INTERNATIONAL STANDARD

ISO/IEC 10373-6

Second edition
2011-01-15

AMENDMENT 5
2014-03-01

Identification cards — Test methods —

Part 6: **Proximity cards**

AMENDMENT 5: Bit rates of 3*fc*/4, *fc*, 3*fc*/2 and 2*fc* from PCD to PICC

Cartes d'identification — Méthodes d'essai —

Partie 6: Cartes de proximité

AMENDEMENT 5: Débits binaires de 3fc/4, fc, 3fc/2 et 2fc de PCD à PICC

Reference number
ISO/IEC 10373-6:2011/Amd.5:2014(E)

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Amendment 5 to ISO/IEC 10373-6:2011 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 17, *Cards and personal identification*.

Identification cards — Test methods —

Part 6:
Proximity cards

AMENDMENT 5: Bit rates of 3*fc*/4, fc, 3*fc*/2 and 2*fc* from PCD to PICC

Page 22

Add new subclause

7.3 Test methods for bit rates of 3*fc*/4, *fc*, 3*fc*/2 and 2*fc* from PCD to PICC

See Annex J.

After Annex I

Add following new annex:

Annex J
(normative)

Test methods for bit rates of 3*fc*/4, *fc*, 3*fc*/2 and 2*fc* from PCD to PICC

J.1 Overview

This annex specifies the test methods for bit rates of 3*fc*/4, *fc*, 3*fc*/2 and 2*fc* from PCD to PICC.

NOTE Future revisions of ISO/IEC 14443 and ISO/IEC 10373-6 may specify new NPV tolerance and phase noise values with corresponding test methods.

J.2 Test of ISO/IEC 14443-2 parameters

J.2.1 PCD Tests

All the tests described below will be done in the operating volume as defined by the PCD manufacturer.

J.2.1.1 PCD phase range and waveform characteristics

J.2.1.1.1 Purpose

This test is used to determine the PR as well as the normalized differential phase noise and inter-symbol interference parameters, ISI_m and ISI_d, as defined in ISO/IEC 14443-2:2010/Amd.5.

J.2.1.1.2 Test procedure

Apply the procedure defined in 7.1.4.2 with the following adaptations:

— After the activation of a bit rate of 3*fc*/4, *fc*, 3*fc*/2 or 2*fc*, the PCD shall transmit an $I(0)_0$(TEST_COMMAND1(1)).

— In steps a) and f) of 7.1.4.2, the waveform characteristics shall be determined using the analysis tool defined in J.3.

J.2.1.1.3 Test report

The test report shall give the measured PR, ISI_m, ISI_d and the normalized differential phase noise values of the PCD field, within the defined operating volume in unloaded and loaded conditions.

NOTE J.3.12 gives some example test reports.

J.2.2 PICC Tests

J.2.2.1 PICC reception

J.2.2.1.1 Purpose

The purpose of this test is to verify the ability of the PICC to receive PCD commands for bit rates of 3*fc*/4, *fc*, 3*fc*/2 and 2*fc*.

J.2.2.1.2 Test conditions

Four test conditions are defined at the border of the PICC signal parameters as defined in ISO/IEC 14443-2:2010/Amd.5. A low pass filtered pseudo-random white noise as defined in J.4.3 is added to the transmitted APVs such that the normalized differential phase noise (rms) is the maximum value as defined in ISO/IEC 14443-2:2010/Amd.5. The test conditions are created using the test PCD assembly in combination with digital pre-conditioning of the transmitted APVs as shown in J.4:

— Condition 1: the test PCD signal is digitally pre-conditioned to have the maximum $\mathrm{ISI_m}$ value for the $\mathrm{ISI_d}$ value of 45° as defined in ISO/IEC 14443-2:2010/Amd.5;

— Condition 2: the test PCD signal is digitally pre-conditioned to have the maximum $\mathrm{ISI_m}$ value for the $\mathrm{ISI_d}$ value of -45° as defined in ISO/IEC 14443-2:2010/Amd.5;

— Condition 3: the test PCD signal is digitally pre-conditioned to have the maximum $\mathrm{ISI_m}$ value for the $\mathrm{ISI_d}$ value of 120° as defined in ISO/IEC 14443-2:2010/Amd.5;

— Condition 4: the test PCD signal is digitally pre-conditioned to have the maximum $\mathrm{ISI_m}$ value for the $\mathrm{ISI_d}$ value of 0° as defined in ISO/IEC 14443-2:2010/Amd.5.

NOTE 1 These conditions are applied after switching to the bit rate under test.

NOTE 2 J.4 informatively describes how to create the above 4 conditions in the base-band domain (on the complex envelope of the signal).

These 4 test conditions shall be tested at least using H_{min} and H_{max}.

J.2.2.1.3 Test procedure

For each supported bit rate of 3*fc*/4, *fc*, 3*fc*/2 and 2*fc*, the PICC shall operate under the defined conditions after the selection of a bit rate. This PICC shall respond correctly to an $\mathrm{I(0)_0}$(TEST_COMMAND1(1)) transmitted at the specified bit rate.

The activation of the bit rates uses S(PARAMETERS) mechanism as defined in ISO/IEC 14443-4:2008/Amd.3.

NOTE For a frame size higher than 256 bytes a frame with error correction as defined in ISO/IEC 14443-4:2008/Amd.4 should be used.

J.2.2.1.4 Test report

The test report shall confirm the intended operation at the bit rates under test. Used test conditions shall be mentioned in the test report.

J.3 PCD waveform characteristics analysis tool for bit rates of 3*fc*/4, *fc*, 3*fc*/2 and 2*fc*

J.3.1 Overview

The working principle of the analysis tool for bit rates of 3*fc*/4, *fc*, 3*fc*/2 and 2*fc* is illustrated in Figure J.1.

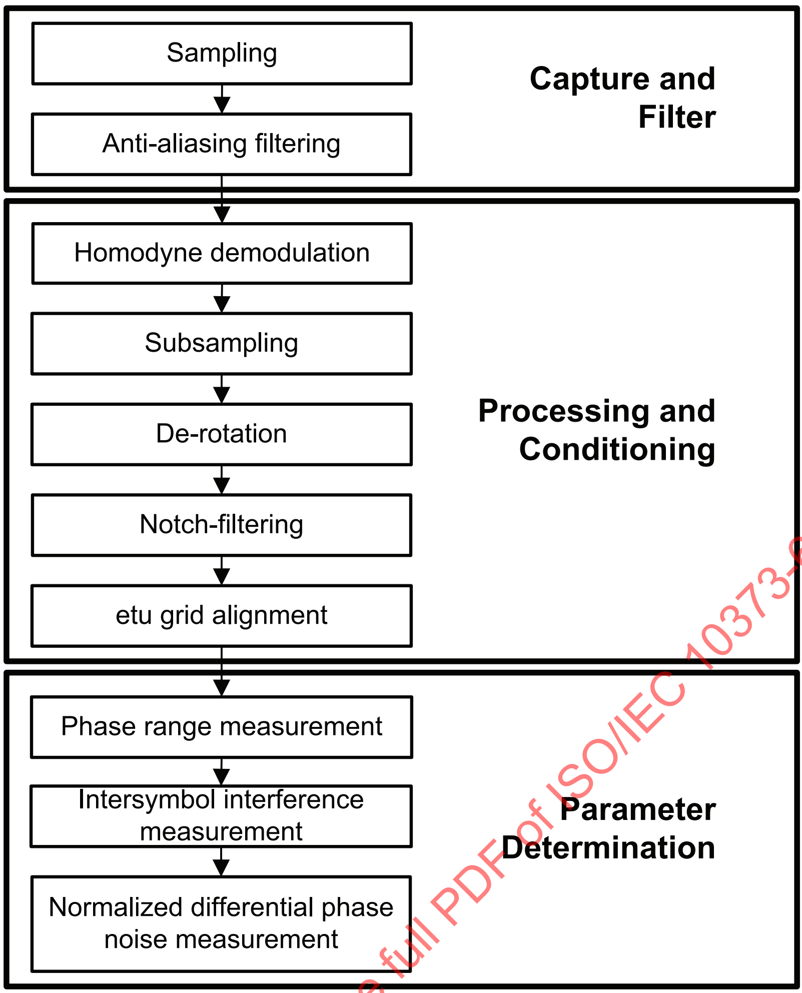

Figure J.1 — Block diagram of the analysis tool for bit rates of 3*fc*/4, *fc*, 3*fc*/2 and 2*fc*

Each block is separately described in the subsequent clauses.

J.3.2 Sampling

The oscilloscope used for signal capturing shall fulfill the requirements defined in 5.1.1. The time and voltage data of at least 1000 non-modulated carrier periods followed by one data frame, followed by at least 10 non-modulated carrier periods (see illustration in Figure J.2) shall be transferred to a suitable computer.

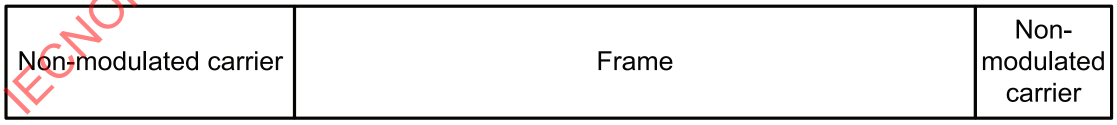

Figure J.2 — Non-modulated carrier followed by one frame, followed by non-modulated carrier

J.3.3 Anti-aliasing filtering

A 4th order, Butterworth type low pass filter with 3-dB cut off frequency at 120 MHz shall be used for filtering higher frequency components The filter characteristic is illustrated in Figure J.3.

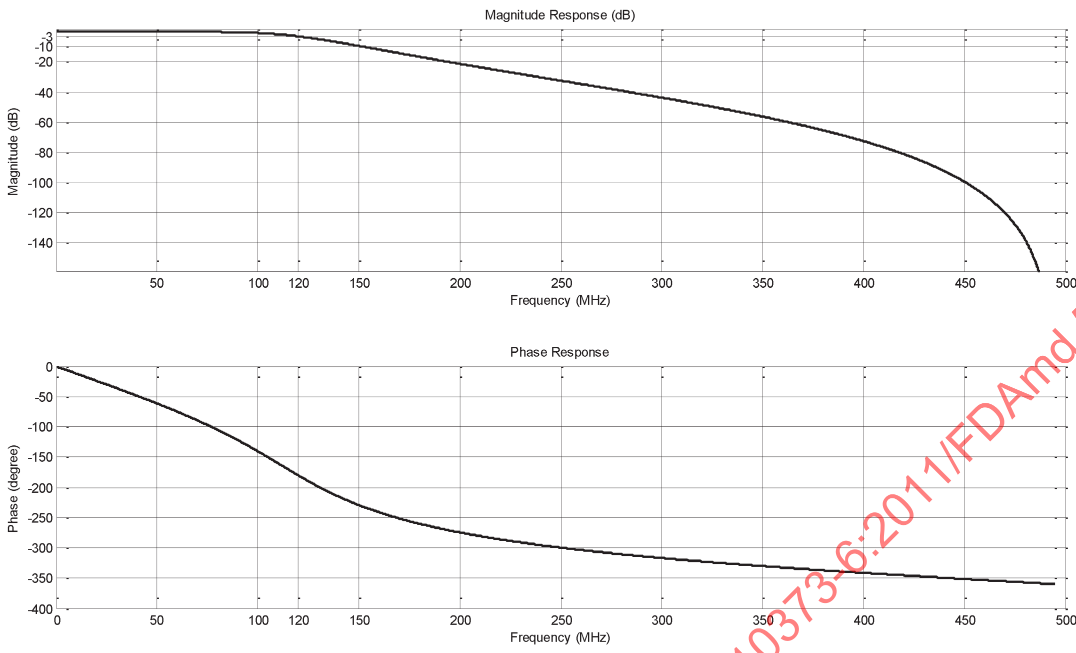

Figure J.3 — Anti-aliasing filter characteristics

J.3.4 Homodyne demodulation

The signal shall be demodulated using a homodyne demodulator (IQ demodulator) and the argument of this complex transform represents the phase signal over time (see Figure J.4).

