++ compatibility & warnings       */
/*          01/29/99    Test for zero 1st difference removed to avoid      */
/*                      problem with low-precision data where two          */
/*                      adjacent frequency data point are identical.       */
/*          03/08/03    Brought into Ver 2.0 of FrequenC lib               */
/*          04/26/03   Adapted for Version 2.0 source code documentation   */
/*                                                                         */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                         */
/***************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcFreqOverlapSigma(F_TYPE y[],
    F_TYPE *p_sig, int m)
{
    double sum_j, sum_i;                /* summing variables to calc sigma */
    float fm=(float) m;                                /* float version of m */
    int i;                                      /* inner index for averaging */
    int j;                         /* outer index for summing 1st differences */
    int num=0;                                         /* # analysis points */
    int n;                                       /* # frequency data points */

    n=(int)(*(y+2)-*(y+1)+2);                   /* total # phase data points */
                                             /* N=M+1 where M is # freq points */
    if( (m<1) || (n<2*m) )
    {
        return(-1);                                                /* error */
    }

    sum_j=0;
    for( j=1; j<=n-(2*m); j++)                               /* outer loop */
                             /* j goes over basic range from 1 to N-2m */
    {
        sum_i=0;
        for(i=(int)(j+(*(y+1)-1)); i<=(int)(j+(*(y+1))-1+m-1); i++)
                                                             /* inner loop */
                          /* i index includes analysis start limit of *(y+1) */
                         /* data array offset of 2 is included in code below */
        {
            if(*(y+i+m+2) && *(y+i+2))
            {
                sum_i += *(y+i+m+2) - *(y+i+2);
            }                                                    /* end if */
        }                                                       /* end for */

        sum_j += SQR(sum_i);
        num++;
    }                                                           /* end for */
    if(num>0)
    {
        *p_sig=sqrt(sum_j/(2.0*fm*fm*num));
        return(num);                                  /* # analysis points */
    }
    else
    {
        return(-1);                                               /* error */
    }
}                                           /* end CalcFreqOverlapSigma() */
```

The FrequenC Library	
NAME: CalcFreqSigma	**FUNCTION:** Calculate Allan deviation for frequency data
SYNOPSIS: int CalcFreqSigma(F_TYPE y[], F_TYPE *p_sig)	

F_TYPE y[]	Frequency data array:
	    y[0] = # data points
	    y[1] = analysis start point
	    y[2] = analysis end point

F_TYPE *p_sig	Pointer to sig, the Allan deviation for the frequency data
RETURN: int	The # of non-gap data points processed, or -1 if error

REMARKS:
Only data points between start and end analysis limits are analyzed.
There must be at least 2 non-gap analysis points.
All zeros are treated as gaps in frequency data.

EXAMPLE:
```
#include "frequenc.h"                              /* FrequenC header file */
F_TYPE y[512];                                     /* frequency data array */
F_TYPE sig;                                             /* Allan deviation */
int num;                                      /* # data points processed */
.
.
num=CalcFreqSigma(y, &sig);                                  /* calc sigma */
if(num==-1)                                             /* check for error */
{
    printf("\nError");                                    /* error message */
}
else
{
    printf("\nSigma = %e", sig);                          /* display sigma */
    printf("\n# Data Points Analyzed = %d", num);           /* display num */
}
```

SEE ALSO: CalcPhaseSigma()

REFERENCE: NIST Technical Note 1337

```
/****************************************************************************/
/*                                                                          */
/*                         CalcFreqSigma()                                  */
/*                                                                          */
/*      Function to calculate Allan variance from frequency data            */
/*                                                                          */
/*      Parameters:     F_TYPE y[]    = frequency data                      */
/*                                      y[0] = # data points                */
/*                                      y[1] = analysis start               */
/*                                      y[2] = analysis end                 */
/*                      F_TYPE *p_sig = pointer to Allan deviation          */
/*                                                                          */
/*      Returns:        int           = # analysis points,                  */
/*                                      or -1 if error                      */
/*                                                                          */
/*      Note:           Zero value treated as gap in frequency data.        */
/*                      Must have at least 1 analysis point; this           */
/*                      requires at least 2 adjacent non-gap frequency      */
/*                      data points.                                        */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/16/91    Created                                             */
/*          12/26/91    Changed to int returning # analysis points          */
/*          12/31/91    Renamed                                             */
/*          01/18/92    Edited title block                                  */
/*                      Added error trap                                    */
/*          02/03/96    Modified for use as Win 3.1 DLL                     */
/*          01/01/98    Changes for MS VC++ compatibility & warnings        */
/*          04/26/03   Adapted for Version 2.0 source code documentation   */
/*                                                                          */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcFreqSigma(F_TYPE y[],
    F_TYPE *p_sig)
{
    double sum;                           /* summing variable to calc sigma */
    int i;                                             /* index for summing */
    int num = 0;                                       /* # analysis points */

    for( i = (int)(*(y+1)+2), sum = 0; i < (int)(*(y+2)+2); i++)
                                             /* index goes from 1 to n-1 */
                     /* for n data points from 1 to n; data starts at y[3] */
    {
        if( *(y+i) && *(y+i+1) )           /* skip if either value is zero */
        {
            sum += SQR( *(y+i+1) - *(y+i) );
            num++;
        }                                                       /* end if */
    }                                                          /* end for */
    if(num>0)
    {
        *p_sig = ( sqrt( sum / (2.0 * num)));
        return(num);                                   /* # analysis points */
    }
else
    {
        return(-1);                                                       /* error */
    }
}                                                          /* end CalcFreqSigma() */
```

The FrequenC Library

NAME: CalcFreqStdDev	**FUNCTION:** Calculate standard deviation for frequency data
SYNOPSIS: `int CalcFreqStdDev(F_TYPE y[], F_TYPE *p_dev)`	

F_TYPE y[]	Frequency data array:
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

F_TYPE *p_dev	Pointer to dev, the standard deviation for the frequency data
RETURN: int	The # of non-gap data points processed, or -1 if error

REMARKS:
Only data points between start and end analysis limits are analyzed.
There must be at least 2 non-gap analysis points.
All zeros are treated as gaps in frequency data.

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE y[512];                                 /* frequency data array */
F_TYPE dev;                                      /* standard deviation */
int num;                                     /* # data points processed */
.
.
num=CalcFreqSigma(y, &dev);                              /* calc sigma */
if(num==-1)                                         /* check for error */
{
    printf("\nError");                                /* error message */
}
else
{
    printf("\nStd Dev = %e", dev);                    /* display sigma */
    printf("\n# Data Points Analyzed = %d", num);       /* display num */
}
```

SEE ALSO: CalcPhaseStdDev()

REFERENCE: NIST Technical Note 1337

```
/*****************************************************************************/
/*                                                                           */
/*                         CalcFreqStdDev()                                  */
/*                                                                           */
/*      Function to calculate standard deviation from frequency data        */
/*                                                                           */
/*      Parameters:     F_TYPE y[]   = frequency data                        */
/*                                     y[0] = # data points                  */
/*                                     y[1]  = analysis start                */
/*                                     y[2] = analysis end                   */
/*                      F_TYPE *p_dev = pointer to standard deviation        */
/*                                                                           */
/*      Return:         int          = # analysis points,                    */
/*                                      or -1 if error                       */
/*                                                                           */
/*      Note:           All zeros treated as gaps in frequency data.         */
/*                      Must have at least 2 non-gap freq data points.       */
/*                                                                           */
/*      Revision record:                                                     */
/*          12/16/91    Created                                              */
/*          12/31/91    Renamed                                              */
/*          01/18/92    Edited title block                                   */
/*          01/19/92    Added error trap                                     */
/*          02/03/96    Modified for use as Win 3.1 DLL                      */
/*          01/01/98    Changes for MS VC++ compatibility & warnings         */
/*          04/26/03   Adapted for Version 2.0 source code documentation    */
/*                                                                           */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcFreqStdDev(F_TYPE y[],
    F_TYPE *p_dev)
{
    double sum=0;                                      /* summing variable */
    double avg;                                /* mean value of freq data */
    int i;                                             /* index for summing */
    int num = (int)(*(y+2)-*(y+1)+1);                /* # analysis points */

    for(i=(int)(*(y+1)+2); i<=(int)(*(y+2))+2; i++)        /* calc average */
    {
        if(*(y+i))
        {
            sum += *(y+i);
        }
        else
        {
            num--;
        }
    }
    if((num-1)>0)                      /* check for <2 non-gap data points */
    {
        avg = sum/num;
    }
    else
    {
        return(-1);                                             /* error */
    }
    sum=0;
    for(i=(int)(*(y+1)+2); i<=(int)(*(y+2)+2); i++)       /* calc variance */
    {
        if(*(y+i))                                    /* skip if value is zero */
        {
            sum += SQR(*(y+i)- avg);
        }                                                            /* end if */
    }                                                               /* end for */
    *p_dev = (sqrt(sum/(num- 1)));
    return(num);                                          /* # analysis points */
}                                                      /* end CalcFreqStdDev() */
```

The FrequenC Library

NAME: ConvFreqToPhase	**FUNCTION:** Convert frequency data to phase data

SYNOPSIS:
```
int ConvFreqToPhase(F_TYPE y[], F_TYPE x[], F_TYPE tau,BOOL do_norm)
```

F_TYPE y[]	Frequency data array:
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

F_TYPE x[]	Phase data array
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE tau	Averaging time of frequency data
BOOL do_norm	Normalization flag: 0 = no, 1 = yes
RETURN: int	The # of non-gap data points processed, or -1 if error

REMARKS:
Only data between start and end analysis limits are converted.
All zeros are treated as gaps in frequency data.
The averaging time (tau) of the phase data will be the same as that of the frequency data. The original frequency data is unchanged.
The do_norm flag determines whether or not the frequency data is normalized to zero mean before conversion to phase data.

EXAMPLE:
```
#include "frequenc.h"                           /* FrequenC header file */
F_TYPE y[512];                                  /* frequency data array */
F_TYPE x[513];                                      /* phase data array */
F_TYPE tau;                                           /* averaging time */
int num;                                     /* # data points processed */
BOOL do_norm;                                     /* normalization flag */
.
.
do_norm=TRUE;                             /* set flag to do normalization */
num=ConvFreqToPhase(y, x, tau, do_norm);  /* convert freq to phase */
 if(num==-1)                                         /* check for error */
{
    printf("\nError");                                 /* error message */
}
else
{
    printf("\n# Data Points Converted = %d", num);  /* display num */
}
```

SEE ALSO: ConvPhaseToFreq()

REFERENCE: None

```
/*****************************************************************************/
/*                                                                           */
/*                        ConvFreqToPhase()                                  */
/*                                                                           */
/*      Function to convert frequency data to phase (time) data              */
/*                                                                           */
/*      Parameters:     F_TYPE y[]   = frequency data                        */
/*                                     y[0] = # data points                  */
/*                                     y[1] = analysis start                 */
/*                                     y[2] = analysis end                   */
/*                      F_TYPE x[]   = phase data                            */
/*                      F_TYPE tau   = time interval between freq samples    */
/*                      BOOL do_norm = normalization flag:                   */
/*                                     0=don't normalize, 1=normalize        */
/*                                                                           */
/*      Return:        int           = # frequency data gaps,                */
/*                                     or -1 if error                        */
/*                                                                           */
/*      Notes:          Calls FrequenC functions mean() and normalize().     */
/*                      Freq data normalized before conversion if do_norm    */
/*                      is TRUE.  This option allows original freq data      */
/*                      to remain unnormalized.  tau must be >0.             */
/*                      Zeros treated as gaps in frequency data.             */
/*                      Phase data in gap extrapolated from last previous    */
/*                      non-zero frequency value.                            */
/*                      Function processes n = y[2] - y[1] +1 freq points    */
/*                      and produces n+1 phase points.                       */
/*                                                                           */
/*      Revision record:                                                     */
/*          08/28/02    Created from ConvFreqToPhase() of FrequenC library   */
/*                      Changed code to preserve gap instesd of linear       */
/*                      integrated phase during gap                          */
/*          08/30/02    Use average freq for gap interpolation               */
/*          03/08/03    Brought into Ver 2.0 of FrequenC lib                 */
/*          04/26/03    Adapted for Version 2.0 source code documentation    */
/*                                                                           */
/* (c) Copyright 1991-2003 Hamilton Technical Services  All Rights Reserved  */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL ConvFreqToPhase(F_TYPE y[],
    F_TYPE x[], F_TYPE tau, BOOL do_norm)
{
    int g;                                                  /* # freq gaps */
    int i;                                                      /* y index */
    int j;                                                      /* x index */
    int num;                                              /* # freq points */
    F_TYPE y_last;                               /* last non-zero freq value */
    F_TYPE y_avg;                                     /* average freq value */
    F_TYPE xbar, *p_xbar;                     /* avg of x[] and pointer to it */

    if(tau<=0)
    {
        return(-1);                                               /* error */
    }

    num = (int)(y[2]-y[1]+1);                   /* set num = # freq points */
    x[0] = num;                                          /* # phase points */
```

```
x[1] = 1;                                       /* # of 1st phase point */
x[2] = num;                                    /* # of last phase point */
p_xbar = &xbar;                            /* assign address to pointer */

/* put freq data to be converted into x[] array */
j=3;                                       /* initialize index for x[] */
for(i = (int)(y[1]+2); i <= (int)(y[2]+2); i++ )
{
    x[j] = y[i];                                      /* copy freq data */
    j++;                                             /* increment index */
}

// Find average frequency value
CalcMean(x, &y_avg, 1);                                /* calc avg freq */

if(do_norm)                /* normalize frequency data to be converted */
{
    if((CalcMean( x, p_xbar, 1))==-1)         /* calc average frequency */
    {
        return(-1);                                /* pass along error */
    }
    if((NormalizeData( x, xbar, 1))==0)      /* normalize frequency data */
    {
        return(-1);                                /* pass along error */
    }
}                                                             /* end if */

/* shift normalized frequency data up 1 position in x[] array */
for(i = num+2; i >= 3; i-- )
{
    x[i+1] = x[i];
}
                                            /* add 1 to # of data points */
x[0]++;
x[2]++;
num++;

/* convert frequency data to phase data */
x[3] = 0;                                 /* set 1st phase value to zero */
y_last = 0;                                         /* ditto for y_last */
g=0;                                               /* initialize # gaps */
for(i = 4; i < num+3; i++ )
                                /* phase data goes from y[3] to y[num+2] */
{
    if( x[i] )                                   /* freq point not a gap */
    {
        y_last = x[i];                              /* save current freq */
        x[i] = x[i-1] + ( x[i] * tau );
               /* x[i] is freq on RH side, converted to phase on LH side */
    }                                                         /* end if */
    else                                           /* freq point is a gap */
    {
        // Avg freq gives better result for large gaps
        x[i] = x[i-1] + ( y_avg * tau );

        // Make phase value zero to preserve gap
        x[i-1]=0.0;

        g++;
    }                                                       /* end else */
    }                                                            /* end for */

    return( g );
}                                                  /* end ConvFreqToPhase() */
```

The FrequenC Library

NAME: CountGaps	**FUNCTION:** Count the number of gaps in phase or frequency phase data
SYNOPSIS: int CountGaps(F_TYPE z[], BOOL datype)	

F_TYPE z[]	Phase or frequency data array:
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysis end point

BOOL datype	Type of data: 0=phase, 1=frequency
RETURN: int	The # of gaps counted

REMARKS:
Only data points between start and end analysis limits are analyzed.
All zeros are counted as gaps in frequency data.
Only imbedded gaps are counted as gaps in phase data.
The datype flag determines whether the data is treated as phase or frequency data. The data is unchanged by the CountGaps() function.
No error codes are returned from this function.

EXAMPLE:

```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE z[512];                                          /* data array */
BOOL datype;                                        /* data type flag */
int num;                                                    /* # gaps */
.
.
datype=FREQ;                              /* set flag for frequency data */
         /* frequenc.h contains #define PHASE 0 and #define FREQ 1 */
num=CountGaps(z, datype);                       /* count gaps in data */
printf("\n# Gaps = %d", num);                         /* display num */
```

SEE ALSO: FillGaps()

REFERENCE: None

```
/*****************************************************************************/
/*                                                                           */
/*                           CountGaps()                                     */
/*                                                                           */
/*      Function counts the number of gaps in a phase or frequency           */
/*      data file.                                                           */
/*                                                                           */
/*      Parameters:     F_TYPE z[]   = phase or frequency data array         */
/*                                     z[0] = # data points                  */
/*                                     z[1] = analysis start                 */
/*                                     z[2] = analysis end                   */
/*                      BOOL datype  = data type: 0=phase, 1=frequency       */
/*                                                                           */
/*      Return:         int          = # gaps in data                        */
/*                                                                           */
/*      Note:           Gaps in data are indicated by zero values.           */
/*                      Leading and trailing zeros are not counted as        */
/*                      gaps in phase data.                                  */
/*                                                                           */
/*      Revision record:                                                     */
/*          12/30/91    Created & edited                                     */
/*          12/31/91    Renamed                                              */
/*          01/18/92    Edited title block                                   */
/*          02/03/96    Modified for use as Win 3.1 DLL                      */
/*          01/01/98    Changes for MS VC++ compatibility & warnings         */
/*          04/26/03    Adapted for Version 2.0 source code documentation    */
/*                                                                           */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved   */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CountGaps(F_TYPE z[], BOOL datype)

{
    int i;                                                  /* array index */
    int count=0;                                                 /* # gaps */

    for(i=(int)(*(z+1)+3-datype); i<=(int)(*(z+2)+1+datype); i++)
                       /* start at 1st and last data points for freq data, */
                                    /* 2nd and next to last for phase data */
    {
        if(!*(z+i))                        /* is data point a gap (zero)? */
        {
            count++;
        }                                                       /* end if */
    }                                                          /* end for */

    return(count);
}                                                    /* end CountGaps() */
```

The FrequenC Library

NAME: CalcGreenhallModSigma()	**FUNCTION:** Calc mod Allan variance for phase data using faster Greenhall algorithm

SYNOPSIS:
```
int CalcGreenhallModSigma(F_TYPE x[], F_TYPE *sig, F_TYPE tau, int m)
```

F_TYPE x[]	Phase data array
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE *sig	Pointer to mod Allan deviation, mod $\sigma_y(\tau)$
F_TYPE tau	Analysis averaging time = τ = m*τ_0, where τ_0 is measurement interval of phase data
int m	Averaging factor, m = τ/τ_0
RETURN: int	# analysis points, or -1 if error

REMARKS:
Only data points between start and end analysis limits are analyzed.
No gaps are allowed in phase data.
Must have at least 3 phase data points.
m must be >0, and cannot exceed (int)(n/3). tau must be >0.

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE x[512];                                     /* phase data array */
F_TYPE sig;                                               /* mod sigma */
F_TYPE tau=1.0;                                       /* meas interval */
int m=100;                                               /* avg factor */
int num;                                       /* # analysis points */
.
.
num=CalcGreenhallModSigma(x, &sig, tau, m);      /* call function */
printf("\nModSigma = %e", sig);                      /* display result */
```

SEE ALSO: CalcPhaseModSigma()

REFERENCE: C.A. Greenhall, "A Shortcut for Computing the Modified Allan Variance", AFCS '92.

The FrequenC Library

NAME: CalcHadamardB1	**FUNCTION:** Description of what function does
SYNOPSIS: int FunctionName(F_TYPE z[], F_TYPE one, F_TYPE two)	

F_TYPE z[]	Phase or frequency data array
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysios end point

F_TYPE one	parameter one, and description of what it is
F_TYPE two	parameter two, and description of what it is
RETURN: int	# non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are modified.
Embedded zeros are treated as gaps in phase data.
All zeros are treated as gaps in frequency data.

EXAMPLE:
```
#include <frequenc.h>                         /* FrequenC header file */
F_TYPE z[512];                                          /* data array */
F_TYPE one;                                          /* parameter one */
F_TYPE two;                                          /* parameter two */
int num;                                  /* # data points processed */
.
.
one=1.0;                                  /* set 1st parameter value */
two=2.0;                                  /* set 2nd parameter value */
num=FunctionName(z, one, two);                       /* call function */
printf("\n# Data Points Scaled = %d", num);           /* display num */
```

SEE ALSO: RelatedFunctionName()

REFERENCE: NIST Technical Note 1337

```
/***************************************************************************/
/*                                                                         */
/*                          CalcHadamardB1()                               */
/*                                                                         */
/*      Function to calc the Hadamard B1 Bias Function BH1(N, 1, mu),      */
/*      the ratio of the standard to Hadamard variances as a function of   */
/*      N = # samples and integer mu = slope of log sigma vs log tau plot  */
/*      for r = 1 (no dead time).  This function applies for integer mu =  */
/*      -1 thru 3 and N > 1.  BH1 = 1 for N = 2 for any mu.  This function */
/*      can be used to ID power law noises from alpha = 0 to -4.           */
/*                                                                         */
/*      Noise types handled:    Alpha   Mu  FM                             */
/*                              -4      3   RR                             */
/*                              -3      2   FW                             */
/*                              -2      1   RW                             */
/*                              -1      0    F                             */
/*                               0     -1    W                             */
/*                                                                         */
/*      References:    (1) NBS Technical Note 375, J.A. Barnes, "Tables    */
/*                         of Bias Functions B1 and B2, for Variances      */
/*                         Based on Finite Samples of Processes with       */
/*                         Power Law Spectral Densities", January, 1969.   */
/*                     (2) NIST Technical Note 1318, J.A. Barnes and D.W.  */
/*                         Allan, "Variances Based on Data with Dead Time  */
/*                         Between the Measurements", 1990.                */
/*                     (3) NIST Technical Note 1337, TN-296 to TN-335,     */
/*                         J.A. Barnes and D.W. Allan, "Variances Based    */
/*                         on Data with Dead Time Between the              */
/*                         Measurements", 1990.                            */
/*                     (4) J.A. Barnes, "The Analysis of Time and          */
/*                         Frequency Data", Austron, Inc, December, 1991.  */
/*                     (5) C.A Greenhall, "A Hadamard B1 Bias Function"    */
/*                         (private communication), 04/23/99.              */
/*                                                                         */
/*      Parameters:    int num = # samples                                 */
/*                               Get -1 error code for n <2                */
/*                     int mu  = noise type                                */
/*                               Get -1 error code for mu < -1 or mu > 3   */
/*                                                                         */
/*      Return:        float   = B1 bias function value,                   */
/*                               or -1 if error                            */
/*                                                                         */
/*      Revision record:                                                   */
/*          04/24/99   Drafted & debugged                                  */
/*          04/26/03    Adapted for Version 2.0 source code documentation     */
/*                                                                         */
/* (c) Copyright 1999-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                         */
/***************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL CalcHadamardB1(int num, int mu)
{
    double n=(double)num;       // Double version of num
    double m=(double)mu;        // Double version of mu
    double b;                   // BH1 bias function
    double b1, b2, b3;          // Working variables
```

```
if(n<2)                     // Num must be at least 2
{
    return(-1.0);           // Error code
}

switch(mu)                  // Switch according to mu
                            // Different formulae are required for
                            // mu = 0 & 2 to avoid singularities
{
    case -1:                // White FM (alpha=0)
    case  1:                // Random Walk FM (alpha=-2)
    case  3:                // Random Run FM (alpha=-4)
    {
        b1=2*pow(1-(1/n), m+2);
        b2=2*pow(1+(1/n), m+2);
        b3=pow(n, m);
        b=b3*(4-b2-b1);
        b1=4/(n*n);
        b2=pow(2, m+3);
        b+=-8+b2+b1;
        b1=pow(3, m);
        b2=pow(2, m);
        b3=-15+24*b2-9*b1;
        b*=n/((n-1)*b3);
        break;
    }

    case 0:                 // Flicker FM (alpha=-1)
    {
        b1=1-(1/n);
        b2=1+(1/n);
        b=log(b1);
        b*=-2*b1*b1;
        b3=log(b2);
        b3*=-2*b2*b2;
        b+=b3;
        b-=4*log(n)/(n*n);
        b+=8*log(2);
        b1=24*log(2)-9*log(3);
        b*=n/((n-1)*b1);
        break;
    }

    case 2:                 // Flicker Walk FM (alpha=-3)
    {
        b1=1-(1/n);
        b2=1+(1/n);
        b=log(b1);
        b*=b1*b1*b1*b1;
        b3=log(b2);
        b3*=b2*b2*b2*b2;
        b+=b3;
        b*=-2*n*n;
        b-=4*log(n)/(n*n);
        b-=24*log(n);
        b+=32*log(2);
        b1=96*log(2)-81*log(3);
        b*=n/((n-1)*b1);
        break;
    }
```

```
        default:                // Unallowed mu
        {
            b=-1.0;             // Error code
            break;
        }
    }

    return((float)b);           // Return valid value of BH1 bias function
}
```

The FrequenC Library

NAME:	**FUNCTION:**
`CalcHadamardDev()`	Calculate Hadamard variance from phase data

SYNOPSIS:
```
 int CalcHadamardDev(F_TYPE x[], F_TYPE *dev, F_TYPE tau)
```

`F_TYPE x[]`	Phase data array
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

`F_TYPE *dev`	Pointer to Hadamard deviation
`F_TYPE tau`	Averaging time
RETURN: `int`	# analysis points, or -1 if error

REMARKS:
```
 Only data points between start and end analysis limits are analyzed.
 Embedded zeros treated as gaps in phase data.
 Must have at least 4 adjacent non-gap phase data points.
 Tau must be >0.
```

EXAMPLE:
```
 #include "frequenc.h"                       /* FrequenC header file */
 F_TYPE x[512];                                  /* phase data array */
 F_TYPE dev;                                         /* Hadamard dev */
 F_TYPE tau=1.0;                                         /* avg time */
 int num;                                     /* # analysis points */
 .
 .
 num=CalcHadamardDev(x, &dev, tau);               /* call function */
 printf("\nHadamard Dev = %e", dev);               /* display result */
```

SEE ALSO: `CalcFreqHadamardDev()`

REFERENCE: T. Walter, "A Multi-Variance Analysis in the Time Domain", Proc. 24th PTTI Meeting, Dec. 1992.

```
/************************************************************************/
/*                                                                      */
/*                      CalcHadamardDev()                               */
/*                                                                      */
/*      Function to calculate Hadamard variance from phase data         */
/*                                                                      */
/*      Parameters:     F_TYPE x[]    = phase data                      */
/*                                      x[0] = # data points            */
/*                                      x[1] = analysis start           */
/*                                      x[2] = analysis end             */
/*                      F_TYPE *p_dev = pointer to Hadamard deviation   */
/*                      F_TYPE tau    = time between phase measurements */
/*                                                                      */
/*      Return:         int           = # analysis points,              */
/*                                      or -1 if error                  */
/*                                                                      */
/*      Note:           Embedded zeros treated as gaps in phase data.   */
/*                      Must have at least 1 analysis point; this       */
/*                      requires at least 4 adjacent non-gap phase data */
/*                      points.  Tau must be >0.                        */
/*                                                                      */
/*      Reference:      T. Walter, "A Multi-Variance Analysis in the Time */
/*                      Domain", Proc. 24th PTTI Meeting, Dec. 1992.    */
/*                                                                      */
/*      Revision record:                                                */
/*          11/10/92    Created from CPS.C                              */
/*          02/03/96    Modified for use as Win 3.1 DLL                 */
/*          01/01/98    Changes for MS VC++ compatibility & warnings    */
/*          04/26/03    Adapted for Version 2.0 source code documentation */
/*                                                                      */
/* (c) Copyright 1992-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                      */
/************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcHadamardDev(F_TYPE x[],
    F_TYPE *p_dev, F_TYPE tau)
{
    double sum;                         /* summing variable to calc sigma */
    int i;                                           /* index for summing */
    int num = 0;                                     /* # analysis points */

    if(tau<=0)
    {
        return(-1);                                              /* error */
    }

    for( i = (int)(*(x+1)+2), sum = 0; i < (int)(*(x+2)); i++)
                                              /* index goes from 1 to n-3 */
                     /* for n data points from 1 to n; data starts at x[3] */
    {
        if(    ( i==*(x+1)+2 && *(x+i+1) && *(x+i+2) && *(x+i+3) )
           ||  ( *(x+i)      && *(x+i+1) && *(x+i+2) && *(x+i+3) )
           ||  ( i==*(x+2)   && *(x+i  ) && *(x+i+1) && *(x+i+2) ) )
                          /* skip if any imbedded phase data value is zero */
        {
            sum += SQR( *(x+i+3) - 3 * *(x+i+2) + 3 * *(x+i+1) - *(x+i) );
            num++;
        }                                                       /* end if */
    }                                                      /* end for */

    if(num>0)
    {
        *p_dev = ( sqrt( sum / (6.0 * num * tau * tau )));
        return(num);                               /* # analysis points */
    }
    else
    {
        return(-1);                                          /* error */
    }
}                                              /* end CalcPhaseSigma() */
```

The FrequenC Library

NAME:	**FUNCTION:**
CalcInvChiSqr	Calculate the chi square value that corresponds to a certain probability and number of degrees of freedom

SYNOPSIS: `float CalcInvChiSqr(float p, float df)`

float p	Chi square probability, p(X^2, df)
float df	# of degrees of freedom, df
RETURN: float	The chi square value, X^2, or -1 if error

REMARKS:
This function uses an iterative process that is slow for very small values of p. It returns an error code (-1) for p $\le$ 0.00001.
The # degrees of freedom does not have to be an integer.
This function defines p(X^2, df) as the area under the upper tail of the chi square distribution from X^2 to ∞. Depending on your usage, you may need to use (1-p) instead of p as defined here.
CalcInvChiSqr() calls CalcChiSqrProb(), which uses CalcNormalProb().

EXAMPLE:
```
#include "frequenc.h"                      /* FrequenC header file */
float p;                                 /* chi square probability */
float df;                                  /* # degrees of freedom */
float x;                                       /* chi square value */
.
.
p=0.95;                                         /* assign p value */
df=10.0;                                       /* assign df value */
x=CalcInvChiSqr(p, df);                         /* calc X²(p, df) */
if(x==-1)
{
    printf("\nError"):                           /* error message */
}
else
{
    printf("\nX² = %5.2f", x);                      /* display X² */
}
```

SEE ALSO: CalcChiSqrProb()

REFERENCE: None

```
/******************************************************************************/
/*                                                                            */
/*                              CalcInvChiSqr()                               */
/*                                                                            */
/*      Code to replace CalcInvChiSqr() of FrequenC32 library                 */
/*                                                                            */
/*      1st function calculates closed-form approximation to inverse chi      */
/*      square for large # degrees of freedom (OK for df>10)                  */
/*                                                                            */
/*      References: (1) C. Greenhall/JPL via e-mail 12/06/96                  */
/*                  (2) Abramowitz & Stegun, Handbook of Mathematical         */
/*                      Functions, Sections 26.2.22 & 26.4.17                 */
/*                                                                            */
/*      2nd function uses a WJR iterative process to calculate the            */
/*      value of chi squared associated with a given probability for          */
/*      a certain # of degrees of freedom.  Calls CalcChiSqrProb().           */
/*                                                                            */
/*      Parameters:    float p  = chi squared probability value               */
/*                     float df = # of degrees of freedom (not necessarily    */
/*                                an integer)                                 */
/*                                                                            */
/*      Return:        float    = chi square value,                           */
/*                                or -1 if error                              */
/*                                                                            */
/*      Revision record:                                                      */
/*          02/03/97   Adapted from CalcInvChiSqr() of FrequenC Library       */
/*          02/04/97   Added closed-form function for large df.  Faster,      */
/*                     and avoids possible convergence problems.              */
/*          02/05/97   Fixed problem with wrong tail for new large df calc    */
/*          05/16/00   Use NR routines to calc chi sqr probability            */
/*                     Fixes discontunity in chi sqr vs. df                   */
/*          04/13/02   Watch out when df<1 - can hang up                      */
/*          03/08/03   Brought into Ver 2.0 of FrequenC lib                   */
/*          04/26/03   Adapted for Version 2.0 source code documentation      */
/*                                                                            */
/* (c) Copyright 1997-2003 Hamilton Technical Services All Rights Reserved    */
/*                                                                            */
/******************************************************************************/

#include "frequenc.h"

// Prototypes for Numerical Recipes functions used here - not exported
float gammln(float xx);
float gammp(float a, float x);
float gammq(float a, float x);
void gcf(float* gammcf, float a, float x, float* gln);
void gser(float* gamser, float a, float x, float* gln);

float __declspec(dllexport) FAR PASCAL CalcInvChiSqr(float p, float df)
{
    // Local variables
    double x;                       // Chi-squared value
    double prob;                    // Degrees of freedom
    double sign;                    // Iteration control sign
    double div;                     // Iteration control divisor

    // Closed-form variables
    double a0=2.30753;              // Coefficients from A&S 26.2.22
    double a1=0.27601;              // "
    double b1=0.99229;              // "
    double b2=0.04481;              // "
    double b;
    double n;

    double p1;
    double s;
    double t;
    double y;

    if(p<=0.00001)                    /* check for small, zero or negative p */
    {                                     /* no/slow convergence for small p */
        return(-1.0);                                          /* error code */
    }

    // Use closed-form approximation for df > 100
    if(df>100)
    {
        // Use opposite tail of distribution
        p=1-p;

        // Calc approximation to inverse chi sqr for large df
        p1=min(p, 1-p);
        t=sqrt(-2.0*log(p1));                         // A&S 26.2.22
        n=t-((a0+(a1*t))/(1+(t*(b1+(t*b2)))));        // "
        b=2.0/(9.0*df);                               // A&S 26.4.17

        if((p-0.5)<0) s=-1.0;                         // Greenhall revision
        else if((p-0.5)>0) s=1.0;                     // 1 function for
        else s=0.0;                                   // both tails

        y=1-b+(s*n*sqrt(b));                          // A&S 26.4.17
        x=df*pow(y, 3.0);                             // "

        return (float)x;
    }

    // Iterative solution for df<=100
    x = df+(0.5-p)*df*0.5;                    /* start value for chi squared */
    prob=gammp(df/2, (float)(x/2));
    prob=1-prob;

    div = 0.1;

    while(fabs((prob-p)/p)>0.0001)                     /* accuracy criterion */
    {
        if(prob-p>0)                                   /* save sign of error */
        {
            sign=1.0;
        }
        else
        {
            sign=-1.0;
        }
        x += df*(prob-p)/div;                         /* iteration increment */
        if(x>0)                                   /* make sure argument is + */
        {
            prob=gammp(df/2, (float)(x/2));
            prob=1-prob;

        }
```

```
        else                      /* otherwise restore it & reduce increment */
        {
            x -= df*(prob-p)/div;
            div*=2;
        }
        if(((prob-p)/sign)<0)                 /* did sign of error reverse? */
        {
            div*=2;                           /* reduce increment if it did */
        }
    }
    return (float)x;
}

/***************************************************************************/
/*                                                                         */
/*    Numerical Recipes Routines to Calculate Chi Square Probability       */
/*                                                                         */
/*    Call with p=gammp(df/2, x/2);                                        */
/*    where p=chi sqr prob, df=deg of freedom, x=chi sqr val               */
/*                                                                         */
/***************************************************************************/

// Defines
#define ITMAX 100
#define EPS 3.0e-7

/***************************************************************************/
/*                                                                         */
/*                           gamlm()                                       */
/*                                                                         */
/***************************************************************************/
float gammln(float 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 (float)(-tmp+log(2.50662827465*ser));
}
/***************************************************************************/
/*                                                                         */
/*                           gammp()                                       */
/*                                                                         */
/***************************************************************************/
float gammp(float a, float x)
{
    float gamser,gammcf,gln;
    if(x < 0.0 || a <= 0.0)
    {
        // TRACE_STR("Invalid arguments in routine GAMMP");
    }
    if(x < (a+1.0))
    {
        gser(&gamser,a,x,&gln);
        return gamser;
    }
    else
    {
        gcf(&gammcf,a,x,&gln);
        return (float)(1.0-gammcf);
    }
}
/****************************************************************************/
/*                                                                          */
/*                           gammq()                                        */
/*                                                                          */
/****************************************************************************/
float gammq(float a, float x)
{
    float gamser,gammcf,gln;

    if(x < 0.0 || a <= 0.0)
    {
        // TRACE_STR("Invalid arguments in routine GAMMQ");
    }
    if(x < (a+1.0))
    {
        gser(&gamser,a,x,&gln);
        return (float)(1.0-gamser);
    }
    else
    {
        gcf(&gammcf,a,x,&gln);
        return gammcf;
    }
}
/****************************************************************************/
/*                                                                          */
/*                           gammcf()                                       */
/*                                                                          */
/****************************************************************************/
void gcf(float* gammcf, float a, float x, float* gln)
{
    int n;
    float gold=0.0,g,fac=1.0,b1=1.0;
    float b0=0.0,anf,ana,an,a1,a0=1.0;

    *gln=gammln(a);
    a1=x;
    for(n=1;n<=ITMAX;n++)
    {
        an=(float) n;
        ana=an-a;
        a0=(a1+a0*ana)*fac;
        b0=(b1+b0*ana)*fac;
        anf=an*fac;
        a1=x*a0+anf*a1;
        b1=x*b0+anf*b1;
        if(a1)
        {
            fac=(float)1.0/a1;
            g=b1*fac;
            if(fabs((g-gold)/g) < EPS)
            {
                *gammcf=(float)exp(-x+a*log(x)-(*gln))*g;
                return;
            }
            gold=g;
        }
    }
}
/****************************************************************************/
/*                                                                          */
/*                           gser()                                         */
/*                                                                          */
/****************************************************************************/
void gser(float* gamser, float a, float x, float* gln)
{
    int n;
    float sum,del,ap;

    *gln=gammln(a);
    if(x <= 0.0)
    {
        if(x < 0.0)
        {
            // TRACE_STR("x less than 0 in routine GSER");
        }
        *gamser=0.0;
        return;
    }
    else
    {
        ap=a;
        del=sum=(float)1.0/a;
        for(n=1;n<=ITMAX;n++)
        {
            ap += 1.0;
            del *= x/ap;
            sum += del;
            if(fabs(del) < fabs(sum)*EPS)
            {
                *gamser=sum*(float)exp(-x+a*log(x)-(*gln));
                return;
            }
        }
        return;
    }
}
```

The FrequenC Library

NAME: `CalcLinFreqDrift`	**FUNCTION:** Calculate the linear drift for frequency data

SYNOPSIS:

```
int CalcLinFreqDrift(F_TYPE y[],F_TYPE *p_a,F_TYPE *p_b,F_TYPE *p_v)
```

`F_TYPE y[]`	Frequency data array:
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

`F_TYPE *p_a`	Pointer to intercept, a, of y(t)=a+bt linear fit to the frequency drift
`F_TYPE *p_b`	Pointer to slope, b, of y(t)=a+bt linear fit to the frequency data
`F_TYPE *p_v`	Pointer to variance, v, of linear fit to the frequency data
RETURN: `int`	The # of non-gap data points processed, or -1 if error

REMARKS:
Only data points between start and end analysis limits are analyzed.
There must be at least 2 non-gap analysis points.
All zeros are treated as gaps in frequency data.

EXAMPLE:

```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE y[512];                                 /* frequency data array */
F_TYPE a;                                                /* intercept */
F_TYPE b;                                                    /* slope */
F_TYPE v;                                                 /* variance */
int num;                                    /* # data points processed */
.
.
num=CalcLinFreqDrift(y, &a, &b, &v);             /* calc linear drift */
if(num==-1)                                        /* check for error */
{
    printf("\nError");                              /* error message */
}
else
{
    printf("\nIntercept = %e", a);                   /* display slope */
    printf("\nSlope = %e", b);                   /* display intercept */
    printf("\nVariance = %e, v);                  /* display variance */
    printf("\n# Data Points Analyzed = %d", num);   /* display num */
}
```

SEE ALSO: `CalcLogFreqDrift()`

REFERENCE: None

```
/*****************************************************************************/
/*                                                                           */
/*                          CalcLinFreqDrift()                               */
/*                                                                           */
/*      Function to calculate linear frequency drift y(t)=a+bt              */
/*                                                                           */
/*      Parameters:     F_TYPE y[]  = frequency data                        */
/*                                    y[0] = # data points                  */
/*                                    y[1] = analysis start                 */
/*                                    y[2] = analysis end                   */
/*                      F_TYPE p_a  = pointer to y-intercept                */
/*                      F_TYPE p_b  = pointer to slope                      */
/*                      F_TYPE p_v  = pointer to fit variance               */
/*                                                                           */
/*      Return:         int         = # analysis points,                    */
/*                                    or -1 if error                        */
/*                                                                           */
/*      Notes:          Zeros treated as gaps in frequency data             */
/*                      # non-gap data points must be at least 2            */
/*                      and at least 3 to calc finite fit variance          */
/*                                                                           */
/*      Revision record:                                                     */
/*          12/16/91    Created                                              */
/*          12/26/91    Changed to int returning # non-gap data points       */
/*          12/31/91    Renamed                                              */
/*          01/16/92    Edited per FrequenC documentation                    */
/*          01/18/92    Edited title block                                   */
/*          01/19/92    Added error traps                                    */
/*          02/03/96    Modified for use as Win 3.1 DLL                      */
/*          01/01/98    Changes for MS VC++ compatibility & warnings         */
/*          04/26/03    Adapted for Version 2.0 source code documentation   */
/*                                                                           */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcLinFreqDrift(F_TYPE y[],
    F_TYPE *p_a, F_TYPE *p_b, F_TYPE *p_v)
{
    int i;
    int num = (int)(*(y+2)-*(y+1)+1);
    double sum_x = 0;
    double sum_y = 0;
    double sum_xx = 0;
    double sum_yy = 0;
    double sum_xy = 0;

    for( i = (int)(*(y+1)+2); i <= (int)(*(y+2)+2); i++)
    {
        if(*(y + i))
        {
            sum_x  += i-2;
            sum_y  += *(y + i);
            sum_xx += SQR ( (float) (i-2) );         /* must cast to float */
            sum_yy += SQR ( *(y + i) );
            sum_xy += (i-2) * *(y + i);
        }                                                        /* end if */
        else
        {
            num--;
        }                                                       /* end else */
    }                                                            /* end for */

    if(num<2)
    {
        return(-1);                                                 /* error */
    }

    *p_b = (sum_xy - ((sum_x) * (sum_y) / num));
    *p_b = *p_b / (sum_xx- ((sum_x) * (sum_x)) / num);
    *p_a = ((sum_y) / num) - (*p_b * sum_x / num);
    *p_v = 0;                           /* fit variance 0 if only 2 points */

    if(num>=3)     /* need at least 3 analysis points to calc fit variance */
    {
        *p_v = sum_yy  - (sum_y * sum_y / num);
        *p_v -= *p_b * *p_b * (sum_xx - (sum_x * sum_x / num ));
        *p_v /= (num - 2);
    }

    return(num);
}                                                 /* end CalcLinFreqDrift() */
```

The FrequenC Library

NAME: CalcLogFreqDrift	**FUNCTION:** Calculate the log drift for frequency data

SYNOPSIS:
```
int CalcLogFreqDrift(F_TYPE y[], F_TYPE *p_a, F_TYPE *p_b,
                    F_TYPE *p_c, F_TYPE *p_v)
```

F_TYPE y[]	Frequency data array:
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

F_TYPE *p_a	Pointer to multiplicative term, a, of y(t)=a·ln(bt+1)+c log fit to frequency data
F_TYPE *p_b	Pointer to time term, b, of log fit
F_TYPE *p_c	Pointer to constant term, c, of log fit
F_TYPE *p_v	Pointer to fit variance, v, of log fit
RETURN: int	The # of non-gap data points processed, or -1 if error

REMARKS:
All zeros are treayed as gaps in frequency data.
Only data points between start and end analysis limits are analyzed.
There must be at least 2 non-gap analysis points.

EXAMPLE:
```
#include "frequenc.h"                       /* FrequenC header file */
F_TYPE y[512];                              /* frequency data array */
F_TYPE a, b, c, v;                                /* fit parameters */
int num;                                 /* # data points processed */
.
.
num=CalcLogFreqDrift(y, &a, &b, &c, &v);          /* calc log drift */
if(num==-1)                                      /* check for error */
{
    printf("\nError");                             /* error message */
}
else
{
    printf("\na = %e", a);                             /* display a */
    printf("\nb = %e", b);                             /* display b */
    printf("\nc = %e", c);                             /* display c */
    printf("\nv = %e", v);                             /* display v */
    printf("\n# Data Points Analyzed = %d", num);   /* display num */
}
```

SEE ALSO: CalcLinFreqDrift()

REFERENCE: MIL-O-55310B, General Specification for Crystal Oscillators

```
/****************************************************************************/
/*                                                                          */
/*                         CalcLogFreqDrift()                               */
/*                                                                          */
/*      Function to calculate log frequency drift                           */
/*      Fit fractional frequency data to log drift model:                   */
/*                      y(t) = aùln(bt+1)+c                                 */
/*                      where: y(t) = fractional frequency                  */
/*                             t    = time                                  */
/*                             a    = multiplicative term                   */
/*                             b    = log time term                         */
/*                             c    = constant term                         */
/*                                                                          */
/*      Reference:      MIL-O-55310B, General Specification for Crystal     */
/*                      Oscillators                                         */
/*                                                                          */
/*      Parameters:     F_TYPE y[]  = frequency data                        */
/*                                    y[0] = # data points                  */
/*                                    y[1] = analysis start                 */
/*                                    y[2] = analysis end                   */
/*                      F_TYPE *p_a  = pointer to multiplicative term       */
/*                      F_TYPE *p_b  = pointer to time constant term        */
/*                      F_TYPE *p_c  = pointer to constant term             */
/*                      F_TYPE *p_v  = pointer to fit variance              */
/*                                                                          */
/*      Return:         int          = # non-zero anlaysis points,          */
/*                                     or -1 if error                       */
/*                                                                          */
/*      Notes:          Zeros treated as gaps in frequency data.            */
/*                      A value of b is chosen, the linear fit parameters   */
/*                      are found, and b is varied iteratively until the    */
/*                      rms error is below a limit criterion.               */
/*                      Need at least 2 non-zero data points.               */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/16/91    Created                                             */
/*          12/31/91    Renamed                                             */
/*          01/18/92    Edited title block                                  */
/*          01/19/92    Changed error trap                                  */
/*          02/03/96    Modified for use as Win 3.1 DLL                     */
/*          01/01/98    Changes for MS VC++ compatibility & warnings        */
/*          04/26/03    Adapted for Version 2.0 source code documentation   */
/*                                                                          */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcLogFreqDrift(F_TYPE y[],
    F_TYPE *p_a, F_TYPE *p_b, F_TYPE *p_c, F_TYPE *p_v)
{
    int i;                                              /* freq data index */
    int num = (int)(*(y+2)-*(y+1)+1);              /* # frequency data points */
    double error_limit=0.00001;     /* fractional error limit for variance */
    double x;                                                /* x=ln(b*t+1) */
    double err=1.0;                                    /* sum of squared error */
    double prev_err=0;  /* initialized so (err-prev_err)/err > error_limit */
    double sum_x;                                         /* sum of x=ln(bt+1) */
    double sum_y;                                                 /* sum of y */
    double sum_xx;                                                 /* sum of x^2 */
    double sum_xy;                                                  /* sum of xy */
    double d=0.1;                                          /* b iteration factor */