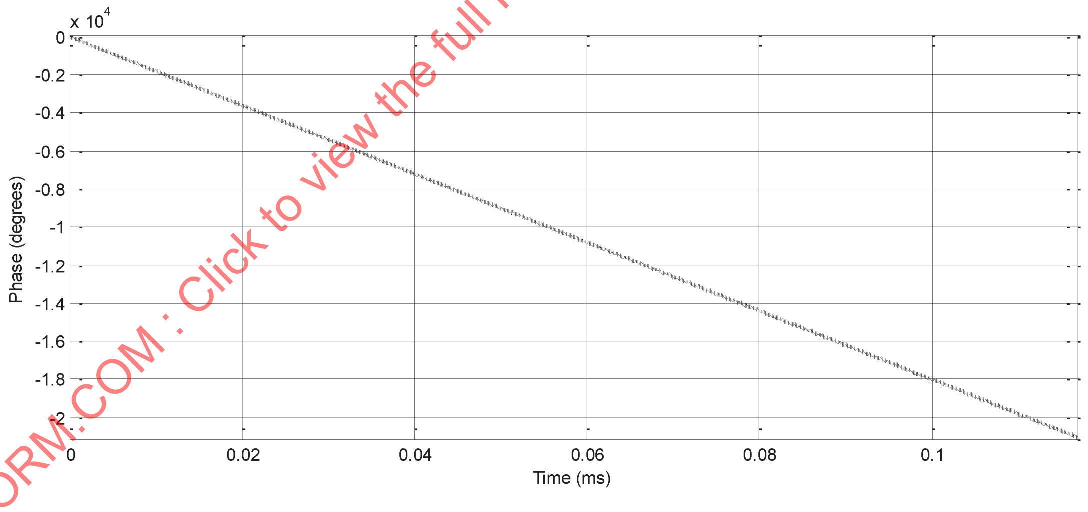

Figure J.4 — Example phase signal over time after homodyne demodulation

J.3.5 Subsampling

The phase signal shall be sub-sampled to an integer number multiple of *fc* using linear interpolation. The integer number shall be at least 32.

J.3.6 De-rotation

This phase signal over time is continuously changing due to the difference between the modulated RF carrier frequency and the demodulator frequency. This frequency mismatch is computed from the constant phase slope of the phase signal during the time of the non-modulated carrier. The complete phase signal is multiplied by a carrier signal whose frequency is the computed frequency difference (see Figure J.5).

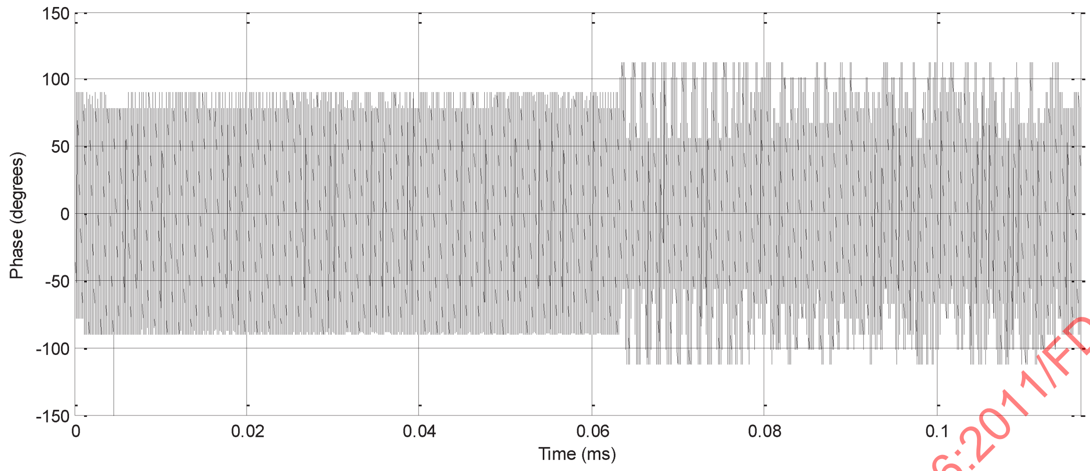

Figure J.5 — Example phase signal after de-rotation

J.3.7 Notch-filtering

The phase signal contains the second harmonic due to demodulation. The phase signal shall be smoothed with a moving average filter having a filter period of 2/*fc* (see Figure J.6).

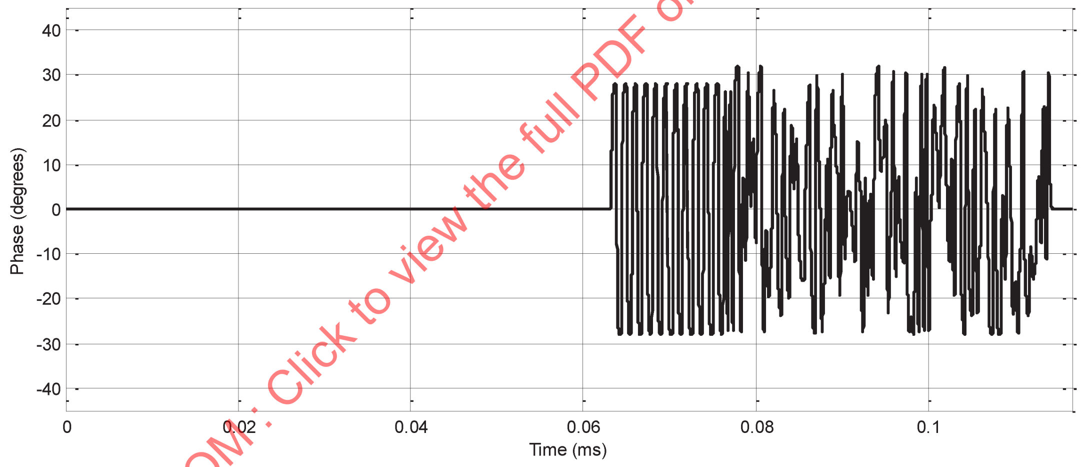

Figure J.6 — Example phase signal after filtering

J.3.8 etu grid alignment

The phase signal shall be aligned to the etu grid of the reference phase signal. The reference phase signal is computed from the SOC using the method defined in ISO/IEC 14443-2:2010/Amd.5. The etu grid alignment is carried out by maximizing the computed correlation, using the phase signal and the reference phase signal.

J.3.9 Phase range measurement

The PR parameter shall be determined as defined in ISO/IEC 14443-2:2010/Amd.5.

J.3.10 Intersymbol interference measurement

The ISI_m and ISI_d parameters shall be determined from the system identification coefficients. The system identification coefficients shall be determined by solving the system identification problem given by the phase signal and the reference phase signal using the Linear Least Squares method. ISI_m and ISI_d values shall be computed for every sampling time within the last carrier period of an etu. The maximum ISI_m value shall be selected with the related ISI_d.

J.3.11 Normalized differential phase noise measurement

The normalized differential phase noise shall be determined during a section of a non-modulated carrier of at least 500 carrier periods according to the definition in ISO/IEC 14443-2:2010/Amd.5.

J.3.12 Program of the PCD waveform characteristics analysis tool for bit rates of 3*fc*/4, *fc*, 3*fc*/2 and 2*fc* (informative)

The following program written in ANSIC language gives an example for the implementation of the analysis tool for bit rates of 3*fc*/4, *fc*, 3*fc*/2 and 2*fc*.