    *p_b=1.0;                                             /* initial value for b */

    while(fabs(((err-prev_err)/err))>error_limit)                  /* iterate */
    {
        prev_err=err;                                     /* save previous error */
        sum_x = 0;                                          /* sum of x=ln(bt+1) */
        sum_y = 0;                                                   /* sum of y */
        sum_xx = 0;                                                /* sum of x^2 */
        sum_xy = 0;                                                 /* sum of xy */
        num = (int)(*(y+2)-*(y+1)+1);           /* # frequency data points */

        for( i=(int)(*(y+1)+2); i<=(int)(*(y+2)+2); i++)
                                                      /* accumulate sums for fit */
        {
            if(*(y+i))                                          /* gap in data? */
            {
                x=log((*p_b * (i-2))+1);
                sum_x  += x;
                sum_y  += *(y+i);
                sum_xx += x*x;
                sum_xy += x * *(y+i);
            }                                                         /* end if */
            else
            {
                num--;
            }                                                       /* end else */
        }                                                            /* end for */

        if(num<2)                           /* must have at least 2 data points */
        {
            return(-1);
        }                                                             /* end if */

        *p_a = sum_xy - (sum_x * sum_y / num);              /* find parameters */
        *p_a /= sum_xx- (sum_x * sum_x / num);
        *p_c = (sum_y - (*p_a * sum_x ))/ num;

        err=0;                                        /* calc sum of squared error */
        for( i=(int)(*(y+1)+2); i<=(int)(*(y+2)+2); i++)
        {
            if(*(y+i))
            {
                err += SQR(*p_a * log((*p_b * (i-2))+1) + *p_c - *(y+i));
            }
        }

        if(err>prev_err)                                           /* wrong way? */
        {
            d *=-0.5;
        }
        *p_b += *p_b * d;                          /* modify b for next iteration */
    }
        *p_v = err;

        return(num);
}                                                      /* end CalcLogFreqDrift() */
```

The FrequenC Library

NAME: CalcMean	**FUNCTION:** Calculate average value of phase or frequency data

SYNOPSIS:
```
int CalcMean(F_TYPE z[], F_TYPE *p_avg, BOOL datype)
```

F_TYPE z[]	Phase or frequency data array:
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysis end point

F_TYPE *p_avg	Pointer to average
BOOL datype	Type of data: 0=phase, 1=frequency
RETURN: int	The # of analysis points, or -1 if error

REMARKS:
Only data between start and end analysis limits are analyzed.
All zeros are treated as gaps in frequency data.
Only embedded gaps are treated as gaps in phase data.
The datype flag determines whether the data is treated as phase or frequency data.

EXAMPLE:
```
#include "frequenc.h"                        /* FrequenC header file */
F_TYPE z[512];                                         /* data array */
F_TYPE avg;                                         /* average value */
BOOL datype;                                       /* data type flag */
int num;                                                   /* # gaps */
.
.
datype=FREQ;                          /* set flag for frequency data */
         /* frequenc.h contains #define PHASE 0 and #define FREQ 1 */
num=CalcMean(z, &avg, datype);                           /* calc avg */
if(num==-1)                                       /* check for error */
{
    printf("\nError);                               /* error message */
}
else
{
    printf("\nAverage = %e", avg);                /* display average */
    printf("\n# Analysis Points = %d", num);          /* display num */
}
```

SEE ALSO: NormalizeData()

REFERENCE: None

```
/****************************************************************************/
/*                                                                          */
/*                              CalcMean()                                  */
/*                                                                          */
/*      Function to calculate average value of phase or frequency data     */
/*                                                                          */
/*      Parameters:     F_TYPE z[]    = phase or frequency data             */
/*                                      z[0] = # data points                */
/*                                      z[1] = analysis start               */
/*                                      z[2] = analysis end                 */
/*                      F_TYPE *p_avg = pointer to average value of data    */
/*                      BOOL datype   = data type: 0=phase, 1=frequency     */
/*                                                                          */
/*      Return:         int           = # non-gap data points,              */
/*                                      or -1 if error                      */
/*                                                                          */
/*      Notes:          All zeros treated as gaps in frequency data.        */
/*                      Embedded zeros treated as gaps in phase data.       */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/16/91    Created                                             */
/*          12/24/91    Changed to make datype a parameter rather than      */
/*                      a global variable.  Made function an int return-    */
/*                      ing # non-gap data points.  y[] changed to z[].     */
/*          12/31/91    Renamed                                             */
/*          01/18/92    Edited title block                                  */
/*          01/19/92    Added error trap                                    */
/*          02/03/96    Modified for use as Win 3.1 DLL                     */
/*          01/01/98    Changes for MS VC++ compatibility & warnings        */
/*          04/26/03    Adapted for Version 2.0 source code documentation  */
/*                                                                          */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcMean(F_TYPE z[], F_TYPE *p_avg,
    BOOL datype)
{
    double sum = 0;
    float num = 0;
    int i;

    for( i=(int)(*(z+1)+2); i<=(int)(*(z+2)+2); i++)
                                          /* loop thru all analysis points */
    {                                                         /* skip gaps */
        if(*(z + i) || (!datype && i==3) || (!datype && i==z[0]+2))
        {
            sum += *(z+i);
            num++;
        }                                                        /* end if */
    }                                                           /* end for */
    if(num>0)
    {
        *p_avg = sum/num;
        return(int)num;                     /* return # non-gap data points */
    }
    else
    {
        return(-1);                                               /* error */
    }
}                                                      /* end CalcMean() */
```

The FrequenC Library

NAME: CalcNormalProb	**FUNCTION:** Calculate the normal probability function
SYNOPSIS: `float CalcNormalProb(float x)`	
float x	The normal variable value
RETURN: float	The value of the normal probability, the area under the normal or Gaussian distribution to the left of the normal variable x.

REMARKS:
No error codes are returned by this function

EXAMPLE:
```
#include "frequenc.h"                       /* FrequenC header file */
float p;                             /* value of normal probability */
float x;                                   /* normal variable value */
.
.
x=1.5;                                       /* assign value to x */
p=CalcNormalProb(x);                     /* calc normal probability */
printf("\np = %5.3f", p);                            /* display p */
```

SEE ALSO: None

REFERENCE: Collected Algorithms from CACM, Vol. I, #209

```
/****************************************************************************/
/*                                                                          */
/*                            CalcNormalProb()                              */
/*                                                                          */
/*      Function to calculate the cumulative normal distribution,           */
/*      the area under the normal or Gaussian distrubution to the           */
/*      left of the normal variable x.                                      */
/*                                                                          */
/*      Reference:     Collected Algorithms from CACM, Vol. I, #209,        */
/*                     D. Ibbetson and E. Brothers, 1963.                   */
/*                                                                          */
/*      Parameter:     float x = normal variable                            */
/*                                                                          */
/*      Return:        float   = p(x) = normal probability value            */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/30/91   Created                                              */
/*          12/31/91   Renamed                                              */
/*          01/18/92   Edited title block                                   */
/*          02/03/96   Modified for use as Win 3.1 DLL                      */
/*          01/01/98   Changes for MS VC++ compatibility & warnings         */
/*          04/26/03   Adapted for Version 2.0 source code documentation   */
/*                                                                          */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL CalcNormalProb(float x)

{
    double w, y, z;

    if (x == 0.0)
    {
        z = 0.0;
    }                                                           /* end if */
    else
    {
        y = fabs(x) / 2.0;
        if (y >= 3.0)
        {
            z = 1.0;
        }                                                       /* end if */
        else
        {
            if (y < 1.0)
            {
                w = y * y;
                z = ((((((((0.000124818987 * w
                -.001075204047) * w + .005198775019) * w
                -.019198292004) * w + .059054035642) * w
                -.151968751364) * w + .319152932694) * w
                -.531923007300) * w + .797884560593) * y * 2.0;
            }                                                   /* end if */
            else
            {
                y = y - 2.0;
                z = (((((((((((((-.000045255659 * y
                +.000152529290) * y - .000019538132) * y
                -.000676904986) * y + .001390604284) * y
                -.000794620820) * y - .002034254874) * y
                +.006549791214) * y - .010557625006) * y
                +.011630447319) * y - .009279453341) * y
                +.005353579108) * y - .002141268741) * y
                +.000535310849) * y + .999936657524;
            }                                                 /* end else */
        }                                                     /* end else */
    }                                                         /* end else */

    if (x > 0.0)
    {
        return (float)((z + 1.0) / 2.0);
    }                                                           /* end if */
    else
    {
        return (float)((1 - z) / 2.0);
    }                                                         /* end else */
}                                                   /* end CalcNormProb() */
```

The FrequenC Library

NAME: CalcPhaseHadamardDev	**FUNCTION:** Calculate Hadamard deviation from phase data

SYNOPSIS:
```
int CalcPhaseHadamardDev(F_TYPE x[], F_TYPE *dev , F_TYPE tau)
```

F_TYPE x[]	Phase data array
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE *dev	Pointer to Hadamard deviation
F_TYPE tau	Averaging time
RETURN: int	# non-gap data points processed or -1 if error

REMARKS:
Only data points between start and end analysis limits are analyzed.
Embedded zeros are treated as gaps in phase data.
Must have at least 4 adjacent non-gap phase data points.
Tau must be > 0.

EXAMPLE:
```
#include "frequenc.h"                     /* FrequenC header file */
F_TYPE x[512];                               /* phase data array */
F_TYPE dev;                                /* Hadamard deviation */
F_TYPE tau=1.0;                                /* averaging time */
int num;                              /* # data points processed */
.
.
num=CalcPhaseHadamardDev(x, &dev, tau);         /* call function */
printf("\nHadamard Dev = %e", dev);            /* display result */
```

SEE ALSO: CalcFreqHadamardDev

REFERENCE: T. Walter, "A Multi-Variance Analysis in the Time Domain", Proc. 24th PTTI Meeting, Dec. 1992.

```
/*****************************************************************************/
/*                                                                           */
/*                          CalcPhaseHadamardDev()                           */
/*                                                                           */
/*      Function to calculate Hadamard variance from phase data              */
/*                                                                           */
/*      Parameters:     F_TYPE x[]    = phase data                           */
/*                                      x[0] = # data points                 */
/*                                      x[1] = analysis start                */
/*                                      x[2] = analysis end                  */
/*                      F_TYPE *p_dev = pointer to Hadamard deviation        */
/*                      F_TYPE tau    = time between phase measurements      */
/*                                                                           */
/*      Return:         int           = # analysis points,                   */
/*                                      or -1 if error                       */
/*                                                                           */
/*      Note:           Embedded zeros treated as gaps in phase data.        */
/*                      Must have at least 1 analysis point; this            */
/*                      requires at least 4 adjacent non-gap phase data      */
/*                      points.  Tau must be >0.                             */
/*                                                                           */
/*      Reference:      T. Walter, "A Multi-Variance Analysis in the Time    */
/*                      Domain", Proc. 24th PTTI Meeting, Dec. 1992.         */
/*                                                                           */
/*      Revision record:                                                     */
/*          04/26/96    Created from STAB_F45.C                              */
/*          01/01/98    Changes for MS VC++ compatibility & warnings         */
/*          04/26/03    Adapted for Version 2.0 source code documentation    */
/*                                                                           */
/* (c) Copyright 1996-2003 Hamilton Technical Services All Rights Reserved   */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcPhaseHadamardDev(F_TYPE x[],
    F_TYPE *p_dev, F_TYPE tau)
{
    double sum;                         /* summing variable to calc sigma */
    int i;                                           /* index for summing */
    int num = 0;                                     /* # analysis points */

    if(tau<=0)
    {
        return(-1);                                              /* error */
    }

    for( i = (int)(*(x+1)+2), sum = 0; i < (int)(*(x+2)); i++)
                                              /* index goes from 1 to n-3 */
                       /* for n data points from 1 to n; data starts at x[3] */
    {
            if(    ( i==*(x+1)+2 && *(x+i+1) && *(x+i+2) && *(x+i+3) )
            ||  ( *(x+i)      && *(x+i+1) && *(x+i+2) && *(x+i+3) )
            ||  ( i==*(x+2)   && *(x+i  ) && *(x+i+1) && *(x+i+2) ) )
                           /* skip if any imbedded phase data value is zero */
        {
            sum += SQR( *(x+i+3) - 3 * *(x+i+2) + 3 * *(x+i+1) - *(x+i) );
            num++;
        }                                                        /* end if */
    }                                                           /* end for */

    if(num>0)
    {
        *p_dev = ( sqrt( sum / (6.0 * num * tau * tau )));
        return(num);                                /* # analysis points */
    }                                                            /* end if */
    else                                           /* no analysis points */
    {
        return(-1);                                               /* error */
    }                                                          /* end else */
}
```

The FrequenC Library

NAME: CalcPhaseModSigma	**FUNCTION:** Calculate the modified Allan deviation for phase data

SYNOPSIS:
```
int CalcPhaseModSigma(F_TYPE x[], F_TYPE *p_sig, F_TYPE tau, int m)
```

F_TYPE x[]	Phase data array:
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE *p_sig	Pointer to sig, the modified Allan deviation for the phase data
F_TYPE tau	The averaging time for the phase data
int m	The averaging factor
RETURN: int	the # of non-gap data points processed, or -1 if error

REMARKS:
Only data points between start and end analysis limits are analyzed.
There must be at least 3 non-gap analysis points.
Embedded zeros are treated as gaps in phase data.
Maximum m = (int) num/3, and m must be >0. Tau must be >0.

EXAMPLE:
```
#include "frequenc.h"                         /* FrequenC header file */
F_TYPE x[512];                                    /* phase data array */
F_TYPE sig;                                        /* modified sigma */
F_TYPE tau;                                        /* avergaing time */
int m;                                           /* averaging factor */
int num;                                  /* # data points processed */
.
.
tau=1.0;                                           /* set tau value */
m=2;                                                 /* set m value */
num=CalcPhaseModSigma(x, &sig, tau, m);           /* calc mod sigma */
if(num==-1)                                      /* check for error */
{
    printf("\nError");                             /* error message */
}
else
{
    printf("\nMod Sigma = %e", sig);               /* display sigma */
    printf("\n# Data Points Analyzed = %d", num);    /* display num */
}
```

SEE ALSO: CalcFastModSigma(), CalcFreqModSigma()

REFERENCE: NIST Technical Note 1337

```
/****************************************************************************/
/*                                                                          */
/*                          CalcPhaseModSigma()                             */
/*                                                                          */
/*      Function to calculate the modified Allan variance for phase data,   */
/*      possibly with gaps.  A faster function, CalcFastModSigma(), can     */
/*      be used if there are no gaps.                                       */
/*                                                                          */
/*      Parameters:     F_TYPE x[]     = phase data                         */
/*                                       x[0] = # data phase points         */
/*                                       x[1] = analysis start              */
/*                                       x[2] = analysis end                */
/*                      F_TYPE *p_sig  = pointer to mod Allan deviation     */
/*                      F_TYPE tau     = analysis averaging time = m*tau_0  */
/*                      int m          = averaging factor = tau/tau_0       */
/*                      BOOL *bAbort   = pointer to abort flag              */
/*                      BOOL bProgress = progress indicator flag            */
/*                      void *ProgressFunction()                            */
/*                                     = pointer to progress indicator      */
/*                                       function with args int nPercentDone */
/*                                       and char *szMessage                */
/*                                                                          */
/*      Return:         int            = # analysis points,                 */
/*                                       or -1 if error                     */
/*                                                                          */
/*      Notes:          Embedded zero phase value treated as gap in data    */
/*                      Phase data is not averaged before calling this      */
/*                      function.  The data consists of num phase samples   */
/*                      at interval tau_0.  Denominator term (num-3m+1)     */
/*                      must be >0.  This sets max m=(int)(num/3),          */
/*                      where num=# phase points before averaging.          */
/*                      m must be >0 and tau must be >0.                    */
/*                      Use faster CalcFastModSigma() if no gaps in data.   */
/*                                                                          */
/*                      The progress function pointer can be NULL, but is   */
/*                      intended as a way to display the % done and a       */
/*                      message about the calculation.                      */
/*                                                                          */
/*                      The abort flag provides a way to abort the calc.    */
/*                      It must exist but does not have to be used.         */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/16/90    Created                                             */
/*          12/31/91    Renamed                                             */
/*          01/18/92    Edited title block                                  */
/*          01/19/92    Added error traps                                   */
/*          02/03/96    Modified for use as Win 3.1 DLL                     */
/*          06/15/02    Copied here for revision.  Added gap flag.          */
/*                      Fixed error for small AF and large sets of low      */
/*                      resolution noise where sum_i can be zero w/o gap    */
/*          01/28/03    Added progress indicator & argument                 */
/*          03/08/03    Brought into Ver 2.0 of FrequenC lib                */
/*                      Added progress message function pointer to arg list */
/*          03/09/03    Added bAbort pointer to arg list                    */
/*          04/26/03    Adapted for Version 2.0 source code documentation   */
/*                                                                          */
/* (c) Copyright 1990-2003 Hamilton Technical Services  All Rights Reserved */
/*                                                                          */
/****************************************************************************/
```

```
#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcPhaseModSigma(F_TYPE x[],
    F_TYPE *p_sig, F_TYPE tau, int m, BOOL *bAbort, BOOL bProgress,
    void *ProgressFunction(int nPercentDone, char *szMessage))
{
    // Local variables
    double sum_i, sum_j;              /* summing variables to calc mod sigma */
    float fm;                                          /* float version of m */
    int i;                                   /* index for inner summing loop */
    int j;                                   /* index for outer summing loop */
    int n_sum=0;                           /* # points in outer summing loop */
    int num = (int)(*(x+2)-*(x+1)+1);    /* # phase data points for analysis */
    int gap;                                                /* gap flag */
    int nTic;                            /* Increment for progress indicator */
    int nLast_j=0;                                /* Previous value of index */

    // Check args
    if( (m<1) || (num<3*m) || (tau<=0) )
    {
        return(-1);                                             /* error */
    }                                                          /* end if */

    // Initialize increment for progress report
    nTic=(int)ceil(num/100);

    // Start
    if(bProgress)
    {
        // Initialize progress display
        ProgressFunction(0,
            "Calculating Mod/Time Sigma (with gaps)");
    }                                                         /* end if */

    // Initialize outer loop
    fm=(float)m;
    sum_j=0;

    // Calc mod sigma
    for(j=1; j<=num-(3*m)+1; j++)                           /* outer loop */
        /* j goes from 1 to N-3m+1 where N=# phase points & m=avg factor */
                                         /* regardless of analysis limits */
    {
        sum_i=0;
        gap=1;
        for(i=j+(int)(*(x+1))-1; i<=m+j-1+(*(x+1))-1; i++)     /* inner loop */
                    /* i is the actual phase data index;  it includes a term */
                      /* *(x+1)-1 to account for the analysis starting point */
                   /* the data array offset +2 is included in the code below */
        {
            if(    ( i==j+(*(x+1))-1     && *(x+i+2+m) && *(x+i+2+(2*m)) )
                ||  ( *(x+i+2)            && *(x+i+2+m) && *(x+i+2+(2*m)) )
                ||  ( i==m+j-1+(*(x+1))-1 && *(x+i+2+m) && *(x+i)        ) )
                            /* skip if any imbedded phase data value is zero */
            {
                sum_i += *(x+i+2+(2*m)) - (2*(*(x+i+2+m))) + *(x+i+2);
                gap=0;                                          /* not a gap */
            }                                                      /* end if */
        }                                                         /* end for */
        if(!gap)
        {
            sum_j += SQR(sum_i);
            n_sum++;
        }                                                          /* end if */

        // Display progress
        // Update when index has changed by at least tic amount
        if(((j-nLast_j)>=nTic) && (bProgress))
        {
            ProgressFunction((int)(0.5+100.0*j/(num-3*m)),
                "Calculating Mod/Time Sigma (with gaps)");
            nLast_j=j;
        }                                                         /* end if */

        // Check for abort
        if(*bAbort)
        {
            break;
        }
    }                                                             /* end for */

    // Done

    // Clear progress display
    ProgressFunction(0, "");

    // Return results
    if(n_sum>0)                                     /* make sure no div by 0 */
    {
        *p_sig = sqrt((sum_j)/(2.0*tau*tau*fm*fm*n_sum));
        return(n_sum);                                  /* # analysis points */
    }
    else
    {
        return(-1);                                                 /* error */
    }
}                                                 /* end CalcPhaseModSigma() */
```

The FrequenC Library

NAME: CalcPhaseOverlap HadamardDev	**FUNCTION:** Description of what function does

SYNOPSIS:
```
int FunctionName( F_TYPE z[], F_TYPE one, F_TYPE two)
```

F_TYPE z[]	Phase or frequency data array
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysios end point

F_TYPE one	parameter one, and description of what it is
F_TYPE two	parameter two, and description of what it is
RETURN: int	# non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are modified.
Embedded zeros are treated as gaps in phase data.
All zeros are treated as gaps in frequency data.

EXAMPLE:
```
#include <frequenc.h>                        /* FrequenC header file */
F_TYPE z[512];                                         /* data array */
F_TYPE one;                                         /* parameter one */
F_TYPE two;                                         /* parameter two */
int num;                                 /* # data points processed */
.
.
one=1.0;                                 /* set 1st parameter value */
two=2.0;                                 /* set 2nd parameter value */
num=FunctionName(z, one, two);                      /* call function */
printf("\n# Data Points Scaled = %d", num);           /* display num */
```

SEE ALSO: RelatedFunctionName()

REFERENCE: NIST Technical Note 1337

```
/****************************************************************************/
/*                                                                          */
/*                        CalcPhaseOverlapHadamardDev()                     */
/*                                                                          */
/*      Function to calc overlapping Hadamard variance from phase data     */
/*                                                                          */
/*      Parameters:     F_TYPE x[]    = phase data                          */
/*                                      x[0] = # data points                */
/*                                      x[1] = analysis start               */
/*                                      x[2] = analysis end                 */
/*                      F_TYPE *p_dev = pointer to Hadamard deviation       */
/*                      F_TYPE tau    = time between phase measurements     */
/*                      int    m      = averaging factor                    */
/*                                                                          */
/*      Return:         int           = # analysis points,                  */
/*                                      or -1 if error                      */
/*                                                                          */
/*      Note:           Embedded zeros treated as gaps in phase data.       */
/*                      Must have at least 1 analysis point; this           */
/*                      requires at least 4 adjacent non-gap phase data     */
/*                      points.  Tau must be >0.                            */
/*                                                                          */
/*      Reference:      T. Walter, "A Multi-Variance Analysis in the Time  */
/*                      Domain", Proc. 24th PTTI Meeting, Dec. 1992.        */
/*                      Modified by WJR to use fully-overlapping samples.  */
/*                                                                          */
/*      Revision record:                                                    */
/*          03/05/99    Created from CPHD.C of FrequenC32 Library           */
/*          04/26/03     Adapted for Version 2.0 source code documentation    */
/*                                                                          */
/* (c) Copyright 1999-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcPhaseOverlapHadamardDev(F_TYPE x[],
    F_TYPE *p_dev, F_TYPE tau, int m)
{
    double sum;                             /* summing variable to calc sigma */
    int i;                                               /* index for summing */
    int num=0;                                           /* # analysis points */
    float fm=(float)m;                                  /* float version of m */

    if(tau<=0)
    {
        return(-1);                                                 /* error */
    }

    for(i=(int)(*(x+1)+2), sum=0; i<=(int)(*(x+2)+2-3*m); i++)
                                               /* index goes from 1 to n-3m */
                     /* for n data points from 1 to n; data starts at x[3] */
    {
           if(    ( i==*(x+1)+2 && *(x+i+m) && *(x+i+2*m) && *(x+i+3*m) )
           ||  ( *(x+i)      && *(x+i+m) && *(x+i+2*m) && *(x+i+3*m) )
           ||  ( i==*(x+2)   && *(x+i  ) && *(x+i+m) && *(x+i+2*m) ) )
                          /* skip if any imbedded phase data value is zero */
        {
            sum+=SQR((*(x+i+(3*m)))-3*(*(x+i+(2*m)))+3*(*(x+i+m))-(*(x+i)));
            num++;
        }                                                         /* end if */
    }                                                            /* end for */

    if(num>0)
    {
        *p_dev=sqrt(sum/(6.0*num*tau*tau*fm*fm));
        return(num);                                  /* # analysis points */
    }                                                             /* end if */
    else                                             /* no analysis points */
    {
        return(-1);                                               /* error */
    }                                                           /* end else */
}
```

The FrequenC Library	
NAME: CalcPhaseOverlapSigma	**FUNCTION:** Calculate the Allan deviation for overlapping samples of phase data
SYNOPSIS: `int CalcPhaseOverlapSigma(F_TYPE x[],F_TYPE *p_sig,F_TYPE tau,int m)`	

`F_TYPE x[]`	Phase data array:
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

`F_TYPE *p_sig`	Pointer to sig, the Allan deviation for overlapping samples of phase data
`F_TYPE tau`	The averaging time for the phase data
`int m`	The averaging factor
RETURN: `int`	the # of non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are analyzed. There must be at least 3 non-gap analysis points.
Embedded zeros are treated as gaps in phase data.
Avg Factor m must be >0 and < # data points/2. Tau must be >0.

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE x[512];                                     /* phase data array */
F_TYPE sig;                                                   /* sigma */
F_TYPE tau;                                          /* averaging time */
int m;                                             /* averaging factor */
int num;                                  /* # data points processed */
.
.
tau=1.0;                                              /* set tau value */
m=2;                                                    /* set m value */
num=CalcPhaseOverlapSigma(x, &sig, tau, m);              /* calc sigma */
if(num==-1)                                         /* check for error */
{
    printf("\nError");                                /* error message */
}
else
{
    printf("\nSigma = %e", sig);                     /* display sigma */
    printf("\n# Data Points Analyzed = %d", num);  /* display num */
}
```

SEE ALSO: CalcPhaseSigma(), CalcFreqOverlapSigma()

REFERENCE: NIST Technical Note 1337

```
/*****************************************************************************/
/*                                                                           */
/*                       CalcPhaseOverlapSigma()                             */
/*                                                                           */
/*      Function to calculate Allan variance using overlapping samples      */
/*      of phase data                                                        */
/*                                                                           */
/*      Parameters:     F_TYPE x[]    = phase data array                    */
/*                                      x[0] =  # phase data points          */
/*                                      x[1] = analysis start                */
/*                                      x[2] = analysis end                  */
/*                      F_TYPE *p_sig = pointer to Allan deviation           */
/*                      F_TYPE tau    = averaging time of phase data         */
/*                      int m         = averaging factor                     */
/*                                                                           */
/*      Return:         int           = # analysis points,                   */
/*                                      or -1 if error                       */
/*                                                                           */
/*      Notes:          Embedded zeros treated as gaps in phase data.        */
/*                      Must have at least 1 analysis point; this            */
/*                      requires at least 3 adjacent non-gap phase data      */
/*                      points.  Averaging time for sigma calculation is     */
/*                      m times the tau parameter of the phase data.         */
/*                      m must be >0 , tau must be >0, and the # non-gap     */
/*                      phase data points N must be > 2m.                    */
/*                                                                           */
/*      Revision record:                                                     */
/*          12/16/90    Created                                              */
/*          12/31/91    Renamed                                              */
/*          01/18/92    Edited title block                                   */
/*          01/19/92    Added error traps                                    */
/*          02/03/96    Modified for use as Win 3.1 DLL                      */
/*          04/26/03    Adapted for Version 2.0 source code documentation    */
/*                                                                           */
/* (c) Copyright 1990-2003 Hamilton Technical Services All Rights Reserved   */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) CalcPhaseOverlapSigma(F_TYPE x[], F_TYPE *p_sig,
    F_TYPE tau, int m)
{
    double sum=0;                          /* summing variable to calc sigma */
    float fm=m;                                       /* float version of m */
    int i;                                           /* index for summation */
    int num=0;                                         /* # analysis points */
    int n;                                      /* # frequency data points */

    n=*(x+2)-*(x+1)+1;                         /* total # phase data points */

    if( (m<1) || (n<2*m) || (tau<=0) )
    {
        return(-1);                                                /* error */
    }
    for( i=*(x+1)+2; i<=n-2*m+2; i++)                      /* summing loop */
    /* i basically goes from 1 to N-2m where N is # phase data points and  */
    /* m is averaging factor.  Term *(x+1) is the analysis start limit (1  */
    /* if all data used) & the 2 accounts for the data array offset        */
    {
        if(    ( i==*(x+1)+2 && *(x+i+m) && *(x+i+(2*m)) )
           ||  ( *(x+i)      && *(x+i+m) && *(x+i+(2*m)) )
           ||  ( i==n-2*m+2  && *(x+i+m) && *(x+i)       ) )
                          /* skip if any imbedded phase data value is zero */
        {
            sum += SQR ( *(x+i+(2*m)) - (2 * (*(x+i+m))) + *(x+i) );
            num++;
        }                                                        /* end if */
    }
    if(num>0)
    {
        *p_sig=sqrt( sum / (2.0*tau*tau*fm*fm*num));
        return(num);                                 /* # analysis points */
    }
    else
    {
        return(-1);                                             /* error */
    }
}                                         /* end CalcPhaseOverlapSigma() */
```

The FrequenC Library

NAME: CalcPhaseSigma	**FUNCTION:** Calculate Allan deviation for phase data
SYNOPSIS: `int CalcPhaseSigma(F_TYPE x[], F_TYPE *p_sig, F_TYPE tau)`	

F_TYPE x[]	Phase data array:
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE *p_sig	Pointer to sig, the Allan deviation for the phase data
F_TYPE tau	The averaging time for the phase data
RETURN: int	the # of non-gap data points processed, or -1 if error

REMARKS:
Only data points between start and end analysis limits are analyzed.
There must be at least 3 non-gap analysis points.
Embedded zeros are treated as gaps in phase data. Tau must be >0.

EXAMPLE:

```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE x[512];                                    /* phase data array */
F_TYPE sig;                                        /* Allan deviation */
F_TYPE tau;                                         /* averaging time */
int num;                                 /* # data points processed */
.
.
tau=1.0;                                             /* set tau value */
num=CalcPhaseSigma(x, &sig, tau);                       /* calc sigma */
if(num==-1)                                        /* check for error */
{
    printf("\nError");                               /* error message */
}
else
{
    printf("\nSigma = %e", sig);                     /* display sigma */
    printf("\n# Data Points Analyzed = %d", num);      /* display num */
}
```

SEE ALSO: CalcFreqSigma()

REFERENCE: NIST Technical Note 1337

```
/***************************************************************************/
/*                                                                         */
/*                          CalcPhaseSigma()                               */
/*                                                                         */
/*      Function to calculate Allan variance from phase data              */
/*                                                                         */
/*      Parameters:     F_TYPE x[]    = phase data                         */
/*                                      x[0] = # data points               */
/*                                      x[1]  = analysis start             */
/*                                      x[2] = analysis end                */
/*                      F_TYPE *p_sig = pointer to Allan deviation         */
/*                      F_TYPE tau    = time between phase measurements    */
/*                                                                         */
/*      Return:         int           = # analysis points,                 */
/*                                      or -1 if error                     */
/*                                                                         */
/*      Note:           Embedded zeros treated as gaps in phase data.      */
/*                      Must have at least 1 analysis point; this          */
/*                      requires at least 3 adjacent non-gap phase data    */
/*                      points.  Tau must be >0.                           */
/*                                                                         */
/*      Revision record:                                                   */
/*          12/16/91    Created                                            */
/*          12/31/91    Renamed                                            */
/*          01/18/92    Edited title block                                 */
/*                      Addded error traps                                 */
/*          02/03/96    Modified for use as Win 3.1 DLL                    */
/*          01/01/98    Changes for MS VC++ compatibility & warnings       */
/*          04/26/03    Adapted for Version 2.0 source code documentation  */
/*                                                                         */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                         */
/***************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcPhaseSigma(F_TYPE x[],
    F_TYPE *p_sig, F_TYPE tau)

{
    double sum;                             /* summing variable to calc sigma */
    int i;                                               /* index for summing */
    int num = 0;                                         /* # analysis points */

    if(tau<=0)
    {
        return(-1);                                                  /* error */
    }

    for( i = (int)(*(x+1)+2), sum = 0; i < (int)(*(x+2)+1); i++)
                                               /* index goes from 1 to n-2 */
                        /* for n data points from 1 to n; data starts at x[3] */
    {
        if(    ( i==*(x+1)+2 && *(x+i+1) && *(x+i+2) )
           ||  ( *(x+i)      && *(x+i+1) && *(x+i+2) )
           ||  ( i==*(x+2)+1 && *(x+i  ) && *(x+i+1) ) )
                          /* skip if any imbedded phase data value is zero */
        {
            sum += SQR( *(x+i+2) - 2 * *(x+i+1) + *(x+i) );
            num++;
        }                                                         /* end if */
    }                                                            /* end for */
    if(num>0)
    {
        *p_sig = ( sqrt( sum / (2.0 * num * tau * tau )));
        return(num);                                  /* # analysis points */
    }
    else
    {
        return(-1);                                                /* error */
    }
}                                                /* end CalcPhaseSigma() */
```

The FrequenC Library

NAME: CalcPhaseStdDev	**FUNCTION:** Calculate standard deviation for phase data
SYNOPSIS: int CalcPhaseStdDev(F_TYPE x[], F_TYPE *p_dev, F_TYPE tau)	

F_TYPE x[]	Phase data array:
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE *p_dev	Pointer to dev, the standard deviation for the phase data
RETURN: int	The # of non-gap data points processed, or -1 if error

REMARKS:
Only data points between start and end analysis limits are analyzed.
There must be at least 3 non-gap analysis points.
Embedded zeros are treated as gaps in phase data. Tau must be >0.

EXAMPLE:
```
#include "frequenc.h"                           /* FrequenC header file */
F_TYPE x[513];                                      /* phase data array */
F_TYPE dev;                                       /* standard deviation */
F_TYPE tau;                               /* averaging time of phase data */
int num;                                     /* # data points processed */
.
.
tau=1.0;                                         /* assign value to tau */
num=CalcPhaseSigma(x, &dev, tau);                      /* calc std dev */
if(num==-1)                                          /* check for error */
{
    printf("\nError");                                 /* error message */
}
else
{
    printf("\nStd Dev = %e", dev);                   /* display std dev */
    printf("\n# Data Points Analyzed = %d", num);    /* display num */
}
```

SEE ALSO: CalcFreqStdDev()

REFERENCE: NIST Technical Note 1337

```
/******************************************************************************/
/*                                                                            */
/*                               CalcPhaseStdDev()                            */
/*                                                                            */
/*       Function to calculate standard deviation from phase data            */
/*                                                                            */
/*       Parameters:      F_TYPE x[]    = phase data                          */
/*                                        x[0] = # data points                */
/*                                        x[1] = analysis start               */
/*                                        x[2] = analysis end                 */
/*                        F_TYPE *p_dev = pointer to standard deviation       */
/*                        F_TYPE tau    = time interval between data points   */
/*                                                                            */
/*       Return:          int           = # analysis points,                  */
/*                                        or -1 if error                      */
/*                                                                            */
/*       Note:            Embedded zeros treated as gaps in phase data.       */
/*                        Must have at least 3 non-gap phase data points,     */
/*                        and tau must be >0.                                 */
/*                                                                            */
/*       Revision record:                                                     */
/*           12/16/90     Created                                             */
/*           12/31/91     Renamed                                             */
/*           01/18/92     Edited title block                                  */
/*           06/05/95     Corrected num per standard_dev() function of        */
/*                        STABLE.  S/B # frequency (not phase) data points.   */
/*           02/03/96     Modified for use as Win 3.1 DLL                     */
/*           01/01/98     Changes for MS VC++ compatibility & warnings        */
/*           05/15/00     Added New to function name & put into Stable_r.c    */
/*                        to fix long-standing error                          */
/*           03/08/03     Brought into Ver 2.0 of FrequenC lib                */
/*           04/26/03     Adapted for Version 2.0 source code documentation   */
/*                                                                            */
/* (c) Copyright 1990-2003 Hamilton Technical Services All Rights Reserved   */
/*                                                                            */
/******************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcPhaseStdDev(F_TYPE x[], F_TYPE *p_dev,
    F_TYPE tau)
{
    double sum=0;                                          /* summing variable */
    double avg;                                    /* mean value of freq data */
    int i;                                                /* index for summing */
    int num = (int)(*(x+2)-*(x+1));                 /* # freq analysis points */

    if(tau<=0)
    {
        return(-1);                                                /* error */
    }

    for(i=(int)(*(x+1)+2); i<=(int)(*(x+2)+1); i++)     /* calc average freq */
    {
        if(   ( *(x+i)        && *(x+i+1) )
           || ( i == *(x+1)+2 && *(x+i+1) )
           || ( i == *(x+2)+1 && *(x+i)   )  )
                                /* if don't have an embedded zero data point */
        {
            sum += *(x+i+1) - *(x+i);           /* freq is 1st diff of phase */
        }

        else
        {
            num--;
        }
    }

    if((num-2)>0)                        /* check for <3 non-gap data points */
    {
        avg = sum/num;
    }
    else
    {
        return(-1);                                                /* error */
    }

    sum=0;
    for(i=(int)(*(x+1)+2); i<=(int)(*(x+2)+1); i++)         /* calc variance */
    {
        if(   ( *(x+i)        && *(x+i+1) )                      /* skip if */
           || ( i == *(x+1)+2 && *(x+i+1) )            /* non-imbedded zero */
           || ( i == *(x+2)+1 && *(x+i)   )  )
        {
            sum += SQR( ( *(x+i+1) - *(x+i) ) - avg );
        }                                                         /* end if */
    }                                                            /* end for */

    *p_dev = sqrt(sum/((num-1)*tau*tau));

    return(num);                                 /* # non-zero data points */
}                                                /* end CalcPhaseStdDev() */
```

The FrequenC Library

NAME: ConvPhaseToFreq	FUNCTION: Convert phase data to frequency data
SYNOPSIS: `int ConvPhaseToFreq(F_TYPE x[], F_TYPE y[], F_TYPE tau)`	

F_TYPE x[]	Phase data array:
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE y[]	Frequency data array:
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

F_TYPE tau	Averaging time of phase data
RETURN: int	The # of non-gap data points processed, or -1 if error

REMARKS:
Only phase data between start and end analysis limits are converted. Embedded zeros are treated as gaps in phase data. Tau must be >0. The averaging time (tau) of the frequency data will be the same as that of the phase data. The original phase data is unchanged.

EXAMPLE:
```
#include "frequenc.h"                         /* FrequenC header file */
F_TYPE x[513];                                    /* phase data array */
F_TYPE y[512];                                /* frequency data array */
F_TYPE tau;                                         /* averaging time */
int num;                                   /* # data points processed */
.
.
tau=1.0;                                         /* assign value to tau */
num=ConvPhaseToFreq(x, y, tau);                /* convert phase to freq */
 if(num==-1)                                         /* check for error */
{
    printf("\nError");                                 /* error message */
}
else
{
    printf("\n# Data Points Converted = %d", num);  /* display num */
}
```

SEE ALSO: ConvFreqToPhase()

REFERENCE: None

```
/***************************************************************************/
/*                                                                         */
/*                        ConvPhaseToFreq()                                */
/*                                                                         */
/*      Enhanced function to convert phase (time) data to frequency data   */
/*                                                                         */
/*      Parameters:     F_TYPE x[]  = phase data                           */
/*                                    x[0] = # phase data points           */
/*                                    x[1] = analysis start                */
/*                                    x[2] = analysis end                  */
/*                      F_TYPE y[]  = frequency data                       */
/*                      F_TYPE tau  = time interval between phase samples  */
/*                      BOOL z      = flag to avoid zero freq              */
/*                                                                         */
/*      Return:         int         = # frequency data gaps,               */
/*                                    or -1 if error                       */
/*                                                                         */
/*      Notes:          Embedded zeros treated as gaps in phase data.      */
/*                      Frequency data in gaps interpolated from nearest   */
/*                      non-gap phase values.  Function processes n =      */
/*                      x[2] - x[1] +1 phase points and produces n-1       */
/*                      frequency points.  tau must be >0.                 */
/*                                                                         */
/*      Revision record:                                                   */
/*          12/27/91    Created                                            */
/*          12/31/91    Renamed                                            */
/*          01/18/92    Edited title block                                 */
/*          01/19/92    Added error trap                                   */
/*          02/03/96    Modified for use as Win 3.1 DLL                    */
/*          01/01/98    Changes for MS VC++ compatibility & warnings       */
/*          05/22/01    Added flag for avoiding gaps if zero freq          */
/*          03/08/03    Brought into Ver 2.0 of FrequenC lib               */
/*          04/26/03    Adapted for Version 2.0 source code documentation  */
/*                                                                         */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                         */
/***************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL ConvPhaseToFreq(F_TYPE x[], F_TYPE y[],
    F_TYPE tau, BOOL z)
{
    int g;                                                  /* # phase gaps */
    int i;                                                       /* x index */
    int j;                                                       /* y index */
    int num;                                              /* # phase points */

    if(tau<=0)
    {
        return(-1);                                                /* error */
    }

    num=(int)(x[2]-x[1]+1);                     /* set num = # phase points */
    y[0]=num-1;                                            /* # freq points */
    y[1]=1;                                         /* # of 1st freq point */
    y[2]=num-1;                                   /* # of last freq point */

    /* put phase data to be converted into y[] array */
    j=3;                                        /* initialize index for y[] */
    for(i=(int)(x[1]+2); i<=(int)(x[2]+2); i++ )

    {
        y[j]=x[i];                                          /* copy phase data */
        j++;                                                /* increment index */
    }

    /* convert phase data to freq data */
    g=0;                                                  /* initialize # gaps */
    for(i=3; i<=num+2; i++)
    {
        if( (y[i] && y[i+1]) || (i==3 && y[i+1]) || (y[i] && i==num+2) )
         /* neither phase point a gap, or 1st point with non-zero 2nd point, */
                          /* or last point with non-zero next-to-last point */
        {
            y[i]=(y[i+1]-y[i])/tau;

            /* avoid zero freq = gap */
            if(!y[i] && z)
            {
                y[i]=1e-99;
            }
        }                                                            /* end if */
        else                                                  /* imbedded gap */
        {
            y[i]=0;
            g++;
        }                                                          /* end else */
    }                                                               /* end for */

    return(g);
}                                                /* end ConvPhaseToFreq() */
```

The FrequenC Library

NAME: CalcQuadraticDrift	**FUNCTION:** Calculate the least-squares quadratic fit for phase data

SYNOPSIS:
```
int CalcQuadraticDrift(F_TYPE x[], F_TYPE tau, F_TYPE *p_a,
                       F_TYPE *p_b, F_TYPE *p_c, F_TYPE *p_v)
```

F_TYPE x[]	Phase data array:
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE tau	Averaging time for phase data
F_TYPE *p_a	Pointer to constant term, a, of quadratic fit $x(t)=a+bt+ct^2$, $y(t)=b+2ct$, freq drift=2c
F_TYPE *p_b	Pointer to linear term, b, of quadratic fit
F_TYPE *p_c	Pointer to quadratic term, c, of fit
F_TYPE *p_v	Pointer to fit variance, v
RETURN: int	The # of non-gap data points processed, or -1 if error

REMARKS: Embedded zeros are treated as gaps in phase data.
Gaps are filled during analysis and restored thereafter.
Only data points between start and end analysis limits are analyzed.
There must be at least 3 non-gap analysis points. Tau must be >0.

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE x[513];                                     /* phase data array */
F_TYPE a, b, c, v;                                   /* fit parameters */
F_TYPE tau;                                          /* averaging time */
int num;                                    /* # data points processed */
.
.
num=CalcQuadraticDrift(y, &a, &b, &c, &v);            /* calc quad fit */
if(num==-1)
{
   printf("\nError");                                 /* error message */
}
else
{
    printf("\na = %e, b = %e, c = %e", a,b,c);  /* disp fit params */
    printf("\nv = %e", v);                     /* display fit variance */
    printf("\n# Data Points Analyzed = %d", num);   /* display num */
}
```

SEE ALSO: CalcLinFreqDrift()

REFERENCE: NIST Technical Note 1337

```
/****************************************************************************/
/*                                                                          */
/*                           CalcQuadraticDrift()                           */
/*                                                                          */
/*      Function to calculate least-squares quadratic fit to phase data    */
/*      x(t) = a + bt + ctý                                                 */
/*                                                                          */
/*      Reference:      J.A. Barnes, "The Measurement of Linear Frequency  */
/*                      Drift in Oscillators", Proc. 15th Annual PTTI      */
/*                      Meeting, 1983, pp. 551-582.                         */
/*                                                                          */
/*      Parameters:     F_TYPE x[]  = phase data array                      */
/*                                    x[0] = # phase data points            */
/*                                    x[1] = analysis start                 */
/*                                    x[2] = analysis end                   */
/*                      F_TYPE tau  = time interval for phase data          */
/*                      F_TYPE *p_a = pointer to phase intercept            */
/*                      F_TYPE *p_b = pointer to frequency intercept        */
/*                      F_TYPE *p_c = pointer to frequency slope            */
/*                      F_TYPE *p_v = pointer to fit variance               */
/*                                                                          */
/*      Return:         int         = # analysis points                     */
/*                                    or -1 if error                        */
/*                                                                          */
/*      Notes:          x(t) , t & a in sec, b in sec/sec, c in sec/secý.  */
/*                      Frequency drift (slope) = 2c; intercept = b.       */
/*                      t is time interval of equally-spaced phase data.   */
/*                      t goes from 1 to n with no missing points.         */
/*                      Calls FrequenC functions CountGaps() and           */
/*                      FillGaps() as needed.  To preserve gaps, save      */
/*                      data before calling this function.                 */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/29/91    Created                                             */
/*          12/31/91    Renamed                                             */
/*          01/18/92    Edited title block                                  */
/*          01/19/92    Added error traps                                   */
/*          01/22/92    Added error trap for tau<=0                         */
/*          11/04/92    Changed gap handling -- was NG                      */
/*          02/03/96    Modified for use as Win 3.1 DLL                     */
/*          04/26/03    Adapted for Version 2.0 source code documentation  */
/*                                                                          */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"                                 /* FrequenC header file */

int F__declspec(dllexport) CalcQuadraticDrift(F_TYPE x[], F_TYPE tau,
    F_TYPE *p_a, F_TYPE *p_b, F_TYPE *p_c, F_TYPE *p_v)
{
    double aa,bb,cc,dd,ee,ff,gg;                                /* fit terms */
    double sx,sxx,snx,snnx;                                   /* summations */
    int n=x[2]-x[1]+1;                                          /* # points */
    int i;                                                    /* data index */
    int j=1;                                             /* summation index */
    /* check arguments */
    if(tau<=0)                                    /* tau negative or zero? */
    {
        return(-1);                                               /* error */
    }

    if(n<3)                   /* need at least 3 points to calc parameters */
    {
        return(-1);                                               /* error */
    }

    /* fill gaps if necessary */
    if( CountGaps(x, PHASE) )                                 /* any gaps? */
    {
        FillGaps(x, PHASE);                    /* fill gaps in phase data */
    }

    /* initialize sums */
    sx = sxx = snx = snnx = 0;

    /* calculate sums for quadratic fit */
    for(i=x[1]; i<=x[2]; i++)
    {
        sx  += x[i+2];
        sxx += SQR(x[i+2]);
        snx += x[i+2]*j;
        snnx+= x[i+2]*j*j;
        j++;
    }                                                           /* end for */

    /* calculate terms for quadratic fit */
    aa = 3*((3*n*(n+1))+2);
    bb = -18*((2*n)+1);
    cc = 30;
    dd = (12*((2*n)+1)*((8*n)+11))/((n+1)*(n+2));
    ee = -180/(n+2);
    ff = 180/((n+1)*(n+2));
    gg = n*(n-1)*(n-2);

    /* calculate coefficients for quadratic fit */
    *p_a = ((aa*sx)+(tau*bb*snx)+(tau*tau*cc*snnx))/(gg        );
    *p_b = ((bb*sx)+(tau*dd*snx)+(tau*tau*ee*snnx))/(gg*tau    );
    *p_c = ((cc*sx)+(tau*ee*snx)+(tau*tau*ff*snnx))/(gg*tau*tau);
    *p_v=0;                    /* fit variance 0 if only 3 analysis points */

    if(n>=4)       /* need at least 4 analysis points to calc fit variance */
    {
        *p_v = (sxx-(*p_a*sx)-(tau**p_b*snx)-(tau*tau**p_c*snnx))/(n-3);
    }

    return(n);
}                                             /* end CalcQuadraticDrift() */
```

The FrequenC Library

NAME: `CalcRatio`	**FUNCTION:** Calculate R(n), the ratio of the modified Allan variance to normal Allan variance

SYNOPSIS: `float CalcRatio(int alpha, float num, float bw)`

`int alpha`	The log-log slope, a, of $S_y(f)$ for a power law noise process:
`float num`	The # of phase data points = # frequency data points + 1
`int bw`	Bandwidth factor = $2 \cdot p \cdot f_h \cdot t_0$ where f_h is measuring system bandwidth
RETURN: `float`	The R(n) ratio function = Mod $s^2_y(t)/s^2_y(t)$, or -1 if error

Noise Type	a
Flicker Walk FM	-3
Random Walk FM	-2
Flicker FM	-1
White FM	0
Flicker PM	1
White PM	2

REMARKS:
The bandwidth factor applies only to the a = +1 Flicker PM case

EXAMPLE:
```
#include "frequenc.h"                      /* FrequenC header file */
x[513];                                        /* phase data array */
float bw;                                      /* bandwidth factor */
float r;                                                   /* R(n) */
int num;                                   /* # phase data points */
int alpha;                                /* noise exponent alpha */
.
.
alpha=0;                                              /* set alpha */
num=(int) y[0];                                /* set # data points */
bw=10.0;                                                 /* set bw */
r=CalcRatio(alpha, num, bw);                           /* calc R(n) */
if(r==-1)                                        /* check for error */
{
    printf("\nError");                             /* error message */
}
else
{
    printf("\ndr = %e", r);                            /* display r */
}
```

SEE ALSO: None

REFERENCE: NIST Technical Note 1337

```
/****************************************************************************/
/*                                                                          */
/*                         CalcQuadraticDrift()                             */
/*                                                                          */
/*      Function to calculate least-squares quadratic fit to phase data    */
/*      x(t) = a + bt + ctý                                                 */
/*                                                                          */
/*      Reference:      J.A. Barnes, "The Measurement of Linear Frequency  */
/*                      Drift in Oscillators", Proc. 15th Annual PTTI      */
/*                      Meeting, 1983, pp. 551-582.                         */
/*                                                                          */
/*      Parameters:     F_TYPE x[]  = phase data array                      */
/*                                    x[0] = # phase data points            */
/*                                    x[1] = analysis start                 */
/*                                    x[2] = analysis end                   */
/*                      F_TYPE tau  = time interval for phase data          */
/*                      F_TYPE *p_a = pointer to phase intercept            */
/*                      F_TYPE *p_b = pointer to frequency intercept        */
/*                      F_TYPE *p_c = pointer to frequency slope            */
/*                      F_TYPE *p_v = pointer to fit variance               */
/*                                                                          */
/*      Return:         int         = # analysis points                     */
/*                                    or -1 if error                        */
/*                                                                          */
/*      Notes:          x(t) , t & a in sec, b in sec/sec, c in sec/secý.  */
/*                      Frequency drift (slope) = 2c; intercept = b.       */
/*                      t is time interval of equally-spaced phase data.   */
/*                      t goes from 1 to n with no missing points.         */
/*                      Calls FrequenC functions CountGaps() and           */
/*                      FillGaps() as needed.  To preserve gaps, save      */
/*                      data before calling this function.                 */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/29/91    Created                                             */
/*          12/31/91    Renamed                                             */
/*          01/18/92    Edited title block                                  */
/*          01/19/92    Added error traps                                   */
/*          01/22/92    Added error trap for tau<=0                         */
/*          11/04/92    Changed gap handling -- was NG                      */
/*          02/03/96    Modified for use as Win 3.1 DLL                     */
/*          04/26/03    Adapted for Version 2.0 source code documentation  */
/*                                                                          */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"                                /* FrequenC header file */

int F__declspec(dllexport) CalcQuadraticDrift(F_TYPE x[], F_TYPE tau,
    F_TYPE *p_a, F_TYPE *p_b, F_TYPE *p_c, F_TYPE *p_v)
{
    double aa,bb,cc,dd,ee,ff,gg;                                /* fit terms */
    double sx,sxx,snx,snnx;                                    /* summations */
    int n=x[2]-x[1]+1;                                           /* # points */
    int i;                                                   /* data index */
    int j=1;                                            /* summation index */
    /* check arguments */
    if(tau<=0)                                    /* tau negative or zero? */
    {
        return(-1);                                               /* error */
    }

    if(n<3)                   /* need at least 3 points to calc parameters */
    {
        return(-1);                                               /* error */
    }

    /* fill gaps if necessary */
    if( CountGaps(x, PHASE) )                                 /* any gaps? */
    {
        FillGaps(x, PHASE);                     /* fill gaps in phase data */
    }

    /* initialize sums */
    sx = sxx = snx = snnx = 0;

    /* calculate sums for quadratic fit */
    for(i=x[1]; i<=x[2]; i++)
    {
        sx  += x[i+2];
        sxx += SQR(x[i+2]);
        snx += x[i+2]*j;
        snnx+= x[i+2]*j*j;
        j++;
    }                                                           /* end for */

    /* calculate terms for quadratic fit */
    aa = 3*((3*n*(n+1))+2);
    bb = -18*((2*n)+1);
    cc = 30;
    dd = (12*((2*n)+1)*((8*n)+11))/((n+1)*(n+2));
    ee = -180/(n+2);
    ff = 180/((n+1)*(n+2));
    gg = n*(n-1)*(n-2);

    /* calculate coefficients for quadratic fit */
    *p_a = ((aa*sx)+(tau*bb*snx)+(tau*tau*cc*snnx))/(gg        );
    *p_b = ((bb*sx)+(tau*dd*snx)+(tau*tau*ee*snnx))/(gg*tau    );
    *p_c = ((cc*sx)+(tau*ee*snx)+(tau*tau*ff*snnx))/(gg*tau*tau);
    *p_v=0;                     /* fit variance 0 if only 3 analysis points */

    if(n>=4)       /* need at least 4 analysis points to calc fit variance */
    {
        *p_v = (sxx-(*p_a*sx)-(tau**p_b*snx)-(tau*tau**p_c*snnx))/(n-3);
    }

    return(n);
}                                             /* end CalcQuadraticDrift() */
```

The FrequenC Library

NAME: CalcStarB1	**FUNCTION:** Description of what function does

SYNOPSIS:
```
int FunctionName( F_TYPE z[], F_TYPE one, F_TYPE two)
```

F_TYPE z[]	Phase or frequency data array
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysios end point

F_TYPE one	parameter one, and description of what it is
F_TYPE two	parameter two, and description of what it is
RETURN: int	# non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are modified.
Embedded zeros are treated as gaps in phase data.
All zeros are treated as gaps in frequency data.

EXAMPLE:
```
#include <frequenc.h>                       /* FrequenC header file */
F_TYPE z[512];                                        /* data array */
F_TYPE one;                                        /* parameter one */
F_TYPE two;                                        /* parameter two */
int num;                                /* # data points processed */
.
.
one=1.0;                                /* set 1st parameter value */
two=2.0;                                /* set 2nd parameter value */
num=FunctionName(z, one, two);                     /* call function */
printf("\n# Data Points Scaled = %d", num);          /* display num */
```

SEE ALSO: RelatedFunctionName()

REFERENCE: NIST Technical Note 1337

```
/*****************************************************************************/
/*                                                                           */
/*                              CalcStarB1()                                 */
/*                                                                           */
/*   Function to calculate the Barnes B1 ratio of the N-sample to 2-sample   */
/*   (standard to Allan) variance for frequency data treated as phase data   */
/*   which is used to resolve alpha=3 or -4 ambiguity for Hadamard sigmas.   */
/*   This is called *B1 to distinguish it from the normal B1 ratio.          */
/*                                                                           */
/*   Parameters:    F_TYPE  *fData          Data array                       */
/*                  BOOL    bDataType       Data type (0=Phase, 1=Freq)      */
/*                  int     fTau                                             */
/*                  int     nAF             Averaging factor                 */
/*                                                                           */
/*   Return:        float   fB             *B1 ratio                         */
/*                                          or -1.0 if error                 */
/*                                                                           */
/*   Notes:         What we are doing here is to use Chuck Greenhall's       */
/*                  idea to calculate the B1 ratio for the freq data         */
/*                  as phase data, and use the standard noise ID method      */
/*                  except for adding 2 to the resulting mu (subtracting     */
/*                  2 from alpha).                                           */
/*                                                                           */
/*   References:    1.  C.A. Greenhall, (private communication)              */
/*                  2.  D. Howe, et al PTTI 2000 paper                       */
/*                                                                           */
/*      Revision record:                                                     */
/*          11/14/00    Created                                              */
/*          11/15/00    Changed parameters                                   */
/*          04/26/03     Adapted for Version 2.0 source code documentation      */
/*                                                                           */
/* (c) Copyright 2000-3  Hamilton Technical Services  All Rights Reserved    */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL CalcStarB1(F_TYPE fData[],
    BOOL bDataType, F_TYPE fTau, int nAF)
{
    // Local variables
    F_TYPE *fWork;   // Pointer to working data array
    F_TYPE fSig;     // Allan deviation
    F_TYPE fStdDev;  // Standard deviation
    int nNum;        // # data points
    int i;           // Index

    // Get # data points in data array
    nNum=(int)fData[0];

    // Allocate working array of freq data
    // Data is subject to being averaged, so must use a working array
    if((fWork=(F_TYPE*)malloc((nNum+ARRAY_OFFSET)*sizeof(F_TYPE)))==NULL)
    {
        // Error
        return -1.0;
    }
    // For phase data, convert it to freq data in working array
    if(bDataType==PHASE_DATA)
    {
        // Conv phase data to freq data
        ConvPhaseToFreq(fData, fWork, fTau, TRUE);
    }
    else // Freq data - no conversion needed - copy it to working array
    {
        // Copy freq data to working array
        for(i=0; i<nNum+ARRAY_OFFSET; i++)
        {
            fWork[i]=fData[i];
        }
    }

    // Average working freq data
    AverageFreqData(fWork, nAF);

    // Calc its sigma and std dev as if it were phase data
    CalcPhaseSigma(fWork, &fSig, fTau);
    CalcPhaseStdDev(fWork, &fStdDev, fTau);

    // Free the working memory
    free(fWork);

    // Calc & return *B1 = ratio of standard to Allan variance
    return (float)(SQR(fStdDev/fSig));
}
```

The FrequenC Library	
NAME: CalcSecondDiff	**FUNCTION:** Calculate the second difference for phase data
SYNOPSIS: int CalcQuadraticDrift(F_TYPE x[], F_TYPE *p_d)	

F_TYPE x[]	Phase data array:
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE *p_d	Pointer to the average of the 2nd differences (frequency drift) for the phase data
RETURN: int	The # of analysis points processed, or -1.0 if error

REMARKS:
Only data points between start and end analysis limits are analyzed.
There must be at least 3 non-gap analysis points.
Embedded zeros are treated as gaps in phase data.

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE x[513];                                     /* phase data array */
F_TYPE d;                                            /* 2nd difference */
int num;                                          /* # analysis points */
.
.
num=CalcSecondDiff(x, &d);                   /* calc second difference */
if(num==-1)
{
    printf("\nError");                                /* error message */
}
else
{
    printf("\n2nd Diff = %e, d);             /* display 2nd difference */
}
```

SEE ALSO: CalcQuadraticDrift()

REFERENCE: NIST Technical Note 1337

```
/****************************************************************************/
/*                                                                          */
/*                           CalcSecondDiff()                               */
/*                                                                          */
/*      Function to calculate average of 2nd differences of phase data     */
/*                                                                          */
/*      Parameters:     F_TYPE x[]  = array of phase data                  */
/*                                    x[0] = # phase data points            */
/*                                    x[1] = analysis start                 */
/*                                    x[2] = analysis end                   */
/*                      F_TYPE *p_d = pointer to average of second          */
/*                                    differences of phase data             */
/*                                  = frequency drift of phase data         */
/*                                                                          */
/*      Return:         int         = # anlaysis points,                     */
/*                                    or -1 if error                        */
/*                                                                          */
/*      Note:           Embedded gaps are treated as gaps in phase data.    */
/*                      There must be at least 1 analysis point,            */
/*                      which requires at least 3 adjacent non-gap          */
/*                      phase data points.                                  */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/29/91    Created                                             */
/*          12/31/91    Renamed                                             */
/*          01/18/92    Edited title                                        */
/*          01/19/92    Added d parameter for result, changed return to     */
/*                      int = # analysis points, added error trap           */
/*          02/03/96    Modified for use as Win 3.1 DLL                     */
/*          01/01/98    Changes for MS VC++ compatibility & warnings        */
/*          04/26/03     Adapted for Version 2.0 source code documentation    */
/*                                                                          */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcSecondDiff(F_TYPE x[], F_TYPE *p_d)
{
    float num;
    double d1;
    double d2;
    double sum;
    int i;

    num = 0;
    sum = 0;

    for(i=(int)(x[1]+2); i<=(int)x[2]; i++)
    {
        if((x[i] && x[i+1] && x[i+2]) || (i==3 && x[4] && x[5]) \
                                  || (x[i+1] && x[i] && i==x[2]))
                                                  /* no phase points are gaps, */
                             /* or 1st point with non-zero 2nd & 3rd points, */
                               /* or last point with non-zero next-to-last & */
                                                /* next-to-next-to-last points */
        {
            d1  =  x[i+1] - x[i];
            d2  =  x[i+2] - x[i+1];
            sum += d2 -d1;
            num++;
        }                                                           /* end if */

    }                                                              /* end for */
    if(num>0)
    {
        *p_d = sum / num;
        return(int)num;
    }
    else
    {
        return(-1);                                                 /* error */
    }
}                                                /* end CalcSecondDiff() */
```

The FrequenC Library

NAME: `CalcThreePointDrift`	**FUNCTION:** Calculate frequency drift using the first, middle and last phase data points.

SYNOPSIS:
```
float CalcThreePointDrift(F_TYPE x[])
```

`F_TYPE x[]`	Phase data array
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

RETURN: `float`	Frequency drift per tau interval

REMARKS:
Only data points between start and end analysis limits are analyzed.
Embedded zeros are treated as gaps in phase data.
This is the best estimator of drift for the combination of white and random walk FM noise (see reference).
Tau=1 is assumed.   Drift is per tau interval.
This function calls FrequenC functions CountGaps()and FillGaps().
To preserve gaps, save original phase data before calling this function.

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE x[512];                                          /* data array */
float drift;                                            /* freq drift */
.
.
drift= CalcThreePointDrift(x);                       /* call function */
printf("\nDrift = %e", drift);                      /* display result */
```

SEE ALSO: `CalcQuadraticDrift()`

REFERENCE: M. Weiss, D. Allan, and D. Howe, "Confidence on the Second Difference Estimation of Frequency Drift", 1992 IEEE Frequency Control Symposium, pp.300-305, June, 1992.

```
/***************************************************************************/
/*                                                                         */
/*                       CalcThreePointDrift()                             */
/*                                                                         */
/*      Function to calculate frequency drift using the first, middle      */
/*      and last phase data points.
/*                                                                         */
/*                                                                         */
/*      Parameters:     x[]     phase data array                           */
/*                                                                         */
/*      Return:         float   frequency drift                            */
/*                                                                         */
/*      Notes:  This is the best estimator of drift for the combination    */
/*              of white and random walk FM noise (see reference).         */
/*              Tau=1 is assumed.   Drift is per tau interval.             */
/*              This function calls FrequenC functions CountGaps() and     */
/*              FillGaps().  To preserve gaps, save original phase data    */
/*              before calling CalcThreePointDrift().                      */
/*                                                                         */
/*      Reference:                                                         */
/*      M. Weiss, D. Allan, and D. Howe, "Confidence on the Second Diff-   */
/*      erence Estimation of Frequency Drift", 1992 IEEE Frequency Control */
/*      Symposium, pp.300-305, June, 1992.                                 */
/*                                                                         */
/*      Revision record:                                                   */
/*          11/01/92    Created                                            */
/*          02/03/96    Removed unused variable: int i                     */
/*                      Modified for use as Win 3.1 DLL                    */
/*          01/01/98    Changes for MS VC++ compatibility & warnings       */
/*          04/26/03    Adapted for Version 2.0 source code documentation  */
/*                                                                         */
/* (c) Copyright 1992-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                         */
/***************************************************************************/

#include "frequenc.h"                                 /* FrequenC header file */

float __declspec(dllexport) FAR PASCAL CalcThreePointDrift(F_TYPE x[])
{
    int start;                                /* first phase analysis point */
    int mid;                                 /* middle phase analysis point */
    int end;                                   /* last phase analysis point */

    float n1, n2;             /* # analysis points in 1st and last halves */
    float n3;                        /* avg # analysis points in each half */
    float d;                           /* frequency drift per tau interval */

    /* fill gaps if necessary */
    if( CountGaps(x, PHASE) )                                /* any gaps? */
    {
        FillGaps(x, PHASE);                      /* fill gaps in phase data */
    }                                                           /* end if */

    /* calculate drift */
    start = (int) x[1];                             /* first analysis point */
    mid = (int) ((x[2]-x[1])+2)/2;                 /* middle analysis point */
    end = (int) x[2];                                 /* last analysis point */
    n1 = (float) mid-start;                          /* # points in 1st half */
    n2 = (float) end-mid;                            /* # points in 2nd half */
    n3 = (n1+n2)/2;                                        /* drift interval */
    d = (float)((((x[end+2]-x[mid+2])/n2) - ((x[mid+2]-x[start+2])/n1))/n3);

    return(d);                                                 /* frequency drift */
}                                                    /* end CalcThreePointDrift() */
```

The FrequenC Library

NAME: DateConv	**FUNCTION:** Convert month & year to day of week, DOY, MJD, and # days in month

SYNOPSIS:
```
void DateConv(int *lpnMonth, int *lpnDaysInMonth,
   int *lpnYear, int *lpnDOW, int *lpnDOY, long *lplnMJD)
```

int *p_month	Pointer to month # (January=1)
int *p_days	Pointer to # days in month
int *p_year	Pointer to 4-digit year (e.g. 2003)
int *p_dow	Pointer to day of week (Sunday=0)
int *p_doy	Pointer to day of year
long *lp_mjd	Pointer to Modified Julian Date
RETURN: void	

REMARKS:
Calling program sets month and year. 1st day of month is assumed. Function calculates corresponding day of week, day of year, MJD and # days in month.

EXAMPLE:
```
#include "frequenc.h"                        /* FrequenC header file */
int month;                                                /* month # */
int days;                                       /* # days in month */
int year;                                                  /* year */
int dow;                                            /* day of week */
int doy;                                                   /* year */
long mjd;                                                   /* MJD */
.
.
month=1;                                              /* set month */
year=2003;                                             /* set year */
DateConv(&month, &days, &year, &dow, &doy, &mjd); /* call function */
printf("\nDOW = %d", DOY = %d, MJD = %ld, Days/Month = %d", dow, doy,
    mjd, days);                                 /* display results */
```

SEE ALSO: MJDtoDOY(), MJDtoDate()

REFERENCE: None

```
/*****************************************************************************/
/*                                                                           */
/*                           DateConv()                                      */
/*                                                                           */
/*      Function to convert month & year to day of week, DOY and MJD,        */
/*      plus # days in month.                                                */
/*                                                                           */
/*      Calling program sets month and year.  1st day of month is assumed.   */
/*      Function sets corresponding day of week, day of year, MJD and        */
/*      # days in month.                                                     */
/*                                                                           */
/*      Uses date functions from Hammer library.                             */
/*                                                                           */
/*      Parameters:                                                          */
/*          int  lpnMonth          pointer to month (January=1)              */
/*          int  lpnDaysInMonth    pointer to # days in month                */
/*          int  lpnYear           pointer to year (e.g. 1995)               */
/*          int  lpnDOW            pointer to day of week (Sunday=0)         */
/*          int  lpnDOY            pointer to day of year                    */
/*          long lplnMJD           pointer to Modified Julian Date           */
/*                                                                           */
/*      Returns nothing (void)                                               */
/*                                                                           */
/*      Revision record:                                                     */
/*          03/30/96    Adapted from date_conv() of SCRN_CAL.C               */
/*          04/08/97    Fixed bug in century leap year logic                 */
/*          07/18/97    Changed to new date function to avoid Hammer Library */
/*          04/26/03    Adapted for Version 2.0 source code documentation    */
/*                                                                           */
/* (c) Copyright 1996-2003 Hamilton Technical Services All Rights Reserved   */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"                             /* FrequenC Library header file */

void __declspec(dllexport) FAR PASCAL DateConv(int *lpnMonth,
    int *lpnDaysInMonth, int *lpnYear, int *lpnDOW, int *lpnDOY, long *lplnMJD)
{
    static int sum[] = { 0, 31, 59, 90, 120, 151, 181, 212,
                          243, 273, 304, 334, 365 };
    static int num_days[] = {31, 28, 31, 30, 31, 30, 31, 31,
                          30, 31, 30, 31};
    // Local variables
    int month;
    int year;
    int dow;                             /* day of week: 0=Sunday ... 6=Saturday */
    int day=1;                                         /* day of month always 1 */
    int doy;                                                    /* day of year */
    int leap=0;                                               /* leap year flag */
    long int gdate;                                          /* Gregorian date */
    long int jdate;                                             /* Julian day # */
    long int mjd;                                         /* Modified Julian Day */

    // Initializations
    month=*lpnMonth;
    year=*lpnYear;

    // Date conversions
    gdate=MakeDate(month, day, year);
    BreakDate(gdate, &month, &day, &year);
    dow=DayOfWeek(gdate);

    doy=sum[month-1]+day;

    if( year % 4 == 0 && year % 100 != 0 || year % 400 == 0 )
    {
        leap=1;
        if(month>2)
        {
            doy++;
        }                                                           /* end if */
    }                                                               /* end if */

    jdate=DateToJulian(gdate);
    mjd=jdate-2400001L;

    // Pass results
    *lplnMJD=mjd;
    *lpnDOW=dow;
    *lpnDOY=doy;

    if(month!=2)
    {
        leap=0;
    }                                                               /* end if */

    *lpnDaysInMonth=num_days[month-1]+leap;
}                                                       /* end date_convert() */
```

The FrequenC Library

NAME: DateToMJD	**FUNCTION:** Convert date to MJD

SYNOPSIS:
`long DateToMJD(int day, int month, int year)`

int day	Day of date to be converted to MJD
int month	Month of date to be converted to MJD
int year	Year of date to be converted to MJD
RETURN: long	Modified Julian date # for date

REMARKS:
Example: November 11, 1992 is MJD 48937.

EXAMPLE:
```
#include "frequenc.h"                        /* FrequenC header file */
int day=11;                                                 /* day */
int month=11;                                             /* month */
int year=1992;                                             /* year */
long mjd;                                                   /* MJD */
.
.
mjd=DateToMJD(day, month, year);                  /* call function */
printf("\nMJD = %ld", mjd);                         /* display MJD */
```

SEE ALSO: MJDtoDate()

REFERENCE:

```
/****************************************************************************/
/*                                                                          */
/*                           DateToMJD()                                    */
/*                                                                          */
/*       Function for date to MJD conversion                                */
/*                                                                          */
/*       Parameters:                                                        */
/*           int  day      day of date to be converted to MJD               */
/*           int  month    month of date to be converted to MJD             */
/*           int  year     year of date to be converted to MJD              */
/*                                                                          */
/*       Return:                                                            */
/*           long mjd      modified Julian day # for date                   */
/*                                                                          */
/*       Example:                                                           */
/*           November 11, 1992 is MJD 48937.                                */
/*                                                                          */
/*       Function revision record:                                          */
/*           11/11/92    Created                                            */
/*           11/28/92    Adapted for FrequenC Library                       */
/*           02/03/96    Changed constant 679019 to L                       */
/*                       Modified for use as Win 3.1 DLL                    */
/*           01/01/98    Changes for MS VC++ compatibility & warnings       */
/*           04/26/03     Adapted for Version 2.0 source code documentation    */
/*                                                                          */
/*   (c) Copyright 1992-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"                              /* FrequenC header file */

long __declspec(dllexport) FAR PASCAL DateToMJD(int day, int month, int year)
{
    if(month>2)
    {
        month++;
    }                                                          /* end if */
    else
    {
        month+=13;
        year--;
    }                                                        /* end else */

    return(-679019L+(long)(365.25*year)+(long)(30.6001*month)+day);

}                                                   /* end DateToMjd() */
```

The FrequenC Library

NAME: DayOfWeek	**FUNCTION:** Find day of week for a given Gregorian date

SYNOPSIS:
```
int WINAPI DayOfWeek(long gdate)
```

long gdate	Gregorian date (yyyymmdd)
RETURN: int	Day of week (0=Sunday, 6=Saturday)

REMARKS:
Covers years 1900-2099
Calls BreakDate()

EXAMPLE:
```
 #include "frequenc.h"                           /* FrequenC header file */
 long gdate;                                            /*Gregorian date */
 int dow;                                                 /* day of week */
 .
 .
 gdate=20030101;                                             /* set date */
 dow=DayOfWeek(gdate);                                  /* call function */
 printf("\nDOW = %d", dow);                               /* display dow */
```

SEE ALSO: MakeDate(), BreakDate()

REFERENCE: NIST Technical Note 1337

```
/*****************************************************************************/
/*                                                                           */
/*                               DayOfWeek()                                 */
/*                                                                           */
/*      Function to find the day of the week for a given Gregorian date      */
/*                                                                           */
/*      Note: Calls Brkdate() above                                          */
/*                                                                           */
/*      Parameters: long    gdate   Gregorian date (yyyymmdd)                */
/*                                                                           */
/*      Return:     int             Day of week (0=Sunday, 6=Saturday)       */
/*                                                                           */
/*      Revision record:                                                     */
/*          07/17/97    Cloned from dayofwk() of Hammer Library              */
/*                      Changed error return to -1                           */
/*                      Modified for 4-digit year from BreakDate()           */
/*          04/26/03     Adapted for Version 2.0 source code documentation      */
/*                                                                           */
/*  (c) Copyright 1997-2003 Hamilton Technical Services All Rights Reserved  */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"                           /* FrequenC Library header file */

int __declspec(dllexport) FAR PASCAL DayOfWeek(long gdate)
{
    // Local variables
    int a, b, mo, da, yr;
    static int mos[12] = { 0,3,3,6,1,4,6,2,5,0,3,5 };

    BreakDate(gdate, &mo, &da, &yr);

    // Convert 4-digit year into 2-digit year
    // yr=yr % 100;  // Only OK 1900-1999
    yr=yr-1900;      // Better; OK 1900-2099

    if ( mo<1 || mo>12 || da<1 || da>31 || yr<0 )
    {
        return (-1);    /* error - bad date */
    }

    a = (yr/4) + yr + da + mos[mo-1];
    b = (a % 7) + 1;

    if (mo<=2 && yr!=0 && (yr % 4)==0)
    {
        b = (b==0 ? 6 : b) - 1;
    }

    b = (b==0 ? 7 : b) - 1;

    return (b); /* return day of week (0-6) */
}
```

The FrequenC Library	
NAME: DateToJulian	**FUNCTION:** Convert Gregorian date to Julian day number
SYNOPSIS: long DateToJulian(long gdate)	
long gdate	Gregorian Date (yyyymmdd)
RETURN: long	Julian day number
REMARKS:	

EXAMPLE:

```
#include "frequenc.h"                           /* FrequenC header file */
long gdate;                                           /* Gregorian date */
long jd;                                               /* Julian day # */
    .
.

jd=DateToJulian(gdate);                                /* call function */
printf("\nJulian Day # = %ld", jd);                  /* display results */
```

SEE ALSO: MJDtoDOY(), MJDtoDate()

REFERENCE: None

```
/******************************************************************************/
/*                                                                            */
/*                              DateToJulian()                                */
/*                                                                            */
/*      Function to convert Gregorian date to Julian day number               */
/*                                                                            */
/*      Parameters: long    gdate   Gregorian date (yyyymmdd)                 */
/*                                                                            */
/*      Return:     long    jdate   Julian day number                         */
/*                                                                            */
/*      Revision record:                                                      */
/*          07/17/97    Cloned from datjul() of Hammer Library                */
/*                      Changed error return to -1                            */
/*                      Modified to use 4-digit year return from BreakDate()  */
/*          04/26/03     Adapted for Version 2.0 source code documentation    */
/*                                                                            */
/* (c) Copyright 1997-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                         */
/******************************************************************************/

#include "frequenc.h"                         /* FrequenC Library header file */

long __declspec(dllexport) FAR PASCAL DateToJulian(long gdate)
{
    // Local variables
    long jdate; /* Julian Day Number for given date */
    int mo, da, yr, c, ya;

    if (gdate<=0L || gdate==19000000L)      /* if zero or negative, return -1 */
    {
        return (-1L); /* error */
    }

    BreakDate(gdate, &mo, &da, &yr);

    if (mo>2)
    {
        mo-=3;
    }
    else
    {
        mo+=9;
        yr--;
    }

    c=yr/100;
    ya=yr%100;
    jdate=(146097L*c)/4L + (1461L*ya)/4L + (153L*mo+2L)/5L + da + 1721119L;

    return (jdate);
}
```

The FrequenC Library	
NAME: FillGaps	**FUNCTION:** Fill gaps in phase or frequency data
SYNOPSIS: int FillGaps(F_TYPE z[], BOOL datype)	

F_TYPE z[]	Phase or frequency data array:
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysis end point

BOOL datype	Type of data: 0=phase, 1=frequency
RETURN: int	The # of gaps filled

REMARKS:
Only data gaps between start and end analysis limits are filled.
All zeros are treated as gaps in frequency data.
Only embedded gaps are treated as gaps in phase data.
The datype flag determines whether the data is treated as phase or frequency data. The gaps processed are filled with interpolated values. No error codes are returned by this function.

EXAMPLE:
```
#include "frequenc.h"                             /* FrequenC header file */
F_TYPE z[512];                                              /* data array */
BOOL datype;                                            /* data type flag */
int num;                                                        /* # gaps */
.
.
datype=FREQ;                                /* set flag for frequency data */
         /* frequenc.h contains #define PHASE 0 and #define FREQ 1 */
num=Fill(z, datype);                                 /* fill gaps in data */
printf("\n# Gaps Filled = %d", num);                       /* display num */
```

SEE ALSO: CountGaps()

REFERENCE: None

```
/***************************************************************************/
/*                                                                         */
/*                          FillGaps()                                     */
/*                                                                         */
/*      Function to fill gaps in data with interpolated values             */
/*                                                                         */
/*      Parameters:     F_TYPE z[]  = phase or frequency data              */
/*                                    z[0] = # data points                 */
/*                                    z[1] = analysis start                */
/*                                    z[2] = analysis end                  */
/*                      BOOL datype = data type: 0=phase, 1=frequency      */
/*                                                                         */
/*      Return:         int         = # gaps removed                       */
/*                                                                         */
/*      Note:           All zeros treated as gaps in frequency data.       */
/*                      Embedded zeros treated as gaps in phase data.      */
/*                                                                         */
/*      Revision record:                                                   */
/*          12/16/91    Created                                            */
/*          12/24/91    Changed to make datype a parameter rather than a   */
/*                      global variable.  Remove call to inc_extension().  */
/*          12/31/91    Renamed                                            */
/*          01/18/92    Edited title                                       */
/*          07/22/93    Fixed bug that changed i inside interior gap loop  */
/*          02/03/96    Modified for use as Win 3.1 DLL                    */
/*          01/01/98    Changes for MS VC++ compatibility & warnings       */
/*          04/26/03    Adapted for Version 2.0 source code documentation  */
/*                                                                         */
/* (c) Copyright 1991-2003 Hamilton Technical Services All Rights Reserved */
/*                                                                         */
/***************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL FillGaps(F_TYPE z[], BOOL datype)

{
    int g = 0;                                           /* total gap count */
    int i;                                         /* general (outer) index */
    int i_start;                   /* index of last data point at start of gap */
    int i_end;                       /* index of next data point at end of gap */
    int j;                                                   /* inner index */
    int k;                                               /* local gap count */
    int n = (int)*z;                                       /* # data points */
    double increment;               /* freq increment for interpolated data */

    for(i=(int)(*(z+1)+3-datype); i<=(int)(*(z+2)+1+datype); i++)
                                                       /* elim leading gaps */
    {
        if(!*(z+i))                                  /* is data point a gap? */
        {
            for(j=i; j<n+2; j++)         /* move all data up to remove gap */
            {
                *(z+j) = *(z+j+1);
            }                                                    /* end for */
            i--;   /* after shifting, must test the first data point again */
            g++;                             /* increment the total gap count */
            n--;                                 /* decrement the data counts */
            *z=n;
            *(z+2)=*(z+2)-1;
        }                                                    /* end if */

        else
        {
            break;        /* stop when first non-gap data point is reached */
        }                                                  /* end else */
    }                                                       /* end for */

    for(i=(int)(*(z+2)+1+datype); i>=(int)(*(z+1)+3-datype); i--)
                                                /* elim trailing gaps */
    {
        if(!*(z+i))                               /* is data point a gap? */
        {
            for(j=i; j<n+2; j++)         /* move all data up to remove gap */
            {
                *(z+j) = *(z+j+1);
            }                                               /* end for */
            g++;                          /* increment the total gap count */
            n--;                              /* decrement the data counts */
            *z=n;
            *(z+2)=*(z+2)-1;
        }                                                    /* end if */
        else
        {
            break;         /* stop when last non-gap data point is reached */
        }
    }                                                       /* end for */

    k=0;                  /* replace interior gaps with interpolated values */
    for(i=(int)(*(z+1)+3-datype); i<=(int)(*(z+2)+1+datype); i++)
    {
        if(!*(z+i))                               /* is data point a gap? */
        {
            k++;                          /* increment the local gap count */
        }                                                    /* end if */
        else
        {
            if(k)                                     /* are we at a gap? */
            {
                i_end = i;        /* save the index for the end of the gap */
                i_start = i-k-1;    /* save index for the start of the gap */
                increment = (*(z+i_end) - *(z+i_start)) / (k+1);
                for(j=1; j<=k; j++)
                {
                    *(z+i_start+j) = *(z+i_start) + increment*j;
                    g++;                        /* increment the gap count */
                }                                           /* end for */
                k=0;                    /* reinitialize count for next gap */
            }                                                /* end if */
        }                                                  /* end else */
    }                                                       /* end for */

    return(g);                             /* return total # gaps removed */
}                                                    /* end FillGaps() */
```

The FrequenC Library

NAME: FindMedian	**FUNCTION:** Find median value in data array

SYNOPSIS:

```
int FindMedian(F_TYPE z[], BOOL datype, F_TYPE *median)
```

F_TYPE z[]	Phase or frequency data array:
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysis end point

BOOL datype	Type of data: 0=phase, 1=frequency
F_TYPE median	Median value
RETURN: int	0 if OK, or -1 if error

REMARKS:

```
Only data between start and end analysis limits are analyzed.
All zeros are treated as gaps in frequency data.
Embedded zeros only are treated as gaps in phase data.
The datype flag determines whether data is treated as phase or freq.
Median is always center value for odd # data points.
Median is average of two center values for even # data points <100,
and is simply the smaller of the two for even # data points >=100.
```

EXAMPLE:

```
#include "frequenc.h"                      /* FrequenC header file */
F_TYPE y[512];                                  /* freq data array */
BOOL datype=FREQ;                                /* data type flag */
                /* frequenc.h contains #defines PHASE 0 and FREQ 1 */
int code;                                           /* return code */
.
.
code=FindMedian(y, datype, &median);              /* call function */
printf("\nMedian = %e", median);                 /* display result */
```

SEE ALSO:

REFERENCE: None

```
/****************************************************************************/
/*                                                                          */
/*                             Median Functions                             */
/*                                                                          */
/*       Functions to find median value in array.                           */
/*       FindMedian() calls FindMedian1() or FindMedian2()depending on      */
/*       array size.  FindMedian1() calls FindMed() for size <100.          */
/*       FindMedian2() calls nrselect() for size >=100.                     */
/*       Only top-level FindMedian() function is exported.                  */
/*                                                                          */
/*       Revision record:                                                   */
/*           03/18/03    File Created                                       */
/*           04/26/03    Adapted for Version 2.0 source code documentation  */
/*                                                                          */
/* (c) Copyright 2003  Hamilton Technical Services  All Rights Reserved     */
/*                                                                          */
/****************************************************************************/

// Include FrequenC header file
#include "frequenc.h"

// Local wrapper function prototypes
int FindMedian1(F_TYPE fData[], BOOL bDataType, F_TYPE *pfMedian);
int FindMedian2(F_TYPE fData[], BOOL bDataType, F_TYPE *pfMedian);

/****************************************************************************/
/*                                                                          */
/*                               FindMedian()                               */
/*                                                                          */
/*       Function to call one of two FindMedian wrapper functions           */
/*                                                                          */
/*       Parameters:     F_TYPE fData     = data array                      */
/*                                          fData[0] = # data points        */
/*                                          fData[1] = analysis start       */
/*                                          fData[2] = analysis end         */
/*                                                                          */
/*                       BOOL bDataType   = data type flag: 0=phase, 1=freq */
/*                                                                          */
/*                       F_TYPE *pfMedian = pointer to median               */
/*                                                                          */
/*       Returns:        int              = 0 if OK or -1 if error          */
/*                                                                          */
/*       Note:           One of two FindMedian wrapper functions is called  */
/*                       depending on size of data array                    */
/*                                                                          */
/*       Revision record:                                                   */
/*           05/07/02    Created                                            */
/*           03/18/03    Adapted for FrequenC.dll                           */
/*           04/26/03    Adapted for Version 2.0 source code documentation  */
/*                                                                          */
/* (c) Copyright 2002-3  Hamilton Technical Services  All Rights Reserved   */
/*                                                                          */
/****************************************************************************/

int __declspec(dllexport) FAR PASCAL FindMedian(F_TYPE fData[],
    BOOL bDataType, F_TYPE *pfMedian)
{
    // Local variables
    int nSize;          // Size of data array
    int nCode;          // Error code
    // Find size of data array
    nSize=(int)(fData[2]-fData[1]+1);

    // Call one of two wrapper functions
    if(nSize<100)
    {
        nCode=FindMedian1(fData, bDataType, pfMedian);
    }
    else // Large array
    {
        nCode=FindMedian2(fData, bDataType, pfMedian);
    }

    // Done
    return nCode;
}

/****************************************************************************/
/*                                                                          */
/*                              FindMedian1()                               */
/*                                                                          */
/*      Wrapper function to safely call FindMed()for small array size       */
/*                                                                          */
/*      Parameters:     F_TYPE fData   = data array                         */
/*                                       fData[0] = # data points           */
/*                                       fData[1] = analysis start          */
/*                                       fData[2] = analysis end            */
/*                                                                          */
/*                      BOOL bDataType = data type flag: 0=phase, 1=freq    */
/*                                                                          */
/*                      F_TYPE *median = pointer to median                  */
/*                                                                          */
/*      Returns:        int            = 0 if OK or -1 if error             */
/*                                                                          */
/*      Note:           Embedded zeros treated as gaps in phase data.       */
/*                      All zeros gaps in frequency data.                   */
/*                      Must have at least 1 data point.                    */
/*                                                                          */
/*      Revision record:                                                    */
/*          09/14/98    Created                                             */
/*          03/18/03    Adapted for FrequenC.dll                            */
/*          04/26/03    Adapted for Version 2.0 source code documentation   */
/*                                                                          */
/* (c) Copyright 1998-2003 Hamilton Technical Services All Rights Reserved  */
/*                                                                          */
/****************************************************************************/

// Prototype for local median function
int FindMed(F_TYPE z[], BOOL datype, F_TYPE *median);

int FindMedian1(F_TYPE fData[], BOOL bDataType, F_TYPE *pfMedian)
{
    // Local variables
    int i;              // General index
    int nNum;           // # data points in input array
    F_TYPE *fTemp;      // Pointer to temporary array to save original data

    // Save data in temporary array
    // Get array size
    nNum=(int)fData[0];
    // Allocate memory
    if((fTemp=(F_TYPE*)malloc((nNum+ARRAY_OFFSET)*sizeof(F_TYPE)))==NULL)
    {
        *pfMedian=0.0;
        return -1; // Error code
    }

    // Save data
    for(i=0; i<nNum+ARRAY_OFFSET; i++)
    {
        fTemp[i]=fData[i];
    }

    // Call FindMed()
    FindMed(fData, bDataType, pfMedian);

    // Restore data
    for(i=0; i<nNum+ARRAY_OFFSET; i++)
    {
        fData[i]=fTemp[i];
    }

    // Free temporary data array
    free(fTemp);

    // Done
    return 0; // OK
}

/*****************************************************************************/
/*                                                                           */
/*                              FindMedian2()                                */
/*                                                                           */
/*      Wrapper function to safely call NR nrselect()                        */
/*      FindMed() gives strictly correct result for # odd or even            */
/*      Faster nrselect() used when # > 100                                  */
/*                                                                           */
/*      Parameters:     F_TYPE fData   = data array                          */
/*                                       fData[0] = # data points            */
/*                                       fData[1] = analysis start           */
/*                                       fData[2] = analysis end             */
/*                      BOOL bDataType = data type flag: 0=phase, 1=freq     */
/*                      F_TYPE *median = pointer to median                   */
/*                                                                           */
/*      Returns:        int            = 0 if OK or -1 if error              */
/*                                                                           */
/*      Note:           Embedded zeros treated as gaps in phase data.        */
/*                      All zeros gaps in frequency data.                    */
/*                      Must have at least 1 data point.                     */
/*                                                                           */
/*                      This function best used for large N, say N>1OO       */
/*                      It handles analysis limits & gaps.  It is X10        */
/*                      faster than full sort for large N.                   */
/*                                                                           */
/*      Revision record:                                                     */
/*          05/06/02    Created and working                                  */
/*          03/18/03    Adapted for FrequenC.dll                             */
/*          04/26/03    Adapted for Version 2.0 source code documentation    */
/*                                                                           */
/* (c) Copyright 2002-3 Hamilton Technical Services All Rights Reserved      */
/*                                                                           */
/*****************************************************************************/

// Prototype for NR select() function used in median calc
float nrselect(unsigned long k, unsigned long n, float arr[]);

int FindMedian2(F_TYPE fData[], BOOL bDataType, F_TYPE *pfMedian)
{
    // Local variables
    int i;              // General index
    int k;              // Secondary index
    int nStart;         // Analysis start point
    int nNum;           // # analysis points
    int nGaps=0;        // # gaps found
    float *fTemp;       // Pointer to temporary array for median calc

    // Copy analysis data into temporary working array using indices 1 to nNum
    // Get start # and # analysis points
    nStart=(int)fData[1];
    nNum=(int)(fData[2]-fData[1]+1);

    // Allocate memory for analysis data only - no header
    // The +1 allows the array index to start at 1 per NR style
    if((fTemp=(float*)malloc((nNum+1)*sizeof(float)))==NULL)
    {
        *pfMedian=0.0;
        return -1; // Error code
    }

    // Copy analysis data into temporary float array
    k=1;
    for(i=1; i<=nNum; i++)
    {
        if(bDataType==FREQ_DATA)
        {
            // All zeros are gaps to be omitted
            if((fTemp[k]=(float)fData[ARRAY_OFFSET+nStart+i-2])==0)
            {
                nGaps++;
            }
            else
            {
                k++;
            }
        }
        else // Phase data
        {
            // All except 1st & last zeros are gaps to be omitted
            if((fTemp[k]=(float)fData[ARRAY_OFFSET+nStart+i-2])==0)
            {
                if( (k=1) || (k=nNum) )
                {
                    nGaps++;
                }
                else
                {
                    k++;
                }
            }
            else
            {
                k++;
            }
        }
    }

    // Adjust # points
    nNum-=nGaps;

    // Call NR function to find median
    // 1st arg is kth smallest value to find = (N+1)/2
    // This is exactly correct median for N odd
    // For N even, strictly speaking, median S/B avg of N/2 and N/2 +1 points
    // We aren't going to worry about that, esp for large N
    *pfMedian=(F_TYPE)nrselect((unsigned long)((nNum+1)/2), (unsigned long)nNum,
        fTemp);

    // Free temporary data array
    free(fTemp);