This ANSIC implementation consists of 7 files which should be placed in the same folder.

```
/*************************************************************************/
/*** psk_defines.h                                                    ***/
/***  DESCRIPTION:                                                     ***/
/***    Constants and LUTs for VHBR PSK wave shape tool                ***/
/*************************************************************************/

#ifndef PSK_DEFINES_H
#define PSK_DEFINES_H

#include "psk_types.h"

#define MAX_SAMPLES 50000

#define PSK_ERR_OK                         0 /**< Successful termination */
#define PSK_ERR_READ_FILE                 -1 /**< File not found or no read permission */
#define PSK_ERR_PARAMETER                 -2 /**< Parameter of function is invalid or unex-
pected */
#define PSK_ERR_OUT_OF_MEM                -3 /**< Memory allocation failed */
#define PSK_ERR_INVALID_SAMPLE_RATE       -4 /**< Sample rate of signal is not supported */
#define PSK_ERR_INVALID_SAMPLE_VEC        -5 /**< Sample vector could not be downsampled */
#define PSK_ERR_SIGNAL_TOO_SHORT          -6 /**< Insufficient amount of input data for calcu-
lation */
#define PSK_SYMBOL_GRID_ALIGNMENT_FAIL   -7 /**< Grid alignment not found during analysis */
#define PSK_ERR_SIGNAL_LEN_MISMATCH       -8 /**< Unsupported signal length */

#ifndef M_PI
#define M_PI 3.14159265358979323846264338327950
#endif

#ifndef NULL
#define NULL 0
#endif

// carrier frequency [Hz]
#define FC 13560000
// internal sample frequency of 32 times FC [Hz]
#define FS_INT 433920000
// index of last unmodulated input sample is 480 * FS_INT / FC
#define IDX_UNMOD 15360

#define MIN_NUM_SAMPLES  30000
#define MAX_NUM_SAMPLES 900000

static const psk_uint32 PSK8[] = {1, 1, 7, 7, 1, 1, 7, 7, 1, 1, 7, 7, 1, 1,
                                  7, 7, 1, 1, 7, 7, 1, 1, 7, 7, 1, 1, 7, 7,
                                  1, 1, 7, 7, 1, 1, 7, 7, 1, 1, 7, 7, 1, 1,
                                  7, 7, 1, 7, 1, 7, 0, 0, 7, 3, 6, 1, 5, 3,
                                  6, 2, 2, 2, 7, 1, 0, 3, 5, 2, 3, 5, 2, 3,
                                  6, 0, 7, 2, 3, 3, 7, 6, 4, 5, 6, 1, 6, 5,
                                  2, 6, 1, 3, 4, 0, 2, 0, 6, 6, 7, 0, 5, 7,
                                  3, 7, 3, 0, 3, 6, 6, 1, 1, 0, 6, 4, 0, 6,
                                  3, 5, 6, 1, 1, 1, 2, 6, 7, 0, 7, 0, 7, 3,
                                  1, 2, 4, 2, 1, 5, 7, 4, 0, 3, 3, 2, 3, 4};

static const psk_uint32 PSK16[] = {1, 1, 15, 15, 1, 1, 15, 15, 1, 1,
                                   15, 15, 1, 1, 15, 15, 1, 1, 15, 15,
                                   1, 1, 15, 15, 1, 1, 15, 15, 1, 1,
                                   15, 15, 1, 1, 15, 15, 1, 1, 15, 15,
                                   1, 1, 15, 15, 1, 15, 1, 15, 0, 0,
                                   15, 6, 11, 0, 8, 4, 10, 1, 0, 15,
                                   9, 13, 11, 1, 4, 13, 14, 2, 11, 13,
                                   3, 7, 4, 10, 12, 12, 4, 1, 13, 15,
                                   0, 6, 15, 12, 5, 12, 1, 4, 6, 13,
                                   0, 11, 7, 7, 9, 11, 4, 7, 15, 6,
                                   13, 7, 12, 1, 0, 6, 5, 3, 14, 9,
                                   0, 12, 6, 10, 11, 0, 15, 14, 15, 6,
                                   15, 0, 15, 0, 15, 7, 2, 4, 8, 3,
                                   0, 7, 11, 5, 13, 2, 1, 14, 15, 0};

static const psk_int32  SOC_PSK8_deg[] = {  0,  24,  24, -24, -24,  24,  24,
                                          -24, -24,  24,  24, -24, -24,  24,
                                           24, -24, -24,  24,  24, -24, -24,
                                           24,  24, -24, -24,  24,  24, -24,
                                          -24,  24,  24, -24, -24,  24,  24,
                                          -24, -24,  24,  24, -24, -24,  24,
                                           24, -24, -24,  24, -24,  24, -24,
                                           32,  32, -24,   8, -16,  24,  -8,
                                            8, -16,  16,  16,  16, -24,  24,
                                           32,   8,  -8,  16,   8,  -8,  16,
                                            8, -16,  32, -24,  16,   8,   8,
                                          -24, -16,   0,  -8, -16,  24, -16,
                                           -8,  16, -16,  24,   8,   0,  32,
                                           16,  32, -16, -16, -24,  32,  -8,
                                          -24,   8, -24,   8,  32,   8, -16,
                                          -16,  24,  24,  32, -16,   0,  32,
                                          -16,   8,  -8, -16,  24,  24,  24,
                                           16, -16, -24,  32, -24,  32, -24,
                                            8,  24,  16,   0,  16,  24,  -8,
                                          -24,   0,  32,   8,   8,  16,   8, 0};

static const psk_int32  SOC_PSK16_deg[] = {  0,  28,  28, -28, -28,  28,  28,
                                           -28, -28,  28,  28, -28, -28,  28,
                                            28, -28, -28,  28,  28, -28, -28,
                                            28,  28, -28, -28,  28,  28, -28,
                                           -28,  28,  28, -28, -28,  28,  28,
                                           -28, -28,  28,  28, -28, -28,  28,
                                            28, -28, -28,  28, -28,  28, -28,
                                            32,  32, -28,   8, -12,  32,   0,
                                            16,  -8,  28,  32, -28,  -4, -20,
                                           -12,  28,  16, -20, -24,  24, -12,
                                           -20,  20,   4,  16,  -8, -16, -16,
                                            16,  28, -20, -28,  32,   8, -28,
                                           -16,  12, -16,  28,  16,   8, -20,
                                            32, -12,   4,   4,  -4, -12,  16,
                                             4, -28,   8, -20,   4, -16,  28,
                                            32,   8,  12,  20, -24,  -4,  32,
                                           -16,   8,  -8, -12,  32, -28, -24,
                                           -28,   8, -28,  32, -28,  32, -28,
                                             4,  24,  16,   0,  20,  32,   4,
                                         -12,  12, -20,  24,  28, -24, -28, 32};


static const psk_uint32 PSK_len = 141;

// output data types
enum TYPE
{
  COMPLEX,
  DOUBLE,
  INTEGER,
  BUTTERCFS
};

/*****************************************************************************/
/**
 *  The order of the input psk signal is either 16 or 8
 */
extern psk_uint32 ORDER;

/*****************************************************************************/
/**
 *  The bit rate of the input signal is a user defined parameter and can be
 *  3/4 * fc
 *        fc
 *  3/2 * fc
 *  2   * fc
 *  where fc is the carrier frequency as defined in #FC.
 */
extern psk_double BIT_RATE;

/*****************************************************************************/
/**
 *  The phase range is the difference between the highest an lowest phase value
 *  and can be either 56 deg or 60 deg
 */
extern psk_uint32 PR;

/*****************************************************************************/
/**
 *  The elementary phase interval. 8 deg or 4 deg depending on #ORDER
 */
extern psk_uint32 EPI;

/*****************************************************************************/
/**
 *  The elementary time unit is always a multiple of #FC.
 *  2/FC for bit rates 1.5*FC and 2.0*FC and 4/FC else
 */
extern psk_double ETU;

#endif // PSK_DEFINES_H



/*****************************************************************************/
/***  psk_types.h                                                         ***/
/***  DESCIRPTION:                                                        ***/
/***    Definition of used types in psk analysis tool                    ***/
/***                                                                      ***/
/*****************************************************************************/

#ifndef PSK_TYPES_H
#define PSK_TYPES_H

#define RE(z) ( (z).re )
#define IM(z) ( (z).im )
#define ABS(a) ( (a > 0) ? (a) : (-a) )
#define MAX(a, b) ( ( (a) > (b) ) ? (a) : (b) )

#define BUTTER_SIZE_A 5
#define BUTTER_SIZE_B 5

typedef double psk_double;
typedef int psk_int32;
typedef unsigned int psk_uint32;

typedef struct
```

```
{
  psk_double re;
  psk_double im;
} psk_complex;

typedef struct
{
  psk_double a[BUTTER_SIZE_A];
  psk_double b[BUTTER_SIZE_B];
} psk_butter_coefs;

#endif // PSK_TYPES_H


/*****************************************************************************/
/*** psk_math.h                                                           ***/
/***  DESCIRPTION: header of psk_math.c                                   ***/
/***  It contains the function declaration of used mathematical functions ***/
/***  for the PSK waveform characteristics analysis tool                  ***/
/*****************************************************************************/

#ifndef PSK_MATH_H
#define PSK_MATH_H

#include "psk_types.h"

/*****************************************************************************/
/**
 * psk_mean
 *  calculate the arithmetic mean of a given vector
 *  @param vec Calculate the mean of the values in this vector
 *  @param len The vector's number of elements
 *  @return The aritmethic mean or zero, if len < 2
 */
psk_double psk_mean( psk_double* vec /*[in]*/, psk_uint32 len /*[in]*/);


/*****************************************************************************/
/**
 * psk_cmpl_mean
 *  calculate the arithmetic mean of a given vector for both, real and
 *  imaginary parts. The result is also a complex number
 *  @param vec Calculate the mean of the values in this vector
 *  @param len The vector's number of elements
 *  @return The aritmethic mean or zero, if len < 2
 */
psk_complex psk_cmpl_mean( psk_complex* vec/*[in]*/, psk_uint32 len /*[in]*/);


/*****************************************************************************/
/**
 * psk_diff
 *  The resulting vector's elements are the differences of two consecutive
 *  elements of a given vector. The resulting vector has a length of len-1
 *  @param vec Calculate the consecutive differences of this vector's values
 *  @param len The vector's number of elements
 *  @return The aritmethic mean or zero, if len < 2
 */
psk_double* psk_diff( psk_double* vec /*[in]*/, psk_uint32 len /*[in]*/);


/*****************************************************************************/
/**
 * psk_max
 *  Find the maximal value in a vector and it's index
 *  @param vec An array of values
 *  @param vec_len The vector's number of elements
 *  @param max_val Pointer where to store the maximum's value
 *  @param max_idx Pointer where to store the maximum's index
 */
void psk_max( psk_double* vec,       /*[in]*/
              psk_uint32 vec_len,    /*[in]*/
              psk_double* max_val,   /*[out]*/
              psk_uint32* max_idx ); /*[out]*/


/******************************************************************************/
/**
 * psk_min
 *  Find the minimal value in a vector and it's index
 *  @param vec An array of values
 *  @param vec_len The vector's number of elements
 *  @param min_val Pointer where to store the minimum's value
 *  @param min_idx Pointer where to store the minimum's index
 */
void psk_min( psk_double* vec,       /*[in]*/
              psk_uint32 vec_len,    /*[in]*/
              psk_double* min_val,   /*[out]*/
              psk_uint32* min_idx ); /*[out]*/


/******************************************************************************/
/**
 * psk_add
 *  Calculate the sum of two complex numbers
 *  @param a First summand
 *  @param b Second summand
 *  @return The complex result
 */
psk_complex psk_add( psk_complex a /*[in]*/, psk_complex b /*[in]*/ );


/******************************************************************************/
/**
 * psk_sub
 *  Calculate the difference of two complex numbers
 *  @param a Minuend
 *  @param b Subtrahend
 *  @return The complex result
 */
psk_complex psk_sub( psk_complex a /*[in]*/, psk_complex b /*[in]*/ );

/******************************************************************************/
/**
 * psk_cmpl_mult
 *  Calculate the product of two complex numbers
 *  @param a First factor
 *  @param b Second factor
 *  @return The complex result
 */
psk_complex psk_cmpl_mult( psk_complex a /*[in]*/, psk_complex b /*[in]*/ );

/******************************************************************************/
/**
 * psk_cmpl_div
 *  Calculate the quotient of two complex numbers
 *  @param a Dividend
 *  @param b Divisor
 *  @return The complex result
 */
psk_complex psk_cmpl_div( psk_complex a /*[in]*/, psk_complex b /*[in]*/ );

/******************************************************************************/
/**
 * psk_cmpl_conj
 *  Get the complex conjugate of a number
 *  @param a The complex number
 *  @return The complex conjugate of a
 */
psk_complex psk_cmpl_conj( psk_complex a /*[in]*/ );
```

```
/****************************************************************************/
/**
 * psk_abs
 *  Get the absolute value of a complex number: sqrt(RE^2+IM^2)
 *  @param num The complex number
 *  @return The absolute value
 */
psk_double psk_abs( psk_complex num /*[in]*/ );


/****************************************************************************/
/**
 * psk_cmpl_vec_mult
 *  Calculate the product of vector elements with the same index and return
 *  the result in a vector. This function makes use of #psk_cmpl_mult
 *  The two vectors must have the same length
 *  @param a Factors of first vector
 *  @param b Factros of second vector
 *  @param len The number of elments. Must be the same for a and b
 *  @return The complex product of a and b
 */
psk_complex* psk_cmpl_vec_mult( psk_complex* a,    /*[in]*/
                                psk_complex* b,    /*[in]*/
                                psk_uint32 len );  /*[in]*/


/****************************************************************************/
/**
 * psk_vec_abs
 *  Apply #psk_abs on every complex element of a vector
 *  @param a The complex input values
 *  @param len The number of elements in vector a
 *  @return The absolute values
 */
psk_double* psk_vec_abs( psk_complex* a /*[in]*/, psk_uint32 len /*[in]*/ );


/****************************************************************************/
/**
 * psk_cmpl_vec_conj
 *  Apply #psk_cmpl_conj on every complex element of a vector
 *  @param vec The complex input values
 *  @param len The number of elements in vector vec
 *  @return The complex conjugate of vec
 */
psk_complex* psk_cmpl_vec_conj( psk_complex* vec, /*[in]*/
                                psk_uint32 len);  /*[in]*/


/****************************************************************************/
/**
 * psk_vec_angle
 *  Apply the built in function atan2 on every complex element of a vector
 *  @param vec The complex input values
 *  @param len The number of elements in vector vec
 *  @return The angle of every input element
 */
psk_double* psk_vec_angle( psk_complex* vec /*[in]*/, psk_uint32 len /*[in]*/ );


/****************************************************************************/
/**
 * psk_variance
 *  Calculate the variance of the elements in a vector
 *  @param vec The real number input values
 *  @param len The number of elements in vector vec
 *  @return The variance of the vector's elements
 */
psk_double psk_variance( psk_double* vec /*[in]*/, psk_uint32 len /*[in]*/ );
```

```
/*****************************************************************************/
/**
 * psk_std
 *  Calculate the standard deviation of the elements in a vector
 *  This function uses #psk_variance and returns it’s square root
 *  @param vec The real number input values
 *  @param len The number of elements in vector vec
 *  @return The standard deviation of the vector’s elements
 */
psk_double psk_std( psk_double* vec /*[in]*/, psk_uint32 len /*[in]*/ );


/*****************************************************************************/
/**
 * psk_normalize
 *  Normalize a vector’s elements. This function uses #psk_vec_abs and #psk_max
 *  @param vec The real number input values.
 *  @param len The number of elements in vector vec
 *  @return The normalized vector
 */
psk_double* psk_normalize( psk_double* vec,  /*[in]*/
                           psk_uint32 len ); /*[in]*/


/*****************************************************************************/
/**
 * psk_cmpl_normalize
 *  Normalize a vector’s complex elements both real and imaginary part separatly
 *  @param vec The complex number input values.
 *  @param len The number of elements in vector vec
 *  @return The normalized vector
 */
psk_complex* psk_cmpl_normalize( psk_complex* vec, /*[in]*/
                                 psk_uint32 len ); /*[in]*/


/*****************************************************************************/
/**
 * psk_linspace
 *  Build a vector with a given start and stop value and defined step size
 *  @param start The value of the first element
 *  @param step The step size. Difference between two consecutive elements
 *  @param stop The value of the last element
 *  @param len Pointer where to save the length of the vector
 *  @return The resulting vector or NULL, if start >= stop
 */
psk_double* psk_linspace( psk_double start,  /*[in]*/
                          psk_double step,   /*[in]*/
                          psk_double stop,   /*[in]*/
                          psk_uint32* len ); /*[out]*/


/*****************************************************************************/
/**
 * psk_idx_linspace
 *  Build a vector with a given start and stop value and defined step size.
 *  This function uses integer values, so the resulting vector can be used
 *  as a indexing vector for signals.
 *  @param start The value of the first element
 *  @param step The step size. Difference between two consecutive elements
 *  @param stop The value of the last element
 *  @param len Pointer where to save the length of the vector
 *  @return The resulting vector or NULL, if start >= stop
 */
psk_uint32* psk_idx_linspace( psk_uint32 start,  /*[in]*/
                              psk_uint32 step,   /*[in]*/
                              psk_uint32 stop,   /*[in]*/
                              psk_uint32* len ); /*[out]*/


#endif //PSK_MATH_H
```

```
/*****************************************************************************/
/*** psk_math.c                                                            ***/
/***  DESCIRPTION: Implementation of psk_math.h                            ***/
/***  Contains the function implementation of used mathematical functions  ***/
/***  for the VHBR PSK wave shape analysis tool                            ***/
/*****************************************************************************/

#include <stdlib.h>
#include <math.h>

#include "psk_defines.h"
#include "psk_types.h"
#include "psk_math.h"

//-----------------------------------------------------------------------------
psk_double psk_mean( psk_double* vec, psk_uint32 len )
{
  psk_double sum = 0.0;
  psk_uint32 idx;

  if( len < 2 )
    return sum;

  for( idx = 0; idx < len; idx++ )
    sum += vec[idx];

  sum /= (double)len;

  return( sum );
}

//-----------------------------------------------------------------------------
psk_complex psk_cmpl_mean( psk_complex* vec, psk_uint32 len )
{
  psk_complex mean;
  RE( mean ) = 0.0;
  IM( mean ) = 0.0;
  psk_uint32 idx;

  if( len < 2 )
    return mean;

  for( idx = 0; idx < len; idx++ )
  {
    RE( mean ) += RE( vec[idx] );
    IM( mean ) += IM( vec[idx] );
  }

  RE( mean ) /= (double)len;
  IM( mean ) /= (double)len;

  return( mean );
}

//-----------------------------------------------------------------------------
psk_double* psk_diff( psk_double* vec , psk_uint32 len )
{
  psk_double* diff = calloc( ( len - 1 ), sizeof(psk_double) );
  psk_uint32 idx;

  for( idx = 0; idx < ( len - 1 ); idx++ )
    diff[idx] = vec[idx + 1] - vec[idx];

  return( diff );
}

//-----------------------------------------------------------------------------
void psk_max( psk_double* vec,
              psk_uint32 vec_len,
              psk_double* max_val,
              psk_uint32* max_idx )
{
  if( vec == NULL )
    return;
  psk_uint32 idx;
  *max_val = vec[0];
  *max_idx = 0;

  for( idx = 1; idx < vec_len; idx++ )
  {
    if( vec[idx] > *max_val )
    {
      *max_val = vec[idx];
      *max_idx = idx;
    }
  }
}

//----------------------------------------------------------------------------
void psk_min( psk_double* vec,
              psk_uint32 vec_len,
              psk_double* min_val,
              psk_uint32* min_idx )
{
  psk_uint32 idx;
  *min_val = vec[0];
  *min_idx = 0;

  for( idx = 1; idx < vec_len; idx++ )
  {
    if( vec[idx] < *min_val )
    {
      *min_val = vec[idx];
      *min_idx = idx;
    }
  }
}

//----------------------------------------------------------------------------
psk_complex psk_add( psk_complex a, psk_complex b )
{
  psk_complex sum;
  RE( sum ) = RE( a ) + RE( b );
  IM( sum ) = IM( a ) + IM( b );

  return( sum );
}

//----------------------------------------------------------------------------
psk_complex psk_sub( psk_complex a, psk_complex b )
{
  psk_complex diff;