    // Done
    return 0; // OK
}

/******************************************************************************/
/*                                                                            */
/*                              FindMed()                                     */
/*                                                                            */
/*      Function to find median of array of phase or frequency data           */
/*                                                                            */
/*      Parameters:     F_TYPE z[]    = phase or frequency data               */
/*                                      z[0] = # data points                  */
/*                                      z[1] = analysis start                 */
/*                                      z[2] = analysis end                   */
/*                      BOOL   datype = data type: 0=phase, 1=frequency       */
/*                      F_TYPE *median= pointer to median value of data       */
/*                                                                            */
/*      Notes:          All zeros treated as gaps in frequency data.          */
/*                      Embedded zeros treated as gaps in phase data.         */
/*                      This function first fills gaps in the data            */
/*                      with interpolated values by calling fill().           */
/*                      It then sorts the data into assending order,          */
/*                      using qsort(), to find the median value.              */
/*                      The median is the center value for an odd # of        */
/*                      data points, and the average of the two center        */
/*                      values for an even # of data points.                  */
/*                                                                            */
/*      Return:         int           = error flag: 0=OK, 1=NG                */
/*                                                                            */
/*      Revision record:                                                      */
/*          05/03/97    Adapted from FindMedian() of STABLE_A.C               */
/*          01/01/98    Changes for MS VC++ compatibility & warnings          */
/*          03/18/03    Adapted for FrequenC.dll                              */
/*          04/26/03    Adapted for Version 2.0 source code documentation     */
/*                                                                            */
/* (c) Copyright 1992-2003 Hamilton Technical Services All Rights Reserved    */
/*                                                                            */
/******************************************************************************/
```

```
// Declare comparison function - not exported */
static int median_compare(const void *a, const void *b);

int FindMed(F_TYPE z[], BOOL datype, F_TYPE *median)
{
    int i;                                                /* 1st index */
    int j=0;                                              /* 2nd index */
    int start=(int)z[1];                             /* analysis start */
    int end=(int)z[2];                                 /* analysis end */
    int num=(int)z[0];                                /* # data points */
    F_TYPE *temp;                /* pointer to temporary data storage array */
    F_TYPE med;                                        /* median value */

    // save data
    if((temp=(F_TYPE *)malloc((num+3)*sizeof(F_TYPE)))==NULL)
                                    /* allocate temporary data storage space */
    {
        return(1);                                     /* error return */
    }                                                        /* end if */

    for(i=0; i<num+3; i++)
    {
        temp[i]=z[i];                                     /* save data */
    }                                                       /* end for */

    /* Fill any gaps in data */
    FillGaps(z, datype);

    /* eliminate non-analysis points */
    for(i=start+2; i<=end+2; i++)
    {
        z[j]=z[i];
        j++;
    }                                                       /* end for */
    num=end-start+1;

    /* sort data into numerical order */
    qsort((void *) z, (size_t) num, sizeof(F_TYPE), median_compare);

    /* find & return median */
    if(num%2)                                          /* is num odd? */
    {
        med=z[num/2];                                  /* middle value */
    }                                                        /* end if */
    else
    {                                                     /* n is even */
        med=(z[(num/2)-1]+z[num/2])/2.0;       /* average of 2 middle values */
    }                                                      /* end else */

    // restore data
    for(i=0; i<num+3; i++)
    {
        z[i]=temp[i];                                  /* restore data */
    }                                                       /* end for */
    free(temp);                                    /* deallocate temp[] */

    *median=med;                     /* set median to med = median of data */
    return(0);                                        /* normal return */

}                                                /* end FindMedian() */
```

```
/*****************************************************************************/
/*                                                                           */
/*                            median_compare()                               */
/*                                                                           */
/*      Qsort compare function for finding median of array                   */
/*                                                                           */
/*****************************************************************************/

static int median_compare(const void *a, const void *b)
{
    F_TYPE x =  (*(F_TYPE *) a) - (*(F_TYPE *) b) ;
    return (x>0) ? 1 : (x<0) ? -1 : 0;
}

/*****************************************************************************/
/*                                                                           */
/*                               nrselect()                                  */
/*                                                                           */
/*      Fast Numerical Recipes function to find median of array              */
/*                                                                           */
/*****************************************************************************/

// Copied from on-line NR, Sec. 8.5 (later version than I have in book)
// Note:  The NR working array arr[] indices go from 1 to n; arr[0] not used
// Note:  The nrselect() array is just a float, but this precision S/B OK

// Local macro
#define SWAPP(a,b) temp=(a);(a)=(b);(b)=temp;

// Start of function - prototype is at top of this file - not in Stable_w.h
// This function returns the kth smallest value in the n-point array arr[]
// k is set to (n+1)/2 to find median
// About x10 faster than sorting - smaller elements moved in arbitrary order
float nrselect(unsigned long k, unsigned long n, float arr[])
{
    unsigned long i,ir,j,l,mid;
    float a,temp;

    l=1;
    ir=n;
    for(;;) {
        if (ir <= l+1) {
            if (ir == l+1 && arr[ir] < arr[l]) {
                SWAPP(arr[l],arr[ir])
            }
            return arr[k];
        } else {
            mid=(l+ir) >> 1;
            SWAPP(arr[mid],arr[l+1])
            if (arr[l] > arr[ir]) {
                SWAPP(arr[l],arr[ir])
            }
            if (arr[l+1] > arr[ir]) {
                SWAPP(arr[l+1],arr[ir])
            }
            if (arr[l] > arr[l+1]) {
                SWAPP(arr[l],arr[l+1])
            }
            i=l+1;
            j=ir;

            a=arr[l+1];
            for (;;) {
                do i++; while (arr[i] < a);
                do j--; while (arr[j] > a);
                if (j < i) break;
                SWAPP(arr[i],arr[j])
            }
            arr[l+1]=arr[j];
            arr[j]=a;
             if (j >=k) ir=j-1;
             if (j <=k) l=i;
        }
    }
}
```

The FrequenC Library

NAME: FindMinMax	**FUNCTION:** Find the minimun and maximum values of phase or frequency data

SYNOPSIS:
```
int FindMinMax(F_TYPE z[], F_TYPE *p_min, F_TYPE *p_max,BOOL datype)
```

F_TYPE z[]	Phase or frequency data array:
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysis end point

F_TYPE *p_min	Pointer to minimum
F_TYPE *p_max	Pointer to maximum
BOOL datype	Type of data: 0=phase, 1=frequency
RETURN: int	The # of analysis points (may be 0 if no non-gap data points)

REMARKS:
Only data between start and end analysis limits are analyzed.
All zeros are treated as gaps in frequency data.
Only embedded gaps are treated as gaps in phase data.
The datype flag determines whether the data is treated as phase (datype=0) or frequency (datype=1) data.

EXAMPLE:
```
#include "frequenc.h"                           /* FrequenC header file */
F_TYPE z[512];                                            /* data array */
F_TYPE min;                                            /* minimum value */
F_TYPE max;                                            /* maximum value */
BOOL datype;                                          /* data type flag */
int num;                                           /* # analysis points */
.
.
datype=FREQ;                             /* set flag for frequency data */
         /* frequenc.h contains #define PHASE 0 and #define FREQ 1 */
num=FindMinMax(z, &min, &max, datype);                /* find min & max */
printf("\nMin = %e", min);                            * display minimum */
printf("\nMax = %e", max);                           /* display maximum */
printf("\n# Points = %d, num);        * display # analysis points */
```

SEE ALSO: FindPlotScale()

REFERENCE: None

```
/***************************************************************************/
/*                                                                         */
/*                            FindMinMax                                   */
/*                                                                         */
/*      Function to find min and max of phase or frequency data            */
/*                                                                         */
/*      Parameters:     F_TYPE z[]    = frequency data                     */
/*                                      z[0] = # data points               */
/*                                      z[1] = analysis start              */
/*                                      z[2] = analysis end                */
/*                      F_TYPE *p_min = pointer to minimum data value      */
/*                      F_TYPE *p_max = pointer to maximum data value      */
/*                      BOOL   datype = data type: 0=phase, 1=frequency    */
/*                                                                         */
/*      Notes:          All zero treated as gaps in frequency data.        */
/*                      Embedded zeros treated as gaps in phase data.      */
/*                      min & max set to 0 if no non-gap data points.      */
/*                                                                         */
/*      Return:         int            = # non-gap data points             */
/*                                                                         */
/*      Revision record:                                                   */
/*          12/16/91    Created                                            */
/*          12/26/91    Changed to make datype a parameter rather than     */
/*                      a global variable.  Made function an int return-   */
/*                      ing # non-gap data points.  y[] changed to z[].    */
/*          12/31/91    Renamed; min & max arguments reversed              */
/*          01/18/92    Edited title block                                 */
/*          01/19/92    Changed to min=max=0 if no non-gap data points     */
/*          02/20/92    Rewritten to eliminate use of MAXFLOAT             */
/*          02/03/96    Modified for use as Win 3.1 DLL                    */
/*          01/01/98    Changes for MS VC++ compatibility & warnings       */
/*                                                                         */
/* (c) Copyright 1991-8  Hamilton Technical Services  All Rights Reserved  */
/*                                                                         */
/***************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL FindMinMax(F_TYPE z[], F_TYPE *p_min,
    F_TYPE *p_max, BOOL datype)
{
    int i;                                                     /* index */
    int num=0;                                  /* # non-gap data points */
    BOOL flag=FALSE;           /* flag to indicate 1st non-gap point found */

    *p_max = 0;                             /* initialize max & min to 0 */
    *p_min = 0;            /* they will remain 0 if no non-gap data points */

    for( i = (int)(*(z+1)+2); i <= (int)(*(z+2)+2); i++)
    {
        if(*(z + i) || (!datype && i==3) || (!datype && i==z[0]+2))
        {
            if(flag)
            {
                if( *(z + i) > *p_max )
                {
                    *p_max = *(z + i);
                }                                             /* end if */
                if( *(z + i) < *p_min )
                {
                    *p_min = *(z + i);
                }                                           /* end if */
            }                                               /* end if */
            else                            /* lst non-gap data point */
            {
                *p_max = *(z + i);
                *p_min = *(z + i);
                flag = TRUE;
            }                                             /* end else */
            num++;
        }                                                   /* end if */
    }                                                      /* end for */

    return(num);
}                                              /* end FindMinMax() */
```

The FrequenC Library

NAME: FindPlotScale	**FUNCTION:** Find plot scale for phase or frequency data

SYNOPSIS:
```
int FindPlotScale(F_TYPE min, F_TYPE max, F_TYPE *p_scale_min,
                  F_TYPE *p_scale_max)
```

F_TYPE min	Data minimum
F_TYPE max	Data maximum
F_TYPE *p_scale_min	Pointer to scale minimum
F_TYPE *p_scale_max	Pointer to scale maximum
RETURN: int	The # of scale tick intervals, or -1 if error

REMARKS:
Error if min > max

EXAMPLE:
```
#include "frequenc.h"                      /* FrequenC header file */
F_TYPE y[512];                             /* frequency data array */
F_TYPE min;                                       /* minimum value */
F_TYPE max;                                       /* maximum value */
F_TYPE scale_min;                                     /* scale min */
F_TYPE scale_max;                                     /* scale max */
int ticks;                                      /* # scale ticks */
.
.
FindMinMax(y, &min, &max, FREQ);        /* find min & max for data */
ticks=FindPlotScale(min, max, &scale_min, &scale_max);     * scale */
if(ticks==-1)
{
    printf("\nError");                            /* error message */
}
else
{
    printf("\nScale Min = %e", scale_min);    /* display scale min */
    printf("\nScale Max = %e", scale_max);     * display scale max */
    printf("\n# Tick Intervals = %d, ticks);    /* display # ticks */
}
```

SEE ALSO: FindMinMax()

REFERENCE: None

```
/***************************************************************************/
/*                                                                         */
/*                              FindPlotScale()                            */
/*                                                                         */
/*      Function to find scale factor for plotting phase or freq data      */
/*                                                                         */
/*      Parameters:     F_TYPE min          = minimum freq value           */
/*                      F_TYPE max          = maximum freq value           */
/*                      F_TYPE *p_scale_min = pointer to scale minimum     */
/*                      F_TYPE *p_scale_max = pointer to scale maximum     */
/*                                                                         */
/*      Return:         int = # tick intervals in scale,                   */
/*                            or -1 if error                               */
/*                                                                         */
/*      Revision record:                                                   */
/*          12/16/91    Created                                            */
/*          12/26/91    Changed to int returning # decades in scale        */
/*          12/31/91    Renamed                                            */
/*          01/04/91    Changed return value to # ticks                    */
/*                      Added 4 and 8 tick choices                         */
/*          01/05/92    Added range check & error code return if range NG  */
/*          01/15/92    Added 6 tick choice; changed to else if style      */
/*                      Changed to tighter range check logic at end        */
/*          01/16/92    (int) casts made floor() or ceil() to avoid O/F    */
/*                      Added e in fine tests; added 3 tick choice         */
/*                      Additional changes to range check/correction logic */
/*          01/18/92    Edited title block                                 */
/*          01/22/92    Changed order of min and max parameter pairs       */
/*          02/17/92    Changed min & max params from pointers to values   */
/*          02/03/96    Modified for use as Win 3.1 DLL                    */
/*          01/01/98    Changes for MS VC++ compatibility & warnings       */
/*                                                                         */
/* (c) Copyright 1991-8  Hamilton Technical Services  All Rights Reserved  */
/*                                                                         */
/***************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL FindPlotScale(F_TYPE min, F_TYPE max,
    F_TYPE *p_scale_min, F_TYPE *p_scale_max)
{
    double range, log_range, coarse, fine, ticks;
    double full_scale, digits, e;
    int k;                                                /* # ticks added */

    e=1e-6;                                             /* error tolerance */
    range = max - min;

    if(range<=0)                                     /* check for NG range */
    {
        *p_scale_min=0;
        *p_scale_max=0;
        return(-1);                                          /* error code */
    }

    log_range = log10(range) + 100;
    coarse = floor(log_range);
    fine = log_range - coarse;
```

```
if(fine < log10(2)+e)
{
    ticks = 2;
}                                                          /* end if */
else if(fine < log10(3)+e)
{
    ticks = 3;
}                                                          /* end if */
else if(fine < log10(4)+e)
{
    ticks = 4;
}                                                          /* end if */
else if(fine < log10(5)+e)
{
    ticks = 5;
}                                                          /* end if */
else if(fine < log10(6)+e)
{
    ticks = 6;
}                                                          /* end if */
else if(fine < log10(8)+e)
{
    ticks = 8;
}                                                          /* end if */
else
{
    ticks = 10;
}                                                        /* end else */

full_scale = pow(10, coarse - 100) * ticks;
if(log10(full_scale) >= 0)
{
    digits = floor(log10(full_scale));
}                                                          /* end if */
else
{
    digits = ceil(log10(full_scale)) - 1;
}                                                        /* end else */

if((min / pow(10, digits)) >= 0)
{
    *p_scale_min = floor(min / pow(10, digits)) * pow(10, digits);
}                                                          /* end if */
else
{
    *p_scale_min = (floor(min / pow(10, digits))-1) * pow(10, digits);
}                                                        /* end else */
*p_scale_max = *p_scale_min + full_scale;

k=0;

if(*p_scale_max < max)                /* check for scale max too small */
{
    while(*p_scale_max < max)                        /* move scale up */
    {
        *p_scale_max += full_scale/ticks;
        *p_scale_min += full_scale/ticks;
    }
```

```
        if(*p_scale_min > min)
        {
            while(*p_scale_min > min)           /* add tic interval to max */
            {
                *p_scale_min -= full_scale/ticks;
                k++;
            }
        }
    }

    else if(*p_scale_min > min)           /* check for scale min too large */
    {
        while(*p_scale_min > min)                       /* move scale down */
        {
            *p_scale_max -= full_scale/ticks;
            *p_scale_min -= full_scale/ticks;
        }
        if(*p_scale_max < max)                  /* add tic interval to min */
        {
            while( *p_scale_max < max)
            {
                *p_scale_max += full_scale/ticks;
                k++;
            }
        }
    }

    ticks+=k;                                  /* adjust # tic intervals */

    return(int)ticks;
}                                                 /* end FindPlotScale() */
```

The FrequenC Library

NAME: HistoCalc	**FUNCTION:** Calculate histogram from phase or frequency data

SYNOPSIS:
```
void HistoCalc(F_TYPE z[], float bins[], float hist[], F_TYPE *max,
int num, BOOL datype)
```

F_TYPE z[]	Phase or frequency data array
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysis end point

float bins[]	Array of histogram bin values
float hist[]	Array of histogram count values
F_TYPE *max	Maximum z[] value
int num	# histogram bins
BOOL datype	Date type (0=phase, 1=frequency)
RETURN: void	

REMARKS:
Only data points between start and end analysis limits are analyzed.
Embedded zeros are treated as gaps in phase data.
All zeros are treated as gaps in frequency data.

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
float *bins, hist;                          */ pointers to float arrays */
F_TYPE z[512];                                          /* data array */
F_TYPE max;                                           /* data maximum */
int num=50;                                           /* # histo bins */
.
/* allocate bins[num] and hist[num] arrays */
.
HistoCalc(z, bins, hist, &max, num, 1);              /* call function */
printf("\n# Histo Points = %d", num);                  /* display num */
```

SEE ALSO:

REFERENCE:

```
/*****************************************************************************/
/*                                                                           */
/*                            HistoCalc()                                    */
/*                                                                           */
/*      Function to calculate histogram from frequency data                  */
/*      Parameters:                                                          */
/*          y[]         = analysis data;         y[0] = # data points        */
/*                        y[1] = analysis start; y[2] = analysis end         */
/*          bins[]      = histogram bin values                               */
/*          hist[]      = histogram counts                                   */
/*          y_max       = maximum y[] value                                  */
/*          num_bins    = # histogram bins                                   */
/*          bDataType   = data type (0=phase, 1=frequency)                   */
/*                                                                           */
/*      Returns nothing (void)                                               */
/*                                                                           */
/*      Revision record:                                                     */
/*          12/28/96    Adapted from Stable/DOS histo_calc() of STAB_F15.C   */
/*          12/29/96    Up and running                                       */
/*          12/30/96    Added gap handling (Histo never had that!)           */
/*          03/20/03    Adapted for FrequenC DLL                             */
/*                      Added BOOL bDataType argument                        */
/*          04/26/03    Adapted for Version 2.0 source code documentation    */
/*                                                                           */
/* (c) Copyright 1996-2003 Hamilton Technical Services  All Rights Reserved  */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

void __declspec(dllexport) FAR PASCAL HistoCalc(F_TYPE y[], float bins[],
    float hist[], F_TYPE *y_max, int num_bins, BOOL bDataType)
{
    // Local variables
    int i;                                                   /* general index */
    int j;                                                       /* bin index */
    int points=(int)(y[2]-y[1]+1);                       /* # analysis points */
    int offset=(int)(y[1]+2);                  /* index of 1st analysis point */

    // Calc histogram of y[] values from [offset] to [points]
    // Using bins[] from [0] to [num_bins]
    // Counts of # data values/bin go into histogrm[] from [0] to [num_bins-1]
    // y_max keeps track of the largest bin count for the normal curve fit

    *y_max=0.0;

    for(i=offset; i<offset+points; i++)
    {
        if( y[i] || (bDataType==PHASE_DATA && i==3) ||
                    (bDataType==PHASE_DATA && i==y[0]+2) )
        {
            for(j=0; j<num_bins; j++)
            {
                if(y[i]<=bins[j+1])
                {
                    hist[j]++;
                    if(hist[j]>*y_max)
                    {
                        *y_max=hist[j];
                    }
```

```
                        break;
                    }
                }
            }
        }
    }
```

The FrequenC Library	
NAME: HadamardEDF	**FUNCTION:** Description of what function does
SYNOPSIS: int FunctionName(F_TYPE z[], F_TYPE one, F_TYPE two)	

F_TYPE z[]	Phase or frequency data array
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysios end point

F_TYPE one	parameter one, and description of what it is
F_TYPE two	parameter two, and description of what it is
RETURN: int	# non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are modified.
Embedded zeros are treated as gaps in phase data.
All zeros are treated as gaps in frequency data.

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE z[512];                                           /* data array */
F_TYPE one;                                           /* parameter one */
F_TYPE two;                                           /* parameter two */
int num;                                  /* # data points processed */
.
.
one=1.0;                                  /* set 1st parameter value */
two=2.0;                                  /* set 2nd parameter value */
num=FunctionName(z, one, two);                        /* call function */
printf("\n# Data Points Scaled = %d", num);             /* display num */
```

SEE ALSO: RelatedFunctionName()

REFERENCE: NIST Technical Note 1337

```
/****************************************************************************/
/*                                                                          */
/*                              HadamardEDF()                               */
/*                                                                          */
/*  Function to calc the estimated # of chi-squared degrees of freedom     */
/*  for the overlapping Hadamard variance of a power law noise process.    */
/*                                                                          */
/*      Parameters:    int N     = # phase data points                      */
/*                     int m     = averaging factor                         */
/*                     int b     = beta (-2 to -6)                          */
/*                                                                          */
/*      Return:        float     = Hadamard edf                             */
/*                                 or -1 if error                           */
/*                                                                          */
/*      Reference:     C. Greenhall, "Hadamard edf Algorithm", May 1999    */
/*                     (private communication via e-mail)                   */
/*                                                                          */
/*      Revision Record:                                                    */
/*          05/15/99   Created from hedf3.c per C. Greenhall suggestions   */
/*                                                                          */
/* (c) Copyright 1999 Hamilton Technical Services  All Rights Reserved     */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

// Local function prototypes - not exported
double Rx(double t, int b);
double Log(double x);

float __declspec(dllexport) FAR PASCAL HadamardEDF(int N, int m, int b)
{
    // Local macros
    #define JMAX 100                    // Max # terms to calc as sum
    #define SUM 0                       // Calc using sum
    #define BOUND 1                     // Calc using upper bound
    #define SMALL 2                     // Calc using smaller sum

    // Local variables
    int M=N-3*m;                        // M=N-3m
    int j;                              // Index
    int jmax;                           // Max j for summation
    int m_prime;                        // Smaller sum m
    int c;                              // Calculation type (SUM, BOUND or SMALL)
    double a0;                          // edf upper bound 1st term
    double a1;                          // edf upper bound 2nd term
    double edfinv;                      // Inverse of Hadamard edf
    double r=0;                         // r(t)
    double r0;                          // r(0)
    double e;                           // edf summation term
    double t;                           // Argument of r(t)
    double p;                           // M/m

    // Check for allowed beta
    if(b<-6 || b>-2)
    {
        return -1;                      // Error code
    }
```

```
// Check for enough points: N must be >3m
if(M<=0)
{
    return -1;                      // Error code
}

// Find jmax
jmax=min(M, 3*m);

switch(b)                           // Different calc for each noise type
{
    case -2:                        // White FM
    {
        // Set r0
        r0=12;

        // Test jmax
        if(jmax<JMAX)               // Calc edf as sum
        {
            // Set calc type
            c=SUM;

            // Find sum
            for(j=1; j<=jmax; j++)
            {
                t=(double)j/(double)m;
                e=Rx(t, b);
                r+=e*e*(1-((double)j/(double)M));
            }
        }
        else                        // Too many terms
        {
            // Test M
            if(M>=3*m)              // Calc edf as upper bound
            {
                // Set calc type
                c=BOUND;

                // Set coefficients
                a0=0.777778;  // 7/9
                a1=0.5;
            }
            else                    // Use smaller sum
            {
                // Set calc type
                c=SMALL;

                // Find nearest integer to JMAX/p
                p=(double)M/(double)m;
                m_prime=(int)(((double)JMAX/p)+0.5);

                // Calc sum using this m
                jmax=min(JMAX, 3*m_prime);
                // Find smaller sum
                for(j=1; j<=jmax; j++)
                {
                    t=(double)j/(double)m_prime;
                    e=Rx(t, b);
                    r+=e*e*(1-((double)j/(double)JMAX));
                }
            }
        }
    }
    break;

    case -3:                    // Flicker FM
    {
        // Set r0
        r0=13.49604;            // 48*log(2)-18*log(3)

        // Test jmax
        if(jmax<JMAX)           // Calc edf as sum
        {
            // Set calc type
            c=SUM;

            // Find sum
            for(j=1; j<=jmax; j++)
            {
                t=(double)j/(double)m;
                e=Rx(t, b);
                r+=e*e*(1-((double)j/(double)M));
            }
        }
        else                    // Too many terms
        {
            // Test M
            if(M>=3*m)          // Calc edf as upper bound
            {
                // Set calc type
                c=BOUND;

                // Set coefficients
                a0=1.0;
                a1=0.62;
            }
            else                // Use smaller sum
            {
                c=SMALL;

                // Find nearest integer to JMAX/p
                p=(double)M/(double)m;
                m_prime=(int)(((double)JMAX/p)+0.5);

                // Calc sum using this m
                jmax=min(JMAX, 3*m_prime);
                // Find smaller sum
                for(j=1; j<=jmax; j++)
                {
                    t=(double)j/(double)m_prime;
                    e=Rx(t, b);
                    r+=e*e*(1-((double)j/(double)JMAX));
                }
            }
        }
    }
    break;

    case -4:                    // Random Walk FM
    {
        // Set r0
        r0=12;

        // Test jmax
        if(jmax<JMAX)           // Calc edf as sum
        {
            // Set calc type
            c=SUM;

            // Find sum
            for(j=1; j<=jmax; j++)
            {
                t=(double)j/(double)m;
                e=Rx(t, b);
                r+=e*e*(1-((double)j/(double)M));
            }
        }
        else                    // Too many terms
        {
            // Test M
            if(M>=3*m)          // Calc edf as upper bound
            {
                // Set calc type
                c=BOUND;

                // Set coefficients
                a0=1.033334;  // 31/30
                a1=0.607143;  // 17/28
            }
            else                // Use smaller sum
            {
                // Set calc type
                c=SMALL;

                // Find nearest integer to JMAX/p
                p=(double)M/(double)m;
                m_prime=(int)(((double)JMAX/p)+0.5);

                // Calc sum using this m
                jmax=min(JMAX, 3*m_prime);
                // Find smaller sum
                for(j=1; j<=jmax; j++)
                {
                    t=(double)j/(double)m_prime;
                    e=Rx(t, b);
                    r+=e*e*(1-((double)j/(double)JMAX));
                }
            }
        }
    }
    break;

    case -5:                    // Flicker Walk FM
    {
        // Set r0
        r0=44.89093;            // -192*log(2)+162*log(3)

        // Test jmax
        if(jmax<JMAX)           // Calc edf as sum
        {
            // Set calc type
            c=SUM;

            // Find sum
            for(j=1; j<=jmax; j++)
            {
                t=(double)j/(double)m;
                e=Rx(t, b);
                r+=e*e*(1-((double)j/(double)M));
            }
        }
        else                    // Too many terms
        {
            // Test M
            if(M>=3*m)          // Calc edf as upper bound
            {
                // Set calc type
                c=BOUND;

                // Set coefficients
                a0=1.06;
                a1=0.53;
            }
            else                // Use smaller sum
            {
                // Set calc type
                c=SMALL;

                // Find nearest integer to JMAX/p
                p=(double)M/(double)m;
                m_prime=(int)(((double)JMAX/p)+0.5);

                // Calc sum using this m
                jmax=min(JMAX, 3*m_prime);
```

```
                // Find smaller sum
                for(j=1; j<=jmax; j++)
                {
                    t=(double)j/(double)m_prime;
                    e=Rx(t, b);
                    r+=e*e*(1-((double)j/(double)JMAX));
                }
            }
        }
    }
    break;

    case -6:                    // Random Run FM
    {
        // Set r0
        r0=132;

        // Test jmax
        if(jmax<JMAX)           // Calc edf as sum
        {
            // Set calc type
            c=SUM;

            // Find sum
            for(j=1; j<=jmax; j++)
            {
                t=(double)j/(double)m;
                e=Rx(t, b);
                r+=e*e*(1-((double)j/(double)M));
            }
        }
        else                    // Too many terms
        {
            // Test M
            if(M>=3*m)          // Calc edf as upper bound
            {
                // Set calc type
                c=BOUND;

                // Set coefficients
                a0=1.30;
                a1=0.54;
            }
            else                // Use smaller sum
            {
                // Set calc type
                c=SMALL;

                // Find nearest integer to JMAX/p
                p=(double)M/(double)m;
                m_prime=(int)(((double)JMAX/p)+0.5);

                // Calc sum using this m
                jmax=min(JMAX, 3*m_prime);
```

```
                    // Find smaller sum
                    for(j=1; j<=jmax; j++)
                    {
                        t=(double)j/(double)m_prime;
                        e=Rx(t, b);
                        r+=e*e*(1-((double)j/(double)JMAX));
                    }
                }
            }
        }
        break;
    }

    switch(c)                       // Different way for each calc type
    {
        case SUM:                   // Calc using sum
        {
            r*=2.0;
            r+=r0*r0;
            edfinv=r/(M*r0*r0);
        }
        break;

        case BOUND:                 // Calc using upper bound
        {
            p=(double)M/(double)m;
            edfinv=(1/p)*(a0-(a1/p));
        }
        break;

        case SMALL:                 // Calc using smaller sum
        {
            r*=2.0;
            r+=r0*r0;
            edfinv=r/(JMAX*r0*r0);
        }
        break;
    }

    return (float)(1/edfinv);       // Return Hadamard edf
}

/*****************************************************************************/
/*                                                                         */
/*                                Rx()                                     */
/*                                                                         */
/*  Function to calculate Rx(t,b) for Hamamard edf                         */
/*                                                                         */
/*      Parameters:    double t  = time (>0)                               */
/*                     int b     = beta (-2 to -6)                         */
/*                                                                         */
/*      Return:        double    = Rx(t,b)                                 */
/*                                                                         */
/*      Revision Record:                                                   */
/*          05/15/99   Created                                             */
/*                                                                         */
/* (c) Copyright 1999 Hamilton Technical Services  All Rights Reserved     */
/*                                                                         */
/*****************************************************************************/
```

```
double Rx(double t, int b)
{
    switch(b)       // Different expression for each noise type
    {
        case -2:    // White FM: Rx(t)=-|t|
        {
            return -20*t+15*(t+1)+15*fabs(t-1)-6*(t+2)-6*fabs(t-2)+
                (t+3)+fabs(t-3);
        }

        case -3:    // Flicker FM: Rx(t)=t*t*ln(|t|)
        {
            return 20*t*t*log(t)-15*(t+1)*(t+1)*Log(t+1)-
                15*(t-1)*(t-1)*Log(fabs((t-1)))+
                6*(t+2)*(t+2)*Log((t+2))+
                6*(t-2)*(t-2)*Log(fabs((t-2)))-
                (t+3)*(t+3)*Log((t+3))-(t-3)*(t-3)*Log(fabs((t-3)));
        }

        case -4:    // Random Walk FM: Rx(t)=-|t|^3
        {
            return 20*t*t*t-
                15*(t+1)*(t+1)*(t+1)-
                15*fabs((t-1)*(t-1)*(t-1))+
                6*(t+2)*(t+2)*(t+2)+
                6*fabs((t-2)*(t-2)*(t-2))-
                (t+3)*(t+3)*(t+3)-
                fabs((t-3)*(t-3)*(t-3));
        }

        case -5:    // Flicker Walk FM: Rx(t)=-t*t*t*t*ln(|t|)
        {
            return -20*t*t*t*t*log((t))+
                15*(t+1)*(t+1)*(t+1)*(t+1)*Log((t+1))+
                15*(t-1)*(t-1)*(t-1)*(t-1)*Log(fabs((t-1)))-
                6*(t+2)*(t+2)*(t+2)*(t+2)*Log((t+2))-
                6*(t-2)*(t-2)*(t-2)*(t-2)*Log(fabs((t-2)))+
                (t+3)*(t+3)*(t+3)*(t+3)*Log((t+3))+
                (t-3)*(t-3)*(t-3)*(t-3)*Log(fabs((t-3)));
        }

        case -6:    // Random Run FM: Rx(t)=-|t|^5
        {
            return -20*t*t*t*t*t+
                15*(t+1)*(t+1)*(t+1)*(t+1)*(t+1)+
                15*fabs((t-1)*(t-1)*(t-1)*(t-1)*(t-1))-
                6*(t+2)*(t+2)*(t+2)*(t+2)*(t+2)-
                6*fabs((t-2)*(t-2)*(t-2)*(t-2)*(t-2))+
                (t+3)*(t+3)*(t+3)*(t+3)*(t+3)+
                fabs((t-3)*(t-3)*(t-3)*(t-3)*(t-3));
        }
    }
}
```

```
/****************************************************************************/
/*                                                                          */
/*                                Log()                                     */
/*                                                                          */
/*  Wrapper function for log(x) modified to return 0 for x=0                */
/*                                                                          */
/*      Parameters:    double x  = argument for log(x)                      */
/*                                                                          */
/*      Return:        double    = log(x)                                   */
/*                                 or 0 if x=0                              */
/*                                                                          */
/*      Note: x can be 0 but must not be negative                           */
/*                                                                          */
/*      Revision Record:                                                    */
/*          05/10/99   Created                                              */
/*                                                                          */
/* (c) Copyright 1999 Hamilton Technical Services  All Rights Reserved     */
/*                                                                          */
/****************************************************************************/

double Log(double x)
{
    if(x)
    {
        return log(x);
    }
    else
    {
        return 0;
    }
}
```

The FrequenC Library	
NAME: HadTotvarBias	**FUNCTION:** Description of what function does
SYNOPSIS: `int FunctionName( F_TYPE z[], F_TYPE one, F_TYPE two)`	

F_TYPE z[]	Phase or frequency data array
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysios end point

F_TYPE one	parameter one, and description of what it is
F_TYPE two	parameter two, and description of what it is
RETURN: int	# non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are modified.
Embedded zeros are treated as gaps in phase data.
All zeros are treated as gaps in frequency data.

EXAMPLE:
```
#include "frequenc.h"                           /* FrequenC header file */
F_TYPE z[512];                                            /* data array */
F_TYPE one;                                            /* parameter one */
F_TYPE two;                                            /* parameter two */
int num;                                   /* # data points processed */
.
.
one=1.0;                                   /* set 1st parameter value */
two=2.0;                                   /* set 2nd parameter value */
num=FunctionName(z, one, two);                         /* call function */
printf("\n# Data Points Scaled = %d", num);              /* display num */
```

SEE ALSO: RelatedFunctionName()

REFERENCE: NIST Technical Note 1337

```
/*****************************************************************************/
/*                                                                           */
/*                              HadTotvarBias()                              */
/*                                                                           */
/*      Function to calculate the Hadamard Totvar bias function              */
/*                                                                           */
/*      Parameters:     int   nAlpha = noise type (-4 to 0)                  */
/*                                                                           */
/*                                                                           */
/*      Return:         float fBias  = Mod Totvar bias factor                */
/*                                     or -1 if error                        */
/*                                                                           */
/*      Note:  fBias is independent of T and tau, and depends only on the    */
/*             the noise type.  It is defined as a=[E{TotHvar}/E{Hvar}]-1,   */
/*             but is calculated and used here as simply the ratio.          */
/*             This fBias is exactly the same as that for MTOT, but a        */
/*             separate function is used just in case that should change,    */
/*             and because of the diffeent alphas that apply.                */
/*                                                                           */
/*      Alpha     Noise        a    HTOT/HVAR = 1+a                          */
/*        0         W FM     -0.005      0.995                                */
/*       -1         F FM     -0.149      0.851                                */
/*       -2        RW FM     -0.229      0.771                                */
/*       -3        FW FM     -0.283      0.717                                */
/*       -4        RR FM     -0.321      0.679                                */
/*                                                                           */
/*      The Hadamard Totvar bias factor is the ratio of the expected value   */
/*      HTOT to HVAR.  To correct a calculated Hadamard Totdev value for a   */
/*      certain noise type, divide the uncorrected Mod Totdev value by the   */
/*      square root of the bias factor returned by this function.            */
/*                                                                           */
/*      Notes:          (1) This function applies to -4 >= alpha >= 0        */
/*                      (2) This function applies for all T/tau              */
/*                                                                           */
/*      References:     (1) E-mail from D.A. Howe/NIST,  10/27/00            */
/*                      (2) Final draft of PTTI 2000 HTOT paper:             */
/*                          Final3-PTTI-00.pdf rcv'd 12/24/00                */
/*                                                                           */
/*      Revision record:                                                     */
/*          10/28/00    Created                                              */
/*          12/24/00    Revise coefficients per Ref. 2                       */
/*                                                                           */
/* (c) Copyright 2000   Hamilton Technical Services   All Rights Reserved    */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL HadTotvarBias(int nAlpha)
{
    // Local variable
    float fBias;

    // Verify parameter
    if(nAlpha<-4 || nAlpha>0)
    {
        return (float) -1.0; // Error
    }
    // Sort by noise type
    switch(nAlpha)
    {
        case  0: // W FM
        {
            fBias=(float) 0.995;
        }
        break;

        case -1: // F FM
        {
            fBias=(float) 0.851;
        }
        break;

        case -2: // RW FM
        {
            fBias=(float) 0.771;
        }
        break;

        case -3: // FW FM
        {
            fBias=(float) 0.717;
        }
        break;

        case -4: // RR FM
        {
            fBias=(float) 0.679;
        }
        break;
    }

    return (fBias);
}
```

The FrequenC Library

NAME:	**FUNCTION:**
HadTotvarCalc	Calculate the Hadamard total deviation from frequency data

SYNOPSIS:

```
int HadTotvarCalc(F_TYPE y[], F_TYPE *dev, int af, BOOL *abort, BOOL
progress, void *ProgressFunc(int percent, char *msg))
```

F_TYPE y[]	Frequency data array
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysios end point

F_TYPE *dev	Pointer to Hadamard total deviation
int af	Averaging factor
BOOL *abort	Pointer to abort flag
BOOL progress	Progress indicator flag
void *ProgressFunc	Pointer to progress indicator function with args int percent and char *msg
RETURN: int	# analysis points, or -1 if memory allocation error, or -2 if no result error

REMARKS:
Only data points between start and end analysis limits are analyzed.
All zeros are treated as gaps in frequency data.
This calculation is done with a new array that is deleted after the function closes.
Use of the progress indicator is optional.

EXAMPLE:

```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE y[512];                                      /* freq data array */
F_TYPE dev;                                         /* Hadamard totdev */
int af=1;                                          /* averaging factor */
BOOL abort;                                              /* abort flag */
int num;                                  /* # data points processed */
.
.
num=HadTotCalc(y, &dev, af, &abort, FALSE, NULL); /* call function */
printf("\n# Data Points Analyzed = %d", num);       /* display num */
```

SEE ALSO: HadTotvarBias(), HadTotvarEDF()

REFERENCE: D. Howe, et al, "A Total Estimator of the Hadamard Function Used for GPS Operations", Proc. 32th Meeting, November 2000.

```
/******************************************************************************/
/*                                                                            */
/*                              HadTotvarCalc()                               */
/*                                                                            */
/*      Function to calculate the Hadamard Total Variance from freq data.    */
/*      Uses uninverted even reflection of offset-removed freq data.         */
/*      This function uses fast inner loop algorithm.                        */
/*                                                                            */
/*      Parameters:     F_TYPE y[]     = freq data                            */
/*                                        y[0] = # data points                */
/*                                        y[1] = analysis start               */
/*                                        y[2] = analysis end                 */
/*                      F_TYPE *fHTDev = pointer to Hadamard Totdev           */
/*                      int nAF        = averaging factor                     */
/*                      BOOL *bAbort   = pointer to abort flag                */
/*                      BOOL bProgress = progress indicator flag              */
/*                      void *ProgressFunction()                              */
/*                                     = pointer to progress indicator        */
/*                                       function with args int nPercentDone */
/*                                       and char *szMessage                  */
/*                                                                            */
/*      Return:         int            = # Hadamard Totvar analysis points,  */
/*                                       or -1 if memory alloc error          */
/*                                       or -2 if no result error             */
/*                                                                            */
/*      Notes:          1. Calculation is done entirely with a new array     */
/*                         that is deleted after this function closes.       */
/*                      2. Progress indicator is optional.                    */
/*                      3. This is fast method, and it handles gaps.         */
/*                                                                            */
/*      Overview:       Similar to mod totvar calc except freq instead of    */
/*                      phase data.  Uses 3m-point subsequence from which    */
/*                      linear freq drift is removed before being doubly     */
/*                      extended by uninverted, even reflection at each end. */
/*                      No tau argument needed.  Different final scaling.    */
/*                                                                            */
/*      Description:    Given the freq data array y[], Had Totvar is         */
/*                      calculated using z[], which is computed from a       */
/*                      subsequence of y[] comprising 3m points, where m     */
/*                      the avg factor nAF.  This subsequence is formed      */
/*                      from y[i] where i=n thru n+3m-1, where n is an       */
/*                      overall index that goes from 1 to the # of freq      */
/*                      data points nNum.  The subsequence z[i] is found     */
/*                      from y[i] by removing the frequency drift by         */
/*                      subtracting the slope found by averaging the 1st     */
/*                      and last halves of the subsequence and dividing by   */
/*                      half the interval.                                    */
/*                                                                            */
/*                      This subsequence is then extended at both ends by    */
/*                      an uninverted even reflection according to z(n-k)=   */
/*                      z(n+k-1) and z(n+3m+k-1)=z(n+3m-k) for k=1 thru 3m,  */
/*                      thus creating a new subsequence having a tripled     */
/*                      range from i=n-3m to i=n+6m-1.                        */
/*                                                                            */
/*                      Had Totvar is then calculated as the average of its  */
/*                      nNum-3m Totvar sub-estimates.                         */
/*                                                                            */
/*      Reference:      D. Howe, et al, "A Total Estimator of the Hadamard   */
/*                      Function Used for GPS Operations", Proc. 32th PTTI   */
/*                      Meeting, November 2000 (to be published).            */
/*                                                                          */
/*      Revision record:                                                    */
/*          10/28/00    Cloned from NewModTotvarCalc()                      */
/*          10/29/00    Running                                             */
/*          03/09/03    Made progress function and abprt flag arguments     */
/*          03/14/03    Moved into FrequenC DLL                             */
/*                      Replaced GlobalAllocPtr() with malloc(), etc.       */
/*          04/26/03     Adapted for Version 2.0 source code documentation     */
/*                                                                          */
/* (c) Copyright 2000-2003 Hamilton Technical Services All Rights Reserved   */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL HadTotvarCalc(F_TYPE fY[], F_TYPE *fHTDev,
    int nAF, BOOL *bAbort, BOOL bProgress,
    void *ProgressFunction(int nPercentDone, char *szMessage))
{
    // Local variables
    int i;              // Main index
    int j;              // Auxilary index
    int k;              // Inner index
    int m;              // Alias for averaging factor nAF
    int nNum;           // # points
    int nStart;         // Analysis start
    int nCount=0;       // # analysis points
    int nTic;           // Increment for progress indicator
    int nLast_i=0;      // Previous value of index

    double *fZ;         // Subsequence array pointer
    double fC1;         // Subsequence 1st half average
    double fC2;         // Subsequence 2nd half average
    double fC;          // Freq slope (drift)
    double fSum=0;      // Overall Had Totvar summation
    double fSubsum=0;   // Subsequence Had Totvar summation
    double fTemp;       // Temporary data value

    double fZ1;         // 1st freq average
    double fZ2;         // 2nd freq average
    double fZ3;         // 3rd freq average

    // Initializations
    nStart=(int)fY[1];          // Analysis start point
    nNum=(int)(fY[2]-fY[1]+1);  // # analysis freq data points = N
    m=nAF;                      // Averaging factor

    nTic=(int)ceil(nNum/100);   // Increment for progress report

    // Set fTemp to mean value of overall freq data
    CalcMean(fY, &fTemp, FREQ_DATA);

    // Start
    if(bProgress)
    {
        // Label progress bar
        ProgressFunction(0, "Calculating Hadamard Totdev");
    }
```

```
// Allocate 9m points of fZ[] working memory with index from 0 to 9m-1
// w/o array header (#, Start, End)
if((fZ=(double*)malloc((9*m)*sizeof(double)))==NULL)
{
    return(-1);  // Error - memory reallocation failed
}

// Outer loop thru all nNum-3m+1 subsequences of 3m freq data points
// A subsequence will be processed during each iteration
// and the average used to find Had-Totvar
// This allows a max m of nNum/3

// To handle analysis limits, initialize index to nStart-1 not 0
// and end it before nStart-1-3*m+1 instead of nNum-3*m+1
for(i=nStart-1; i<nStart-1+nNum-3*m+1; i++)
{
    // Copy data subsequence into working array starting at index 3m
    // Note:  The fY[] array needs offset for its header info
    for(j=0; j<3*m; j++)
    {
        // Test for gap (zero value)
        if(fY[i+j+ARRAY_OFFSET])
        {
            fZ[3*m+j]=fY[i+j+ARRAY_OFFSET];
            fTemp=fZ[3*m+j];
        }
        else
        {
            // Replace gap with interpolated value
            // if adjacent points are themselves not gaps
            if( (fY[i+j+ARRAY_OFFSET-1]) && (fY[i+j+ARRAY_OFFSET+1]) )
            {
                fZ[3*m+j]=
                    (fY[i+j+ARRAY_OFFSET-1]+fY[i+j+ARRAY_OFFSET+1])/2.0;
            }
            else
            {
                // For lack of a better idea,
                // replace gap with last OK value
                // or, if haven't found one, the overall mean value
                fZ[3*m+j]=fTemp;
            }
        }
    }

    // Do removal of frequency drift
    // Find slope fC by averaging the 1st and last halves of the subsequence
    // subtracting them, and dividing by half the interval
    fC1=0;
    fC2=0;
    for(j=0; j<3*m/2; j++)
    {
        fC1+=fZ[3*m+j];
        fC2+=fZ[3*m+(int)ceil(3.0*(double)m/2.0)+j];
    }
    fC1/=(double)(3*m/2);
    fC2/=(double)(3*m/2);
    fC=(fC2-fC1)/ceil(3.0*(double)m/2.0);
```

```
// Then subtract slope from each point
for(j=0; j<3*m; j++)
{
    fZ[3*m+j]-=fC*(double)j;
}

// Do uninverted even reflection at bottom and top
for(j=0; j<3*m; j++)
{
    // Bottom reflection
    fZ[j]=fZ[6*m-1-j];

    // Top reflection
    fZ[6*m+j]=fZ[j];
}

// Calc Hvar for the fZ[] subsequence
// Will need to add gap testing and analysis limits
// This data array does not need an offset
// This code implements the summation of Eq(4) of the reference

// Initialize inner sum
fSubsum=0;

// We have a doubly-extended subsequence fZ[] of 9m points
// and are going to form m-point averages
// in 6m+1 groups of 3 (called fZ1, fZ2 and fZ3)
// from which all possible fully overlapping 2nd differences
// will be used to calculate Hadamard Totvar
// Note: Only 6m of the 6m+1 possible 2nd differences are used
// because the last one is a duplicate
// This implements Eq. 1 of the reference
for(j=0; j<6*m; j++)
{
    if(j==0)
    {
        // Find the 3 freq averages for the first 2nd difference
        // Initialize freq averages
        fZ1=0.0;
        fZ2=0.0;
        fZ3=0.0;