  RE( diff ) = RE( a ) - RE( b );
  IM( diff ) = IM( a ) - IM( b );

  return( diff );
}

//----------------------------------------------------------------------------
psk_complex psk_cmpl_mult( psk_complex a, psk_complex b )
{
  psk_complex prod;
  RE( prod ) = RE( a ) * RE( b ) - IM( a ) * IM( b );
  IM( prod ) = IM( a ) * RE( b ) + RE( a ) * IM( b );

  return( prod );
}

//----------------------------------------------------------------------------
psk_complex psk_cmpl_div( psk_complex a, psk_complex b )
```

```
{
  psk_complex quot;
  RE( quot ) = RE( a ) * RE( b ) + IM( a ) * IM( b );
  RE( quot ) = RE( quot ) / ( RE( b ) * RE( b ) + IM( b ) * IM( b ) );

  IM( quot ) = IM( a ) * RE( b ) - RE( a ) * IM( b );
  IM( quot ) = IM( quot ) / ( RE( b ) * RE( b ) + IM( b ) * IM( b ) );
  return( quot );
}

//----------------------------------------------------------------------------
psk_complex psk_cmpl_conj( psk_complex a )
{
  psk_complex conj;
  RE( conj ) = RE( a );
  IM( conj ) = -IM( a );

  return( conj );
}

//----------------------------------------------------------------------------
psk_double psk_abs( psk_complex num )
{
  return( sqrt( pow( RE( num ), 2 ) + pow( IM( num ), 2 ) ) );
}

//----------------------------------------------------------------------------
psk_complex* psk_cmpl_vec_mult( psk_complex* a,
                                psk_complex* b,
                                psk_uint32 len )
{
  psk_complex *prod_vec = calloc( len, sizeof(psk_complex) );
  psk_uint32 idx;

  for( idx = 0; idx < len; idx++ )
    prod_vec[idx] = psk_cmpl_mult( a[idx], b[idx] );

  return( prod_vec );
}

//----------------------------------------------------------------------------
psk_double* psk_vec_abs( psk_complex* a, psk_uint32 len )
{
  psk_double *abs_vec = calloc( len, sizeof(psk_complex) );
  psk_uint32 idx;

  for( idx = 0; idx < len; idx++ )
    abs_vec[idx] = psk_abs( a[idx] );

  return( abs_vec );
}

//----------------------------------------------------------------------------
psk_complex* psk_cmpl_vec_conj( psk_complex* vec, psk_uint32 len )
{
  psk_complex *conj_vec = calloc( len, sizeof(psk_complex) );
  psk_uint32 idx;

  for( idx = 0; idx < len; idx++ )
    conj_vec[idx] = psk_cmpl_conj( vec[idx] );

  return( conj_vec );
}

//----------------------------------------------------------------------------
psk_double* psk_vec_angle( psk_complex* vec, psk_uint32 len)
{
  psk_double *angle_vec = calloc( len, sizeof(psk_complex) );
  psk_uint32 idx;

  for( idx = 0; idx < len; idx++ )
```

```
    angle_vec[idx] = atan2( IM( vec[idx] ), RE( vec[idx] ) );

  return( angle_vec );
}

//------------------------------------------------------------------------------
psk_double psk_variance( psk_double* vec, psk_uint32 len )
{
  psk_double variance = 0.0;
  psk_double mean = psk_mean( vec, len );
  psk_uint32 idx;

  for( idx = 0; idx < len; idx++ )
    variance += pow( vec[idx] - mean, 2.0 );
  variance /= (double)len;

  return( variance );
}

//------------------------------------------------------------------------------
psk_double psk_std( psk_double* vec, psk_uint32 len )
{
  psk_double std_deviation = 0.0;
  std_deviation = psk_variance( vec, len );
  return( sqrt( std_deviation ) );
}

//------------------------------------------------------------------------------
psk_double* psk_normalize( psk_double* vec, psk_uint32 len )
{
  psk_double max_val = ABS( vec[0] );
  psk_double* norm_vec = calloc( len, sizeof(psk_double) );
  psk_uint32 idx;

  for( idx = 0; idx < len; idx++ )
  {
    if ( ABS( vec[idx] ) > max_val )
      max_val = ABS( vec[idx] );
  }

  for( idx = 0; idx < len; idx++ )
    norm_vec[idx] = vec[idx] / max_val;

  return( norm_vec );
}

//------------------------------------------------------------------------------
psk_complex* psk_cmpl_normalize( psk_complex* vec, psk_uint32 len )
{
  psk_double max_abs = psk_abs( vec[0] );
  psk_complex* norm_vec = calloc( len, sizeof(psk_complex) );
  psk_uint32 idx;
  psk_double tmp = 0.0;

  // find absolute maxima
  for( idx = 1; idx < len; idx++ )
  {
    tmp = psk_abs( vec[idx] );
    if( tmp > max_abs )
      max_abs = tmp;
  }

  // normalize
  for( idx = 0; idx < len; idx++ )
  {
    RE( norm_vec[idx] ) = RE( vec[idx] ) / max_abs;
    IM( norm_vec[idx] ) = IM( vec[idx] ) / max_abs;
  }

  return( norm_vec );
}
```

```
//------------------------------------------------------------------------------
psk_double* psk_linspace( psk_double start,
                          psk_double step,
                          psk_double stop,
                          psk_uint32* len )
{
  *len = ABS( ( (stop - start ) / step ) ) + 1.0;
  if( *len < 2 )
    return( NULL );
  psk_double* out = calloc( *len, sizeof(psk_double) );
  psk_uint32 idx = 0;

  out[idx] = start;
  for( idx = 1; idx < *len; idx++ )
    out[idx] = out[idx - 1] + step;

  return( out );
}

//------------------------------------------------------------------------------
psk_uint32* psk_idx_linspace( psk_uint32 start,
                              psk_uint32 step,
                              psk_uint32 stop,
                              psk_uint32* len )
{
  if( start >= stop || step == 0 )
    return( NULL );
  *len = ( ( ( (psk_double)stop - (psk_double)start ) /
           (psk_double)step ) + 1.0 );
  psk_uint32 *out = calloc( *len, sizeof(psk_uint32) );
  psk_uint32 idx = 0;

  out[idx] = start;
  for( idx = 1; idx < *len; idx++ )
    out[idx] = out[idx - 1] + step;

  return( out );
}


/*****************************************************************************/
/*** psk_dsp.h                                                            ***/
/***  DESCIRPTION: header of psk_dsp.c                                    ***/
/***    Collection of functions used for dsp in the PSK waveform          ***/
/***  characteristics analysis tool.                                      ***/
/*****************************************************************************/

#ifndef PSK_DSP_H
#define PSK_DSP_H

#include "psk_types.h"
#include "psk_math.h"

/*****************************************************************************/
/**
 * psk_compute_butter_lp_coef
 *  Computes the low-pass filter coefficients of a 4th order Butterworth
 *  IIR filter with a cut-off frequency of 120MHz
 *  @param sample_rate Sampling rate
 *  @return Struct with the b and a polynom coefficients
 */
psk_butter_coefs psk_compute_butter_lp_coef( psk_double sample_rate /*[in]*/);

/*****************************************************************************/
/**
 * psk_antialiasing filter
 *  Filters the signal with a 4th order IIR filter. This function makes use of
 *  #psk_compute_butter_lp_coef to get the filter coefficients.
 *  @param v_in Input signal amplitude
 *  @param len_in Length of v_in
 *  @param v_out_ptr This pointer will be set to the array containing the
 *         output signal.
 *  @param len_out Length of the output signal
 *  @param sample_rate Sampling rate of the input signal
 *  @return error value as defined in psk_defines.h
 */
psk_int32 psk_antialiasing( psk_double* v_in,          /*[in]*/
                            psk_uint32 len_in,         /*[in]*/
                            psk_double** v_out_ptr,    /*[out]*/
                            psk_uint32* len_out,       /*[out]*/
                            psk_double sample_rate ); /*[in]*/

/******************************************************************************/
/**
 * psk_downsample
 *  resamples input signal to internal sampling rate (FS_INT)
 *  defined in "psk_defines.h" using linear interpolation
 *  @param samples_in Array of amplitude values
 *  @param time_in Array of time values
 *  @param len_in Length of samples_in and time_in
 *  @param samples_out_ptr Pointer to output array of amplitudes
 *  @param time_out_ptr Pointer to output array of time values
 *  @param len_out Length of output arrays
 *  @return error value as defined in psk_defines.h
 */
psk_int32 psk_downsample( psk_double* samples_in        /*[in]*/,
                          psk_double* time_in           /*[in]*/,
                          psk_uint32 len_in             /*[in]*/,
                          psk_double** samples_out_ptr  /*[out]*/,
                          psk_double** time_out_ptr     /*[out]*/,
                          psk_uint32* len_out           /*[out]*/ );

/******************************************************************************/
/**
 * psk_ma_filter
 *  Moving average implementation
 *  @param samples_in Array of amplitude values
 *  @param len_in Length of samples_in
 *  @param samples_out_ptr Pointer to output array of amplitudes
 *  @param len_out Length of output array
 *  @param len_filter Window size of the moving average filter
 *  @return error value as defined in psk_defines.h
 */
psk_int32 psk_ma_filter( psk_double* samples_in        /*[in]*/,
                         psk_uint32 len_in              /*[in]*/,
                         psk_double** samples_out_ptr   /*[out]*/,
                         psk_uint32* len_out            /*[out]*/,
                         psk_uint32 len_filter         /*[in]*/ );

/******************************************************************************/
/**
 * psk_ma_filter_cmpl
 *  Moving average on a complex input signal: real and imaginary part are
 *  each MA filtered
 *  @param samples_in Array of amplitude values
 *  @param len_in Length of samples_in
 *  @param samples_out_ptr Pointer to output array of amplitudes
 *  @param len_out Length of output array
 *  @param len_filter Window size of the moving average filter
 *  @return error value as defined in psk_defines.h
 */
psk_int32 psk_ma_filter_cmpl( psk_complex* samples_in        /*[in]*/,
                              psk_uint32 len_in               /*[in]*/,
                              psk_complex** samples_out_ptr   /*[out]*/,
                              psk_uint32* len_out             /*[out]*/,
                              psk_uint32 len_filter          /*[in]*/);


/******************************************************************************/
/**
 * psk_FIR_filter
 *  finite impulse response filter
 *  @param b Polynomial ff filter coefficients "Direct Form II Transposed"
 *  @param num_coef Number of coefficients
 *  @param sample_in Array of amplitude values
 *  @param len_in Length of sample_in
 *  @return filter signals of the same length as input signal
 */
psk_double* psk_FIR_filter( psk_double* b          /*[in]*/,
                            psk_uint32 num_coef    /*[in]*/,
                            psk_double* sample_in  /*[in]*/,
                            psk_uint32 len_in      /*[in]*/);

/******************************************************************************/
/**
 * psk_FIR_filter_cmpl