        // Sum freq averages for 1st 3 sets of m points
        // using points 0 thru m-1, m thru 2m-1 and 2m thru 3m-1
        // Note: These freq sums are divided by m = # points of each
        // sum in the 2nd difference expression below
        for(k=0; k<m; k++)
        {
            fZ1 += fZ[k];
            fZ2 += fZ[k+m];
            fZ3 += fZ[k+2*m];
        }
    }
    else
    {
        // Now do the rest of the points w/o summing all each time
        // Just subtract the old point and add the new one
        // Sum freq averages for 3 sets of m points
        // using points j thru j+m-1, j+m thru j+2m-1 and
        // j+2m thru j+3m-1
```

```
                // Note: These freq sums are divided by m = # points of
                // each sum in the 2nd difference expression below
                fZ1 = fZ1 -fZ[j-1]     +fZ[j+m-1];
                fZ2 = fZ2 -fZ[j+m-1]   +fZ[j+2*m-1];
                fZ3 = fZ3 -fZ[j+2*m-1] +fZ[j+3*m-1];
            }

            // Calc 2nd diff of one of the rest of the 6m sets of 3 freq avgs
            fSubsum += SQR((fZ1 - 2*fZ2 + fZ3)/m);
        }

        // Scale the subsum and add it to overall sum
        fSum+=fSubsum/(6*m);

        // Increment the sum count
        nCount++;

        // Display progress
        // Update when index has changed by at least tic amount
        if(((i-nLast_i)>=nTic) && (bProgress))
        {
            ProgressFunction((int)(0.5+100.0*i/(nNum-3*m)),
                "Calculating Hadamard Totdev");
            nLast_i=i;
        }

        // Check for abort
        if(*bAbort)
        {
            break;
        }
   }

    // Free the subsequence array
    free(fZ);

    // Scale result - See Eq (4) of Reference
    // Without gaps, nCount=nNum-3m
    // Note: Divisor S/B 6
    // But 3 gives result that agrees exactly with normal and overlapping
    // Hadamard deviation for AF=1 for FREQ.DAT test data
    // 6 OK for other AFs
    // Results agree exactly with T.Peppler/NIST with 6 for all AFs
    // So 6 is clearly right - low result for AF=1 is apparently a property of HTOT
    fSum/=(6*nCount);

    // Find Hadamard Totdev
    if(nCount && fSum>0)
    {
        *fHTDev=(F_TYPE)sqrt(fSum);
    }
    else // No result error
    {
        *fHTDev=0.0;
        nCount=-2;
    }
    // Done
    // Clear progress display
    ProgressFunction(0, "");

    return(nCount);  // Return # freq analysis points
}
```

The FrequenC Library	
NAME: `HadTotvarEDF`	**FUNCTION:** `Description of what function does`
SYNOPSIS: `int FunctionName( F_TYPE z[], F_TYPE one, F_TYPE two)`	

`F_TYPE z[]`	`Phase or frequency data array`
	`z[0] = # data points`
	`z[1] = analysis start point`
	`z[2] = analysios end point`

`F_TYPE one`	`parameter one, and description of what it is`
`F_TYPE two`	`parameter two, and description of what it is`
RETURN: `int`	`# non-gap data points processed`

REMARKS:

```
Only data points between start and end analysis limits are modified.
Embedded zeros are treated as gaps in phase data.
All zeros are treated as gaps in frequency data.
```

EXAMPLE:

```
#include "frequenc.h"                           /* FrequenC header file */
F_TYPE z[512];                                           /* data array */
F_TYPE one;                                           /* parameter one */
F_TYPE two;                                           /* parameter two */
int num;                                   /* # data points processed */
.
.
one=1.0;                                   /* set 1st parameter value */
two=2.0;                                   /* set 2nd parameter value */
num=FunctionName(z, one, two);                        /* call function */
printf("\n# Data Points Scaled = %d", num);         /* display num */
```

SEE ALSO: `RelatedFunctionName()`

REFERENCE: NIST Technical Note 1337

```
/*****************************************************************************/
/*                                                                           */
/*                             HadTotvarEDF()                                */
/*                                                                           */
/*  Function to determine the estimated # of chi-squared degrees of freedom  */
/*  for the Hadamard total variance (HTOT) of a power law noise process.     */
/*                                                                           */
/*  Params:          int   nAlpha = alpha                                    */
/*                   float fRatio = T/tau = N/m                              */
/*                                  where: N = # phase data points           */
/*                                         m = avg factor = tau/tau0         */
/*                                                                           */
/*  Return:          float edf = estimated degrees of freedom                */
/*                               or -1 if error                              */
/*                                                                           */
/*  The edf is modeled by the expression (T/tau)/[b0+b1(tau/T)], where:      */
/*                                                                           */
/*  Alpha    Noise     b0      b1       Values per D. Howe, et.al            */
/*   0        W FM    0.559   1.004     12/24/00 draft of PTTI'00 paper      */
/*  -1        F FM    0.868   1.140                                          */
/*  -2       RW FM    0.938   1.696                                          */
/*  -3       FW FM    0.974   2.554                                          */
/*  -4       RR FM    1.276   3.149                                          */
/*                                                                           */
/*  Notes:           (1) This function applies only to -4 >= alpha >= 0      */
/*                   (2) This function applies only for m > 16               */
/*                   (3) T is defined as tau0 * # phase data points          */
/*                                                                           */
/*  References:      (1) E-Mail from D.A. Howe/NIST, 10/27/00                */
/*                   (2) Final draft of PTTI 2000 HTOT paper:                */
/*                       Final3-PTTI-00.pdf rcv'd 12/24/00                   */
/*                                                                           */
/*  Revision record:                                                         */
/*      10/28/00    Created                                                  */
/*      12/24/00    Revised per Ref. 2                                       */
/*                                                                           */
/* (c) Copyright 2000  Hamilton Technical Services   All Rights Reserved     */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL HadTotvarEDF(int nAlpha, float fRatio)
{
    // Local variables
    float fEDF;
    double b0;
    double b1;

    // Verify parameters
    if(nAlpha<-4 || nAlpha>0)
    {
        return -1.0; // Error
    }

    switch(nAlpha)
    {
        case 0: // W FM noise
        {
            b0=0.559;
            b1=1.004;
        }
        break;

        case -1: // F FM noise
        {
            b0=0.868;
            b1=1.140;
        }
        break;

        case -2: // RW FM noise
        {
            b0=0.938;
            b1=1.696;
        }
        break;

        case -3: // FW FM noise
        {
            b0=0.974;
            b1=2.554;
        }
        break;

        case -4: // RR FM noise
        {
            b0=1.276;
            b1=3.149;
        }
        break;
    }

    fEDF=(float)((fRatio)/(b0+(b1/fRatio)));
    return fEDF;
}
```

The FrequenC Library

NAME: JulianToDate	**FUNCTION:** Convert Julian day number to Gregorian date
SYNOPSIS: long JulianToDate(long jdate)	
long jdate	Julian day number
RETURN: long	Gregorian date (yyyymmdd)
REMARKS: Calls MakeDate()	

EXAMPLE:

```
#include "frequenc.h"                              /* FrequenC header file */
long jdate;                                                /* Julian day # */
long gdate;                                              /* Gregorian date */
.
.
jdate=24050000;                                        /* set Julian day # */
gdate=JulianToDate(jdate);                                /* call function */
printf("\nGregorian Date = %ld", gdate);                 /* display result */
```

SEE ALSO: DateToJulian(), MakeDate()

REFERENCE:

```
/******************************************************************************/
/*                                                                            */
/*                              JulianToDate()                                */
/*                                                                            */
/*      Function to convert Julian day number to Gregorian date               */
/*                                                                            */
/*      Parameters: long    jdate   Julian day number                         */
/*                                                                            */
/*      Return:     long            Gregorian date (yyyymmdd)                 */
/*                                                                            */
/*      Revision record:                                                      */
/*          07/17/97    Cloned from juldat() of Hammer Library                */
/*                      Changed error code to -1                              */
/*                                                                            */
/******************************************************************************/

#include "frequenc.h"                         /* FrequenC Library header file */

long __declspec(dllexport) FAR PASCAL JulianToDate(long jdate)
{
    extern long int makdate();
    long int gdate; /* Gregorian Date (yyyymmdd) for given Julian Day */
    int mo, da, yr;
    long int td, tm, ty;

    if (jdate<=0L)  /* if zero or negative, return -1 */
    {
        return (-1L);
    }

    jdate-=1721119L;
    ty=(4L*jdate-1L)/146097L;
    jdate=4L*jdate - 1L - 146097L*ty;
    td=jdate/4L;

    jdate=(4L*td+3)/1461L;
    td=4L*td + 3L - 1461L*jdate;
    td=(td+4L)/4L;

    tm=(5L*td - 3L)/153L;
    td=5L*td - 3L - 153L*tm;
    td=(td+5L)/5L;

    ty=100L*ty + jdate;
    if (tm<10L)
    {
        tm+=3L;
    }
    else
    {
        tm-=9L, ty++;
    }

    da=(int) td; mo=(int) tm; yr=(int) ty;
    gdate=MakeDate(mo, da, yr);

    return(gdate);
}
```

The FrequenC Library

NAME: MedianDeviation	**FUNCTION:** Calculate the median absolute deviation from freqency data

SYNOPSIS:
```
int MedDev(F_TYPE y[], float *mad)
```

F_TYPE y[]	Frequency data array
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

float *mad	Pointer to median absolute deviation
RETURN: int	# analysis points (including filled gaps)

REMARKS:
The median absolute deviation (MAD) is a robust statistic based on the median of the data. It is the median of the scaled absolute deviations of the data points from their median value, and is defined as MAD = Median { | y(i) - m | / 0.6745 }, where m=Median { y(i) }, and the factor 0.6745 makes the MAD equal to the standard deviation for normally distributed data.

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE y[512];                                      /* freq data array */
float mad;                                /* median absolute deviation */
.
.
MedianDeviation(y, &mad);                             /* call function */
printf("\nMAD = %e", mad);                          /* display results */
```

SEE ALSO: FindMedian()

REFERENCES: (1) Gernot M.R. Winkler, "Introduction to Robust Statistics and Data Filtering, (2) D.B. Percival, "Use of Robust Statistical Techniques in Time Scale Formation.

```
/*****************************************************************************/
/*                                                                           */
/*                              MedDev()                                     */
/*                                                                           */
/*      Function to calc median of absolute deviation from freqency data     */
/*      Based on Stable/DOS function F37 of STAB_F37.C                       */
/*                                                                           */
/*      meddev = med { | x - Median | / 0.6745 }                             */
/*                                                                           */
/*      Procedure for calculating meddev:                                    */
/*          1. Sort data                                                     */
/*          2. Find Median of data                                           */
/*          3. Subtract Median from all data points                          */
/*          4. Calculate absolute value of these differences                 */
/*          5. Divide values by 0.6745 (This makes the expected value of     */
/*             meddev = the std dev for normally distributed data.)          */
/*          6. Resort data                                                   */
/*          7. Find median value = meddev                                    */
/*      These operations are performed in-place on a working data array t[]  */
/*                                                                           */
/*      Parameters:                                                          */
/*          y[]   = frequency data; y[0] = # data points                     */
/*          y[1]  = analysis start; y[2] = analysis end                      */
/*          p_mad = pointer to median absolute deviation                     */
/*          zero freq value treated as gap in data                           */
/*      Returns num = # analysis points (including filled gaps)              */
/*                                                                           */
/*      References: (1) Gernot M.R. Winkler, "Introduction to Robust         */
/*                      Statistics and Data Filtering".                      */
/*                  (2) D.B. Percival, "Use of Robust Statistical            */
/*                      Techniques in Time Scale Formation".                 */
/*                                                                           */
/*      Revision record:                                                     */
/*          01/14/95    Created                                              */
/*          01/15/95    Added 0.6745 factor for unbiased estimate            */
/*          04/30/96    Modified as MedDev() for Stable/Win                  */
/*          01/01/98    Changes for MS VC++ compatibility & warnings         */
/*          04/26/03    Adapted for Version 2.0 source code documentation       */
/*                                                                           */
/* (c) Copyright 1995-2003 Hamilton Technical Services All Rights Reserved   */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

// Declare comparison function - not exported
static int MedDevCompare(const void *a, const void *b);


int __declspec(dllexport) FAR PASCAL MedDev(F_TYPE y[], float *p_mad)
{
    int i;                                                      /* 1st index */
    int j=0;                                                    /* 2nd index */
    int start=(int)y[1];                                   /* analysis start */
    int end=(int)y[2];                                       /* analysis end */
    int num=(int)y[0];                                      /* # data points */
    F_TYPE *t;                       /* pointer to temporary data storage array */
    F_TYPE med;                                              /* median value */
```

```
// Allocate storage for temporary array
if((t=(F_TYPE *)malloc((num+3)*sizeof(F_TYPE)))==NULL)
                                /* allocate temporary data storage space */
{
    return(1);                                          /* error return */
}                                                             /* end if */

// Copy data
for(i=0; i<num+3; i++)
{
    t[i]=y[i];                                             /* copy data */
}                                                            /* end for */

// Fill any gaps in data
if(CountGaps(t, FREQ))
{
    FillGaps(t, FREQ);                                     /* fill gaps */
}                                                             /* end if */

// Eliminate non-analysis points
for(i=start+2; i<=end+2; i++)
{
    t[j]=t[i];
    j++;
}                                                            /* end for */
num=end-start+1;

// Sort data into numerical order
qsort((void *)t, (size_t) num, sizeof(F_TYPE), MedDevCompare);

// Find median
if(num%2)                                              /* is num odd? */
{
    med=t[num/2];                                       /* middle value */
}                                                             /* end if */
else
{                                                          /* n is even */
    med=(t[(num/2)-1]+t[num/2])/2.0;       /* average of 2 middle values */
}                                                           /* end else */

// Subtract median, get absolute value and scale data points
for(i=0; i<num; i++)
{
    t[i]-=med;
    t[i]=fabs(t[i])/0.6745;
}                                                            /* end for */

// Resort data into numerical order
qsort((void *)t, (size_t) num, sizeof(F_TYPE), MedDevCompare);

// Find meddev
if(num%2)                                              /* is num odd? */
{
    *p_mad=(float)t[num/2];                             /* middle value */
}                                                             /* end if */
else
{                                                          /* n is even */
    *p_mad=(float)((t[(num/2)-1]+t[num/2])/2.0);  /* avg of 2 mid values */
}                                                           /* end else */
```

```
    free(t);                                    /* deallocate temporary array */

    // Done
    return(num);                                    /* # non-zero data points */
}                                                             /* end MedDev() */

/******************************************************************************/
/*                                                                            */
/*      Comparison Function for qsort()                                       */
/*                                                                            */
/******************************************************************************/

int MedDevCompare(const void *a, const void *b)
{
    F_TYPE x =  (*(F_TYPE *) a) - (*(F_TYPE *) b) ;
    return (x>0) ? 1 : (x<0) ? -1 : 0;
}
```

The FrequenC Library

NAME:	**FUNCTION:**
MakeDate	Make Gregorian date from month, day and year

SYNOPSIS:
`long MakeDate(int month, int day, int year)`

int month	Month (1-12)
int day	Day (1-31)
int year	Year (4-digit, e.g. 2003)
RETURN: long	Gregorian date (yyyymmdd)

REMARKS:
Year must be 1900 or greater

EXAMPLE:
```
#include "frequenc.h"                         /* FrequenC header file */
long gdate;                                         /* Gregorian date */
int month;                                                   /* month */
int day;                                                       /* day */
int year;                                                     /* year */
.
.
month=1;                                                 /* set month */
day=1;                                                     /* set day */
year=2003;                                                /* set year */
gdate=MakeDate(month, day, year);                    /* call function */
printf("\nDate = %ld", gdate);                     /* display results */
```

SEE ALSO: BreakDate()

REFERENCE: None

```
/*****************************************************************************/
/*                                                                           */
/*                              MakeDate()                                   */
/*                                                                           */
/*      Function to make Gregorian date from month, day and year             */
/*                                                                           */
/*      Parameters: int     mo      Month (1-12)                             */
/*                  int     da      Day (1-31)                               */
/*                  int     yr      Year (2 or 4-digit)                      */
/*                                                                           */
/*      Return:     long            Gregorian date (yyyymmdd)                */
/*                                                                           */
/*      Revision record:                                                     */
/*          07/17/97    Cloned from makdate() of Hammer Library              */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"                       /* FrequenC Library header file */

long __declspec(dllexport) FAR PASCAL MakeDate(int mo, int da, int yr)
{
    yr = (yr<=100 ? yr+1900 : yr);
    return ((long int) yr*10000L + mo*100L + da);
}
```

The FrequenC Library	
NAME: MultitaperSpectrumCalc	**FUNCTION:** Description of what function does
SYNOPSIS: `int FunctionName( F_TYPE z[], F_TYPE one, F_TYPE two)`	

`F_TYPE z[]`	Phase or frequency data array
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysios end point

`F_TYPE one`	parameter one, and description of what it is
`F_TYPE two`	parameter two, and description of what it is
RETURN: `int`	# non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are modified.
Embedded zeros are treated as gaps in phase data.
All zeros are treated as gaps in frequency data.

EXAMPLE:
```
#include "frequenc.h"                           /* FrequenC header file */
F_TYPE z[512];                                           /* data array */
F_TYPE one;                                           /* parameter one */
F_TYPE two;                                           /* parameter two */
int num;                                  /* # data points processed */
.
.
one=1.0;                                  /* set 1st parameter value */
two=2.0;                                  /* set 2nd parameter value */
num=FunctionName(z, one, two);                        /* call function */
printf("\n# Data Points Scaled = %d", num);            /* display num */
```

SEE ALSO: RelatedFunctionName()

REFERENCE: NIST Technical Note 1337

```
/*********************************************************************************/
/*                                                                               */
/*                                     MT.C                                      */
/*                                                                               */
/*                  ANSI C Code for Multitaper Spectral Analysis                 */
/*                                                                               */
/*  A simple set of subroutines in ANSI-C are presented for multiple taper       */
/*  spectrum estimation. The multitaper approach provides an optimal spectrum    */
/*  estimate by minimizing spectral leakage while reducing the variance of the   */
/*  estimate by averaging orthogonal eigenspectrum estimates. The orthogonal     */
/*  tapers are Slepian np prolate functions used as tapers on the windowed time  */
/*  series. Since the taper functions are orthogonal, combining them to achieve  */
/*  an average spectrum does not introduce spurious correlations as standard     */
/*  smoothed single-taper estimates do. Furthermore, estimates of the degrees of */
/*  freedom and F-test values at each frequency provide diagnostics for          */
/*  determining levels of confidence in narrow band (single frequency)           */
/*  periodicities. The program provided is portable and has been tested on both  */
/*  Unix and Macintosh systems.                                                  */
/*                                                                               */
/*  References:                                                                  */
/*  1. J.M. Lees and J. Park, "A C-Subroutine for Computing Multiple-Taper       */
/*     Spectral Analysis", Computers in Geoscience, Vol. 21, 1995, pp. 195-236.  */
/*  2. D.J. Thomson, "Spectrum Estimation and Harmonic Analysis", Proc. IEEE,    */
/*     Vol. 70, No. 9, September 1982, pp. 1055-1096.                            */
/*  3. D.B. Percival and A.T. Walden, Spectral Analysis for Physical Applica-    */
/*     tions, Cambridge University Press, 1993.                                  */
/*                                                                               */
/*  Downloaded from: ftp://ftp.iamg.org/VOL21/v21-2-2.zip                        */
/*                                                                               */
/*  Required project source files are JFOUR1.C, JREALFT.C, JTAP.C and JL.H       */
/*  The three C files are combined into this one.                                */
/*  It is not necessary to invoke the JL.H header file                           */
/*  All necessary macros and prototypes are included in this source file         */
/*                                                                               */
/*  Adapted by W. Riley, Hamilton Technical Services, as MTAP July 9, 2002       */
/*                                                                               */
/*  Revision Record:                                                             */
/*      07/19/02    Up-and running OK as separate program.                       */
/*      07/20/02    Moved MTAP into Stable32 as Stable_c.c.                      */
/*                  Made many revisions to adapt for Stable32 usage.             */
/*                  Basically working - getting reasonable plots.                */
/*      07/25/02    Trim down unwanted features                                  */
/*      03/15/03    Adapted for FrequenC DLL                                     */
/*      04/26/03    Adapted for Version 2.0 source code documentation               */
/*                                                                               */
/*  (c) 2002-3  W. Riley  Hamilton Technical Services  All Rights Reserved       */
/*                                                                               */
/*********************************************************************************/

// Includes
#include "frequenc.h"
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <math.h>
#include <memory.h>

// Local Macros
#define PI 3.14159265358979
#define ABS(a) ((a) < (0) ? -(a) : (a))
```

```
#define MAX(a,b) ((a) >= (b) ? (a) : (b))
#define DIAG1 0
#define JM_ADD 0

// Memory allocation functions
float *alloc_fvec(long first , long last)
{
    float *v;
    v=(float *)malloc((size_t) ((last-first+1+JM_ADD)*sizeof(float)));
    return v-first+JM_ADD;
}

double *alloc_dvec(long first , long last)
{
    double *v;
    v=(double *)malloc((size_t) ((last-first+1+JM_ADD)*sizeof(double)));
    return v-first+JM_ADD;
}

int *alloc_ivec(long first , long last)
{
    int *v;
    v=(int *)malloc((size_t) ((last-first+1+JM_ADD)*sizeof(int)));
    return v-first+JM_ADD;
}

// Local function prototypes
int get_pow_2(int inum);
float get_cos_taper(int n, int k);
int adwait(double *sqr_spec, double *dcf, double *el, int nwin, int num_freq,
    double *ares, double *degf, double avar);
void get_F_values(double *sr, double *si, int nf, int nwin, float *Fvalue,
    double *b);
void do_mtap_spec(float *data, int npoints, int kind, int nwin, float npi,
    int inorm, float dt, float *ospec, int klen);
int hires(double *sqr_spec, double *el, int nwin, int num_freq, double *ares);
void jfour1(float data[], unsigned long nn, int isign);
void jrealft(float data[], unsigned long n, int isign);
int jtridib_(int *n, double *eps1, double *d, double *e, double *e2, double *lb,
    double *ub, int *m11, int *m, double *w, int *ind, int *ierr, double *rv4,
    double *rv5);
int jtinvit_(int *nm, int *n, double *d, double *e, double *e2, int *m,
    double *w, int *ind, double *z, int *ierr, double *rv1, double *rv2,
    double *rv3, double *rv4, double *rv6);
void mt_get_spec(float *series, int inum, int klength, float *amp);
int multitap(int n, int nwin, double *el, float npi, double *tapers,
    double *tapsum);
void zero_pad(float output[], int start, int olength);
double remove_mean(float *x, int lx);
float get_cos_taper(int n, int k);

/*******************************************************************************/
/*                                                                             */
/*                           MultiTaperSpctrumCalc()                           */
/*                                                                             */
/*  Function to calculate and plot multi-taper power spectrum, similar to      */
/*  SpectrumCalc() that uses ordinary periodogram method.                      */
/*                                                                             */
/*  Parameters:                                                                */
/*      F_TYPE  y[]         = time-domain data; y[0] = *y = # data points      */
/*      float   data[]      = real data points -> PSD results                  */
/*      float   freq[]      = Fourier freq results                             */
/*      float   dt          = time interval of data samples                    */
/*      int     num_points  = # time series data points                        */
/*      float   npi         = # pi-prolate functions                           */
/*      int     nwin        = # summing windows                                */
/*      int     kind        = kind of analysis (way eigenspectra are combined)  */
/*                            1 = hires                                        */
/*                            2 = adwait                                       */
/*                            3 = naive periodogram                            */
/*      int     inorm       = normalization method                             */
/*                            1 = standard = npoints                           */
/*                            2 = other = 1/dt                                 */
/*                            3 = sqrt(npoints)                                */
/*      int     win_type    = windowing method code:                           */
/*                            0 = Rectangular                                  */
/*                            1 = Hanning                                      */
/*                            2 = Hamming                                      */
/*      int     p_type      = phase_noise_type:                                */
/*                            0 = Sx(f)                                        */
/*                            1 = So(f)                                        */
/*                            2 = L(f)                                         */
/*      float   corr_factor = correction_factor to adjust PSD for carrier freq  */
/*      BOOL    bDataType   = data type:                                       */
/*                            0 = Phase                                        */
/*                            1 = Frequency                                    */
/*                                                                             */
/*  Returns:                                                                   */
/*       int    num_freqs   = # FFT frequencies                                */
/*                                                                             */
/*  Note: Calling function must allocate the correct data & freq array size.   */
/*  This size is 1+klen/2, where klen is twice the smallest power of 2         */
/*  that encompasses the data (e.g. for 500 data points, klen is 1024 & array  */
/*  size is 1+512=513).                                                        */
/*                                                                             */
/*  Revision record:                                                           */
/*      07/20/02    Adapted from SpectrumCalc()                                */
/*                  Objective is to call from SpectrumCalcDlgProc() instead of  */
/*                  SpectrumCalc() to offer multi-taper spectral analysis      */
/*      07/25/02    Eliminate O/P file and Fvalues calc                        */
/*                  Eliminate naive spectrum calc                              */
/*                  Eliminate dof calc and file                                */
/*      07/30/02    Revised to handle analysis limits                          */
/*      03/15/03    Adapted for FrequenC DLL                                   */
/*      03/16/03    Added bDataType parameter                                  */
/*                  Changed from void to int to return # freqs                 */
/*      04/26/03    Adapted for Version 2.0 source code documentation             */
/*                                                                             */
/*******************************************************************************/

int __declspec(dllexport) FAR PASCAL MultiTaperSpectrumCalc(F_TYPE y[],
    float data[], float freq[], float dt, int num_points, float npi, int nwin,
    int kind, int inorm, int win_type, int p_type, float corr_factor,
    BOOL bDataType)
{
    // Local Variables
    // Ints
    int i;              // General index
    int k;
    int npoints;
    int klen;
    int num_freqs;
    int K;
    int freqwin;
    int increase;
    int isign = 1;
    int offset;         // Index of first data point to be analyzed

    // Floats
    float *spec;
    float fWidth;
    float df;
    float nyquist;
    float bw_corr;      // Bandwidth correction factor
    float log_mult;     // Log PSD multiplier scale factor

    // Doubles
    double mean;

    // WJR Notes re original input parameters 7/19/02
    // name = input time series data filename
    // npi = order of pi-prolate taper functions
    // Choose some small integer (j=2, 3 or 4)
    // This sets resolution BW 2W
    // W is set to a small multiple of the fundamental frequency 1/(N*Delta t)
    // so W = j/(N*Delta t).  See Percival & Walden p.335.  Use 3.
    // nwin = # summing windows (Percival recommends 6 for lowest bias) Use 5.
    // kind = way the eigenspectra are combined
    //   1 = hires=best for white noise
    //   2 = adwait=optimal adaptive weights=best for colored noise
    //   3 = naive periodogram
    //   For our purposes, where colored noise is often encountered, use 2
    // inorm=normalization method
    //   1 = standard
    //   2 = other
    // So, our parameters are typically filename 3 5 2 1

    // Input parameters
    // dt = sampling interval
    // num_points = # input points
    // npi = # pi-prolate functions
    // nwin = # summing windows
    // kind = kind of analysis
    // inorm = normalization method

    // Notes re parameters:
    // See p. 335, Percival and Walden
    // Choose npi = 2,3,4 (some small integer)
    // Then W = npi/(num_points*dt) or num_points*W = npi/dt
    // K < 2*num_points*W*dt
    // nwin = 0...K-1

    // Calc BW params
    fWidth=npi/((float)num_points*dt);
    K=(int)(2*num_points*fWidth*dt);

    // Calc Nyquist frequency (2 samples/cycle)
    nyquist=(float)(0.5/dt);

    // Calc zero-padded power-of-2 data length
    klen=get_pow_2(num_points);
```

```
increase=1;
klen=(int)(klen*pow((double)2, (double)increase));  // This doubles klen

// Calc # Fourier frequencies
num_freqs=1+klen/2;

// Data entry from array
// The xdata array gets loaded with the time domain data.
// This array is allocated with space for the # points & analysis limits.
// These are put into the xdata array initially to allow it to use the
// NewFillFloatGaps() function.
// Then the header info is removed from the xdata array for the FFT calcs.

// Find index of first data point to be analyzed
offset=(int)y[1]+2;

// Copy y[] analysis data into data[] with header info
data[0]=(float)num_points;
data[1]=1.0;
data[2]=(float)num_points;
for(i=0; i<num_points; i++)
{
    data[i+ARRAY_OFFSET]=(float)y[i+offset];
}

// Make sure no there are gaps in the time domain data
FillFloatGaps(data, bDataType);

// Eliminate header from analysis data
for(i=0; i<num_points; i++)
{
    data[i]=data[ARRAY_OFFSET+i];
}

npoints=i;
k=1;

// Remove average
mean=remove_mean(data, npoints);

df=2*nyquist/klen;
freqwin=(int)(fWidth/df)/2;

spec=alloc_fvec((long)0, (long)klen);

do_mtap_spec(data, npoints, kind, nwin, npi, inorm, dt, spec, klen);

// Must allocate correct size at calling function
for(i=1; i<num_freqs-1; i++) // Start index excludes dc component
{
    // Copy Fourier freqs to freq array
    freq[i-1]=df*i;
}

// Calc bandwidth correction
// Note:  There are 2 other ways of doing this:
// 1.  Use 2W where W=j/N*delta t
// 2.  Normalize the results appropriately
bw_corr=(float)log10(freq[0]);
```

```
    // Set log multiplier factor
    if(p_type==2) // L(f)
    {
        log_mult=10;
    }
    else // Sx(f) & So(f)
    {
        log_mult=1;
    }

    // Convert spectral data to log form
    // Note that data is PSD result & freq is Fourier freq
    // for(i=0; i<num_freqs; i++)
    for(i=1; i<num_freqs-1; i++) // Start index excludes dc component
    {
        // Save power spectrum values plus correction factors
        data[i-1]=(float)(log_mult*(log10(spec[i]*corr_factor)- bw_corr));
    }

    // Save PSD data - eliminated 03/16/03 - now done by calling function
    // PSDWrite(szPSDFile, freq, data, num_freqs-2);

    // Free allocated arrays
    free(spec);

    // Done - return # freqs
    return num_freqs;
}

/*--------------------------------------------------------------------------------*/
/*                                   mt_get_spec                                  */
/*--------------------------------------------------------------------------------*/

void mt_get_spec(float *series, int inum, int klength, float *amp)
{
                                                     /* series = input time series */
                                                 /* inum   = length of time series */
                /* klength = number of elements in power spectrum (a power of 2) */
                                                  /* amp = returned power spectrum */
    int i, isign = 1;
    unsigned long   nn;
    nn = klength;

    /* copy amp onto series and apply zero padding to  klength */
    for(i = 0; i < inum; i++)
    {
        amp[i] = series[i];
    }

    zero_pad(amp, inum, klength);

             /* Fast Fourier Transform Routine:  here we are using the Numerical */
               /* Recipes routine jrealft which returns the fft in the 1-D input */
                   /* array packed as pairs of real numbers. The jrealft routine */
            /* requires the input array to start at index=1 so we must decrement */
                                                              /* the index of amp */
    jrealft(amp - 1, nn, isign);
}
```

```
/*----------------------------------------------------------------------------*/
/*                                  do_mtap_spec                                */
/*----------------------------------------------------------------------------*/

void do_mtap_spec(float *data, int npoints, int kind, int nwin, float npi,
    int inorm, float dt, float *ospec, int klen)
{
                                      /* data = floating point input time series */
                                           /* npoints = number of points in data */
            /* kind = flag for choosing hires or adaptive weighting coefficients */
                                  /* nwin = number of taper windows to calculate */
                                         /* npi = order of the slepian functions */
                                     /* inorm = flag for choice of normalization */
                         /* dt = sampling interval (time) ospec = output spctrum */
                          /* klen = number of frequecies calculated (power of 2) */
    // Local variables
    int i, j, k;
    double *lambda, *tapers;
    FILE *fopen();
    float *b;
    int iwin, kk;
    double anrm, norm;
    double *ReSpec, *ImSpec;
    double *sqr_spec, *amu;
    float *amp;
    double avamp, temp, sqramp;
    double sum, *tapsum;
    int num_freqs;
    int len_taps, num_freq_tap;
    double *dcf, *degf, avar;
    int n1, n2, kf;

           /* lambda = vector of eigenvalues   tapsum = sum of each taper, saved */
            /* for use in adaptive weighting  tapers =  matrix of slepian tapers */
                                                /* packed in a 1D double array */

    lambda = alloc_dvec((long)0, (long)nwin);
    tapsum = alloc_dvec((long)0,(long)nwin);
    len_taps = npoints * nwin;
    tapers = alloc_dvec((long)0,(long)len_taps);
    num_freqs = 1 + klen / 2;
    num_freq_tap = num_freqs * nwin;

    // Get a Slepian taper
    k = multitap(npoints, nwin, lambda, npi, tapers, tapsum);

    // Choose normalization based on inorm flag
    anrm = 1.;

    switch(inorm)
    {
        case 1:
            anrm = npoints;
            break;
        case 2:
            anrm = 1 / dt;
            break;
        case 3:
            anrm = sqrt((double) npoints);
            break;
        default:
            anrm = 1.;
            break;

    // Apply the taper in the loop.  Do this nwin times
    b = alloc_fvec((long)0, (long)npoints);
    amu = alloc_dvec((long)0, (long)num_freqs);
    sqr_spec = alloc_dvec((long)0, (long)num_freq_tap);
    ReSpec =alloc_dvec((long)0, (long)num_freq_tap);
    ImSpec = alloc_dvec((long)0,(long) num_freq_tap);

    for(iwin = 0; iwin < nwin; iwin++)
    {
        kk = iwin * npoints;
        kf = iwin * num_freqs;

        for(j = 0; j < npoints; j++)
        {
            // Application of iwin-th taper
            b[j] = (float)(data[j] * tapers[kk + j]);
        }

        amp = alloc_fvec((long)0, (long)klen);

        // Calculate the eigenspectrum
        mt_get_spec(b, npoints, klen, amp);

        // This causes error & program termination
        // Removed WJR 7/19/02 - Program then runs to completion
        // free(b);

        // New code to eliminate multiple free()s WJR 7/19/02
        if(iwin==nwin-1)
        {
            free(b);
        }

        sum = 0.0;

        /* get spectrum from real fourier transform */
        norm = 1.0 / (anrm * anrm);

        for(i = 1; i < num_freqs - 1; i++)
        {
            if(2 * i + 1 > klen)
            {
                fprintf(stderr, "error in index\n");
            }

            if(i + kf > num_freq_tap)
            {
                fprintf(stderr, "error in index\n");
            }

            sqramp = SQR(amp[2 * i + 1]) + SQR(amp[2 * i]);
            ReSpec[i + kf] = amp[2 * i];
            ImSpec[i + kf] = amp[2 * i + 1];
            sqr_spec[i + kf] = norm * (sqramp);
            sum += sqramp;
        }
```

```
        sqr_spec[0 + kf] = norm * SQR(fabs(amp[0]));
        sqr_spec[num_freqs - 1 + kf] = norm * SQR(fabs(amp[1]));
        ReSpec[0 + kf] = amp[0];
        ImSpec[0 + kf] = 0.0;
        ReSpec[num_freqs - 1 + kf] = amp[1];
        ImSpec[num_freqs - 1 + kf] = 0.0;
        sum += sqr_spec[0 + kf] + sqr_spec[num_freqs - 1 + kf];

        if(num_freqs - 1 + kf > num_freq_tap)
        {
            fprintf(stderr, "error in index\n");
        }

        temp = sum / (double) num_freqs;

        if(temp > 0.0)
        {
            avamp = sqrt(temp) / anrm;
        }
        else
        {
            avamp = 0.0;
            // fprintf(stderr," avamp = 0.0! \n");
        }

    free(amp);
    }

    // Choice of hi-res or adaptive weighting for spectra
    switch(kind)
    {
        case 1:
        {
            hires(sqr_spec, lambda, nwin, num_freqs, amu);

            for(i = 0; i < num_freqs; i++)
            {
                ospec[i] = (float)(amu[i]);
            }
            break;
        }
        case 2:
        {
            /* get avar = variance */
            n1 = 0;
            n2 = npoints;
            avar = 0.0;

            for(i = n1; i < n2; i++)
            {
                avar += (data[i]) * (data[i]);
            }

            switch(inorm)
            {
                case 1:
                {
                    #if(0) // Original code - NG for npoints >=65536 - get int O/F
                    avar = avar / (npoints * npoints);
                    #endif
                    #if(1) // New code - WJR - 07/28/02
                    avar = avar / ((double)(npoints) * (double)(npoints));
                    #endif
                    break;
                }
                case 2:
                {
                    avar = avar * dt * dt;
                    break;
                }
                case 3:
                {
                    avar = avar / npoints;
                    break;
                }
                default:
                {
                    break;
                }
            }
        }

        dcf = alloc_dvec((long)0, (long)num_freq_tap);
        degf =alloc_dvec((long)0, (long)num_freqs);
        adwait(sqr_spec, dcf, lambda, nwin, num_freqs, amu, degf, avar);

        // Wrap up
        for(i = 0; i < num_freqs; i++)
        {
            ospec[i] = (float)(amu[i]);
        }

        free(dcf);
        free(degf);

        break;
    }

    // Free up memory and return
    free(amu);
    free(sqr_spec);
    free(ReSpec);
    free(ImSpec);
    free(lambda );
    free(tapers);
}

/*-------------------------------------------------------------------------------*/
/*                                      multitap                                 */
/*-------------------------------------------------------------------------------*/

int multitap(int num_points, int nwin, double *lam, float npi, double *tapers,
    double *tapsum)
{
                /* get the multitaper slepian functions: num_points = number of */
               /* points in data stream nwin = number of windows lam= vector of */
           /* eigenvalues npi = order of slepian functions tapsum = sum of each */
             /* taper, saved for use in adaptive weighting  tapers =  matrix of */
                                  /* slepian tapers, packed in a 1D double array */
```

```
int i, k, kk;
double ww, cs, ai, an, eps, rlu, rlb, aa;
double dfac, drat, gamma, bh, tapsq, TWOPI, DPI;
double *diag, *offdiag, *offsq;
double *scratch1, *scratch2, *scratch3, *scratch4, *scratch6;

/* need to initialize iwflag = 0 */
double anpi;
double *ell;
int *ip;
double *evecs;
long len;
int ierr;
int m11;
DPI = (double) PI;
TWOPI = (double) 2 *DPI;
anpi = npi;
an = (double) (num_points);
ww = (double) (anpi) / an;             /* this corresponds to P&W's W value  */
cs = cos(TWOPI * ww);
ell = alloc_dvec((long)0,(long) nwin );
diag = alloc_dvec((long)0, (long)num_points  );
offdiag = alloc_dvec((long)0, (long)num_points  );
offsq = alloc_dvec((long)0, (long)num_points  );
scratch1 = alloc_dvec((long)0, (long)num_points  );
scratch2 = alloc_dvec((long)0, (long)num_points );
scratch3 = alloc_dvec((long)0, (long)num_points );
scratch4 = alloc_dvec((long)0, (long)num_points );
scratch6 = alloc_dvec((long)0, (long)num_points  );

// Make the diagonal elements of the tridiag matrix
for(i = 0; i < num_points; i++)
{
    ai = (double) (i);
    diag[i] = -cs * (((an - 1.) / 2. - ai)) * (((an - 1.) / 2. - ai));
    offdiag[i] = -ai * (an - ai) / 2.;
    offsq[i] = offdiag[i] * offdiag[i];
}

eps = 1.0e-13;
m11 = 1;
ip = alloc_ivec((long)0,(long) nwin);

// Call the eispac routines to invert the tridiagonal system
jtridib_(&num_points, &eps, diag, offdiag, offsq, &rlb, &rlu, &m11, &nwin,
    lam, ip, &ierr, scratch1, scratch2);

#if(DIAG1)
fprintf(stderr, "ierr=%d rlb=%.8f rlu=%.8f\n", ierr, rlb, rlu);

fprintf(stderr, "eigenvalues for the eigentapers\n");

for(k = 0; k < nwin; k++)
{
    fprintf(stderr, "%.20f ", lam[k]);
}

fprintf(stderr, "\n");
#endif
```

```
len = num_points * nwin;
evecs = alloc_dvec( (long)0, (long)len);
jtinvit_(&num_points, &num_points, diag, offdiag, offsq, &nwin, lam, ip,
evecs, &ierr, scratch1, scratch2, scratch3, scratch4, scratch6);

free(scratch1);
free(scratch2);
free(scratch3);
free(scratch4);
free(scratch6);

        /* We calculate the eigenvalues of the Dirichlet-kernel problem i.e. */
             /* the bandwidth retention factors from Slepian 1978 asymptotic */
             /* formula, gotten from Thomson 1982 eq 2.5 supplemented by the */
            /* asymptotic formula for k near 2n from Slepian 1978 eq 61 more */
           /* precise values of these parameters, perhaps useful in adaptive */
              /* spectral estimation, can be calculated explicitly using the */
       /* Rayleigh-quotient formulas in Thomson (1982) and Park et al (1987) */

dfac = (double) an *DPI * ww;
drat = (double) 8. *dfac;
dfac = (double) 4. *sqrt(DPI * dfac) * exp((double) (-2.0) * dfac);

for(k = 0; k < nwin; k++)
{
    lam[k] = (double) 1.0 - (double) dfac;
    dfac = dfac * drat / (double) (k + 1);
    /* fails as k -> 2n */
}

gamma = log((double) 8. * an * sin((double) 2. * DPI * ww)) +
    (double) 0.5772156649;

for (k = 0; k < nwin; k++)
{
    bh = -2. * DPI * (an * ww - (double) (k) /
        (double) 2. - (double) .25) / gamma;
    ell[k] = (double) 1. / ((double) 1. + exp(DPI * (double) bh));
}

for (i = 0; i < nwin; i++)
{
    lam[i] = MAX(ell[i], lam[i]);
}

          /* Normalize the eigentapers to preserve power for a white process */
                                             /* i.e. they have rms value unity */
         /* tapsum is the average of the eigentaper, should be near zero for */
                                                       /* antisymmetric tapers */

for(k = 0; k < nwin; k++)
{
    kk = (k) * num_points;
    tapsum[k] = 0.;
    tapsq = 0.;
```

```
        for(i = 0; i < num_points; i++)
        {
            aa = evecs[i + kk];
            tapers[i + kk] = aa;
            tapsum[k] = tapsum[k] + aa;
            tapsq = tapsq + aa * aa;
        }

        aa = sqrt(tapsq / (double) num_points);
        tapsum[k] = tapsum[k] / aa;

        for(i = 0; i < num_points; i++)
        {
            tapers[i + kk] = tapers[i + kk] / aa;
        }
    }

    /* Free Memory */
    free(ell);
    free(diag);
    free(offdiag);
    free(offsq);
    free(ip);
    free(evecs);

    return 1;
}

/*--------------------------------------------------------------------------------*/
/*                                      adwait                                    */
/*--------------------------------------------------------------------------------*/

int adwait(double *sqr_spec, double *dcf, double *el, int nwin, int num_freq,
    double *ares, double *degf, double avar)
{
                    /* This version uses thomson's algorithm for calculating the */
                                                   /* adaptive spectrum estimate */
    double as, das, tol, a1, scale, ax, fn, fx;
    double *spw, *bias;
    double test_tol, dif;
    int jitter, i,  k, kpoint, jloop;
    float df;
                                      /* Set tolerance for iterative scheme exit */

    #if(0)
    fprintf(stderr, "test input\n adwait: %d %d %f\n", nwin, num_freq, avar);
    fprintf(stderr, "\n Data=\n");
    for (i = 0; i < num_freq; i++)
    {
        fprintf(stderr, "%d %f \n", i, sqr_spec[i]);
    }
    #endif

    tol = 3.0e-4;
    jitter = 0;
    scale = avar;
```

```
        /* We scale the bias by the total variance of the frequency transform */
                                                     /* from zero freq to the nyquist */
        /* in this application we scale the eigenspectra by the bias to avoid */
                                                  /* possible floating point overflow */

spw = alloc_dvec((long)0,(long)nwin);
bias = alloc_dvec((long)0, (long)nwin);
for(i = 0; i < nwin; i++)
{
    bias[i] = (1.00 - el[i]);
}

#if(0)
for( i=1;i<=nwin; i++) fprintf(stderr,"%f %f\n",el[i], bias[i]);
fprintf(stderr,"\n");
#endif

// START do 100
for(jloop = 0; jloop < num_freq; jloop++)
{
    for(i = 0; i < nwin; i++)
    {
        kpoint = jloop + i * num_freq;
        spw[i] = (sqr_spec[kpoint]) / scale;
    }

    // First guess is the average of two lowest-order eigenspectral estimates
    as = (spw[0] + spw[1]) / 2.00;

    // START do 300
    // Find coefficients
    for(k = 0; k < 20; k++)
    {
        fn = 0.00;
        fx = 0.00;
        for(i = 0; i < nwin; i++)
        {
            a1 = sqrt(el[i]) * as / (el[i] * as + bias[i]);
            a1 = a1 * a1;
            fn = fn + a1 * spw[i];
            fx = fx + a1;
        }

        ax = fn / fx;
        dif = ax - as;
        das = ABS(dif);

        #if(0)
        fprintf(stderr,"adwait: jloop = %d k=%d %g %g %g %g\n",
            jloop,k, fn,fx,ax,das);
        #endif

        test_tol = das / as;
        if(test_tol < tol)
        {
            break;
        }
        as = ax;
    }
```

```
        #if(0)
        fprintf(stderr,"adwait: k=%d test_tol=%f\n",k, test_tol);
        #endif
                                                                /* end  300  */
        // Flag if iteration does not converge
        if(k >= 20)
        {
            jitter++;
        }
        ares[jloop] = as * scale;

        // Calculate degrees of freedom
        df = 0.0;
        for (i = 0; i < nwin; i++)
        {
            kpoint = jloop + i * num_freq;
            dcf[kpoint] = sqrt(el[i]) * as / (el[i] * as + bias[i]);
            df = (float)(df + dcf[kpoint] * dcf[kpoint]);
        }
                   /* We normalize degrees of freedom by the weight of the first */
       /* eigenspectrum this way we never have fewer than two degrees of freedom */

        degf[jloop] = df * 2. / (dcf[jloop] * dcf[jloop]);

    }                                                             /* end 100 */

    fprintf(stderr, "%d failed iterations\n", jitter);
    free(spw);
    free(bias);

    return jitter;
}

/*-----------------------------------------------------------------------------*/
/*                                get_F_values                                 */
/*-----------------------------------------------------------------------------*/

void get_F_values(double *sr, double *si, int nf, int nwin, float *Fvalue,
    double *b)
{
                   /* b is fft of slepian eigentapers at zero freq sr si are the */
      /* eigenspectra amu contains line frequency estimates and f-test parameter */

    double sum, sumr, sumi, sum2;
    int i, j, k;
    double *amur, *amui;
    sum = 0.;
    amur = alloc_dvec((long)0, (long)nf);
    amui = alloc_dvec((long)0, (long)nf);

    for(i = 0; i < nwin; i++)
    {
        sum = sum + b[i] * b[i];
    }
    for(i = 0; i < nf; i++)
    {
        amur[i] = 0.;
        amui[i] = 0.;

        for(j = 0; j < nwin; j++)
        {
            k = i + j * nf;
            amur[i] = amur[i] + sr[k] * b[j];
            amui[i] = amui[i] + si[k] * b[j];
        }

        amur[i] = amur[i] / sum;
        amui[i] = amui[i] / sum;
        sum2 = 0.;

        for(j = 0; j < nwin; j++)
        {
            k = i + j * nf;
            sumr = sr[k] - amur[i] * b[j];
            sumi = si[k] - amui[i] * b[j];
            sum2 = sum2 + sumr * sumr + sumi * sumi;
        }

        Fvalue[i] =
            (float)((nwin - 1) * (SQR(amui[i]) + SQR(amur[i])) * sum / sum2);
    }

    free(amur);
    free(amui);
    return;
}

/*---------------------------------------------------------------------------------*/
/*                                      HIRES                                      */
/*---------------------------------------------------------------------------------*/

int hires(double *sqr_spec, double *el, int nwin, int num_freq, double *ares)
{
    int i, j, k, kpoint;
    float a;

    for(j = 0; j < num_freq; j++)
    {
                                                                         ares[j] = 0.;
    }

    for(i = 0; i < nwin; i++)
    {
        k = i * num_freq;
        a = (float)(1. / (el[i] * nwin));

        for(j = 0; j < num_freq; j++)
        {
            kpoint = j + k;
            ares[j] = ares[j] + a * (sqr_spec[kpoint]);
        }
    }
    for(j = 0; j < num_freq; j++)
    {
        if (ares[j] > 0.0)
        ares[j] = sqrt(ares[j]);
        else
        printf("sqrt problem in hires pos=%d %f\n", j, ares[j]);
    }

    return 1;
}

/*--------------------------------------------------------------------------------*/
/*                                  jtinvit                                       */
/*--------------------------------------------------------------------------------*/

#include <math.h>
#define abs(x) ((x) >= 0 ? (x) : -(x))
#define dabs(x) (double)abs(x)
#define dmin(a,b) (double)min(a,b)
#define dmax(a,b) (double)max(a,b)

int jtinvit_(int *nm, int *n, double *d, double *e, double *e2, int *m,
    double *w, int *ind, double *z, int *ierr, double *rv1, double *rv2,
    double *rv3, double *rv4, double *rv6)
{
    // Initialized data
    static double machep = 1.25e-15;

    // System generated locals
    int z_dim1, z_offset, i1, i2, i3;
    double d1, d2;

    // Builtin functions
    double sqrt();

    // Local variables
    static double   norm;
    static int      i, j, p, q, r, s;
    static double   u, v, order;
    static int      group;
    static double   x0, x1;
    static int      ii, jj, ip;
    static double   uk, xu;
    static int      tag, its;
    static double   eps2, eps3, eps4;
    static double   rtem;

    /* This subroutine is a translation of the inverse iteration technique       */
    /* in the Algol procedure tristurm by Peters and Wilkinson                   */
    /* Handbook for Auto. Comp., Vol.II-Linear Algebra, 418-439(1971).           */
    /* This subroutine finds those eigenvectors of a tridiagonal symmetric       */
    /* matrix corresponding to specified eigenvalues using inverse iteration.    */
    /* On input:                                                                 */
    /* nm must be set to the row dimension of two-dimensional array              */
    /* parameters as declared in the calling program dimension statement         */
    /* n is the order of the matrix                                              */
    /* d contains the diagonal elements of the input matrix                      */
    /* e contains the subdiagonal elements of the input matrix in its last       */
    /* n-1 positions                                                             */
    /* e(1) is arbitrary                                                         */
    /* e2 contains the squares of the corresponding elements of e, with          */
    /* zeros corresponding to negligible elements of e                           */
    /* e(i) is considered negligible if it is not larger than the product        */
    /* of the relative machine precision and the sum of the magnitudes of        */
    /* d(i) and d(i-1).                                                          */
    /* e2(1) must contain 0.0d0 if the eigenvalues are in ascending order,       */
    /* or 2.0d0 if the eigenvalues are in descending order.  If  bisect,         */
    /* tridib, or imtqlv has been used to find the eigenvalues, their output     */
    /* e2 array is exactly what is expected here                                 */
    /* m is the number of specified eigenvalues                                  */
    /* w contains the m eigenvalues in ascending or descending order             */
    /* ind contains in its first m positions the submatrix indices               */
    /* associated with the corresponding eigenvalues in w --                     */
    /* 1 for eigenvalues belonging to the first submatrix from                   */
    /* the top, 2 for those belonging to the second submatrix, etc.              */
    /* On output:                                                                */
    /* all input arrays are unaltered                                            */
    /* z contains the associated set of orthonormal eigenvectors.                */
    /* any vector which fails to converge is set to zero                         */
    /* ierr is set to:                                                           */
    /* zero or normal return,                                                    */
    /* -r if the eigenvector corresponding to the r-th                           */
    /* eigenvalue fails to converge in 5 iterations                              */
    /*     rv1, rv2, rv3, rv4, and rv6 are temporary storage arrays.             */
    /* machep is a machine dependent parameter specifying                        */
    /* the relative precision of floating point arithmetic.                      */
    /* machep = 16.0d0**(-13) for long form arithmetic on s360                   */
    /* for f_floating dec fortran data machep/1.1d-16/                           */
    /* for g_floating dec fortran                                                */
    /*                                                                           */
    /* Questions and comments should be directed to B. S. Garbow,                */
    /* Applied Mathematics Division, Argonne National Laboratory                 */

    // Parameter adjustments
    --rv6;
    --rv4;
    --rv3;
    --rv2;
    --rv1;
    --e2;
    --e;
    --d;
    z_dim1 = *nm;
    z_offset = z_dim1 + 1;
    z -= z_offset;
    --ind;
    --w;

    // Function body
    *ierr = 0;
    if (*m == 0) {
        goto L1001;
    }
    tag = 0;
    order = 1. - e2[1];
    q = 0;
    /* establish and process next submatrix */
    L100:
    p = q + 1;
    i1 = *n;
```

```
for (q = p; q <= i1; ++q) {
    if (q == *n) {
        goto L140;
    }
    if (e2[q + 1] == 0.) {
        goto L140;
    }
}

/* find vectors by inverse iteration */
L140:
++tag;
s = 0;
i1 = *m;
for (r = 1; r <= i1; ++r) {
    if (ind[r] != tag) {
        goto L920;
    }
    its = 1;
    x1 = w[r];
    if (s != 0) {
        goto L510;
    }
    /* check for isolated root */
    xu = 1.;
    if (p != q) {
        goto L490;
    }
    rv6[p] = 1.;
    goto L870;
    L490:
    norm = (d1 = d[p], abs(d1));
    ip = p + 1;
        i2 = q;
    for (i = ip; i <= i2; ++i) {
        norm = norm + (d1 = d[i], abs(d1)) + (d2 = e[i], abs(d2));
    }
    /* eps2 is the criterion for grouping, */
    /* eps3 replaces zero pivots and equal */
    /* roots are modified by eps3, */
    /* eps4 is taken very small to avoid overflow */
    eps2 = norm * .001;
    eps3 = machep * norm;
    uk = (double) (q - p + 1);
    eps4 = uk * eps3;
    uk = eps4 / sqrt(uk);
    s = p;
    L505:
    group = 0;
    goto L520;
    /* look for close or coincident roots */
    L510:
    if ((d1 = x1 - x0, abs(d1)) >= eps2) {
        goto L505;
    }
    ++group;
    if (order * (x1 - x0) <= 0.) {
        x1 = x0 + order * eps3;
    }
```

```
/* elimination with interchanges and initialization of vector */
L520:
v = 0.;
i2 = q;
for (i = p; i <= i2; ++i) {
    rv6[i] = uk;
    if (i == p) {
        goto L560;
    }
    if ((d1 = e[i], abs(d1)) < abs(u)) {
        goto L540;
    }
    /* warning -- a divide check may occur here if */
    /* e2 array has not been specified correctly   */
    xu = u / e[i];
    rv4[i] = xu;
    rv1[i - 1] = e[i];
    rv2[i - 1] = d[i] - x1;
    rv3[i - 1] = 0.;
    if (i != q) {
        rv3[i - 1] = e[i + 1];
    }
    u = v - xu * rv2[i - 1];
    v = -xu * rv3[i - 1];
    goto L580;
    L540:
    xu = e[i] / u;
    rv4[i] = xu;
   rv1[i - 1] = u;
    rv2[i - 1] = v;
    rv3[i - 1] = 0.;
    L560:
    u = d[i] - x1 - xu * v;
    if (i != q) {
        v = e[i + 1];
    }
    L580:
    ;
}
if (u == 0.) {
    u = eps3;
}
rv1[q] = u;
rv2[q] = 0.;
rv3[q] = 0.;
/* back substitution for i=q step -1 until p do -- */
L600:
i2 = q;
for (ii = p; ii <= i2; ++ii) {
    i = p + q - ii;
    rtem = rv6[i] - u * rv2[i] - v * rv3[i];
    rv6[i] = (rtem) / rv1[i];
    v = u;
    u = rv6[i];
}
```

```
/* orthogonalize with respect to previous members of group */
if (group == 0) {
    goto L700;
}
j = r;
i2 = group;
for (jj = 1; jj <= i2; ++jj) {
    L630:
    --j;
    if (ind[j] != tag) {
        goto L630;
    }
    xu = 0.;
    i3 = q;
    for (i = p; i <= i3; ++i) {
        /* L640: */
        xu += rv6[i] * z[i + j * z_dim1];
    }
    i3 = q;
    for (i = p; i <= i3; ++i) {
        rv6[i] -= xu * z[i + j * z_dim1];
    }
}
L700:
norm = 0.;
i2 = q;
for (i = p; i <= i2; ++i) {
    norm += (d1 = rv6[i], abs(d1));
}
if (norm >= 1.) {
    goto L840;
}
/* forward substitution */
if (its == 5) {
    goto L830;
}
if (norm != 0.) {
    goto L740;
}
rv6[s] = eps4;
++s;
if (s > q) {
    s = p;
}
goto L780;
L740:
xu = eps4 / norm;
i2 = q;
for (i = p; i <= i2; ++i) {
    rv6[i] *= xu;
}
/* elimination operations on next vector */
/* iterate */
L780:
i2 = q;
for (i = ip; i <= i2; ++i) {
    u = rv6[i];
```

```
             /* if rv1(i-1) .eq. e(i), a row interchange was performed earlier */
             /* in the triangularization process */
            if (rv1[i - 1] != e[i]) {
                goto L800;
            }
            u = rv6[i - 1];
            rv6[i - 1] = rv6[i];
            L800:
            rv6[i] = u - rv4[i] * rv6[i - 1];
        }
        ++its;
        goto L600;
        /* set error -- non-converged eigenvector */
        L830:
        *ierr = -r;
        xu = 0.;
        goto L870;
        /* normalize so that sum of squares is 1 and expand to full order */
        L840:
        u = 0.;
        i2 = q;
        for (i = p; i <= i2; ++i) {
            /* Computing 2nd power */
            d1 = rv6[i];
            u += d1 * d1;
        }
        xu = 1. / sqrt(u);
        L870:
        i2 = *n;
        for (i = 1; i <= i2; ++i) {
            z[i + r * z_dim1] = 0.;
        }
        i2 = q;
        for (i = p; i <= i2; ++i) {
            z[i + r * z_dim1] = rv6[i] * xu;
        }
        x0 = x1;
        L920:
        ;
    }
    if (q < *n) {
        goto L100;
    }
    L1001:
    return 0;
}
```

```
/*---------------------------------------------------------------------------*/
/*                              jtridib                                      */
/*---------------------------------------------------------------------------*/

int jtridib_(int *n, double *eps1, double *d, double *e, double *e2, double *lb,
    double *ub, int *m11, int *m, double *w, int *ind, int *ierr, double *rv4,
    double *rv5)
{
    // Initialized data
    static double machep = 1.25e-15;

    // System generated locals
    int i1, i2;
    double d1, d2, d3;

    // Local variables
    static int i, j, k, l, p, q, r, s;
    static double u, v;
    static int m1, m2;
    static double t1, t2, x0, x1;
    static int m22, ii;
    static double xu;
    static int isturm, tag;

    /* This subroutine is a translation of the Algol procedure bisect,          */
    /* Num. Math. 9, 386-393(1967) by Barth, Martin, and Wilkinson.             */
    /* Handbook for Auto. Comp., Vol.II-Linear Algebra, 249-256(1971).          */
    /* This subroutine finds those eigenvalues of a tridiagonal symmetric       */
    /* matrix between specified boundary indices, using bisection.              */
    /* On input:                                                                */
    /*     n is the order of the matrix                                         */
    /*     eps1 is an absolute error tolerance for the computed eigenvalues.    */
    /*     if the input eps1 is non-positive, it is reset for each submatrix    */
    /*     to a default value, namely, minus the product of the relative machine */
    /*     precision and the 1-norm of the submatrix                            */
    /*     d contains the diagonal elements of the input matrix                 */
    /*     e contains the subdiagonal elements of the input matrix              */
    /*     in its last n-1 positions.                                           */
    /*     e(1) is arbitrary                                                    */
    /*     e2 contains the squares of the corresponding elements of e.          */
    /*     e2(1) is arbitrary                                                   */
    /*     m11 specifies the lower boundary index for the desired eigenvalues   */
    /*     m specifies the number of eigenvalues desired.  The upper boundary   */
    /*     index m22 is then obtained as m22=m11+m-1.                           */
    /* On output:                                                               */
    /*     eps1 is unaltered unless it was reset to its (last) default value    */
    /*     d and e are unaltered elements of e2, corresponding to elements of e  */
    /*     regarded as negligible, have been replaced by zero causing the       */
    /*     matrix to split into a direct sum of submatrices.                    */
    /*     e2(1) is also set to zero                                            */
    /*     lb and ub define an interval containing exactly the desired          */
    /*     eigenvalues                                                          */
    /*     w contains, in its first m positions, the eigenvalues between        */
    /*     indices m11 and m22 in ascending order                               */
    /*     ind contains in its first m positions the submatrix indices          */
    /*     associated with the corresponding eigenvalues in w --                */
    /*     1 for eigenvalues belonging to the first submatrix from              */
    /*     the top, 2 for those belonging to the second submatrix, etc.         */
    /*     ierr is set to:                                                      */
    /*         zero for normal return                                           */
    /*          3*n+1 if multiple eigenvalues at index m11 make unique           */
    /*          selection impossible                                             */
    /*          3*n+2 if multiple eigenvalues at index m22 make unique           */
    /*          selection impossible                                             */
    /* rv4 and rv5 are temporary storage arrays                                  */
    /* Note that subroutine tql1, imtql1, or tqlrat is generally faster than     */
    /* tridib, if more than n/4 eigenvalues are to be found.                     */
    /* machep is a machine dependent parameter specifying the relative precision */
    /* of floating point arithmetic.                                             */
    /* machep = 16.0d0**(-13) for long form arithmetic on s360                   */
    /* for f_floating dec fortran                                                */
    /* data machep/1.1d-16/ for g_floating dec fortran                           */
    /*                                                                           */
    /* Questions and comments should be directed to B. S. Garbow,                */
    /* Applied Mathematics Division, Argonne National Laboratory                  */

    // Parameter adjustments
    --rv5;
    --rv4;
    --e2;
    --e;
    --d;
    --ind;
    --w;

    /* Function Body */

    *ierr = 0;
    tag = 0;
    xu = d[1];
    x0 = d[1];
    u = 0.;
    /* look for small sub-diagonal entries and determine an */
    /* interval containing all the eigenvalues */
    i1 = *n;
    for (i = 1; i <= i1; ++i) {
        x1 = u;
        u = 0.;
        if (i != *n) {
            u = (d1 = e[i + 1], abs(d1));
        }
        /* Computing MIN */
        d1 = d[i] - (x1 + u);
        xu = min(d1, xu);
        /* Computing MAX */
        d1 = d[i] + (x1 + u);
        x0 = max(d1, x0);
        if (i == 1) {
            goto L20;
        }
        if ((d1 = e[i], abs(d1)) > machep * ((d2 = d[i], abs(d2)) + (
            d3 = d[i - 1], abs(d3)))) {
        goto L40;
        }
        L20:
        e2[i] = 0.;
        L40:
        ;
    }
```

```
/* Computing MAX */
d1 = abs(xu), d2 = abs(x0);
x1 = max(d1, d2) * machep * (double) (*n);
xu -= x1;
t1 = xu;
x0 += x1;
t2 = x0;
/* determine an interval containing exactly the desired eigenvalues */
p = 1;
q = *n;
m1 = *m11 - 1;
if (m1 == 0) {
    goto L75;
}
isturm = 1;
L50:
v = x1;
x1 = xu + (x0 - xu) * .5;
if (x1 == v) {
    goto L980;
}
goto L320;
L60:
if ((i1 = s - m1) < 0) {
    goto L65;
} else if (i1 == 0) {
    goto L73;
} else {
    goto L70;
}
L65:
xu = x1;
goto L50;
L70:
x0 = x1;
goto L50;
L73:
xu = x1;
t1 = x1;
L75:
m22 = m1 + *m;
if (m22 == *n) {
    goto L90;
}
x0 = t2;
isturm = 2;
goto L50;
L80:
if ((i1 = s - m22) < 0) {
    goto L65;
} else if (i1 == 0) {
    goto L85;
} else {
    goto L70;
}
L85:
t2 = x1;
L90:
q = 0;
r = 0;
```

```
/* establish and process next submatrix, refining */
/* interval by the gerschgorin bounds */
L100:
if (r == *m) {
    goto L1001;
}
++tag;
p = q + 1;
xu = d[p];
x0 = d[p];
u = 0.;
i1 = *n;
for (q = p; q <= i1; ++q) {
    x1 = u;
    u = 0.;
    v = 0.;
    if (q == *n) {
        goto L110;
    }
    u = (d1 = e[q + 1], abs(d1));
    v = e2[q + 1];
    L110:
    /* Computing MIN */
    d1 = d[q] - (x1 + u);
    xu = min(d1, xu);
    /* Computing MAX */
    d1 = d[q] + (x1 + u);
    x0 = max(d1, x0);
    if (v == 0.) {
        goto L140;
    }
}
L140:
/* Computing MAX */
d1 = abs(xu), d2 = abs(x0);
x1 = max(d1, d2) * machep;
if (*eps1 <= 0.) {
    *eps1 = -x1;
}
if (p != q) {
    goto L180;
}
/* check for isolated root within interval */
if (t1 > d[p] || d[p] >= t2) {
    goto L940;
}
m1 = p;
m2 = p;
rv5[p] = d[p];
goto L900;
L180:
x1 *= (double) (q - p + 1);
/* Computing MAX */
d1 = t1, d2 = xu - x1;
*lb = max(d1, d2);
/* Computing MIN */
d1 = t2, d2 = x0 + x1;
*ub = min(d1, d2);
x1 = *lb;
isturm = 3;
```

```
goto L320;
L200:
m1 = s + 1;
x1 = *ub;
isturm = 4;
goto L320;
L220:
m2 = s;
if (m1 > m2) {
    goto L940;
}
/* find roots by bisection */
x0 = *ub;
isturm = 5;
i1 = m2;
for (i = m1; i <= i1; ++i) {
    rv5[i] = *ub;
    rv4[i] = *lb;
}
/* loop for k-th eigenvalue */
/* for k=m2 step -1 until m1 do */
/* do not used to legalize computed go to */
k = m2;
L250:
xu = *lb;
/* for i=k step -1 until m1 do */
i1 = k;
for (ii = m1; ii <= i1; ++ii) {
    i = m1 + k - ii;
    if (xu >= rv4[i]) {
        goto L260;
    }
    xu = rv4[i];
    goto L280;
    L260:
    ;
}
L280:
if (x0 > rv5[k]) {
    x0 = rv5[k];
}
/* next bisection step */
L300:
x1 = (xu + x0) * .5;
if (x0 - xu <= machep * 2. * (abs(xu) + abs(x0)) + abs(*eps1)) {
    goto L420;
}
/* in-line procedure for sturm sequence */
L320:
s = p - 1;
u = 1.;
    i1 = q;
for (i = p; i <= i1; ++i) {
    if (u != 0.) {
        goto L325;
    }
    v = (d1 = e[i], abs(d1)) / machep;
    if (e2[i] == 0.) {
        v = 0.;
    }
```

```
        goto L330;
        L325:
        v = e2[i] / u;
        L330:
        u = d[i] - x1 - v;
        if (u < 0.) {
            ++s;
        }
    }
    switch ((int) isturm) {
    case 1:
        goto L60;
    case 2:
        goto L80;
    case 3:
        goto L200;
    case 4:
        goto L220;
    case 5:
        goto L360;
    }
    /* refine intervals */
    L360:
    if (s >= k) {
        goto L400;
    }
    xu = x1;
    if (s >= m1) {
        goto L380;
    }
    rv4[m1] = x1;
    goto L300;
    L380:
    rv4[s + 1] = x1;
    if (rv5[s] > x1) {
        rv5[s] = x1;
    }
    goto L300;
    L400:
    x0 = x1;
    goto L300;
    /* k-th eigenvalue found */
    L420:
    rv5[k] = x1;
    --k;
    if (k >= m1) {
        goto L250;
    }
    /* order eigenvalues tagged with their submatrix associations */
    L900:
    s = r;
    r = r + m2 - m1 + 1;
    j = 1;
    k = m1;
        i1 = r;
    for (l = 1; l <= i1; ++l) {
        if (j > s) {
            goto L910;
        }
        if (k > m2) {
            goto L940;
        }
        if (rv5[k] >= w[l]) {
            goto L915;
        }
        i2 = s;
        for (ii = j; ii <= i2; ++ii) {
            i = l + s - ii;
            w[i + 1] = w[i];
            ind[i + 1] = ind[i];
        }
        L910:
        w[l] = rv5[k];
        ind[l] = tag;
        ++k;
        goto L920;
        L915:
        ++j;
        L920:
        ;
    }
    L940:
    if (q < *n) {
        goto L100;
    }
    goto L1001;
    /*  set error -- interval cannot be found containing */
    /* exactly the desired eigenvalues */
    L980:
    *ierr = *n * 3 + isturm;
    L1001:
    *lb = t1;
    *ub = t2;
    return 0;
}

/*--------------------------------------------------------*/
/*---------------get_pow_2--------------------------------*/
/*--------------------------------------------------------*/
int
get_pow_2(int inum)
{
    int j, klength;
    /* find smallest power of 2 that encompasses the data */

    for (j = 1; pow((double) 2, (double) j) < inum; j++);
    return klength = (int)(pow((double) 2, (double) j));
}
```

```
/*--------------------------------------------------------*/
/*-----------remove_mean-----------------------------------*/
/*--------------------------------------------------------*/

double remove_mean(float *x, int lx)
{
    int k;
    double mean;
    mean = 0.;

    if (lx < 2)
        return mean;

    for (k = 0; k <= lx; k++) {
        mean = x[k] + mean;
    }

    mean = mean / (float) lx;
    for (k = 0; k <= lx; k++) {
        x[k] = (float)(x[k] - mean);
    }

    return mean;
}

/*--------------------------------------------------------*/
/*-----------zero_pad-------------------------------   ---*/
/*--------------------------------------------------------*/

void zero_pad(float output[], int start, int olength)
{
    int i;
    for (i = start; i < olength; i++) {
        output[i] = 0.0;
    }
}

/*--------------------------------------------------------*/
/*-----------get_cos_taper---------------------------------*/
/*--------------------------------------------------------*/

float get_cos_taper(int n, int k)
{
    int l;
    float vwin;
    vwin = 0.0;

    if (k < 0 || k > n)
        return vwin;
    vwin = 1.0;

    l = (n - 2) / 10;
    if (k <= l)
        vwin = (float)(0.5 * (1.0 - cos(k * PI / (l + 1))));
    if (k >= n - l - 2)
        vwin = (float)(0.5 * (1.0 - cos((n - k - 1) * PI / (l + 1))));

    return vwin;
}
```

```
/*************************************************************************************/
/*                                                                                   */
/* File JREALFT.C                                                                    */
/*                                                                                   */
/*************************************************************************************/

#include <math.h>

void jrealft(float data[], unsigned long n, int isign)
{
    void jfour1(float data[], unsigned long nn, int isign);
    unsigned long i,i1,i2,i3,i4,np3;
    float c1=0.5,c2,h1r,h1i,h2r,h2i;
    double wr,wi,wpr,wpi,wtemp,theta;

    theta=3.141592653589793/(double) (n>>1);
    if (isign == 1) {
        c2 = -0.5;
        jfour1(data,n>>1,1);
    } else {
    c2=0.5;
    theta = -theta;
    }
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0+wpr;
    wi=wpi;
    np3=n+3;
    for (i=2;i<=(n>>2);i++) {
        i4=1+(i3=np3-(i2=1+(i1=i+i-1)));
        h1r=c1*(data[i1]+data[i3]);
        h1i=c1*(data[i2]-data[i4]);
        h2r = -c2*(data[i2]+data[i4]);
        h2i=c2*(data[i1]-data[i3]);
        data[i1]=(float)(h1r+wr*h2r-wi*h2i);
        data[i2]=(float)(h1i+wr*h2i+wi*h2r);
        data[i3]=(float)(h1r-wr*h2r+wi*h2i);
        data[i4]=(float)(-h1i+wr*h2i+wi*h2r);
        wr=(wtemp=wr)*wpr-wi*wpi+wr;
        wi=wi*wpr+wtemp*wpi+wi;
    }
    if (isign == 1) {
        data[1] = (h1r=data[1])+data[2];
        data[2] = h1r-data[2];
    } else {
        data[1]=c1*((h1r=data[1])+data[2]);
        data[2]=c1*(h1r-data[2]);
        jfour1(data,n>>1,-1);
    }
}
```

```
/************************************************************************************/
/*                                                                                  */
/* File JFOUR1.C                                                                    */
/*                                                                                  */
/************************************************************************************/

#include <math.h>
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void jfour1(float data[], unsigned long nn, int isign)
{
    unsigned long n,mmax,m,j,istep,i;
    double wtemp,wr,wpr,wpi,wi,theta;
    float tempr,tempi;

    n=nn << 1;
    j=1;
    for (i=1;i<n;i+=2) {
        if (j > i) {
            SWAP(data[j],data[i]);
            SWAP(data[j+1],data[i+1]);
        }
        m=n >> 1;
        while (m >= 2 && j > m) {
            j -= m;
            m >>= 1;
        }
        j += m;
    }
    mmax=2;
    while (n > mmax) {
        istep=mmax << 1;
        theta=isign*(6.28318530717959/mmax);
        wtemp=sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
        wpi=sin(theta);
        wr=1.0;
        wi=0.0;
        for (m=1;m<mmax;m+=2) {
            for (i=m;i<=n;i+=istep) {
                j=i+mmax;
                tempr=(float)(wr*data[j]-wi*data[j+1]);
                tempi=(float)(wr*data[j+1]+wi*data[j]);
                data[j]=data[i]-tempr;
                data[j+1]=data[i+1]-tempi;
                data[i] += tempr;
                data[i+1] += tempi;
            }
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
        }
        mmax=istep;
    }
}
#undef SWAP
```

The FrequenC Library	
NAME: ModTotvarBias	**FUNCTION:** Description of what function does
SYNOPSIS: int FunctionName(F_TYPE z[], F_TYPE one, F_TYPE two)	

F_TYPE z[]	Phase or frequency data array
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysios end point

F_TYPE one	parameter one, and description of what it is
F_TYPE two	parameter two, and description of what it is
RETURN: int	# non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are modified.
Embedded zeros are treated as gaps in phase data.
All zeros are treated as gaps in frequency data.

EXAMPLE:
```
#include "frequenc.h"                           /* FrequenC header file */
F_TYPE z[512];                                            /* data array */
F_TYPE one;                                            /* parameter one */
F_TYPE two;                                            /* parameter two */
int num;                                   /* # data points processed */
.
.
one=1.0;                                   /* set 1st parameter value */
two=2.0;                                   /* set 2nd parameter value */
num=FunctionName(z, one, two);                         /* call function */
printf("\n# Data Points Scaled = %d", num);              /* display num */
```

SEE ALSO: RelatedFunctionName()

REFERENCE: NIST Technical Note 1337

```
/*****************************************************************************/
/*                                                                           */
/*                              ModTotvarBias()                              */
/*                                                                           */
/*      Function to calculate the Mod Totvar bias function                   */
/*                                                                           */
/*      Parameters:     int   nAlpha = noise type (-2 to +2)                 */
/*                                                                           */
/*                                                                           */
/*      Return:         float fBias  = Mod Totvar bias factor                */
/*                                     or -1 if error                        */
/*                                                                           */
/*      Alpha     Noise     MTOT/MVAR                                        */
/*       +2        W PM        0.94                                          */
/*       +1        F PM        0.83                                          */
/*        0        W FM        0.73                                          */
/*       -1        F FM        0.70                                          */
/*       -2       RW FM        0.69                                          */
/*                                                                           */
/*      The Mod Totvar bias factor is the ratio of the expected value of     */
/*      MTOT to MVAR.  To correct a calculated Mod Totdev value for a        */
/*      certain noise type, divide the uncorrected Mod Totdev value by the   */
/*      square root of the bias factor returned by this function.            */
/*                                                                           */
/*      Notes:          (1) This function applies to -2 >= alpha >= 2        */
/*                          (W PM, F PM, W FM, F FM, RW FM noise)            */
/*                      (2) This function applies for all T/tau              */
/*                                                                           */
/*      Reference:      E-mail from D.A. Howe/NIST,  3/13/00                 */
/*                                                                           */
/*      Revision record:                                                     */
/*          03/25/00    Created                                              */
/*          05/05/00    Corrected values per D. Howe e-mail                  */
/*                                                                           */
/* (c) Copyright 2000   Hamilton Technical Services   All Rights Reserved    */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL ModTotvarBias(int nAlpha)
{
    // Local variable
    float fBias;

    // Verify parameter
    if(nAlpha<-2 || nAlpha>2)
    {
        return (float) -1.0; // Error
    }

    // Sort by noise type
    switch(nAlpha)
    {
        case -2: // RW FM
        {
            fBias=(float) 0.69;
        }
        break;
        case -1: // F FM
        {
            fBias=(float) 0.70;
        }
        break;

        case -0: // W FM
        {
            fBias=(float) 0.73;
        }
        break;

        case 1: // F PM
        {
            fBias=(float) 0.83;
        }
        break;

        case 2: // W PM
        {
            fBias=(float) 0.94;
        }
        break;
    }

    return (fBias);
}
```

The FrequenC Library	
NAME: ModTotvarCalc	**FUNCTION:** Description of what function does
SYNOPSIS: int FunctionName(F_TYPE z[], F_TYPE one, F_TYPE two)	

F_TYPE z[]	Phase or frequency data array
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysis end point

F_TYPE one	parameter one, and description of what it is
F_TYPE two	parameter two, and description of what it is
RETURN: int	# non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are modified.
Embedded zeros are treated as gaps in phase data.
All zeros are treated as gaps in frequency data.

EXAMPLE:

```
#include "frequenc.h"                        /* FrequenC header file */
F_TYPE z[512];                                         /* data array */
F_TYPE one;                                         /* parameter one */
F_TYPE two;                                         /* parameter two */
int num;                                  /* # data points processed */
.
.
one=1.0;                                  /* set 1st parameter value */
two=2.0;                                  /* set 2nd parameter value */
num=FunctionName(z, one, two);                      /* call function */
printf("\n# Data Points Scaled = %d", num);           /* display num */
```

SEE ALSO: RelatedFunctionName()

REFERENCE: NIST Technical Note 1337

```
/***************************************************************************/
/*                                                                         */
/*                             ModTotvarCalc()                             */
/*                                                                         */
/*      Function to calculate the Modified Total Variance from phase data  */
/*      Uses uninverted even reflection of offset-removed phase data       */
/*                                                                         */
/*      Parameters:     F_TYPE x[]      = phase data                       */
/*                                        x[0] = # data points             */
/*                                        x[1] = analysis start            */
/*                                        x[2] = analysis end              */
/*                      F_TYPE *fMTDev  = pointer to Mod Totdev            */
/*                      F_TYPE fTau     = interval between phase data points */
/*                      int nAF         = averaging factor                 */
/*                      BOOL *bAbort    = pointer to abort flag            */
/*                      BOOL bProgress  = progress indicator flag          */
/*                      void *ProgressFunction()                           */
/*                                      = pointer to progress indicator    */
/*                                        funct with args int nPercentDone */
/*                                        and char *szMessage              */
/*                                                                         */
/*      Return:         int           = # Mod Totvar analysis points,      */
/*                                      or -1 if memory alloc error        */
/*                                      or -2 if no result error           */
/*                                                                         */
/*      Notes:          1. Calculation is done entirely with a new array.  */
/*                         that is deleted after this function closes.     */
/*                      2. Progress indicator and abort not necessary.     */
/*                      3. The phase data need not be endmatched before    */
/*                         calling this function.                          */
/*                                                                         */
/*      Description:    Given the phase data array x[], Mod Totvar is      */
/*                      calculated using z[], which is computed from a     */
/*                      subsequence of x[] consisting of 3m points, where  */
/*                      m is the averaging factor nAF.  This subsequence is */
/*                      given by x[i] where i=n thru n+3m-1, where n is an */
/*                      overall index that goes from 1 to the # of phase   */
/*                      data points nNum.  The subsequence z[i] is found   */
/*                      from x[i] by removing the frequency offset by      */
/*                      subtracting the slope found by averaging the 1st   */
/*                      and last halves of the subsequence and dividing by */
/*                      half the interval.                                 */
/*                                                                         */
/*                      This subsequence is then extended at both ends by  */
/*                      an uninverted even reflection according to z(n-k)= */
/*                      z(n+k-1) and z(n+3m+k-1)=z(n+3m-k) for k=1 thru 3m, */
/*                      thus creating a new subsequence having a tripled   */
/*                      range from i=n-3m to i=n+6m-1.                     */
/*                                                                         */
/*                      Mod Totvar is then calculated as the average of its */
/*                      nNum-3m Totvar sub-estimates.                      */
/*                                                                         */
/*      Reference:      D.A. Howe and F. Vernotte, "Generalization of the  */
/*                      Total Variance Approach to the Modified Allan      */
/*                      Variance", Proc. 31th PTTI Meeting, Dec. 9, 1999,  */
/*                      (to be published).                                 */
/*                                                                         */
/*      Revision record:                                                   */
/*          12/12/99    Adapted from TotvarCalc() and CalcPhaseModSigma()  */
/*                      of FrequenC Library.                               */
/*          12/16/99    Basic code drafted - no gaps or analysis limits      */
/*          12/26/99    Changed progress bar argument from lpfn to flag      */
/*          12/30/99    Fixed & cleaned up a few details                     */
/*          02/06/00    Changed to fully overlapping per D. Howe review      */
/*          02/08/00    Changed inner sum index limit from 6m+1 to 6m to     */
/*                      elim duplicate term, and change divisor to 6m        */
/*          02/12/00    Added trace code to aid debugging                    */
/*          02/23/00    Added alternative freq offset removal method         */
/*          05/06/00    Added code to handle analysis limits                 */
/*                      Not sure about gaps - no specific code for them      */
/*                      But seems like they S/B OK as-is                     */
/*          03/09/03    Added abort and progress args                        */
/*                      Changed from GlobalAllocPtr() to malloc()            */
/*                                                                           */
/* (c) Copyright 1999-2003  Hamilton Technical Services  All Rights Reserved */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL ModTotvarCalc(F_TYPE fX[], F_TYPE *fMTDev,
    F_TYPE fTau, int nAF, BOOL *bAbort, BOOL bProgress,
    void *ProgressFunction(int nPercentDone, char *szMessage))
{
    // Local variables
    int i;              // Main index
    int j;              // Auxilary index
    int k;              // Inner index
    int m;              // Alias for averaging factor nAF
    int nNum;           // # points
    int nStart;         // Analysis start
    int nCount=0;       // # analysis points
    int nTic;           // Increment for progress indicator
    int nLast_i=0;      // Previous value of index

    double *fZ;         // Subsequence array pointer
    double fC1;         // Subsequence 1st half average
    double fC2;         // Subsequence 2nd half average
    double fC;          // Phase slope=frequency offset
    double fSum=0;      // Overall Mod Totvar summation
    double fSubsum=0;   // Subsequence Mod Totvar summation

    double fZ1;         // 1st phase average
    double fZ2;         // 2nd phase average
    double fZ3;         // 3rd phase average

    // Initializations
    nStart=(int)fX[1];          // Analysis start point
    nNum=(int)(fX[2]-fX[1]+1);  // # analysis phase data points = N
    m=nAF;                      // Averaging factor

    nTic=(int)ceil(nNum/100);   // Increment for progress report

    // Start
    if(bProgress)
    {
        // Label progress bar
        ProgressFunction(0, "Calculating Mod Totdev");
    }
```

```
// Allocate 9m points of fZ[] working memory with index from 0 to 9m-1
// w/o array header (#, Start, End)
if((fZ=(double*)malloc((9*m)*sizeof(double)))==NULL)
{
    return(-1);  // Error - memory reallocation failed
}

// Outer loop thru all nNum-3m+1 subsequences of 3m phase data points
// A subsequence will be processed during each iteration
// and the average used to find Mod-Totvar
// This should allow a max m of nNum/3
// Note: PTTI Paper uses only nNum-3m subsequences
// but at my suggestion the "official" version has now been expanded
// to use nNum-3m+1 as done here

// To handle analysis limits, need to initialize index to nStart-1 not 0
// and end it before nStart-1-3*m+1 instead of nNum-3*m+1
for(i=nStart-1; i<nStart-1+nNum-3*m+1; i++)
{
    // Copy data subsequence into working array starting at index 3m
    // Note:  The fX[] array needs offset for its header info
    for(j=0; j<3*m; j++)
    {
        fZ[3*m+j]=fX[i+j+ARRAY_OFFSET];
    }

    // Do removal of frequency offset
    // Find slope fC by averaging the 1st and last halves of the subsequence
    // subtracting them, and dividing by half the interval
    fC1=0;
    fC2=0;
    for(j=0; j<3*m/2; j++)
    {
        fC1+=fZ[3*m+j];
        fC2+=fZ[3*m+(int)ceil(3.0*(double)m/2.0)+j];
    }
    fC1/=(double)(3*m/2);
    fC2/=(double)(3*m/2);
    fC=(fC2-fC1)/ceil(3.0*(double)m/2.0);

    // Then subtract slope from each point
    for(j=0; j<3*m; j++)
    {
        fZ[3*m+j]-=fC*(double)j;
    }

    // Do uninverted even reflection at bottom and top
    for(j=0; j<3*m; j++)
    {
        // Bottom reflection
        fZ[j]=fZ[6*m-1-j];

        // Top reflection
        fZ[6*m+j]=fZ[j];
    }

    // Calc Mvar for the fZ[] subsequence
    // Will need to add gap testing and analysis limits
    // This data array does not need an offset
    // This code implements the right summation of Eq(8) of the reference
```

```
        // Initialize inner sum
        fSubsum=0;

        // We have a doubly-extended subsequence fZ[] of 9m points
        // and are going to form m-point averages
        // in 6m+1 groups of 3 (called fZ1, fZ2 and fZ3)
        // from which all possible fully overlapping 2nd differences
        // will be used to calculate Mod Totvar
        // Note: Only 6m of the 6m+1 possible 2nd differences are used
        // because the last one is a duplicate
        // NOTE: For 6m+1 version, use j<6*m+1 instead of j<6*m
        for(j=0; j<6*m; j++)
        {
            // Find the 3 phase averages for this 2nd difference
            // Initialize phase averages
            fZ1=0.0;
            fZ2=0.0;
            fZ3=0.0;

            // Sum phase averages for 3 sets of m points
            // using points j thru j+m-1, j+m thru j+2m-1 and j+2m thru j+3m-1
            // Note: These phase sums are divided by m = # points of each sum
            // in the 2nd difference expression below
            for(k=j; k<j+m; k++)
            {
                fZ1 += fZ[k];
                fZ2 += fZ[k+m];
                fZ3 += fZ[k+2*m];
            }

            // Calc 2nd difference of one of the 6m sets of 3 phase averages
            fSubsum += SQR((fZ1 - 2*fZ2 + fZ3)/m);
        }

        // Scale the subsum and add it to overall sum
        // NOTE: For 6m+1 version this expression uses (6*m+1) instead of (6*m)
        fSum+=fSubsum/(6*m);

        // Increment the sum count
        nCount++;

        // Display progress
        // Update when index has changed by at least tic amount
        if(((i-nLast_i)>=nTic) && (bProgress))
        {
            ProgressFunction((int)(0.5+100.0*i/(nNum-3*m)),
                "Calculating Mod Totdev");
            nLast_i=i;
        }

        // Check for abort
        if(*bAbort)
        {
            break;
        }
    }

    // Free the subsequence array
    free(fZ);
    // Scale result - See Eq (8) of Reference
    // Without gaps, nCount=nNum-3m
    fSum/=(fTau*fTau*nAF*nAF*2*nCount);

    // Find Mod-Totdev
    if(nCount && fSum>0)
    {
        *fMTDev=(F_TYPE)sqrt(fSum);
    }
    else // No result error
    {
        *fMTDev=0.0;
        nCount=-2;
    }

    // Done
    // Clear progress display
    ProgressFunction(0, "");

    return(nCount);  // Return # phase analysis points
}
```

The FrequenC Library	
NAME: MJDtoDOY	**FUNCTION:** Convert Modified Julian Date to Day of Year
SYNOPSIS: int MJDtoDOY(long mjd)	
long mjd	Modified Julian Date
RETURN: int	Day of year
REMARKS: Calls MJDToDate()	

EXAMPLE:

```
#include "frequenc.h"                             /* FrequenC header file */
long mjd;                                                       /* MJD */
int doy;                                                        /* DOY */
.
.
mjd=50000;                                                  /* set MJD */
doy=MJDtoDOY(mjd);                                    /* call function */
printf("\nDOY = %d", doy);                           /* display result */
```

SEE ALSO: MJDToDate()

REFERENCE: None

```
/*****************************************************************************/
/*                                                                           */
/*                              MJDtoDOY()                                   */
/*                                                                           */
/*      Function to convert long integer MJD to DOY                          */
/*                                                                           */
/*      Parameters:                                                          */
/*                  long    nMJD    MJD value                                */
/*                                                                           */
/*      Return:     int     nDOY    Day of year                              */
/*                                                                           */
/*      Dependencies:                                                        */
/*                  MJDToDate()                                              */
/*                                                                           */
/*      Revision record:                                                     */
/*          03/20/02    Created                                              */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"                       /* FrequenC Library header file */

int __declspec(dllexport) FAR PASCAL MJDtoDOY(long lnMJD)
{
    // Local variables
    // Ints
    int nYear;
    int nDOY;
    int nMonth;
    int nDay;
    static int nSum[] = { 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304,
        334, 365 };

    // Start of code
    // Convert integer MJD to date
    MJDToDate(lnMJD, &nDay, &nMonth, &nYear);

    // Convert data to DOY
    nDOY=nSum[nMonth-1]+nDay;
    if(nYear%4==0 && nYear%100!=0 || nYear%400==0)
    {
        if(nMonth>2)
        {
            nDOY++;
        }
    }

    // Return DOY
    return nDOY;
}
```

The FrequenC Library

NAME: MJDtoDate	**FUNCTION:** Convert MJD to calendar date

SYNOPSIS:
`void MJDtoDate(long mjd, int *day, int *month, int *year)`

long mjd	Modified Julian Date
int *day	Pointer to day #
int *month	Pointer to month #
int *year	Pointer to year
RETURN: void	

REMARKS:

EXAMPLE:
```
 #include "frequenc.h"                        /* FrequenC header file */
 long mjd;                                                     /* MJD */
 int day;                                                      /* day */
 int month;                                                  /* month */
 int year;                                                    /* year */
 .
 .
 mjd=50000;                                                /* set MJD */
 MJDtoDate(mjd, &day, &month, &year);              /* call function */
 printf("\nDay = %d, Month = %d, Year = %d", day, month, year);
                                                   /* display results
*/
```

SEE ALSO:

REFERENCE: None

```
/*****************************************************************************/
/*                                                                           */
/*                             MJDtoDate()                                   */
/*                                                                           */
/*      Function for MJD to calendar date conversion                         */
/*                                                                           */
/*      Parameters:                                                          */
/*          long mjd     modified Julian day # to be converted into date     */
/*          int *day     pointer to day of date to be found from MJD         */
/*          int *month   pointer to month of date to be found from MJD       */
/*          int *year    pointer to year of date to be found from MJD        */
/*                                                                           */
/*      Return:                                                              */
/*          none (void)                                                      */
/*                                                                           */
/*      Function revision record:                                            */
/*          11/11/92    Created                                              */
/*                                                                           */
/*  (c) 1992    W. Riley  Hamilton Technical Services  All Rights Reserved   */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"                        /* FrequenC Library header file */

void __declspec(dllexport) FAR PASCAL MJDtoDate(long mjd, int *day,
    int *month, int *year)
{
    mjd+=679019;
    *year=(int)((mjd-122.1)/365.25);
    *month=(int)((mjd-(long)(365.25**year))/30.6001);
    *day=mjd-(int)(365.25**year)-(int)(30.6001**month);

    if(*month<14)
    {
        *month-=1;
    }                                                           /* end if */
    else
    {
        *month-=13;
    }                                                         /* end else */

    if(*month<3)
    {
        *year+=1;
    }                                                           /* end if */

}                                                   /* end MjdToDate() */
```

The FrequenC Library

NAME: ModTotvarEDF	**FUNCTION:** Description of what function does

SYNOPSIS:
```
int FunctionName( F_TYPE z[], F_TYPE one, F_TYPE two)
```

F_TYPE z[]	Phase or frequency data array
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysis end point

F_TYPE one	parameter one, and description of what it is
F_TYPE two	parameter two, and description of what it is
RETURN: int	# non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are modified.
Embedded zeros are treated as gaps in phase data.
All zeros are treated as gaps in frequency data.