 *  complex finite impulse response filter "Direct Form II Transposed"
 *  @param b Complex valued polynomial ff filter coefficients
 *  @param num_coef Number of coefficients
 *  @param sample_in Complex samples_in (arra of amplitude values)
 *  @param len_in Length of samples_
 *  @return complex filter signals of the same length as input signal
 */
psk_complex* psk_FIR_filter_cmpl( psk_complex* b          /*[in]*/,
                                  psk_uint32 num_coef     /*[in]*/,
                                  psk_complex* sample_in  /*[in]*/,
                                  psk_uint32 len_in       /*[in]*/);


/******************************************************************************/
/**
 * psk_demodulation
 *  Homodyne demodulator (IQ demodulator)
 *  @param samples_in Array of amplitude values
 *  @param time_in Array of time values
 *  @param len Length of inputs
 *  @param v_out_bb complex output signal in baseband of size len
 *  @return nothing
 */
void psk_demodulation( psk_double*   samples_in /*[in]*/,
                       psk_double*   time_in    /*[in]*/,
                       psk_uint32    len        /*[in]*/,
                       psk_complex** v_out_bb   /*[out]*/);

/******************************************************************************/
/**
 * psk_exp_1i_wt
 *  complex multiplication with exp(j *w*t) term
 *  @param v_in_bb Array of amplitude values
 *  @param time_in Array of time values
 *  @param len Length of inputs
 *  @param freq Frequency parameter for computing 'w'
 *  @param v_out_bb complex output signal in baseband
 *         (memory not allocated in function)
 *  @return nothing
 */
void psk_exp_1i_wt( psk_complex* v_in_bb    /*[in]*/,
                    psk_double*  time_in    /*[in]*/,
                    psk_uint32   len        /*[in]*/,
                    psk_double   freq       /*[in]*/,
                    psk_complex** v_out_bb  /*[out]*/);

/******************************************************************************/
/**
 * psk_derotation
 *  derotate input baseband signal by frequency f
 *  @param v_in_bb Complex valued array of amplitude values
 *  @param time_in Array of time values of v_in_bb
 *  @param len Length of input arrays
 *  @param v_out_bb complex output signal in baseband
 *         (memory not allocated in function)
 *  @return error value as defined in psk_defines.h
 */
psk_int32 psk_derotation( psk_complex* v_in_bb    /*[in]*/,
                          psk_double* time_in     /*[in]*/,
                          psk_uint32  len         /*[in]*/,
                          psk_complex** v_out_bb  /*[out]*/);

/******************************************************************************/
/**
 * psk_estimate_carrier_freq
 *  estimate frequency difference of input signal to demodulation signal
 *  @param v_in_bb (complex valued array)
 *  @param len_unmod Length of unmodulated signal
 *  @return frequency error [Hz]
 */
psk_double psk_estimate_carrier_freq( psk_complex* v_in_bb  /*[in]*/,
                                      psk_uint32 len_unmod  /*[in]*/);

/******************************************************************************/
/**
 * psk_cross_coveriance
 *  resample signal by rate 'osf' using nearest neighbor interpolation
 *  @param v_in1 (complex valued array)
 *  @param len_in1 Length of v_in1
 *  @param v_in2 (complex valued array)
 *  @param len_in2 Length v_in2
 *  @param lags The signals are shifted 'lags' times
 *  @param v_out Complex valued array containing the xcov function of v_in1 and
 *         v_in2
 *  @param lagIdx_vec Array of lag indices
 *  @return error value as defined in psk_defines.h
 */
psk_int32 psk_cross_covariance( psk_complex* v_in1       /*[in]*/,
                                psk_uint32   len_in1     /*[in]*/,
                                psk_complex* v_in2       /*[in]*/,
                                psk_uint32   len_in2     /*[in]*/,
                                psk_uint32   lags        /*[in]*/,
                                psk_complex** v_out      /*[out]*/,
                                psk_int32**   lagIdx_vec /*[out]*/);

/******************************************************************************/
/**
 * psk_ff_lms
 *  System identification using LMS algorithm
 *  @param v_in1 (complex valued array ==> reference data (send data))
 *  @param v_in2 (complex valued array ==> filtered data (received data)
 *  @param len Length v_in1, v_in2
 *  @param n_taps (filter length used for SI)
 *  @return complex filter coefficients
 */
 psk_complex* psk_ff_lms( psk_complex* v_in1  /*[in]*/,
                          psk_complex* v_in2  /*[in]*/,
                          psk_uint32 len      /*[in]*/,
                          psk_uint32 n_taps   /*[in]*/);

/******************************************************************************/
/**
 * psk_cross_coveriance
 *  resample signal by rate 'osf' using nearest neighbor interpolation
 *  @param v_in (input phase array in radians)
 *  @param len_in Length of v_in
 *  @return v_out Unwrapped output phase array in radians
 */
psk_double* psk_unwrap(psk_double* v_in /*[in]*/, psk_uint32 len_in /*[in]*/);


/******************************************************************************/
/**
 * psk_get_phase_noise
 *  calculate the differential phase noise of a given complex signal
 *  @param sig Complex signal of an unmodulated carrier sequence
 *  @param sig_len The number of samples in sig
 *  @param phase_noise Pointer where to store the result
 *  @return error value as defined in psk_defines.h
 */
psk_int32 psk_get_phase_noise( psk_complex* sig,          /*[in]*/
                               psk_uint32 sig_len,        /*[in]*/
                               psk_double* phase_noise ); /*[out]*/


/******************************************************************************/
/**
 * psk_isi_param
 *  calculate the differential phase noise of a given complex signal
 *  @param sig_bb_int Grid aligned complex input BB signal
 *  @param sig_bb_len The number of samples in sig_bb_int
 *  @param isi_m Pointer where the ISI magnitude [EPI] will be stored
 *  @param isi_d Pointer where the ISI rotation [degree] will be stored
 *  @param phase_range Pointer where the measured phase range will be stored
 *  @param sig_out Pointer where the grid aligned signal at symbol rate will
 *                be stored
 *  @return error value as defined in psk_defines.h
 */
psk_int32 psk_isi_param( psk_complex* sig_bb_int,  /*[in]*/
                         psk_uint32 sig_bb_len,    /*[in]*/
                         psk_double* isi_m,        /*[out]*/
                         psk_double* isi_d,        /*[out]*/
                         psk_double* phase_range,  /*[out]*/
                         psk_complex** sig_out );  /*[out]*/


 /******************************************************************************/
/**
 * psk_re_align_symbol_grid
 *  align the PSK modulated input signal to the reference SOC signal
 *  @param v_in Complex base band signal
 *  @param len The number of samples in v_in
 *  @param strt_idx Pointer to field storing the index of the starting point
 *  @param end_idx Pointer to field storing the index of the end point
 *  @return error value as defined in psk_defines.h
 */
 psk_int32 psk_re_align_symbol_grid( psk_complex* v_in,       /*[in]*/
                                     psk_uint32 len,          /*[in]*/
                                     psk_uint32* strt_idx,    /*[out]*/
                                     psk_uint32* end_idx );   /*[out]*/

 /******************************************************************************/
/**
 * psk_find_approx_delay
 *  align the PSK modulated input signal to the reference SOC signal
 *  @param v_ref_in Complex reference base band signal
 *  @param ref_len The number of samples in v_ref_in
 *  @param v_in Complex base band signal
 *  @param len The number of samples in v_in
 *  @param max_shift Maximum shift used for cross-covaraince computation
  * @param delay
 *  @return error code as defined in psk_defines.h
 */
 psk_int32 psk_find_approx_delay( psk_complex* v_ref_in,       /*[in]*/
                                  psk_uint32 ref_len,           /*[in]*/
                                  psk_complex* v_in,            /*[in]*/
                                  psk_uint32 len,               /*[in]*/
                                  psk_uint32 max_shift,         /*[in]*/
                                  psk_uint32 *delay );          /*[out]*/


/******************************************************************************/
#endif // PSK_DSP_H


/******************************************************************************/
/*** psk_dsp.c                                                              ***/
/***  DESCIRPTION: Implementation of psk_dsp.h                              ***/
```

```
/***  Contains the function implementation of used DSP functions           ***/
/***  for the VHBR PSK wave shape analysis tool                             ***/
/*****************************************************************************/
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#include "psk_defines.h"
#include "psk_types.h"
#include "psk_math.h"
#include "psk_dsp.h"

//----------------------------------------------------------------------------
psk_butter_coefs psk_compute_butter_lp_coef( psk_double sample_rate )
{
  psk_uint32 i = 0;
  psk_uint32 butter_order = ( BUTTER_SIZE_A - 1 );
  psk_double wd = 120e6 * 2*M_PI; // cutoff frequency in radians ==> 120 Mhz
  psk_double Ts = 1.0 / (psk_double)sample_rate;
  psk_double wa = 0.0;
  psk_double wa_p2 = 0.0;
  psk_double g = 0.0;
  psk_double a_biquad[3 * butter_order / 2];
  psk_double b_biquad[3 * butter_order / 2 ];
  psk_double norm_fact = 0.0;

  psk_butter_coefs coefs;

  memset( a_biquad, 0, 3 * butter_order / 2 );
  memset( b_biquad, 0, 3 * butter_order / 2 );

  if( butter_order == 0 ) // filter order must be greater then Zero and even
    return( coefs );

  if( butter_order % 2 ) // filter order must be even
    return(coefs);

  // Step 1: pre-warping of cut-off frequency
  wa = tan( wd * Ts / 2.0 );
  //wa1 = 2.0 / Ts * wa;

  wa_p2 = pow( wa, 2 );
  // Step 2: compute biquad filter coefficients

  for( i = 0 ; i < ( butter_order / 2 ); i++ )
  {
    norm_fact = 1.0 + 2.0 * cos( M_PI * ( 2.0 * (psk_double)i + 1.0 ) /
                ( 2.0 * (psk_double)butter_order ) ) * wa + wa_p2 ;
    g = wa_p2 / norm_fact;

    b_biquad[i * 3    ] = 1.0 * g;
    b_biquad[i * 3 + 1] = 2.0 * g;
    b_biquad[i * 3 + 2] = 1.0 * g;

    a_biquad[i * 3    ] = 1.0;
    a_biquad[i * 3 + 1] = ( 2.0 * wa_p2 - 2.0 ) / norm_fact;
    a_biquad[i * 3 + 2] = ( 1.0 - 2.0 * cos( M_PI * ( 2.0 * (psk_double)i + 1.0 ) /
              ( 2.0 * (psk_double)butter_order ) ) * wa + wa_p2 ) / norm_fact;
  }

  // compute polynomial from 2 biquads (SOS)
  if( butter_order == 4 )
  {
    //(a0*s^2+a1*s+a2)*(b0*s^2+b1*s+b2)= s^4(a0*b0)+s^3(a1*b0+a0*b1)+s^2(a2*b0+a1*b1+a0*b2
)+s(a2*b1+a1*b2)+a2*b2
    coefs.b[0] = b_biquad[0] * b_biquad[3];
    coefs.b[1] = b_biquad[1] * b_biquad[3] + b_biquad[0] * b_biquad[4];
    coefs.b[2] = b_biquad[2] * b_biquad[3] + b_biquad[1] * b_biquad[4] + b_biquad[0] * b_
biquad[5];
    coefs.b[3] = b_biquad[2] * b_biquad[4] + b_biquad[1] * b_biquad[5];
    coefs.b[4] = b_biquad[2] * b_biquad[5];

    coefs.a[0] = a_biquad[0] * a_biquad[3];
    coefs.a[1] = a_biquad[1] * a_biquad[3] + a_biquad[0] * a_biquad[4];
    coefs.a[2] = a_biquad[2] * a_biquad[3] + a_biquad[1] * a_biquad[4] + a_biquad[0] * a_
biquad[5];
    coefs.a[3] = a_biquad[2] * a_biquad[4] + a_biquad[1] * a_biquad[5];
    coefs.a[4] = a_biquad[2] * a_biquad[5];

  }

  return( coefs );
}

//------------------------------------------------------------------------------
psk_int32 psk_antialiasing( psk_double*  v_in,
                            psk_uint32   len_in,
                            psk_double** v_out_ptr,
                            psk_uint32*  len_out,
                            psk_double   sample_rate )
{

  psk_double len_butter_out = (psk_double)len_in - (psk_double)BUTTER_SIZE_A;
  psk_double* v_out = calloc( sizeof(psk_double), len_butter_out );

  if( v_out == NULL )
    return( PSK_ERR_OUT_OF_MEM );

  psk_double* out_tmp = calloc( sizeof(psk_double), len_in );
  if( out_tmp == NULL )
  {
    if( v_out )
      free( v_out );

    return( PSK_ERR_OUT_OF_MEM );
  }

  psk_int32 i = 0;
  psk_uint32 t_0 = 0;
  psk_uint32 t_1 = 0;
  psk_uint32 t_2 = 0;
  psk_uint32 t_3 = 0;
  psk_uint32 t_4 = 0;
  psk_butter_coefs coefs;


  coefs = psk_compute_butter_lp_coef( sample_rate );
  for( i = 0; i < len_in; i++ )
  {
    if( i < BUTTER_SIZE_A - 1 )
    {
      t_0 = fmax( 0, i     );
      t_1 = fmax( 0, i - 1 );
      t_2 = fmax( 0, i - 2 );
      t_3 = fmax( 0, i - 3 );
      t_4 = fmax( 0, i - 4 );
      out_tmp[i] = coefs.b[0] * v_in[t_0] + coefs.b[1] * v_in[t_1] +
                   coefs.b[2] * v_in[t_2] + coefs.b[3] * v_in[t_3] +
                   coefs.b[4] * v_in[t_4] -
                   ( coefs.a[0] * out_tmp[t_0] + coefs.a[1] * out_tmp[t_1] +
                   coefs.a[2] * out_tmp[t_2] + coefs.a[3] * out_tmp[t_3] +
                   coefs.a[4] * out_tmp[t_4] );
    }
    else
    {
      out_tmp[i] = coefs.b[0] * v_in[i] + coefs.b[1] * v_in[i - 1] +
                   coefs.b[2] * v_in[i - 2] + coefs.b[3] * v_in[i - 3] +
                   coefs.b[4] * v_in[i - 4] -
                   ( coefs.a[0] * out_tmp[i] + coefs.a[1] * out_tmp[i - 1] +
                   coefs.a[2] * out_tmp[i - 2] + coefs.a[3] * out_tmp[i - 3] +
                   coefs.a[4] * out_tmp[i - 4] );
    }
    if( i > BUTTER_SIZE_A - 1 )
    {
      v_out[i - BUTTER_SIZE_A] = out_tmp[i];
    }
  }

  *v_out_ptr = v_out;
  *len_out = len_butter_out;

  if( out_tmp )
    free( out_tmp );
  out_tmp = NULL;

  return( PSK_ERR_OK );
}

//------------------------------------------------------------------------------
psk_int32 psk_downsample( psk_double*  samples_in,
                          psk_double*  time_in,
                          psk_uint32   len_in,
                          psk_double** samples_out_ptr,
                          psk_double** time_out_ptr,
                          psk_uint32*  len_out)
{
  psk_uint32 k, i; // indexing counters
  psk_double d_y, d_t1, d_t2;
  k = i = 1;
  d_y = d_t1 = d_t2 = 0;
  psk_double t_step = 1.0 / (psk_double)FS_INT;

  psk_double* t_out  = NULL;
  psk_uint32 vec_size;
  t_out = psk_linspace( time_in[0], t_step, time_in[len_in - 1], &vec_size );

  psk_double* v_out = calloc( vec_size, sizeof(psk_double) );
  if( v_out == NULL )
    return( PSK_ERR_OUT_OF_MEM );

  v_out[0] = samples_in[0];

  for( i = 1; i < vec_size; i++)
  {
    k--;
    while ( t_out[i] > time_in[i+k] )
    {
      k++;
    }
    d_y = samples_in[ i + k ] - samples_in[ i + k - 1 ];
    d_t1 = time_in[ i + k ] - time_in[ i + k - 1 ];
    d_t2 = t_out[ i ] - time_in[ i + k - 1 ];
    v_out[i] = samples_in[ i + k - 1 ] + d_y * ( d_t2 / d_t1 );
  }

  *samples_out_ptr = v_out;
  *time_out_ptr = t_out;
  *len_out = vec_size;

  return( PSK_ERR_OK );
}


// ------------------------------------------------------------------------------
psk_int32 psk_ma_filter( psk_double*  samples_in,
                         psk_uint32   len_in,
                         psk_double** samples_out_ptr,
                         psk_uint32*  len_out,
                         psk_uint32   len_filter )
{
  psk_double out_sum, out_sum_prev, a_prev;
  psk_uint32 i, k;
  out_sum = out_sum_prev = a_prev = 0;
  psk_double* filt_buffer = NULL;
  psk_uint32 N_out = len_in - len_filter;
  if( N_out < 1 )
    return( PSK_ERR_PARAMETER );

  i = k = 0;

  filt_buffer = calloc( len_filter, sizeof(psk_double) );
  if( filt_buffer == NULL )
    return( PSK_ERR_OUT_OF_MEM );

  psk_double* v_out = calloc( N_out , sizeof(psk_double) );
  if( v_out == NULL )
  {
    free( filt_buffer );
    return( PSK_ERR_OUT_OF_MEM );
  }

  for( i = 0; i < len_in; i++ )
  {
    a_prev = filt_buffer[len_filter - 1];

    for( k = len_filter - 1; k > 0; k-- )
      filt_buffer[k] = filt_buffer[k - 1];

    filt_buffer[0] = samples_in[i];
    out_sum_prev = filt_buffer[0] - a_prev;
    out_sum += out_sum_prev;

    if( i > len_filter - 1 )
      v_out[i - len_filter] = out_sum / (psk_double)len_filter;
  }

  if( filt_buffer )
    free( filt_buffer );

  *samples_out_ptr = v_out;
  *len_out = N_out;
  return( PSK_ERR_OK );
}

// ---------------------------------------------------------------------------
psk_int32 psk_ma_filter_cmpl( psk_complex*  samples_in,
                              psk_uint32    len_in,
                              psk_complex** samples_out_ptr,
                              psk_uint32*   len_out,
                              psk_uint32    len_filter )
{
  psk_complex out_sum, out_sum_prev, a_prev;
  psk_uint32 i, k;
  RE( out_sum ) = RE( out_sum_prev ) = RE( a_prev ) = 0.0;
  IM( out_sum ) = IM( out_sum_prev ) = IM( a_prev ) = 0.0;
  psk_complex* filt_buffer = NULL;
  psk_uint32 N_out = len_in - len_filter;
  if( N_out < 1 )
    return( PSK_ERR_PARAMETER );

  i = k = 0;

  filt_buffer = calloc( len_filter, sizeof(psk_complex) );
  if( filt_buffer == NULL )
    return( PSK_ERR_OUT_OF_MEM );

  psk_complex* v_out = calloc( N_out , sizeof(psk_complex) );
  if( v_out == NULL )
  {
    free( filt_buffer );
    return( PSK_ERR_OUT_OF_MEM );
  }
  for( i = 0; i < len_in; i++ )
  {
    a_prev = filt_buffer[len_filter - 1];
    for( k = len_filter - 1; k > 0; k-- )
      filt_buffer[k] = filt_buffer[k - 1];

    filt_buffer[0] = samples_in[i];
    out_sum_prev = psk_sub( filt_buffer[0] , a_prev );
    out_sum = psk_add( out_sum, out_sum_prev );

    if( i > len_filter - 1 )
    {
      RE( v_out[i - len_filter] ) = RE( out_sum ) / (psk_double)len_filter;
      IM( v_out[i - len_filter] ) = IM( out_sum ) / (psk_double)len_filter;
    }
  }

  if( filt_buffer )
    free( filt_buffer );

  *samples_out_ptr = v_out;
  *len_out = N_out;

  return( PSK_ERR_OK );
}

// ----------------------------------------------------------------------------------
psk_complex* psk_FIR_filter_cmpl( psk_complex* b,
                                  psk_uint32   num_coef,
                                  psk_complex* sample_in,
                                  psk_uint32   len_in )

{
  psk_uint32 i, k;
  psk_uint32 idx_ptr, idx;
  i = k = idx_ptr = idx = 0;
  psk_complex* circ_buffer = calloc( num_coef, sizeof(psk_complex) );
  psk_complex* out         = calloc( len_in,   sizeof(psk_complex) );
  psk_complex y, res_mult;

  for( i = 0; i < len_in; i++ )
  {
    circ_buffer[idx_ptr] = sample_in[i];

    idx_ptr = ( (idx_ptr + 1) % num_coef);
    RE(y) = 0.0;
    IM(y) = 0.0;
    for( k = 0; k < num_coef; k++ )
    {
      idx = ( ( idx_ptr + k ) % num_coef );
      res_mult = psk_cmpl_mult( b[num_coef - k - 1], circ_buffer[idx] );
      y = psk_add( y, res_mult );
    }
    out[i] = y;
  }

  if( circ_buffer )
    free( circ_buffer );

  return( out );
}

// ----------------------------------------------------------------------------------
psk_double* psk_FIR_filter( psk_double* b,
                            psk_uint32  num_coef,
                            psk_double* sample_in,
                            psk_uint32  len_in )
{
  psk_uint32 i, k;
  psk_uint32 idx_ptr, idx;
  i = k = idx_ptr = idx = 0;
  psk_double* circ_buffer = calloc( num_coef, sizeof(psk_double) );
  psk_double* out         = calloc(   len_in, sizeof(psk_double) );

  psk_double y;

  for( i = 0; i < len_in; i++ )
  {
    circ_buffer[idx_ptr] = sample_in[i];
    idx_ptr = ( ( idx_ptr + 1 ) % num_coef );
    y = 0.0;
    for (k = 0; k < num_coef; k++)
    {
      idx = ( ( idx_ptr + k ) % num_coef );
      y += ( b[num_coef - k - 1] * circ_buffer[idx] );
    }
    out[i] = y;
  }

  if( circ_buffer )
    free( circ_buffer );

  return( out );
}


// --------------------------------------------------------------------------------
void psk_demodulation( psk_double*   samples_in,
                       psk_double*   time_in,
                       psk_uint32    len,
                       psk_complex** v_out_bb )
{
  psk_uint32 i = 0;
  psk_double w_c = 2.0 * (psk_double)M_PI * FC;
  psk_complex* v_out_bb1 = calloc( len, sizeof(psk_complex) );

  for( i = 0; i < len; i++ )
  {
    RE( v_out_bb1[i] ) = samples_in[i] * cos( w_c * time_in[i] );
    IM( v_out_bb1[i] ) = -1.0 * samples_in[i] * sin( w_c * time_in[i] );
  }
  *v_out_bb = v_out_bb1;
}

// --------------------------------------------------------------------------------
void psk_exp_1i_wt( psk_complex*  v_in_bb,
                    psk_double*   time_in,
                    psk_uint32    len,
                    psk_double    freq,
                    psk_complex** v_out_bb )
{
  psk_complex* out_bb = calloc( len, sizeof(psk_complex) );
  psk_uint32 idx = 0;
  psk_double w_c = 2.0 * (psk_double)M_PI * freq;
  psk_complex arg2;

  for( idx = 0; idx < len; idx++ )
  {
    RE( arg2 ) = cosl( w_c * time_in[idx] );
    IM( arg2 ) = sinl( w_c * time_in[idx] );
    out_bb[idx] = psk_cmpl_mult( v_in_bb[idx], arg2 );

  }
  *v_out_bb = out_bb;
}

// --------------------------------------------------------------------------------
psk_int32 psk_derotation( psk_complex*  v_in_bb,
                          psk_double*   time_in,
                          psk_uint32    len,
                          psk_complex** v_out_bb )
{
  psk_complex* ma_out = NULL;
  psk_uint32 filter_len = 4.0 * (psk_double)FS_INT / (psk_double)FC;
  psk_uint32 ma_len = 0;
  psk_int32 status = 0;

  psk_double fc_err = 0.0;