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE z[512];                                           /* data array */
F_TYPE one;                                           /* parameter one */
F_TYPE two;                                           /* parameter two */
int num;                                    /* # data points processed */
.
.
one=1.0;                                    /* set 1st parameter value */
two=2.0;                                    /* set 2nd parameter value */
num=FunctionName(z, one, two);                        /* call function */
printf("\n# Data Points Scaled = %d", num);            /* display num */
```

SEE ALSO: RelatedFunctionName()

REFERENCE: NIST Technical Note 1337

```
/******************************************************************************/
/*                                                                            */
/*                             ModTotvarEDF()                                 */
/*                                                                            */
/*  Function to determine the estimated # of chi-squared degrees of freedom  */
/*  for the modified total variance (MTOT) of a power law noise process.     */
/*                                                                            */
/*  Params:          int   nAlpha = alpha                                     */
/*                   float fRatio = T/tau = N/m                               */
/*                                                                            */
/*  Return:          float edf = estimated degrees of freedom                 */
/*                               or -1 if error                               */
/*                                                                            */
/*  The edf is modeled by the expression b(T/tau) - c, where:                 */
/*                                                                            */
/*  Alpha    Noise     b      c        Original values                        */
/*   2         W PM    1.90   2.10                                            */
/*   1         F PM    1.20   1.40                                            */
/*   0         W FM    1.10   1.20                                            */
/*  -1         F FM    0.85   0.50                                            */
/*  -2        RW FM    0.75   0.31                                            */
/*                                                                            */
/*  Alpha    Noise     b      c        New values per D. Howe                 */
/*   2         W PM    2.16   2.28     e-mail of 05/20/00                     */
/*   1         F PM    1.73   2.99                                            */
/*   0         W FM    1.33   1.89                                            */
/*  -1         F FM    0.92   0.76                                            */
/*  -2        RW FM    0.79   0.37                                            */
/*                                                                            */
/*  Notes:           (1) This function applies only to -2 >= alpha >= 2       */
/*                       (W PM, F PM, W FM, F FM and RW FM noise)             */
/*                   (2) This function applies only for m > 8.0               */
/*                   (3) T is defined as tau0 * # phase data points           */
/*                   (4) The t/tau ratio should never be < 3                  */
/*                                                                            */
/*  References:      E-Mails from D.A. Howe/NIST, 3/13/00 & 5/20/00           */
/*                                                                            */
/*  Revision record:                                                          */
/*      03/25/00    Created                                                   */
/*      03/30/00    Parameter verification revised                            */
/*      05/21/00    Parameters revised per D. Howe update                     */
/*                                                                            */
/* (c) Copyright 2000  Hamilton Technical Services   All Rights Reserved     */
/*                                                                            */
/******************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL ModTotvarEDF(int nAlpha, float fRatio)
{
    // Local variables
    float fEDF;

    // Verify parameters
    if(fRatio<3.0)
    {
        return -1.0; // Error
    }
    if(nAlpha<-2 || nAlpha>2)
    {
        return -1.0; // Error
    }

    switch(nAlpha)
    {
        case 2: // W PM noise
        {
            // fEDF=(float)(1.90*fRatio - 2.10);
            fEDF=(float)(2.16*fRatio - 2.28);
        }
        break;

        case 1: // F PM noise
        {
            // fEDF=(float)(1.20*fRatio - 1.40);
            fEDF=(float)(1.73*fRatio - 2.99);
        }
        break;

        case 0: // W FM noise
        {
            // fEDF=(float)(1.10*fRatio - 1.20);
            fEDF=(float)(1.33*fRatio - 1.89);
        }
        break;

        case -1: // F FM noise
        {
            // fEDF=(float)(0.85*fRatio - 0.50);
            fEDF=(float)(0.92*fRatio - 0.76);
        }
        break;

        case -2: // RW FM noise
        {
            // fEDF=(float)(0.75*fRatio - 0.31);
            fEDF=(float)(0.79*fRatio - 0.37);
        }
        break;
    }

    return fEDF;
}
```

The FrequenC Library

NAME: MJDtoGPS	**FUNCTION:** Convert MJD to GPS Week #

SYNOPSIS:
`int MJDtoGPS(long MJD)`

long MJD	Modified Julian Date
RETURN: int	Global Positioning System week #

REMARKS:
The origin (time zero) of GPS System Time was 00:00:00 UTC on
6 January 1980, Julian Day 2,444,244(.5), MJD 44244.
A GPS cycle is 1024 weeks.

EXAMPLE:

```
#include "frequenc.h"                              /* FrequenC header file */
long mjd;                                                         /* MJD */
int week;                                                  /* GPS week # */
.
.
mjd=50000;                                                     /* set MJD */
week=MJDtoGPS(mjd);                                      /* call function */
printf("\nGPS Week = %d", week);                        /* display result */
```

SEE ALSO: MJDtoDOY(), MJDtoDate()

REFERENCE:

```
/*****************************************************************************/
/*                                                                           */
/*                              MJDtoGPS()                                   */
/*                                                                           */
/*      Function to convert MJD to GPS Week #                                */
/*                                                                           */
/*      Parameters: long    MJD                                              */
/*                                                                           */
/*      Return:     int     GPS Week #                                       */
/*                                                                           */
/*      Note:   The origin (time zero) of GPS System Time was 00:00:00 UTC   */
/*              6 January 1980, Julian Day 2,444,244.500, MJD 44244.  A      */
/*              GPS Cycle is 1024 weeks.                                     */
/*                                                                           */
/*      Revision record:                                                     */
/*          01/25/98    Created                                              */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"                        /* FrequenC Library header file */

int __declspec(dllexport) FAR PASCAL MJDtoGPS(long MJD)
{
    return ((int)((MJD-44244L)/7)%1024);
}
```

The FrequenC Library

NAME: CalcMTIE	**FUNCTION:** Calc maximum time interval error for phase data

SYNOPSIS:
```
int CalcMTIE(F_TYPE x[], F_TYPE *MTIE, int af, BOOL *abort,
             BOOL show, void *ProgressFunc(int percent, char *msg))
```

F_TYPE x[]	Phase data array
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE *MTIE	Pointer to MTIE result
int af	Averaging factor
BOOL *abort	Pointer to abort flag (FALSE = no abort)
BOOL show	Flag to show progress (FALSE = no display)
void *ProgressFunc(int percent, char *msg)	Pointer to function to display progress, with int arg set to % calc complete, and char* arg set to text message to display. Can be NULL.
RETURN: int	# non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are used. Embedded zeros are treated as gaps in phase data.

EXAMPLE:
```
#include "frequenc.h"                        /* FrequenC header file */
F_TYPE x[512];                                  /* phase data array */
F_TYPE MTIE;                                          /* MTIE value */
int af=1;                                             /* avg factor */
int num;                                 /* # data points processed */
.
.
num=CalcMTIE(x, &MTIE, af, FALSE, FALSE, NULL);    /* call function */
                                  /* No aborting or progress display */
printf("\nMTIE = %e", MTIE);                        /* display MTIE */
```

SEE ALSO: TIErms()

REFERENCE: S. Bregni, "Clock Stability Characterization and Measurement in Telecommunications", IEEE Transactions on Instrumentation and Measurement, Vol. 46, No. 6, Dec 1997, pp 1284-1294, Eq. 13.

```
/****************************************************************************/
/*                                                                          */
/*                              CalcMTIE()                                  */
/*                                                                          */
/*      Function to calc MTIE (maximum time interval error) for phase data  */
/*      Parameters:                                                         */
/*          F_TYPE fX[]    = data array;     F_TYPE fX[0] = # data points   */
/*          F_TYPE fX[1]   = analysis start; F_TYPE fX[2] = analysis end    */
/*          F_TYPE *fMTIE  = pointer to MTIE                                */
/*          int nAF        = averaging factor                               */
/*          BOOL *bAbort   = pointer to abort flag                          */
/*          BOOL bProgress = progress indicator flag                        */
/*          void *ProgressFunction()                                        */
/*                         = pointer to progress indicator function with    */
/*                           args int nPercentDone and char *szMessage      */
/*                                                                          */
/*      Return:                                                             */
/*          num   = # analysis points (not very useful)                     */
/*                                                                          */
/*      Algorithm:                                                          */
/*          An n-point wide window is moved thru the phase data and the     */
/*          difference between the maximum and minimum phase (time) values  */
/*          is found at each window position.  MTIE is the overall maximum  */
/*          of this time interval error over the whole data set.            */
/*                                                                          */
/*      Notes:                                                              */
/*          (1) This function handles gaps because FindMinMax() does.       */
/*          (2) No test suite values to check results against.  Results are */
/*              consistent with the p-p range of the test data.             */
/*          (3) MTIE has no tau or AF dependence for white pm noise.  It has */
/*              a tau^+1/2 log slope for W FM noise. It is very sensitive to */
/*              peak values (transients or outliers).                       */
/*          (4) # time intervals is N-1, where N = # phase data points, so  */
/*              AF can go from 1 to N-1.                                    */
/*                                                                          */
/*      Reference:      S. Bregni, "Clock Stability Characterization and    */
/*                      Measurement in Telecommunications", IEEE            */
/*                      Transactions on Instrumentation and Measurement,    */
/*                      Vol. 46, No. 6, Dec 1997, pp 1284-1294, Eq. 14.     */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/07/96    Drafted - seems to run OK - patched into Sigma      */
/*                      function instead of TOTALVAR (for freq data!) since */
/*                      that call has same signature.                       */
/*          06/03/98    Put into Stable32                                   */
/*          06/04/98    Refined progress display                            */
/*          01/27/01    Extended allowable index by 1                       */
/*          03/09/03    Added abort and progress args                       */
/*                                                                          */
/* (c) Copyright 1996-2003  Hamilton Technical Services  All Rights Reserved */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL CalcMTIE(F_TYPE fX[], F_TYPE *fMTIE,
    int nAF, BOOL *bAbort, BOOL bProgress,
    void *ProgressFunction(int nPercentDone, char *szMessage))

{
```

```
// Define local variables
// Ints
int i;                          // Index
int last_i=0;                   // Previous index
int tic;                        // Progress interval
int nStart;                     // Analysis start
int nEnd;                       // Analysis end
int nNum;                       // # analysis points
// F_TYPES
F_TYPE fMin;                    // Min phase data value within window
F_TYPE fMax;                    // Max phase data value within window

// Set analysis limits & progress interval
nStart=(int)fX[1];
nEnd=(int)fX[2];
nNum=nEnd-nStart+1;
tic=(int)ceil((nNum-nStart-nAF)/100);  /* 1% interval for progress report */

// Initialize MTIE value
*fMTIE=0.0;

// Calc MTIE
// The analysis window is nAF points wide.  It starts at the 1st data point
// and moves thru all the data until the end of the window is at the end of
// the analysis data.

// Initialize progress display
if(bProgress)
{
    ProgressFunction(0, "Calculating MTIE");
}

// Was i<nNum-nAF until 01/27/00
for(i=nStart; i<=nNum-nAF; i++)
{
    // Find the min & max values within the data window
    fX[1]=i;
    fX[2]=i+nAF;
    FindMinMax(fX, &fMin, &fMax, PHASE_DATA);

    // MTIE is the overall maximum p-p excursion
    *fMTIE=max(*fMTIE, fMax-fMin);

    // Display progress
    // Note:  This code provides updates only if index has changed
    // by at least tic amount (1%)
    if( ((i-last_i)>=tic) && (bProgress) )
    {
        ProgressFunction((int)(0.5+100.0*i/(nNum-nStart-nAF)),
            "Calculating MTIE");
        last_i=i;
    }
```

```
            // Check for abort
            if(*bAbort)
            {
                // Restore analysis limits
                fX[1]=nStart;
                fX[2]=nEnd;

                // Bail out
                break;
            }
        }

        // Restore analysis limits
        fX[1]=nStart;
        fX[2]=nEnd;

        // Done
        // Clear progress display
        if(bProgress)
        {
            ProgressFunction(0, "");
        }

        // Return # analysis points
        return nNum;
    }
```

The FrequenC Library

NAME: NormalizeData	**FUNCTION:** Normalize phase or frequency data by removing mean value
SYNOPSIS: `int NormalizeData(F_TYPE z[], F_TYPE avg, BOOL datype)`	

F_TYPE z[]	Phase or frequency data array:
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysis end point

F_TYPE avg	Average (mean) value to be removed from data
BOOL datype	Type of data: 0=phase, 1=frequency
RETURN: int	The # of analysis points (can be 0)

REMARKS:
Only data between start and end analysis limits are analyzed.
All zeros are treated as gaps in frequency data.
Only embedded zeros are treated as gaps in phase data.
The datype flag determines whether the data is treated as phase (datype=0) or frequency (datype=1) data.
CalcMean() may be used to calculate the value of avg to be removed.
No error codes are returned from this function.
Returns 0 if all gaps; an array of constant data can be normalized to all zeros, and thus will be treated as all gaps.

EXAMPLE:
```
#include "frequenc.h"                              /* FrequenC header file */
F_TYPE y[512];                                          /* freq data array */
F_TYPE avg;                                                     /* average */
BOOL datype;                                             /* data type flag */
int num;                                             /* # analysis points */
.
.
datype=FREQ;                               /* set flag for frequency data */
         /* frequenc.h contains #define PHASE 0 and #define FREQ 1 */
CalcMean(y, &avg, datype)                                      /* calc avg */
                 /* include error check for CalcMean() as required */
num=NormalizeData(y, avg, datype);                       /* normalize data */
printf("\n# Points Processed = %d", num);                   /* display num */
```

SEE ALSO: CalcMean()

REFERENCE: None

```
/*****************************************************************************/
/*                                                                           */
/*                          NormalizeData()                                  */
/*                                                                           */
/*      Function to normalize phase or freq data by removing mean value     */
/*                                                                           */
/*      Parameters:     F_TYPE z[]  = phase or freq data                    */
/*                                    z[0] = # data points                   */
/*                                    z[1] = analysis start                  */
/*                                    z[2] = analysis end                    */
/*                      F_TYPE avg  = average value to be removed           */
/*                      int datype  = data type: 0=phase, 1=frequency        */
/*                                                                           */
/*      Return:         int         = # non-gap data points (may be 0)       */
/*                                                                           */
/*      Notes:          All zeros treated as gaps in frequency data          */
/*                      Embedded zeros are gaps in phase data                */
/*                                                                           */
/*      Revision record:                                                     */
/*          12/16/91    Created                                              */
/*          12/24/91    Added datype parameter & int return value            */
/*          12/26/91    Changed avg parameter from pointer to value          */
/*          12/31/91    Renamed                                              */
/*          01/18/92    Edited title block                                   */
/*          02/03/96    Modified for use as Win 3.1 DLL                      */
/*          01/01/98    Changes for MS VC++ compatibility & warnings         */
/*                                                                           */
/* (c) Copyright 1991-8  Hamilton Technical Services  All Rights Reserved  */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL NormalizeData(F_TYPE z[], F_TYPE avg,
    BOOL datype)
{
    int i;
    int n=0;

    for( i = (int)(*(z+1)+2); i <= (int)(*(z+2)+2); i++)
    {
        if(*(z+i) || (!datype && i==3 || !datype && i==z[0]+2))
        {
            *(z+i) -=  avg;
            n++;
        }                                                       /* end if */
    }                                                          /* end for */
    return(n);
}                                                /* end NormalizeData() */
```

The FrequenC Library

NAME: NoiseID	**FUNCTION:** Determine power law noise type from B1 or R(n) ratio

SYNOPSIS:
```
 int WINAPI NoiseID(int num, int sigtype, int af, float bw, float
ratio)
```

int num	# non-gap data points
int sigtype	Sigma type (e.g. NORMAL_SIGMA macro)
int af	Averaging factor
float bw	Bandwidth factor
float ratio	B1 or R(n) ratio (from data analysis)
RETURN: int	Alpha (power law noise type) -4 <= α <= 2 (RR FM to W PM)

REMARKS:
Use CalcStarB1() before call to this function with NORMAL_SIGMA as alternative to using CalcHadamardB1() with Hadamard sigmas.
See sigma type macros in frequenc.h.

EXAMPLE:
```
 #include <frequenc.h>                         /* FrequenC header file */
 int alpha;                                                  /* alpha */
 int num=100;                                         /* # data points */
 int sigtype=OVERLAPPING_SIGMA;                          /* sigma type */
 int af=1;                                         /* averaging factor */
 float bw=10.0;                                           /* BW factor */
 float ratio=50.0;                                         /* B1 ratio */
    /* B1 found as ratio of standard & Allan variances for the data */
 .
 .
 alpha=NoiseID(num, sigtype, af, bw, ratio);          /* call function */
 printf("\nAlpha = %d", alpha);                       /* display alpha */
```

SEE ALSO: CalcBias(), CalcRatio(), CalcHadamardB1(), CalcStarB1()

REFERENCE: NIST Technical Note 1337

```
/*****************************************************************************/
/*                                                                           */
/*                              NoiseID()                                    */
/*                                                                           */
/*  Function to determine power law noise type fron B1 or R(n) ratio, where: */
/*      B1 is ratio of normal variance to Allan variance                     */
/*      R(n) is ratio of modified to normal Allan variance                   */
/*                                                                           */
/*  Parameters:     int     nNum            # non-gap data points            */
/*                                          (# data points-# gaps)           */
/*                  int     nSigmaType      Sigma type                       */
/*                  int     nAvgFactor      Averaging factor                 */
/*                  float   fBW             Bandwidth factor                 */
/*                  float   fRatio          B1 or  R(n) ratio                */
/*                                                                           */
/*  Return          int     nAlpha          Alpha (noise type)               */
/*                                          -4 <= alpha <= 2                 */
/*                                          RR FM to W PM                    */
/*                                                                           */
/*  Note 1: Use CalcStarB1() before call to this function with NORMAL_SIGMA  */
/*          as alternative to using CalcHadamardB1() herein with Hadamard    */
/*          sigmas.                                                          */
/*                                                                           */
/*  Note 2: Sigma types are as follows:     NORMAL_SIGMA                     */
/*                                          OVERLAPPING_SIGMA                */
/*                                          MODIFIED_SIGMA                   */
/*                                          TIME_SIGMA                       */
/*                                          TOTAL_SIGMA                      */
/*                                          MODTOTAL_SIGMA                   */
/*                                          TIMETOTAL_SIGMA                  */
/*                                          HADAMARD_SIGMA                   */
/*                                          OVERLAPHAD_SIGMA                 */
/*                                          HADTOTAL_SIGMA                   */
/*                                                                           */
/*      Revision record:                                                     */
/*          03/14/99    Added OVERLAPHAD_SIGMA sigma type                    */
/*          04/24/99    Added usage of CalcHadamardB1()                      */
/*          12/26/99    Added MODTOTAL_SIGMA sigma type                      */
/*          05/06/00    Added TIMETOTAL_SIGMA sigma type                     */
/*          11/05/00    Added HADTOTAL_SIGMA sigma type                      */
/*          11/15/00    Moved TIME_SIGMA to with other mod sigma cases       */
/*          11/23/00    Changed Drift or FW FM case to alpha=-3              */
/*                                                                           */
/*      (c) 1996-9 W. Riley Hamilton Technical Services All Rights Reserved  */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL NoiseID(int nNum, int nSigmaType,
    int nAvgFactor, float fBW, float fRatio)
{
    // Local variables
    // Ints
    int nAlpha;                                                 // Alpha exps
    // Floats
    float fB_rrfm, fB_fwfm, fB_rwfm, fB_ffm, fB_wfm, fB_pm;   // Bias functs
    float fR_rwfm, fR_ffm,  fR_wfm, fR_fpm, fR_wpm;           // Ratio functs
```

```
#ifdef LOGGING
// Chars
char szBuffer[BUFFER_SIZE+1];   // Text buffer
#endif

switch(nSigmaType)
{
    case NORMAL_SIGMA:
    case OVERLAPPING_SIGMA:
    case TOTAL_SIGMA:
    {
        // Note: 2nd arg in CalcBias() is mu
        fB_fwfm=CalcBias(nNum,  2);
        fB_rwfm=CalcBias(nNum,  1);
        fB_ffm =CalcBias(nNum,  0);
        fB_wfm =CalcBias(nNum, -1);
        fB_pm  =CalcBias(nNum, -2);

        #ifdef LOGGING
        sprintf(szBuffer, "NoiseID() using "
            "fB_fwfm=%f, fB_rwfm=%f, fB_ffm=%f, fB_wfm=%f, fB_pm=%f",
            fB_fwfm, fB_rwfm, fB_ffm, fB_wfm, fB_pm);
        WriteLog(60, szBuffer);
        #endif

        if(fRatio>((fB_fwfm+fB_rwfm)/2))
        {
            // Drift or FW FM
            // Note: Can make it -2 to avoid alpha<-2
            // which causes trouble in CI calcs
            // Or can let it be -3 to detect FW FM noise
            // and deal with potential CI calc problem separately
            // The latter approach is used
            nAlpha=-3;
        }
        else if(fRatio>sqrt(fB_rwfm*fB_ffm))
        {
            // Random Walk FM
            nAlpha=-2;
        }
        else if(fRatio>sqrt(fB_ffm*fB_wfm))
        {
            // Flicker FM
            nAlpha=-1;
        }
        else if(fRatio>sqrt(fB_wfm*fB_pm))
        {
            // White FM
            nAlpha=0;
        }
        else
        {
            // White or Flicker PM
            // Make it +1 to agree with what Run function does
            nAlpha=1;  // Could also be +2
        }
    }
    break;
```

```
case MODIFIED_SIGMA:
case TIME_SIGMA:
case MODTOTAL_SIGMA:
case TIMETOTAL_SIGMA:
{
    // Note: 1st arg in CalcRatio() is alpha = nAvg Factor
    // Was wrong (nNum) until 02/09/02
    fR_rwfm=CalcRatio(-2, nAvgFactor, fBW);
    fR_ffm =CalcRatio(-1, nAvgFactor, fBW);
    fR_wfm =CalcRatio( 0, nAvgFactor, fBW);
    fR_fpm =CalcRatio( 1, nAvgFactor, fBW);
    fR_wpm =CalcRatio( 2, nAvgFactor, fBW);

    #ifdef LOGGING
    sprintf(szBuffer, "NoiseID() using "
        "fR_rwfm=%f, fR_ffm=%f, fR_wfm=%f, fR_fpm=%f, fR_wpm=%f",
        fR_rwfm, fR_ffm, fR_wfm, fR_fpm, fR_wpm);
    WriteLog(60, szBuffer);
    #endif

    if(nAvgFactor==1)
    {
        // Undefined
        nAlpha=0;
    }
    else if(fRatio>sqrt(fR_rwfm*fR_ffm))
    {
        // Random Walk FM
        nAlpha=-2;
    }
    else if(fRatio>sqrt(fR_ffm*fR_wfm))
    {
        // Flicker FM
        nAlpha=-1;
    }
    else if(fRatio>sqrt(fR_wfm*fR_fpm))
    {
        // White FM
        nAlpha=0;
    }
    else if(fRatio>sqrt(fR_fpm*fR_wpm))
    {
        // Flicker PM
        nAlpha=1;
    }
    else
    {
        // White PM
        nAlpha=2;
    }
}
break;
```

```
        case HADAMARD_SIGMA:
        case OVERLAPHAD_SIGMA:
        case HADTOTAL_SIGMA:
        {
            // Note: 2nd arg in CalcHadamardB1() is mu
            fB_rrfm=CalcHadamardB1(nNum,  3);
            fB_fwfm=CalcHadamardB1(nNum,  2);
            fB_rwfm=CalcHadamardB1(nNum,  1);
            fB_ffm =CalcHadamardB1(nNum,  0);
            fB_wfm =CalcHadamardB1(nNum, -1);
            // Use ordinary B1 ratio for mu=-2 (F or F PM)
            fB_pm  =CalcBias(nNum, -2);

            #ifdef LOGGING
            sprintf(szBuffer, "NoiseID() using "
                "fB_rrfm=%f, fB_fwfm=%f, fB_ffm=%f, fB_wfm=%f, fB_pm=%f",
                fB_rrfm, fB_fwfm, fB_ffm, fB_wfm, fB_pm);
            WriteLog(60, szBuffer);
            #endif

            if(fRatio>((fB_rrfm*fB_fwfm)))
            {
                // RR FM
                nAlpha=-4;
            }
            else if(fRatio>((fB_fwfm*fB_rwfm)))
            {
                // Drift or FW FM
                nAlpha=-3;
            }
            else if(fRatio>sqrt(fB_rwfm*fB_ffm))
            {
                // Random Walk FM
                nAlpha=-2;
            }
            else if(fRatio>sqrt(fB_ffm*fB_wfm))
            {
                // Flicker FM
                nAlpha=-1;
            }
            else if(fRatio>sqrt(fB_wfm*fB_pm))
            {
                // White FM
                nAlpha=0;
            }
            else
            {
                // White or Flicker PM

                // Make it +1 to agree with what Run function does
                nAlpha=1;  // Could also be +2
            }
        }
        break;
    }

    return nAlpha;
}
```

The FrequenC Library

NAME: RoundAxes	**FUNCTION:** Find plot scale
SYNOPSIS: `void RoundAxes(F_TYPE *min, F_TYPE *max, F_TYPE *tics)`	
F_TYPE min	Pointer to data minimum input; becomes scale minimum output
F_TYPE max	Pointer to data maximum input; becomes scale maximum output
F_TYPE *tics	Pointer to tic size output
RETURN: void	

REMARKS:
This function is recommended for a large relative data range.

EXAMPLE:

```
 #include "frequenc.h"                           /* FrequenC header file */
 F_TYPE y[512];                                       /* freq data array */
 double min, max;                            /* data and scale min & max */
 double tics;                                                /* tic size */
 .
 .
 FindMinMax(y, &min, &max, FREQ_DATA);          /* get data min & max */
 RoundAxes(&min, &max, &tics);                        /* get plot scale */
 printf("\nMin = %e, Max = %e, Tics = %e", min, max, tics);
                                                      /* display results */
```

SEE ALSO: FindPlotScale(), BasScale()

REFERENCE: Ref: Quinn-Curtis S&E Tools IPC-TC-006 SEGRAPH.C

```
/***************************************************************************/
/*                                                                         */
/*                              RoundAxes()                                */
/*                                                                         */
/*      Still another function to find plot scale                          */
/*      Ref: Quinn-Curtis S&E Tools IPC-TC-006 SEGRAPH.C                   */
/*                                                                         */
/*      Parameters:  realtype *a1   =   pointer to data min input          */
/*                                      becomes scale min output           */
/*                   realtype *a2   =   pointer to data max input          */
/*                                      becomes scale max output           */
/*                   realtype *tics =   pointer to tics size output        */
/*                                                                         */
/*      Return:        None (void)                                         */
/*                                                                         */
/*      Revision record:                                                   */
/*          03/10/02    Copied and adapted                                 */
/*          03/12/02    Added log 0 trap                                   */
/*          03/19/03    Adapted for FrequenC DLL                           */
/*                                                                         */
/***************************************************************************/

// FrequenC header
#include "frequenc.h"

// Local function prototypes
F_TYPE log10x(F_TYPE realnum);
int NumExp(F_TYPE realnum);
F_TYPE PowerCalc(F_TYPE realnum, F_TYPE power);

// RoundAxes function
void  __declspec(dllexport) FAR PASCAL RoundAxes(F_TYPE *a1, F_TYPE *a2,
    F_TYPE *tics)
{
   F_TYPE dr1, dr2, px2, pc2;
   int di1, di2, digits, a2neg, a1neg;

   a1neg = (*a1 < 0);
   a2neg = (*a2 > 0);
   if (NumExp(*a2) > NumExp(*a1))
     px2 = NumExp(*a2) - 1;
   else
     px2 = NumExp(*a1) - 1;
   pc2 = PowerCalc(10.0,px2);
   dr2 = (*a2) / pc2;
   dr1 = (*a1) / pc2;

   di2 = (int)ceil(dr2 + 0.000001+(dr2-dr1)*0.05);
   if ((di2 > 0) && (dr2 <= 0.0)) di2 = 0;

   di1 = (int)floor(dr1 - 0.000001- (dr2-dr1)*0.05);
   if ((di1 < 0) && (dr1 >= 0.0)) di1 = 0;

   if ( abs(di2) < 10 )
      di2 = di2 + a2neg;
   else
     if ( abs(di2) < 60 )
    di2 = ((di2 / 5) + a2neg ) * 5;
     else
       di2 = ((di2 / 10) + a2neg) * 10;
   if ( abs(di1) < 60 )
    di1 = ((di1 / 5) - a1neg) * 5;
   else
    di1 = ((di1 / 10) - a1neg) * 10;

   *a1 = di1 * pc2;
   *a2 = di2 * pc2;
   digits = abs(di2 - di1);
   if ( digits < 10 )
      *tics =  1.0;
   else
     if ( digits < 20 )
       *tics = 2.0;
     else
       if ( digits < 40 )
     *tics = 5.0;
       else
     if ( digits < 80 )
       *tics = 10.0;
     else
       if ( digits < 100 )
         *tics = 20.0;
       else
         if ( digits < 151 )
           *tics = 20.0;
         else
           if ( digits < 200 )
         *tics = 50.0;
           else
         *tics = 100.0;

   *tics = *tics * pc2 / 10.0;
}

// Local subroutines adapted from SELIB.C
F_TYPE log10x(F_TYPE realnum)
{  F_TYPE fret;
    // Avoid log 0
    if(realnum==0.0)
    {
        realnum=1e-99;
    }
   fret = log10(realnum);
   return(fret);
}

int NumExp(F_TYPE realnum)
{
   int fret;
   fret = (int)floor(log10x(fabs(realnum)));
   return(fret);
}

F_TYPE PowerCalc(F_TYPE realnum, F_TYPE power)
{
   F_TYPE fret;
   fret = pow(realnum,power);
   return(fret);
}
```

The FrequenC Library	
NAME: RemoveDiffusionFreq Drift	**FUNCTION:** Remove the diffusion fit drift from frequency data
SYNOPSIS: int RemoveDiffusionFreqDrift(F_TYPE y[], F_TYPE a, F_TYPE b, F_TYPE c)	

F_TYPE y[]	Frequency data array:
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

F_TYPE a	Parameter a of y(t) = a + b(t+c)^1/2 diffusion fit to the frequency drift to be removed
F_TYPE b	Parameter b of y(t) = a + b(t+c)^1/2 diffusion fit to the frequency drift to be removed
F_TYPE c	Parameter c of y(t) = a + b(t+c)^1/2 diffusion fit to the frequency drift to be removed
RETURN: int	The # of non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are processed. All zeros are retained as gaps in the frequency data. Use CalcDiffusionFreqDrift() to calculate fit parameter of drift to be removed. No error codes are returned from this function.

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE y[512];                                 /* frequency data array */
F_TYPE a;                                               /* fit param a */
F_TYPE b;                                               /* fit param b */
F_TYPE c;                                               /* fit param c */
num;                                       /* # data points processed */
.
.
CalcDiffusionFreqDrift(y, &a, &b, &c);           /* calc diffusion fit */
   /* include error check for CalcDiffusionFreqDrift() as required */
num=RemoveDiffusionFreqDrift(y, a, b, c);         /* remove diff drift */
printf("\n# Data Points Processed = %d", num);          /* display num */
```

SEE ALSO: CalcDiffusionFreqDrift()

REFERENCE:

```
/*****************************************************************************/
/*                                                                           */
/*                          RemoveDiffusionFreqDrift()                       */
/*                                                                           */
/*      Function to remove diffusion frequency drift y(t) = a + bû(t+c)      */
/*                                                                           */
/*      Parameters:     F_TYPE y[] = frequency data                          */
/*                                   y[0] = # frequency data points          */
/*                                   y[1] = analysis start                   */
/*                                   y[2] = analysis end                     */
/*                      F_TYPE a   = a term of diffusion fit                 */
/*                      F_TYPE b   = b term of diffusion fit                 */
/*                      F_TYPE c   = c term of diffusion fit                 */
/*                                                                           */
/*      Return:         int        = # data points processed                 */
/*                                                                           */
/*      Notes:          All zeros treated as gaps in frequency data.         */
/*                      Use function CalcDiffusionFreqDrift() to             */
/*                      calculate diffusion fit parameters a and b.          */
/*                                                                           */
/*      Revision record:                                                     */
/*          06/11/97    Created from first RemoveDiffusionFreqDrift()        */
/*          12/09/97    Added to FrequenC Library                            */
/*          01/01/98    Changes for MS VC++ compatibility & warnings         */
/*                                                                           */
/* (c) Copyright 1997-8 Hamilton Technical Services   All Rights Reserved    */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL RemoveDiffusionFreqDrift(F_TYPE y[],
    F_TYPE a, F_TYPE b, F_TYPE c)
{
    int i;                                                        /* index */
    int num=0;                                       /* # points processed */

    for(i = (int)(*(y+1)+2); i <= (int)(*(y+2)+2); i++)   /* analysis limits */
    {
        if(*(y+i))                                          /* check for gap */
        {
            *(y+i) -=  a + (b * sqrt((double) (i-2)+c));     /* remove drift */
            num++;                                        /* increment count */
        }                                                        /* end if */
    }                                                           /* end for */

    return(num);                              /* return # points processed */
}                                     /* end NewRemoveDiffusionFreqDrift() */
```

The FrequenC Library

NAME: RemoveFreqOffset	**FUNCTION:** Remove the frequency offset from phase data
SYNOPSIS: int RemoveFreqOffset(F_TYPE x[], F_TYPE a, F_TYPE b, BOOL bA)	

F_TYPE x[]	Phase data array:
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE a	Intercept, a, of x(t)=a+bt linear fit to the frequency offset to be removed
F_TYPE b	Slope, b, of x(t)=a+bt linear fit to the frequency offset to be removed
BOOL c	Flag for constant term removal
RETURN: int	The # of non-gap data points processed

REMARKS:
All embedded zeros treated as gaps in phase data.
Use CalcFreqOffset() to calculate linear fit parameters a and b.
No error codes are returned from this function.

EXAMPLE:
```
#include "frequenc.h"                           /* FrequenC header file */
F_TYPE x[512];                                      /* phase data array */
F_TYPE a;                                                  /* intercept */
F_TYPE b;                                                      /* slope */
F_TYPE v;                                               /* fit variance */
BOOL c=TRUE;                                   /* constant removal flag */
int num;                                     /* # data points processed */
.
.
CalcFreqOffset(x, &a, &b, &v);           /* calc linear fit parameters */
num=RemoveFreqOffset(x, a, b, c);                      /* call function */
printf("\n# Data Points Processed = %d", num);           /* display num */
```

SEE ALSO: CalcFreqOffset()

REFERENCE:

```
/****************************************************************************/
/*                                                                          */
/*                           RemoveFreqOffset()                             */
/*                                                                          */
/*      Function to remove freq offset from phase data: x(t) = a + bt       */
/*                                                                          */
/*      Parameters:     F_TYPE x[] = phase data                             */
/*                                   x[0] = # phase data points             */
/*                                   x[1] = analysis start                  */
/*                                   x[2] = analysis end                    */
/*                      F_TYPE a   = intercept of linear fit                */
/*                      F_TYPE b   = slope of linear fit = freq offset      */
/*                      BOOL bA    = switch for constant term removal       */
/*                                                                          */
/*      Return:         int        = # data points processed                */
/*                                                                          */
/*      Notes:          All embedded zeros treated as gaps in phase data.   */
/*                      Use function CalcFreqOffset() to calculate linear   */
/*                      fit parameters a and b.                             */
/*                                                                          */
/*      Revision record:                                                    */
/*          02/22/97    Adapted from RemoveLinFreqDrift() of FrequenC       */
/*          02/27/97    Added BOOL bA parameter                             */
/*          01/01/98    Changes for MS VC++ compatibility & warnings        */
/*          04/23/03    Adapted for Version 2.0 source code documentation   */
/*                                                                          */
/* (c) Copyright 1997-2003 Hamilton Technical Services All Rights Reserved  */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL RemoveFreqOffset(F_TYPE x[], F_TYPE a,
    F_TYPE b, BOOL bA)
{
    int i;                                                       /* index */
    int num=0;                                      /* # points processed */

    // To avoid compiler warning if paramater a isn't used
    a=a;

    for(i = (int)(*(x+1)+2); i <= (int)(*(x+2)+2); i++)   /* analysis limits */
    {
        if( (*(x+i)) || (i==3) || (i==*(x+2)+2) )           /* check for gap */
        {
            if(bA)
            {
                // Remove slope and intercept - make average zero
                *(x+i) -=  a + (b * (i-3));    /* remove slope and intercept */
            }
            else
            {
                // Remove slope only
                *(x+i) -=  b * (i-2);                         /* remove slope */
            }
            num++;                                         /* increment count */
        }
    }
    return(num);                                 /* return # points processed */
}
```

The FrequenC Library

NAME: RemoveLinFreqDrift	**FUNCTION:** Remove the linear drift from frequency data
SYNOPSIS: `int RemoveLinFreqDrift(F_TYPE y[], F_TYPE a, F_TYPE b)`	

F_TYPE y[]	Frequency data array:
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

F_TYPE a	Intercept, a, of y(t)=a+bt linear fit to the frequency drift to be removed
F_TYPE b	Slope, b, of y(t)=a+bt linear fit to the frequency data to be removed
RETURN: int	The # of non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are processed. All zeros are retained as gaps in the frequency data. Use CalcLinFreqDrift() to calculate intercept and slope to be removed. No error codes are returned from this function.

EXAMPLE:
```
#include "frequenc.h"                      /* FrequenC header file */
F_TYPE y[512];                             /* frequency data array */
F_TYPE a;                                             /* intercept */
F_TYPE b;                                                 /* slope */
F_TYPE v;                                              /* variance */
int num;                                /* # data points processed */
.
.
CalcLinFreqDrift(y, &a, &b, &v);          /* calc linear fit params */
         /* include error check for CalcLinFreqDrift() as required */
num=RemoveLinFreqDrift(y, a, b);             /* remove linear drift */
printf("\n# Data Points Processed = %d", num);      /* display num */
```

SEE ALSO: CalcLinFreqDrift()

REFERENCE: NIST Technical Note 1337, TN-264 to TN-295

```
/****************************************************************************/
/*                                                                          */
/*                          RemoveLinFreqDrift()                            */
/*                                                                          */
/*      Function to remove linear frequency drift y(t) = a + bt             */
/*                                                                          */
/*      Parameters:     F_TYPE y[] = frequency data                         */
/*                                   y[0] = # frequency data points         */
/*                                   y[1] = analysis start                  */
/*                                   y[2] = analysis end                    */
/*                      F_TYPE a   = intercept of linear fit                */
/*                      F_TYPE b   = slope of linear fit                    */
/*                                                                          */
/*      Return:         int        = # data points processed                */
/*                                                                          */
/*      Notes:          All zeros treated as gaps in frequency data.        */
/*                      Use FrequenC function CalcLinFreqDrift() to         */
/*                      calculate linear fit parameters a and b.            */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/16/91    Created                                             */
/*          12/26/91    Changed to int returning # data points processed   */
/*                      Changed slope & intercept from pointers to values  */
/*          12/30/91    Changed to include internal call to drift_calc()   */
/*                      with slope and intercept parameters removed         */
/*          12/31/91    Renamed                                             */
/*          01/18/92    Edited title block                                  */
/*          02/03/96    Modified for use as Win 3.1 DLL                     */
/*          01/01/98    Changes for MS VC++ compatibility & warnings        */
/*                                                                          */
/* (c) Copyright 1991-8  Hamilton Technical Services  All Rights Reserved  */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL RemoveLinFreqDrift(F_TYPE y[],
    F_TYPE a, F_TYPE b)
{
    int i;                                                      /* index */
    int num=0;                                     /* # points processed */

    for(i = (int)(*(y+1)+2); i <= (int)(*(y+2)+2); i++) /* analysis limits */
    {
        if(*(y+i))                                        /* check for gap */
        {
            *(y+i) -=  a + (b * (i-2));                  /* remove drift */
            num++;                                    /* increment count */
        }                                                      /* end if */
    }                                                         /* end for */

    return(num);                            /* return # points processed */
}                                            /* end RemoveLinFreqDrift() */
```

The FrequenC Library	
NAME: RemoveLogFreqDrift	**FUNCTION:** Remove the log drift from frequency data
SYNOPSIS: int RemoveLogFreqDrift(F_TYPE y[], F_TYPE a, F_TYPE b, F_TYPE c)	

F_TYPE y[]	Frequency data array:
	y[0] = # data points
	y[1] = analysis start point
	y[2] = analysis end point

F_TYPE a	Multiplicative term, a, of y(t)=a·ln(bt+1)+c log fit to be removed
F_TYPE b	Time term, b, of log fit to be removed
T_TYPE c	Constant term, c, of log fit to be removed
RETURN: int	The # of non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are analyzed.
All zeros are retained as gaps in frequency data.
Use CalcLogFreqDrift() to calculate quadratic fit to phase data.
No error codes are returned from this function.

EXAMPLE:
```
#include "frequenc.h"                         /* FrequenC header file */
F_TYPE y[512];                                     /* freq data array */
F_TYPE a, b, c, v;                              /* log fit parameters */
int num;                                   /* # data points processed */
.
.
CalcLogFreqDrift(y, &a, &b, &c, &v);          /* calc log fit params */
         /* include error check for CalcLogFreqDrift() as required */
num=RemoveLogFreqDrift(y, a, b, c);              /* remove log drift */
printf("\n# Data Points Processed = %d", num);      /* display num */
```

SEE ALSO: CalcLogFreqDrift()

REFERENCE: MIL-O-55310B, General Specification for Crystal Oscillators

```
/****************************************************************************/
/*                                                                          */
/*                          RemoveLogFreqDrift()                            */
/*                                                                          */
/*      Function to remove frequency drift using log model                  */
/*                           y(t) = a*ln(bt+1)+c, where:                    */
/*                           y(t) = fractional frequency                    */
/*                           t    = time                                    */
/*                           a    = multiplicative term                     */
/*                           b    = log time term                           */
/*                           c    = constant term                           */
/*                                                                          */
/*      Parameters:     F_TYPE y[] = frequency data                         */
/*                                   y[0] = # frequency data points         */
/*                                   y[1] = analysis start                  */
/*                                   y[2] = analysis end                    */
/*                      F_TYPE a   = log fit parameter a                    */
/*                      F_TYPE b   = log fit parameter b                    */
/*                      F_TYPE c   = log fit parameter c                    */
/*                                                                          */
/*      Return:         int        = # data points processed                */
/*                                                                          */
/*      Note:           All zeros treated as gaps in frequency data         */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/16/91    Created                                             */
/*          12/26/91    Changed to int returning # data points processed    */
/*                      Changed a, b & c from pointers to values            */
/*          12/31/91    Renamed                                             */
/*          01/18/92    Edited title block                                  */
/*          02/03/96    Modified for use as Win 3.1 DLL                     */
/*          01/01/98    Changes for MS VC++ compatibility & warnings        */
/*                                                                          */
/* (c) Copyright 1991-8  Hamilton Technical Services  All Rights Reserved  */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL RemoveLogFreqDrift(F_TYPE y[],
    F_TYPE a, F_TYPE b, F_TYPE c)
{
    int i;                                                      /* index */
    int num=0;                                     /* # points processed */

    for(i = (int)(*(y+1)+2); i <= (int)(*(y+2)+2); i++) /* analysis limits */
    {
        if(*(y+i))                                         /* check for gap */
        {
            *(y+i) -=  c + (a * (log((b * (i-2))+1)));      /* remove drift */
            num++;                                       /* increment count */
        }                                                         /* end if */
    }                                                            /* end for */

    return(num);                                 /* return # points processed */
}                                                /* end RemoveLogFreqDrift() */
```

The FrequenC Library

NAME:	**FUNCTION:**
RemoveQuadraticDrift	Remove the quadratic drift from phase data

SYNOPSIS:

```
int RemoveQuadraticDrift(F_TYPE x[], F_TYPE a, F_TYPE b, F_TYPE c)
```

F_TYPE x[]	Frequency data array:
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE a	Constant term, a, of $x(t)=a+bt+ct^2$ quadratic fit to be removed
F_TYPE b	Linear term, b, of quadratic fit to be removed
T_TYPE c	Quadratic term, c, of quadratic fit to be removed
RETURN: int	The # of non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are processed. Embedded zeros are retained as gaps in phase data. Use CalcQuadraticDrift() to calculate log fit to frequency data. No error codes are returned from this function.

EXAMPLE:

```
#include "frequenc.h"                      /* FrequenC header file */
F_TYPE x[513];                                 /* phase data array */
F_TYPE a, b, c, v;                               /* fit parameters */
int num;                                /* # data points processed */
.
.
CalcQuadraticDrift(x, &a, &b, &c, &v);          /* calc fit params */
       /* include error check for CalcQuadraticDrift() as required */
num=RemoveQuadraticDrift(x, a, b, c);         /* remove quad drift */
printf("\n# Data Points Processed = %d", num);      /* display num */
```

SEE ALSO: CalcLogFreqDrift()

REFERENCE: NIST Technical Note 1337, TN-264 to TN-295

```
/****************************************************************************/
/*                                                                          */
/*                         RemoveQuadraticDrift()                           */
/*                                                                          */
/*      Function to remove frequency drift from phase data                  */
/*      by removing quadratic fit x(t) = aùln(bt+1) + c                     */
/*                                                                          */
/*      Parameters:     F_TYPE x[] = phase data                             */
/*                                   x[0] = # phase data points             */
/*                                   x[1] = analysis start                  */
/*                                   x[2] = analysis end                    */
/*                      F_TYPE a  =  multiplicative term                    */
/*                      F_TYPE b  =  log term                               */
/*                      F_TYPE c  =  constant term                          */
/*                                                                          */
/*      Return:         int       = # data points processed                 */
/*                                                                          */
/*      Notes:          Embedded zeros treated as gaps in phase data.       */
/*                      Use FrequenC function CalcQuadraticDrift() to       */
/*                      calculate quadratic fit parameters a, b and c.      */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/16/90    Created                                             */
/*          12/31/91    Renamed                                             */
/*          01/18/92    Edited title block                                  */
/*          02/03/96    Modified for use as Win 3.1 DLL                     */
/*                                                                          */
/* (c) Copyright 1990-6  Hamilton Technical Services  All Rights Reserved  */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) RemoveQuadraticDrift(F_TYPE x[], F_TYPE a,
    F_TYPE b, F_TYPE c)
{
    int i;                                                      /* index */
    int num=0;                                     /* # points processed */

    for( i = *(x+1)+2; i <= *(x+2)+2; i++)        /* remove quadratic fit */
    {
        if(*(x+i))                                       /* check for gap */
        {
            *(x+i) -=  a + (b * i) + (c * i * i);      /* remove quad term */
            num++;                                     /* increment count */
        }                                                       /* end if */
    }                                                          /* end for */
    return(num);
}                                          /* end RemoveQuadraticDrift() */
```

The FrequenC Library

NAME: SpectrumCalc

FUNCTION: Calculate power spectrum

SYNOPSIS: `int SpectrumCalc(F_TYPE z[], float x[], float y[], float tau, int n, int win_num, int win_type, int pn_type, float corr, int af, int data_type)`

F_TYPE z[]	Time-domain data; z[0] = *z = # data points
float x[]	Real data points -> PSD results
float y[]	Im data points (0) -> Fourier freq results
float tau	Time interval of data samples
int n	# analysis data points
int win_num	# windowings
int win_type	Windowing method: Rect=0, Han=1, Ham=2
int pn_type	Phase noise type: 0=S_x(f), 1=S_ϕ(f), 2=£(f)
float corr	Corr factor: 0=1, 1=$(2\pi\nu_0)$ý, 2=($\pi\nu_0$)ý & x10
int af	Averaging factor (int power of 2)
int data_type	Time-domain data type: 0=phase, 1=freq
RETURN: int	# FFT frequencies

REMARKS:
# data points must be a power of 2
Data must not have any gaps

EXAMPLE:
```
 #include "frequenc.h"                          /* FrequenC header file */
 F_TYPE z[512];                                     /* time-domain data */
 float x[512], y[512];                                      /* FFT data */
 float tau=1.0, corr=0.0;
 int n=512, win_num=1, win_type=0, pn_type=2, af=1, data_type=0,
freqs;
 .
 .
 freqs=SpectrumCalc(z, x, y, tau, n, win_num, win_type, pn_type, corr,
 af, data_type);                                       /* call function */
 printf("\n# FFT Points = %d", freqs);           /* display # results */
```

SEE ALSO:

REFERENCE:

```
/****************************************************************************/
/*                                                                          */
/*                          SpectrumCalc()                                  */
/*                                                                          */
/*      Function to calculate & plot power spectrum                         */
/*      Uses FFT functions from Science & Engineering Tools                 */
/*      # data points must be a power of 2                                  */
/*      Data must not have any gaps                                         */
/*                                                                          */
/*      Function calls from Quinn-Curtis S&E Library:                       */
/*          WindowFFTData()                                                 */
/*          PowerSpectrumCalc()                                             */
/*                                                                          */
/*      Parameters:                                                         */
/*          y[]         = time-domain data; y[0] = *y = # data points       */
/*          xdata[]     = real data points -> PSD results                   */
/*          ydata[]     = imaginary data points (0) -> Fourier freq results */
/*          tau         = time interval of data samples                     */
/*          n           = # analysis data points                            */
/*          win_num     = # windowings                                      */
/*        2*win_type    = windowing method code:                            */
/*                  0   = Rectangular            win_type = 0               */
/*                  1   = Parzen (not used)                                 */
/*                  2   = Hanning                           1               */
/*                  3   = Welch (not used)                                  */
/*                  4   = Hamming                           2               */
/*                  5   = Blackman (not used)                               */
/*          phase_noise_type:  0=Sx(f), 1=So(f)    , 2=L(f)                 */
/*          correction_factor: 0=1    , 1=(2*ã*v)ý , 2=(ã*v)ý & x10         */
/*          af          = averaging factor (int power of 2)                 */
/*          data_type   = time-domain data type: 0=phase, 1=freq            */
/*      Return          = # FFT frequencies                                 */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/26/96    Adapted from spectrum_calc() of Stable/DOS          */
/*          04/12/97    Added code to preserve original data                */
/*          04/01/02    Added af parameter & trial code for averaging       */
/*          07/11/02    Added PSD data saving to disk                       */
/*          07/27/02    Added zero padding                                  */
/*          07/30/02    Revised to handle analysis limits                   */
/*          03/20/03    Adapted to FrequenC DLL                             */
/*                      Added data_type argument and return                 */
/*                                                                          */
/****************************************************************************/

// FrequenC header file
#include "frequenc.h"

// Local defines
#define realtype float
#ifndef RECTANG
#define RECTANG 0
#define PARZEN  1
#define HANNING 2
#define WELCH   3
#define HAMMING 4
#define EXACTB  5
#endif
```

```
// FFT Windowing Types
#define RECTANGULAR_WIN 0
#define HANNING_WIN     1
#define HAMMING_WIN     2
#define MULTITAPER_WIN  3

// Local function prototypes - not exported
double RemoveMean(float *x, int lx);
void WindowData(realtype *x,int numdat,int window);
void WindowFFTData(realtype *xreal,realtype *yimag, int numdat,int window);
void fft(realtype *xreal, realtype *yimag, int numdat, int flag);
void PowerSpectrumCalc(realtype *xreal, realtype *yimag,int numdat,realtype delta);
void fft2 (realtype *aa, realtype *bb, int n, int m, int ks);
void reorder (realtype *aa, realtype *bb, int n, int m, int ks, int reel);

// SpectrumCalc() top-level function
int __declspec(dllexport) FAR PASCAL SpectrumCalc(F_TYPE y[], float xdata[],
    float ydata[], float tau, int n, int win_num, int win_type,
    int phase_noise_type, float correction_factor, int af, int data_type)
{
    int i;                          // General index
    int j;                          // Index for # windowings
    int k;                          // Index for averaging
    static int offset;              // Index of first analysis data point
    static int nn;                  // Padded array size
    static int nNum;                // # data points in input array
    static int nFFT;                // # data points in one FFT section
    static float pad_factor;        // Correction factor for zero padding
    static float correction[3][4] =
    {
         { (float)0.00, (float)0.000, (float)0.000, (float)0.000 },
            // Rectangular
         { (float)0.00, (float)0.426, (float)0.563, (float)0.647 },
            // Hanning
         { (float)0.00, (float)0.401, (float)0.542, (float)0.627 }
            // Hamming
    };

   // These are log power amplitude corrections factors for wideband noise
   // 1st index is win_type; 2nd index is win_num
   static float bw_correction;
   static float log_mult;
   static F_TYPE avg;

    // The xdata array gets loaded with the time domain data.
    // This array is allocated with space for the # points & analysis limits.
    // These are put into the xdata array initially to allow it to use the
    // FillFloatGaps() function.
    // Then the header info is removed from the xdata array for the FFT calcs.

   // Input parameter n the # of analysis points - not necessarily a power-of-2

   // Find data offset
    offset=(int)y[1]+2; // Index of first data point to be analyzed

   // Copy analysis limits info into xdata[]
    xdata[0]=(float)n;
    xdata[1]=(float)1;
    xdata[2]=(float)n;
```

```
// Load analysis data into the xdata array with header info
for(i=0; i<n; i++)
{
    xdata[i+ARRAY_OFFSET]=(float)y[i+offset];
}

// Make sure no there are gaps in the analysis data
FillFloatGaps(xdata, data_type);

// Eliminate header from the analysis data
for(i=0; i<n; i++)
{
    xdata[i]=xdata[ARRAY_OFFSET+i];                      /* real data points */
    ydata[i]=0.0;                               /* imaginary data points */
}

// Remove DC component
RemoveMean(xdata, n);

// The original FFT code just analyzes the entire data set as one block
// w/o averaging.  Thus a longer record at the same tau increases the
// frequency resolution but does not reduce the variance of the FFT
// results.  To do that, one must break the data into sections, analyze
// each section separately, and then average them to produce better FFT
// values.  Doing so obviously shortens each individual data record, so
// the FFT results won't extend to as low a Fourier frequency.  The variance
// of the FFT values is thereby reduced from 100% to 100% divided by the
// sqrt of the # averages, which should be an integer power of 2.
// The averaged FFT can be done in the same pair of xdata and ydata arrays.
// The xdata array starts with the real data points, and is filled with the
// PSD results.  The ydata array starts with 0's and gets the Fourier freqs.
// For averaging, just FFT each section separately in-place.  Then sum the
// PSDs and Fourier freqs in the 1st section, and divide them by # averages.
// Then continue to apply BW correftion and log mult to 1st section.
// Both nNum and nAF are integer powers of 2, so k is also
// e.g. if n is 512 and nAF is 8 then k goes from 0 to 7
// and each section of FFT has nFFT=64 points.

// Find size of next higher power-of-two = allocated size
nn=(int)pow(2.0, (double)(ceil(log(n)/log(2))));

// Zero the padded portion of the data
for(i=n; i<nn; i++)
{
    xdata[i]=0.0;                                    /* real data points */
    ydata[i]=0.0;                               /* imaginary data points */
}

// Zero padding correction factor
// Square of ratio of total to actual (non-zero) FFT points
pad_factor=((float)(nn)*(float)(nn))/((float)(n)*(float)(n));

// Calc the # of averaged FFT points
nFFT=nn/af;

// Perform FFT on each section
for(k=0; k<af; k++)
{
    // Apply the desired windowing function
    j=win_num;
    while(j--)
    {
        WindowFFTData(&xdata[k*nFFT], &ydata[k*nFFT], nFFT, win_type);
                                                /* window win_num times */
    }

    // Calculate the power spectrum using S&E function
    PowerSpectrumCalc(&xdata[k*nFFT], &ydata[k*nFFT], nFFT, tau);
                                                  /* calc power spectrum */
}

// Sum the results into the 1st section
// Note that summation starts a k=1 (not k=0) - don't add 1st section to itself
for(k=1; k<af; k++)
{
    for(i=0; i<nFFT; i++)
    {
        xdata[i]+=xdata[k*nFFT+i];
        ydata[i]+=ydata[k*nFFT+i];
    }
}

// Find the average values in 1st section
for(i=0; i<nFFT; i++)
{
    xdata[i]/=af;
    ydata[i]/=af;
}

// Calc bandwidth correction
bw_correction=(float)log10(ydata[1]);

// Apply zero padding correction
correction_factor*=pad_factor;

// Set log multiplier factor
if(phase_noise_type==2)                                          /* L(f) */
{
    log_mult=10;
}
else                                                    /* Sx(f) & So(f) */
{
    log_mult=1;
}

// Drop dc term from PSD and convert spectral data to log form
// Note that xdata is PSD & ydata is Fourier freq
for(i=0; i<(nFFT/2)-1; i++)
{
    ydata[i]=ydata[i+1];                    /* Fourier frequency values */
    if(xdata[i+1]>0)
    {
        xdata[i]=(float)(log_mult*(log10(xdata[i+1]*correction_factor)
                 - bw_correction + correction[win_type/2][win_num]));
                        /* power spectrum values plus correction factors */
    }
```

```
        else
        {
            // Error in power spectrum calc
            break;
        }
    }

    // Save PSD data - Must move to Stable32 calling function
    // PSDWrite(szPSDFile, ydata, xdata, (nFFT/2)-1);

    return (nFFT/2)-1;
}

/******************************************************************************/
/*                                                                            */
/*      Quinn-Curtis Science & Engineering Library FFT Functions              */
/*                                                                            */
/*      TC-006 S/N 18146 Rev 8.01 Date: 6/1/92 File: FFT.C (Excerpt)          */
/*                                                                            */
/*      Functions:                                                            */
/*      Parzen()    Hanning()       Hamming()         ExactBlackman()         */
/*      Welch()     WindowData()    WindowFFTData()   fft()                   */
/*      fft2()      reorder()       SumAnd Square()   PowerSpectrumCalc()     */
/*                                                                            */
/*      Note: The Stable/Win usage of these functions is with realtype        */
/*      defined as a float.  The local header fft_.h is a special version     */
/*      of fft.h that calls in reeltype.h which defines realtype as float     */
/*      and has prototypes for these functions.                               */
/*                                                                            */
/*      Copyright (c) Quinn-Curtis, 1992                                      */
/*                                                                            */
/******************************************************************************/

// File macros
#ifndef M_PI // Defined in math.h under Borland C++ but not Visual C++
#define M_PI        3.14159265358979323846
#endif

/* Apply a parzen window */
realtype Parzen (realtype n, realtype j)
{
    realtype fret;

    fret = (realtype) (1.0 - fabs((j - 0.5 * (n - 1.0)) / (0.5 * (n - 1.0))));
    return(fret);
}

/* Apply a hanning window */
realtype Hanning (realtype n, realtype j)
{
    realtype fret;

    fret =  (realtype) (0.5 * (1.0 - cos(2.0 * M_PI * j / (n-1.0))));
    return(fret);
}
```

```
/* Apply a hamming window */
realtype Hamming (realtype n, realtype j)
{
    realtype fret;

    fret =  (realtype) (0.54 - 0.46 * cos(2.0 * M_PI * j / (n-1.0)));
    return(fret);
}

/* Apply a blackman window */
realtype ExactBlackman (realtype n, realtype j)
{
    realtype fret;

    fret =  (realtype) (0.42 - 0.50 * cos(2.0 * M_PI * j /(n-1.0)) +
            0.08 * cos(4.0 * M_PI * j / (n-1.0)));
    return(fret);
}

/* Apply a welch window */
realtype Welch (realtype n, realtype j)
{
    realtype fret;
    fret =  (realtype) ((j - 0.5 * (n - 1.0)) / (0.5 * (n + 1.0)));
    fret = (realtype) (1.0 - fret*fret);
    return(fret);
}

/* Apply a window for an array */
void WindowData (realtype *x, int numdat, int window)
{
   int i;
   realtype multiplier;

   for ( i = 0; i < numdat; i++ )
   {
      switch ( window )
      {
         case RECTANG:
            multiplier = (realtype) 1.0;
            break;
         case PARZEN:
            multiplier = Parzen ((realtype)numdat, (realtype)i);
            break;
         case HANNING:
            multiplier = Hanning ((realtype)numdat, (realtype)i);
            break;
         case WELCH:
            multiplier = Welch ((realtype)numdat, (realtype)i);
            break;
         case HAMMING:
            multiplier = Hamming ((realtype)numdat, (realtype)i);
            break;
         case EXACTB:
            multiplier = ExactBlackman ((realtype)numdat, (realtype)i);
            break;
         default:
            multiplier =  (realtype) 1.0;
            break;
      }
      x[i] = multiplier * x[i];
   }
}

/* Apply a window for both x and y data sets */
void WindowFFTData (realtype *xdata, realtype *ydata, int numdat, int window)
{
    WindowData (xdata, numdat, window);
    WindowData (ydata, numdat, window );
}

/* Complex Fourier Transform */
/* number of points n must be a power of 2 */
void fft (realtype *xr, realtype *yi, int n, int inverse)
{
    int nn, j;
    int      m;
    realtype pp, qq;

    m = -1;
    nn = n;

    /* determine power of 2 */
    while (nn > 0)
    {
        nn = nn >> 1;
        m++;
    }

    fft2 (xr, yi, n, m, n);
    reorder (xr, yi, n, m, n, 0);
    if (inverse)
    {
        pp = (realtype) 1.0 / n;
        qq = -pp;
        for (j = 0; j < n; j++)
        {
            xr [j] *= pp;
            yi [j] *= qq;
        }
    }
    else
    {
        for (j = 0; j < n; j++)
            yi[j] = -yi[j];
    }
}

/* FFT for one variable of dimension n = 2^m */
void fft2 (realtype *x, realtype *y, int n, int m, int ks)
{
    int        indx, k0, k1, k2, k3, span, j, k, kb, kn, mm, mk;
    realtype    rad, c1, c2, c3, s1, s2, s3, ck, sk, sq;
    realtype    x0, x1, x2, x3, y0, y1, y2, y3;
    int *cc;

    cc = (int *) calloc (m + 1, sizeof (int));

    sq = (realtype) 0.707106781187;
    sk = (realtype) 0.382683432366;
    ck = (realtype) 0.92387953251;

    cc[m] = ks;
    mm = (m / 2) * 2;
    kn = 0;
    for (k = m - 1; k >= 0; k--)
        cc[k] = cc[k+1] / 2;
    rad = (realtype) 6.28318530718 / (cc[0] * ks);
    mk = m - 5;

    do
    {
        kb = kn;
        kn = kn + ks;

        if (mm != m)
        {
            k2 = kn;
            k0 = cc[mm] + kb;
            do
            {
                k2--;
                k0--;
                x0 = x [k2];
                y0 = y [k2];
                x[k2] = x[k0] - x0;
                x[k0] = x[k0] + x0;
                y[k2] = y[k0] - y0;
                y[k0] = y[k0] + y0;
            }  while (k0 > kb);
        }

        c1 = (realtype) 1.0, s1 = (realtype) 0.0;
        indx = 0;
        k = mm - 2;
        j = 3;

        if (k >= 0)
            goto l2;
        else
            if (kn < n)
                continue;
            else
                break;

l1:     while (1)
        {
            if (cc[j] <= indx)
            {
                indx = indx - cc[j];
                j--;
                if (cc[j] <= indx)
                {
                    indx = indx - cc[j];
                    j--;
                    k += 2;
                }
            }
            else
                break;
        }

        indx = cc[j] + indx;
        j = 3;

l2:     span = cc[k];
        if (indx != 0)
        {
            c2 = indx * span * rad;
            c1 = (realtype) cos (c2);
            s1 = (realtype) sin (c2);
l5:
            c2 = c1 * c1 - s1 * s1;
            s2 = (realtype) 2.0 * s1 * c1;
            c3 = c2 * c1 - s2 * s1;
            s3 = c2 * s1 + s2* c1;
        }
        for (k0 = kb + span - 1; k0 >= kb; k0--)
        {
            k1 = k0 + span;
            k2 = k1 + span;
            k3 = k2 + span;
            x0 = x[k0];
            y0 = y [k0];
            if (s1 == 0)
            {
                x1 = x[k1]; y1 = y [k1];
                x2 = x[k2]; y2 = y [k2];
                x3 = x[k3]; y3 = y [k3];
            }
            else
            {
                x1 = x[k1] * c1 - y [k1] * s1;
                y1 = x[k1] * s1 + y [k1] * c1;
                x2 = x[k2] * c2 - y [k2] * s2;
                y2 = x[k2] * s2 + y [k2] * c2;
                x3 = x[k3] * c3 - y [k3] * s3;
                y3 = x[k3] * s3 + y [k3] * c3;
            }
            x [k0] = x0 + x2 + x1 + x3; y[k0] = y0 + y2 + y1 + y3;
            x [k1] = x0 + x2 - x1 - x3; y[k1] = y0 + y2 - y1 - y3;
            x [k2] = x0 - x2 - y1 + y3; y[k2] = y0 - y2 + x1 - x3;
            x [k3] = x0 - x2 + y1 - y3; y[k3] = y0 - y2 - x1 + x3;
        }
        if (k > 0)
        {
            k = k - 2;
            goto l2;
        }
        kb = k3 + span;
        if (kb < kn)
        {
            if (j == 0)
            {
                k = 2;
                j = mk;
                goto l1;
            }
            j--;
            c2 = c1;
            if (j == 1)
            {
                c1 = c1 * ck + s1 * sk;
                s1 = s1 * ck - c2 * sk;
            }
            else
            {
                c1 = (c1 - s1) * sq;
                s1 = (s1 + c2) * sq;
            }
            goto    l5;
        }
    }
    while (kn < n);

    free (cc);
}

void reorder ( realtype *x, realtype *y, int n, int m, int ks, int reel)
{
    int i, j, indx, k, kk, kb, k2, ku, lim, p;
    realtype  temp;

    int *cc, *lst;

    cc = (int *) calloc (m + 1, sizeof (int));
    lst = (int *) calloc (m + 1, sizeof (int));

    cc [m] = ks;
    for (k = m; k > 0; k--)
        cc [k-1] = cc [k] / 2;
    p = j = m - 1;
    i = kb = 0;
    if (reel)
    {
        ku = n - 2;
        for (k = 0; k <= ku; k += 2)
        {
            temp = x [k + 1];
            x [k + 1] = y [k];
            y [k] = temp;
        }
    }
    else
        m--;
    lim = (m + 2) / 2;
    if (p > 0)
    do
    {
        ku = k2 = cc [j] + kb;
        indx = cc [m - j];
        kk = kb + indx;

        do
        {
             k = kk + indx;
             do
             {
                 temp = x [kk];
                  x [kk] = x [k2];
                  x [k2] = temp;
                  temp = y [kk];
                  y [kk] = y [k2];
                  y [k2] = temp;
                  kk++;
                  k2++;
             }
             while (kk < k);

             kk = kk + indx;
             k2 = k2 + indx;
        }
        while (kk < ku);

        if (j > lim)
        {
            j--, i++;
            lst [i] = j;
            continue;
        }
        kb = k2;
        if (i > 0)
        {
            j = lst [i];
            i--;
            continue;
        }
        if (kb < n)
            j = p;
        else
            break;
    }
    while (1);

    free (cc);
    free (lst);
}

realtype SquareAndSum (realtype a, realtype b)
{
   realtype fret;

   fret = a * a + b * b;
   return(fret);
}

/* Calculate the power spectrum periodogram of a sampled dataset */
void PowerSpectrumCalc(realtype *xreal, realtype *yimag,
    int numdat, realtype delta)
{
    realtype timespan, normal, n;
    int i;

    n = (float)numdat;
    normal =  (realtype) 1.0 / (n * n);
    fft (xreal, yimag, numdat, 0);
    xreal[0] = SquareAndSum (xreal[0], yimag[0]) * normal;

    for (i = 1; i <= (numdat / 2) - 1; i++)
    {
        xreal[i] = normal * (SquareAndSum(xreal[i], yimag[i]) +
        SquareAndSum (xreal[numdat - i], yimag[numdat - i]));
    }
    i = numdat / 2;
    xreal[i] = SquareAndSum(xreal[i], yimag[i]) * normal;
    timespan = (realtype) ((1.0*numdat) * delta);
    for (i = 0; i <= (numdat/2); i++)
    {
        yimag[i] = (realtype) (1.0*i) / timespan;
    }
}

/* Remove mean from data array */
/* Cloned from same code used in multitaper PSD calc */
double RemoveMean(float *x, int lx)
{
    int k;
    double mean;

    mean = 0.;
    if(lx<2)
    {
        return mean;
    }

    for(k=0; k<=lx; k++)
    {
        mean= x[k]+mean;
    }

    mean=mean/(float)lx;

    for(k=0; k<=lx; k++)
    {
        x[k]=(float)(x[k]-mean);
    }

    return mean;
}
```

The FrequenC Library

NAME: ScaleData	**FUNCTION:** Scale phase or frequency data by a+bx

SYNOPSIS: `int ScaleData(F_TYPE z[], F_TYPE add, F_TYPE mult)`

F_TYPE z[]	Phase or frequency data array:
	z[0] = # data points
	z[1] = analysis start point
	z[2] = analysis end point

F_TYPE add	Addend, the number to be added to all non-gap data points
F_TYPE mult	Multiplier, the number to multiply all data points by
RETURN: int	The # of non-gap data points processed

REMARKS:
Only data points between start and end analysis limits are modified.
Multiplication is done first, then addition.
Embedded zeros are treated as gaps in phase data.
All zeros are treated as gaps in frequency data.
No error codes are returned from this function.

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
F_TYPE z[512];                                          /* data array */
F_TYPE add;                                                 /* addend */
F_TYPE mult;                                            /* multiplier */
int num;                                 /* # data points processed */
.
.
add=1.0;                                          /* set addend value */
mult=5.0;                                     /* set multiplier value */
num=ScaleData(z, add, mult);                            /* scale data */
printf("\n# Data Points Scaled = %d", num);            /* display num */
```

SEE ALSO: None

REFERENCE: None

```
/****************************************************************************/
/*                                                                          */
/*                          ScaleData()                                     */
/*                                                                          */
/*      Function to scale phase or frequency data by multipling by a        */
/*      multiplier constant and adding an addend constant:                  */
/*      z[i] = addend + z[i] * multiplier                                   */
/*                                                                          */
/*      Parameters:     F_TYPE z[]  = data array                            */
/*                                    z[0] = # data points                  */
/*                                    z[1] = analysis start                 */
/*                                    z[2] = analysis end                   */
/*                      F_TYPE add  = addend                                */
/*                      F_TYPE mult = multiplier                            */
/*                      int datype  = data type: 0=phase, 1=frequency       */
/*                                                                          */
/*      Notes:          All zeros treated as gaps in frequency data         */
/*                      Embedded zeros are gaps in phase data               */
/*                                                                          */
/*      Return:         int          = # non-gap values processed           */
/*                                                                          */
/*      Revision record:                                                    */
/*          12/16/91    Created                                             */
/*          12/26/91    Changed to int returning # data points processed    */
/*                      Added datype parameter; changed array to z[]        */
/*                      Changed add and mult to values not pointers         */
/*          12/31/91    Renamed                                             */
/*          01/18/92    Edited title block                                  */
/*          02/03/96    Modified for use as Win 3.1 DLL                     */
/*          01/01/98    Changes for MS VC++ compatibility & warnings        */
/*                                                                          */
/* (c) Copyright 1991-8  Hamilton Technical Services  All Rights Reserved   */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL ScaleData(F_TYPE z[], F_TYPE add,
    F_TYPE mult, BOOL datype)
{
    int i;
    int num=0;

    for(i = (int)(*(z+1)+2); i <= (int)(*(z+2)+2); i++)
    {
        if(*(z+i) || (!datype && i==3 || !datype && i==z[0]+2))
        {
            *(z+i) *=  mult;                /* multiply data by multiplier */
            *(z+i) +=  add;                          /* add addend to data */
            num++;
        }                                                     /* end if */
    }                                                        /* end for */

    return(num);
}                                                  /* end ScaleData() */
```

The FrequenC Library

NAME: TotvarBias	**FUNCTION:** Calculate TOTVAR bias
SYNOPSIS: float TotvarBias(int nAlpha, float fRatio)	
int alpha	Alpha of power law noise type (-2 to +2)
float ratio	Record length/tau = $T/\tau = \tau_0 \cdot N / \tau$
RETURN: float	TOTVAR bias factor, or -1 if error

REMARKS:
The TOTALVAR bias factor is the ratio of the expected value of TOTVAR to AVAR. To correct a calculated TOTDEV value for a certain noise type, record length & analysis tau, divide it by the square root of the bias factor returned by this function. It applies to $-2 \geq \alpha \geq 2$ (W PM, F PM, W FM, F FM, RW FM noise), and for $T/\tau \geq 2.0$, where $T = \tau_0 \cdot \#$ phase data points. No bias correction is necessary for W PM, F PM & W FM noise ($\alpha = 2$, 1 or 0).

EXAMPLE:
```
#include "frequenc.h"                          /* FrequenC header file */
int alpha=0;                                                  /* alpha */
float ratio=10.0;                                /* record length/tau */
float bias;                                     /* TOTVAR bias factor */
.
.
bias=TotvarBias(alpha, ratio);                        /* call function */
if(bias==-1.0)                                      /* check for error */
{
    printf("\nError");                                /* error message */
}
else
{
    printf("\nTOTVAR Bias = %e", bias);              /* display result */
}
```

SEE ALSO: TotvarEDF(), TotvarCalc()

REFERENCES: D.A. Howe, "Method of Improving the Estimation of Long-Term Frequency Variance", Proc. 11th EFTF, March, 1997, and private e-mail communication from C.A. Greenhall/JPL via D.A. Howe/NIST 03/28/97.

```
/*****************************************************************************/
/*                                                                           */
/*                              TotvarBias()                                 */
/*                                                                           */
/*      Function to calculate the TOTVAR bias function                       */
/*                                                                           */
/*      Parameters:     int   nAlpha = noise type (-2 to +2)                 */
/*                      float fRatio = T/ç                                   */
/*                                                                           */
/*                                                                           */
/*      Return:         float fBias  = TOTVAR bias factor                    */
/*                                     or -1 if error                        */
/*                                                                           */
/*      The TOTVAR bias factor is the ratio of the expected value of         */
/*      TOTVAR to AVAR.  To correct a calculated TOTDEV value for a          */
/*      certain noise type, record length & analysis tau, calc the T/ç       */
/*      ratio, call this function, and divide the uncorrected TOTDEV value   */
/*      by the square root of the bias factor returned by this function.     */
/*                                                                           */
/*      Notes:          (1) This function applies to -2 >= alpha >= 2        */
/*                      (W PM, F PM, W FM, F FM, RW FM noise)                */
/*                      (2) This function applies only for T/ç  >= 2.0       */
/*                      (3) T is defined as ç0 * # phase data points         */
/*                      (4) No bias correction is necessary for W PM,        */
/*                          F PM & W FM noise (alpha=2, 1 or 0)              */
/*                                                                           */
/*      Reference:      D.A. Howe and C.A. Greenhall, "Total Variance: A     */
/*                      Progress Report on a New Frequency Stability         */
/*                      Characterization", Proc. 29th PTTI Meeting,          */
/*                      Dec. 2, 1997, (to be published).                     */
/*                                                                           */
/*                      TOTVAR Bias Factor =  1 - a * (tau / T)              */
/*                      where   a = 0 for W FM noise (no bias) (alpha=0)     */
/*                                = 1/(3 ln 2) for F FM noise  (alpha=-1)    */
/*                                = 3/4        for RW FM noise (alpha=-2)    */
/*                                                                           */
/*                      Note that the fRatio entered is T/tau not tau/T!     */
/*                                                                           */
/*      Revision record:                                                     */
/*          12/06/97    Created & debugged                                   */
/*          12/09/97    Added to FrequenC Library                            */
/*          12/21/97    Fixed error reported by D. Howe/NIST:                */
/*                      F FM and RW FM constants were reversed               */
/*          01/01/98    Changes for MS VC++ compatibility & warnings         */
/*                                                                           */
/* (c) Copyright 1997   Hamilton Technical Services   All Rights Reserved    */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL TotvarBias(int nAlpha, float fRatio)
{
    // Local variables
    float fBias;
    // Verify parameters
    if(fRatio<2.0)
    {
        return -1.0; // Error
    }

    if(nAlpha<-2 || nAlpha>2)
    {
        return -1.0; // Error
    }

    // Sort by noise type
    switch(nAlpha)
    {
        case -2: // RW FM
        {
            fBias=(float)(1.0 - 0.7500/fRatio);
        }
        break;

        case -1: // F FM
        {
            fBias=(float)(1.0 - 0.4809/fRatio);
        }
        break;

        case -0: // W FM
        {
            fBias=1.0;
        }
        break;

        case 1: // F PM
        {
            fBias=1.0;
        }
        break;

        case 2: // W PM
        {
            fBias=1.0;
        }
        break;
    }

    return (fBias);
}
```

The FrequenC Library

NAME: TotvarCalc	**FUNCTION:** Calculate TOTVAR using doubly reflected phase data

SYNOPSIS: `int TotvarCalc(F_TYPE x[], F_TYPE *totdev, F_TYPE tau, int af, void *progress)`

`F_TYPE x[]`	Phase data array:
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

`F_TYPE *totdev`	Pointer to total deviation
`F_TYPE tau`	Averaging time of data τ_0
`int af`	Averaging factor τ/τ_0
`void *progress( int percent, char *msg)`	Pointer to function to display progress, with int arg set to % calc complete, and char* arg set to text message to display. Can be NULL.
RETURN: int	# TOTVAR analysis points, or -1 if memory alloc error, or -2 if no result error

REMARKS:
Calculation is done in new array that is deleted after function closes.
The progress indicator is not necessary.
The phase data need not be end matched before calling this function.

EXAMPLE:
```
#include "frequenc.h"                         /* FrequenC header file */
F_TYPE x[512];                                         /* data array */
F_TYPE totdev;                                             /* TOTDEV */
F_TYPE tau=1.0;                                    /* meas interval */
int af=2;                                              */ avg factor */
int num;                                              /* return code */
.
.
num=TotvarCalc(x, totdev, tau, af, NULL);          /* call function */
if(num<0)                                         /* check for error */
{
    printf("\nError");                             /* error message */
}
else
{
    printf("\nTOTDEV = %e", totdev);              /* display result */
}
```

SEE ALSO: TotvarEDF(), TotvarBias()

REFERENCE: D.A. Howe and C.A. Greenhall, "Total Variance: A Progress Report on a New Frequency Stability Characterization", Proc. 29th PTTI Meeting, Dec. 1997.

```
/*****************************************************************************/
/*                                                                           */
/*                          TotvarCalc()                                     */
/*                                                                           */
/*      Function to calculate TOTVAR using doubly reflected phase data       */
/*                                                                           */
/*      Parameters:     F_TYPE x[]      =   phase data                       */
/*                                          x[0] = # data points             */
/*                                          x[1] = analysis start            */
/*                                          x[2] = analysis end              */
/*                      F_TYPE *TotDev= pointer to TOTDEV                    */
/*                      F_TYPE fTau     = interval between phase data points */
/*                      int nAF         = averaging factor                   */
/*                      void *lpfnProgress = ptr to progress display funct   */
/*                                                                           */
/*      Return:         int           = # TOTALVAR analysis points,          */
/*                                      or -1 if memory alloc error          */
/*                                      or -2 if no result error             */
/*                                                                           */
/*      Notes:          1. Calculation is done entirely with a new array.    */
/*                         that is deleted after this function closes.       */
/*                      2. Progress indicator and abort not necessary.       */
/*                      3. The phase data need NOT be endmatched before      */
/*                         calling this function.                            */
/*                                                                           */
/*      Reference:      D.A. Howe and C.A. Greenhall, "Total Variance: A     */
/*                      Progress Report on a New Frequency Stability         */
/*                      Characterization", Proc. 29th PTTI Meeting,          */
/*                      Dec. 2, 1997, (to be published).                     */
/*                                                                           */
/*      Revision record:                                                     */
/*          12/06/97    Adapted from ReflectedTotalvarCalc()                 */
/*          12/09/97    Added to FrequenC Library                            */
/*          01/01/98    Changes for MS VC++ compatibility & warnings         */
/*                                                                           */
/* (c) Copyright 1997-8 Hamilton Technical Services   All Rights Reserved    */
/*                                                                           */
/*****************************************************************************/

// Headers
#include "frequenc.h"
#include <windowsx.h>   // For GlobalAllocPtr() & GlobalAllocFree()

// For Win 3.1 and Win32 Compatibility
#ifndef WIN32
#define HUGE huge
#else
#define HUGE
#endif

// Defines
#define ARRAY_OFFSET 3  // Offset to 1st data point

int __declspec(dllexport) FAR PASCAL TotvarCalc(F_TYPE fX[], F_TYPE *fTotDev,
    F_TYPE fTau, int nAF, void *lpfnProgress)
{
    // Local variables
    int i;              // Index
    int j;              // Auxilary index
    int nNum;           // # points
    int nStart;          // Analysis start
    int nDummy;          // Dummy variable to avoid complier warning
    int nCount=0;        // # analysis points

    #if(0) // Not needed if no progress indicator
    int nTic;            // Increment for progress indicator
    int nLast_i=0;       // Previous value of index
    #endif

    double HUGE *fP;     // TOTVAR phase array pointer
    F_TYPE fSum=0;       // TOTVAR summation
    F_TYPE fEdge;        // Phase data value at edge of reflection

    // Avoid compiler warning if no progress indicator
    lpfnProgress=lpfnProgress;

    // Initializations
    nStart=(int)fX[1];          // Analysis start point
    nNum=(int)(fX[2]-fX[1]+1);  // # analysis phase data points = N

    #if(0) // Not needed if no progress indicator
    nTic=(int)ceil(nNum/100);   // Increment for progress report

    if(lpfnProgress!=NULL)
    {
        lpfnProgress(0, "Calculating Total Sigma"); // Label progress bar
    }
    #endif

    #if(0) // For testing - check input params
    TRACE_EXP(fTau);
    TRACE_INT(nStart);
    TRACE_INT(nNum);
    #endif

    // Allocate a new "virtual" phase data array to size 3N-4, where N is
    // the # of phase data points to be analyzed.  This virtual array is the
    // result of extension by reflection about both endpoints.  N-2 reflected
    // points are added at both ends of the original phase data to be analyzed.
    // Do NOT need the array header (#, Start, End)
    // Note that fP is cast to a huge double ptr and that array size is long
    if((fP=(double HUGE *)GlobalAllocPtr(GHND,
        (long)(3*nNum-4)*sizeof(double)))==NULL)
    {
        #if(0) // Eliminated for FrequenC DLL usage
        AllocErrorMessage();
        #endif
        return(-1);  // Error - memory reallocation failed
    }

    // Note that in the referenced paper the index of the virtual phase data
    // array goes from 3-N (a negative number) to 2N-2 (a positive number),
    // with the original data having indices from 1 to N.  Our indices start at
    // 0 and go to 3N-5.  The lower reflected data has indices from 0 to N-3.
    // The original data in the middle of the virtual array has indices from
    // N-2 to 2N-3. The upper reflected data has indices from 2N-2 to 3N-5.

    // Copy original phase data array fX[] to (headerless) working array fP[]
    // Analysis limits are handled by copying only the points between the
    // start and end.  i is the index into the virtual array fP[].  j is the
    // variable part of the index into the original phase data array fX[].
    j=0;
    for(i=nNum-2; i<2*nNum-2; i++)
    {
        fP[i]=fX[j+nStart+ARRAY_OFFSET-1];
        j++;
    }

    // Fill the lower reflected phase data
    // These values are twice the first phase data point minus the particular
    // data point value to be reflected
    j=0;
    fEdge=2*fP[nNum-2];

    for(i=0; i<nNum-2; i++)
    {
        fP[i]=fEdge-fP[2*nNum-4-j];
        j++;
    }

    // Fill the upper reflected phase data
    // These values are twice the last phase data point minus the particular
    // data point value to be reflected
    j=0;
    fEdge=2*fP[2*nNum-3];

    for(i=2*nNum-2; i<3*nNum-4; i++)
    {
        fP[i]=fEdge-fP[2*nNum-4-j];
        j++;
    }

    #if(0) // For testing - examine virtual array
    for(i=0; i<3*nNum-4; i++)
    {
        TRACE_EXP(fP[i]);
    }
    #endif

    // Calc TOTVAR - See Eq (3) of Reference
    for(i=nNum-1; i<2*nNum-3; i++)
    {
        #if(1) // To eliminate gap checking
        // Check for gap
        if( (fP[i-nAF] && fP[i] && fP[i+nAF]) ||
            (i==nNum+nAF-2                  ) ||
            (i==2*nNum-nAF-3                ) )
        #endif
        {
            // Sum 2nd differences
            fSum+=SQR(fP[i-nAF]-2*fP[i]+fP[i+nAF]);
            nCount++;
        }
        #if(0) // For testing - display gap index
        else
        {
            TRACE_INT(i);
        }
        #endif
```

```
            #if(0) // Not needed if no progress indicator or abort
            // Display progress
            // Note:  This code provides updates only if index has changed
            // by at least tic amount
            // Uses pointer to progress display function PFVi argument
            // Pass NULL for no progress display
            if(((i-nLast_i)>=nTic) && (lpfnProgress!=NULL))
            {
                lpfnProgress(0.5+100.0*i/nNum, "Calculating Total Sigma");
                nLast_i=i;
            }

            // Check for abort
            if(bAbort)
            {
                nCount=0;                                               /* force error */
                break;
            }
            #endif
        }


        #if(0) // For testing - display # points analyzed
        TRACE_INT(nCount);
        #endif

        // Scale result - See Eq (3) of Reference
        fSum/=(fTau*fTau*nAF*nAF*2*nCount);

        // Find TOTDEV
        if(nCount && fSum>0)
        {
            *fTotDev=sqrt(fSum);
        }
        else // No result error
        {
            *fTotDev=0.0;
            nCount=-2;
        }

        // Free memory
        nDummy=GlobalFreePtr(fP);
        nDummy=nDummy;

        return(nCount);  // Return # phase analysis points
    }
```

The FrequenC Library

NAME: TotvarEDF	**FUNCTION:** Determine the equivalent # of chi-squared degrees of freedom for the total variance.
SYNOPSIS: float TotvarEDF(int alpha, float ratio)	
int alpha	Alpha of power law noise process
float ratio	$T/\tau = \tau_0 \cdot N/\tau$ = Record length/averaging time
RETURN: float	float edf = equivalent # of degrees of freedom or -1 if error

REMARKS:
This function applies only to $-2 \leq \alpha \leq 0$ (W FM, F FM and RW FM noise), and for $T/\tau \geq 2.0$, where $T = \tau_0 \cdot$ # phase data points.

EXAMPLE:
```
 #include "frequenc.h"                           /* FrequenC header file */
 int alpha=0;                                                  /* alpha */
 float ratio=10.0;                                             /* ratio */
 float edf;                                                     */ edf */
 .
 .
 edf=TotvarEDF(alpha, ratio);                          /* call function */
 if(num==-1)                                         /* check for error */
 {
     printf("\nError");                                /* error message */
 }
 else
 {
     printf("\nEDF = %f", edf);                       /* display result */
 }
```

SEE ALSO: TotvarCalc(), TotvarBias()

REFERENCE: D.A. Howe and C.A. Greenhall, "Total Variance: A Progress Report on a New Frequency Stability Characterization", Proc. 29th PTTI Meeting, Dec. 1997.

```
/*****************************************************************************/
/*                                                                           */
/*                             TotvarEDF()                                   */
/*                                                                           */
/*  Function to determine the estimated # of chi-squared degrees of freedom  */
/*  for the total variance (TOTVAR) of a power law noise process.            */
/*                                                                           */
/*  Params:         int   nAlpha = alpha                                     */
/*                  float fRatio = T/ç                                       */
/*                                                                           */
/*  Return:         float edf = estimated degrees of freedom                 */
/*                              or -1 if error                               */
/*                                                                           */
/*  Notes:          (1) This function applies only to -2 >= alpha >= 0      */
/*                      (W FM, F FM and RW FM noise)                         */
/*                  (2) This function applies only for T/ç  >= 2.0          */
/*                  (3) T is defined as ç0 * # phase data points             */
/*                                                                           */
/*  Reference:      D.A. Howe and C.A. Greenhall, "Total Variance: A         */
/*                  Progress Report on a New Frequency Stability             */
/*                  Characterization", Proc. 29th PTTI Meeting,              */
/*                  Dec. 2, 1997, (to be published).                         */
/*                                                                           */
/*  Revision record:                                                         */
/*      12/06/97    Created                                                  */
/*      12/09/97    Added to FrequenC Library                                */
/*      01/01/98    Changes for MS VC++ compatibility & warnings             */
/*                                                                           */
/* (c) Copyright 1997-8  Hamilton Technical Services   All Rights Reserved   */
/*                                                                           */
/*****************************************************************************/

#include "frequenc.h"

float __declspec(dllexport) FAR PASCAL TotvarEDF(int nAlpha, float fRatio)
{
    // Local variables
    float fEDF;

    #if(0) // For testing
    TRACE_INT(nAlpha);
    TRACE_EXP(fRatio);
    #endif

    // Verify parameters
    if(fRatio<2.0)
    {
        return -1.0; // Error
    }

    if(nAlpha<-2 || nAlpha>0)
    {
        return -1.0; // Error
    }
    switch(nAlpha)
    {
        case 0:     // W FM noise
        {
            fEDF=(float)(1.500*fRatio);
        }
        break;

        case -1:    // F FM noise
        {
            fEDF=(float)(1.16832*fRatio - 0.222);
        }
        break;

        case -2:    // RW FM noise
        {
            fEDF=(float)(0.92715*fRatio -0.358);
        }
        break;
    }

    #if(0) // For testing
    TRACE_EXP(fEDF);
    #endif

    return fEDF;

}
```

The FrequenC Library

NAME: TIErms	**FUNCTION:** Calc rms time interval error for phase data

SYNOPSIS:
```
int TIErms(F_TYPE x[], F_TYPE *TIE, int af)
```

F_TYPE x[]	Phase data array
	x[0] = # data points
	x[1] = analysis start point
	x[2] = analysis end point

F_TYPE *TIE	Pointer to TIE rms result
int af	Averaging factor
RETURN: int	# non-gap data points processed, or -1 if error

REMARKS:
Only data points between start and end analysis limits are used.
Embedded zeros are treated as gaps in phase data.
Must have at least 1 analysis point; this requires at least 2 adjacent
non-gap phase data points.

EXAMPLE:
```
#include <frequenc.h>                          /* FrequenC header file */
F_TYPE x[512];                                    /* phase data array */
F_TYPE TIE;                                              /* TIE value */
int af=1;                                               /* avg factor */
int num;                                       /* # analysis points */
.
.
num=TIErms(x, &TIE, af);                            /* call function */
printf("\nTIE rms = %e", TIE);                     /* display TIE rms */
```

SEE ALSO: CalcMTIE()

REFERENCE: S. Bregni, "Clock Stability Characterization and Measurement in Telecommunications", IEEE Transactions on Instrumentation and Measurement, Vol. 46, No. 6, Dec 1997, pp 1284-1294, Eq. 13.

```
/****************************************************************************/
/*                                                                          */
/*                              TIErms()                                    */
/*                                                                          */
/*      Function to calc rms time interval error from phase data            */
/*                                                                          */
/*      Parameters:     F_TYPE fX[]   = phase data array                    */
/*                                      fX[0] = # data points               */
/*                                      fX[1] = analysis start              */
/*                                      fX[2] = analysis end                */
/*                      F_TYPE *fTIE  = pointer to TIE rms                  */
/*                      int nAF       = averaging factor                    */
/*                                                                          */
/*      Returns:        int           = # analysis points,                  */
/*                                      or -1 if error                      */
/*                                                                          */
/*      Note:           Embedded zero value treated as gap in phase data.   */
/*                      Must have at least 1 analysis point; this           */
/*                      requires at least 2 adjacent non-gap phase          */
/*                      data points.                                        */
/*                                                                          */
/*      Reference:      S. Bregni, "Clock Stability Characterization and    */
/*                      Measurement in Telecommunications", IEEE            */
/*                      Transactions on Instrumentation and Measurement,    */
/*                      Vol. 46, No. 6, Dec 1997, pp 1284-1294, Eq. 13.     */
/*                                                                          */
/*      Revision record:                                                    */
/*          06/03/98    Created                                             */
/*                                                                          */
/* (c) Copyright 1998  Hamilton Technical Services  All Rights Reserved     */
/*                                                                          */
/****************************************************************************/

#include "frequenc.h"

int __declspec(dllexport) FAR PASCAL TIErms(F_TYPE fX[], F_TYPE *fTIE,
    int nAF)
{
    F_TYPE fSum=0;                           /* summing variable to calc TIE */
    int i;                                               /* index for summing */
    int nNum=0;                                          /* # analysis points */

    for( i = (int)(fX[1]+2); i <= (int)(fX[2]+2-nAF); i++)
              /* For full analysis limits, index goes from 3 to nNum+2-nAF */
                  /* In summation, smallest index is 1st data point=3, and */
                                              /* largest index is nNum+2 */
    {
        if(    (i==fX[1]+2 && fX[i+nAF])
            || (fX[i] && fX[i+nAF])
            || (i==fX[2]+2-nAF && fX[i]))
                          /* skip if any embedded phase data value is zero */
        {
            fSum += SQR(fX[i+nAF] - fX[i]);
            nNum++;
        }
    }

    if(nNum>0)
    {
        *fTIE=sqrt(fSum/nNum);

        return(nNum);                                    /* # analysis points */
    }
    else
    {
        *fTIE=0;
        return(-1);                                                  /* error */
    }
}
```

Notes