  status = psk_ma_filter_cmpl( v_in_bb,
                               (psk_uint32)IDX_UNMOD,
                               &ma_out,
                               &ma_len,
                               filter_len );

  fc_err = psk_estimate_carrier_freq( ma_out, ma_len );

  fc_err *= ( -1.0 * (psk_double)FS_INT );
  psk_exp_1i_wt( v_in_bb, time_in, len, fc_err, v_out_bb );

  return( status );
}

// -----------------------------------------------------------------------------
psk_double psk_estimate_carrier_freq( psk_complex* v_in_bb,
                                      psk_uint32   len_unmod )
{
  psk_complex* v_in_diff      = calloc( len_unmod - 1, sizeof(psk_complex) );
  psk_complex* v_derotated_bb = NULL;
  psk_double* re_v_in_norm    = calloc( len_unmod, sizeof(psk_double) );
  psk_double* im_v_in_norm    = calloc( len_unmod, sizeof(psk_double) );
  psk_double* re_diff, *im_diff, *phase, *v_abs;
  re_diff = im_diff = phase = v_abs = NULL;
  psk_double *derotate_idx = NULL;

  psk_uint32 idx, idx_len;
  psk_double cmp_abs, mu_v_abs, FcEst, FcEst_2, phase_rot;

  idx = idx_len = 0;
  psk_uint32 mu1_end_idx= 0;
  psk_double mu_phase_start, mu_phase_end;
  mu_phase_start = mu_phase_end = 0.0;

  // ---------------------------------------------------------------------------
  // 1 Step: frequency estimate based on first derivative
  // normalize to abs of each element
  // ---------------------------------------------------------------------------

  for( idx = 0; idx < len_unmod; idx++ )
  {
    cmp_abs = psk_abs( v_in_bb[idx] );

    re_v_in_norm[idx] = RE( v_in_bb[idx] ) / cmp_abs;
    im_v_in_norm[idx] = IM( v_in_bb[idx] ) / cmp_abs;
  }

  // compute first difference
  re_diff = psk_diff( re_v_in_norm, len_unmod );
  im_diff = psk_diff( im_v_in_norm, len_unmod );

  // form complex vector
  for( idx = 0; idx < ( len_unmod - 1 ); idx++ )
  {
    RE( v_in_diff[idx] ) = re_diff[idx];
    IM( v_in_diff[idx] ) = im_diff[idx];
  }
  // computs complex abs
  v_abs = psk_vec_abs( v_in_diff, ( len_unmod - 1 ) );

  mu_v_abs = psk_mean( v_abs, ( len_unmod - 1 ) );
  // estimated frequency offset to FC
  FcEst  = asinl( ( mu_v_abs / 2.0 ) ) / (psk_double)M_PI;
  derotate_idx = psk_linspace( 1.0, 1.0, (psk_double)len_unmod, &idx_len );

  // derotate signal for step 2 of frequency offset determination
  psk_exp_1i_wt( v_in_bb, derotate_idx,
                 len_unmod,
                 (-1.0) * FcEst,
                 &v_derotated_bb );

  // compute phase
  phase = psk_vec_angle( v_derotated_bb, len_unmod );
  // unwrap phase
  psk_double* uw_phase = psk_unwrap( phase, len_unmod );

  // free not use memory

  free(re_v_in_norm);   re_v_in_norm   = NULL;
  free(im_v_in_norm);   im_v_in_norm   = NULL;
  free(re_diff);        re_diff        = NULL;
  free(im_diff);        im_diff        = NULL;
  free(v_abs);          v_abs          = NULL;
  free(v_derotated_bb); v_derotated_bb = NULL;
  free(phase);

  // ---------------------------------------------------------------------------
  // Step 2: fine estimate of frequency difference to FC based on
  // slope computation
  // ---------------------------------------------------------------------------
  mu1_end_idx = len_unmod / 2;
  // compute mean of start and end section
  mu_phase_start = psk_mean( uw_phase, mu1_end_idx );

  for( idx = mu1_end_idx - 1; idx < len_unmod; idx++ )
    mu_phase_end += uw_phase[idx];

  mu_phase_end /= (psk_double)( len_unmod - mu1_end_idx + 1 );

  // compute slope and transform to frequency error with respect to FC
  phase_rot = mu_phase_end - mu_phase_start;

  FcEst_2 = phase_rot / 2.0 / (psk_double)M_PI /
          ( ( (psk_double)len_unmod - 1.0 ) / 2.0 );
  // complete frequency difference to FC
  FcEst += FcEst_2;

  free( uw_phase );
  phase = NULL;
  free( derotate_idx );
  derotate_idx = NULL;

  return( FcEst );
}

// -----------------------------------------------------------------------------
psk_int32 psk_cross_covariance( psk_complex*  v_in1,
                                psk_uint32    len_in1,
                                psk_complex*  v_in2,
                                psk_uint32    len_in2,
                                psk_uint32    lags,
                                psk_complex** v_out,
                                psk_int32**   lagIdx_vec )
{
  psk_complex mu_in1, mu_in2;
  psk_uint32 lagIdx_len, i;
  lagIdx_len = i = 0;

  psk_complex* v_in1_wo_mu = calloc( len_in1, sizeof(psk_complex) );
  psk_complex* v_in2_wo_mu = calloc( len_in2, sizeof(psk_complex) );

  // compute mean of input signal
  mu_in1 = psk_cmpl_mean( v_in1, len_in1 );
  mu_in2 = psk_cmpl_mean( v_in2, len_in2 );
```

```
// subtract mean
for( i = 0; i < len_in2; i++ )
  v_in2_wo_mu[i] = psk_sub( v_in2[i], mu_in2 );

// compute conjugate complex
v_in2_wo_mu = psk_cmpl_vec_conj( v_in2_wo_mu, len_in2 );

// subtract mean
if( lags == 0 )
{
  psk_complex res1;
  psk_complex* out1 = calloc( 1, sizeof(psk_complex) );
  psk_int32* lagIdx1 = calloc( 1, sizeof(psk_int32) );
  *lagIdx1 = 0;
  RE(*out1) = 0.0;
  IM(*out1) = 0.0;
  for( i = 0; i < len_in1; i++ )
    v_in1_wo_mu[i] = psk_sub( v_in1[i], mu_in1 );

  for( i = 0; i< len_in1; i++ )
  {
    res1 = psk_cmpl_mult( v_in1_wo_mu[i], v_in2_wo_mu[i] );
    *out1 = psk_add( *out1, res1 );
  }

  *v_out = out1;
  *lagIdx_vec = lagIdx1;
}
else
{
  psk_complex* out = NULL;
  psk_double* lagIdx = NULL;

  for( i = 0; i < len_in1; i++ )
    v_in1_wo_mu[len_in1 - ( 1 + i )] = psk_sub( v_in1[i], mu_in1 );

  out = psk_FIR_filter_cmpl( v_in1_wo_mu, len_in1, v_in2_wo_mu, lags );
  lagIdx = psk_linspace( (psk_double)( len_in1 - 1 ),
                         -1.0,
                         (psk_double)len_in1 - (psk_double)lags ,
                         &lagIdx_len );

  psk_int32* lagIdx_int32 = calloc(lagIdx_len, sizeof(psk_int32) );

  for( i = 0; i < lagIdx_len; i++ )
    lagIdx_int32[i] = (psk_int32)lagIdx[i];

  if( lagIdx_len != lags )
  {
    if( lagIdx )
      free( lagIdx );
    lagIdx = NULL;

    if( lagIdx_int32 )
      free( lagIdx_int32 );
    lagIdx_int32 = NULL;
    return( PSK_ERR_SIGNAL_LEN_MISMATCH );
  }

  *v_out = out;
  *lagIdx_vec = lagIdx_int32;

  if( lagIdx )
    free( lagIdx );
  lagIdx = NULL;
}

if( v_in1_wo_mu )
  free( v_in1_wo_mu );
v_in1_wo_mu = NULL;
```

```
  if( v_in2_wo_mu )
    free( v_in2_wo_mu );
  v_in2_wo_mu = NULL;

  return( PSK_ERR_OK );
}


// -----------------------------------------------------------------------------
psk_double* psk_unwrap( psk_double* v_in, psk_uint32 len_in )
{
  psk_uint32 idx, k_p, k_n;
  idx = k_p = k_n = 0;
  psk_double* out = calloc( len_in, sizeof(psk_double) );
  psk_double* in_diff = NULL;

  in_diff = psk_diff( v_in, len_in );
  out[0] = v_in[0];

  for( idx = 1; idx < len_in; idx++ )
  {
    if( in_diff[idx - 1] >= (psk_double)M_PI )
      k_p += 1;

    if( in_diff[idx - 1] <= ( -1.0 * (psk_double)M_PI ) )
      k_n += 1;

    out[idx] = v_in[idx] +
               ( ( (psk_double)k_n - (psk_double)k_p ) * 2.0 * (psk_double)M_PI );
  }
  free( in_diff );
  in_diff = NULL;

  return( out );
}

// -----------------------------------------------------------------------------
 psk_complex* psk_ff_lms( psk_complex* v_in1,
                          psk_complex* v_in2,
                          psk_uint32   len,
                          psk_uint32   n_taps )
{
  psk_uint32 i, k;
  i = k = 0;
  psk_complex* shift_reg = calloc( n_taps, sizeof(psk_complex) );
  psk_double lambda, tmp, tau, mu;
  lambda = tmp = tau = mu = 0.0;
  psk_double slowDown = 1.4; // default: 1.0
  psk_complex mu_in1 = psk_cmpl_mean( v_in1, len );
  psk_complex* h_est = calloc( n_taps, sizeof(psk_complex) );
  psk_complex* h_est_out = calloc( n_taps, sizeof(psk_complex) );


  for( i = 0; i < len; i++ )
  {
    tmp = psk_abs( psk_sub( v_in1[i], mu_in1 ) );
    lambda += pow( tmp, 2 );
  }
  lambda *= ( 1.0 / ( (psk_double)len - 1.0 ) );
  tau = (psk_double)len / ( 9.0 / slowDown );
  mu = ( 1.0 - expl( -1.0 / tau ) ) / lambda;

  psk_complex update_val, err, q;
  psk_complex* conj_shift_reg;
  psk_complex* q_vec;
  // initialize h_est
  RE( h_est[0] ) = 0.1;

  for( i = 0; i < len; i++ )
  {
    if( h_est_out )
    {
      free(h_est_out);
      h_est_out = NULL;
    }

    for( k = n_taps - 1; k > 0; k-- )
      shift_reg[k] = shift_reg[k - 1];

    shift_reg[0] = v_in1[i];

    q_vec = psk_cmpl_vec_mult( h_est, shift_reg, n_taps );
    RE( q ) = 0.0;
    IM( q ) = 0.0;
    for( k = 0; k < n_taps; k++ )
      q = psk_add( q, q_vec[k] );

    if( q_vec )
    {
      free( q_vec );
      q_vec = NULL;
    }

    err = psk_sub( v_in2[i], q );
    RE( err ) = RE( err ) * mu;
    IM( err ) = IM( err ) * mu;

    conj_shift_reg = psk_cmpl_vec_conj( shift_reg, n_taps );

    for( k = 0; k < n_taps; k++ )
    {
      update_val = psk_cmpl_mult( conj_shift_reg[k], err );
      h_est[k] = psk_add( h_est[k], update_val );
    }
    if( conj_shift_reg )
    {
      free( conj_shift_reg );
      conj_shift_reg = NULL;
    }
    h_est_out = psk_cmpl_vec_conj( h_est, n_taps );
  }
  // free memory
  if( h_est )
    free( h_est );
  if( shift_reg )
    free( shift_reg );

  return( h_est_out );
}
//---------------------------------------------------------------------------
psk_int32 psk_get_phase_noise( psk_complex* sig,
                               psk_uint32   sig_len,
                               psk_double*  phase_noise )
{
  psk_double dpn = 0.0; // differential phase noise value
  psk_double downsample_factor = round( (psk_double)FS_INT * ETU );

  psk_uint32 start = 20; // disregard transient at beginning
  psk_uint32 step  = downsample_factor;
  psk_uint32 stop  = (psk_double)IDX_UNMOD;
  psk_uint32 len = 0;
  psk_uint32* idx_vec = psk_idx_linspace( start, step, stop, &len );
  if( idx_vec == NULL )
    return( PSK_ERR_OUT_OF_MEM );

  if( sig_len < stop || sig_len < len )
  {
    if( idx_vec )
      free( idx_vec );
    return( PSK_ERR_INVALID_SAMPLE_VEC );
  }
  // get the right symbols
  psk_complex iq_symbols[len];
  psk_uint32 idx;
  for( idx = 0; idx < len; idx++ )
    iq_symbols[idx] = sig[ idx_vec[idx] ];

  // get angles of elements
  psk_double* phi = NULL;
  phi = psk_vec_angle( iq_symbols, len );
  if( phi == NULL )
  {
    if( idx_vec )
      free( idx_vec );
    return( PSK_ERR_OUT_OF_MEM );
  }

  // get error of angles
  psk_double* phi_err = psk_diff( phi, len );
  if( phi_err == NULL )
  {
    if( idx_vec )
      free( idx_vec );
    return( PSK_ERR_OUT_OF_MEM );
  }

  // get rms value of phase noise.
  for ( idx = 0; idx < ( len - 1 ); idx++ )
    dpn += pow( phi_err[idx], 2.0 );

  dpn = sqrt( dpn / (psk_double)(len - 1) ) - 0.01 * ( (psk_double)FC/4.0 * (psk_double)
ETU) * (M_PI/180.0); // subtract machine-internal noise

  dpn /= ( EPI * (psk_double)M_PI / 180.0 );

  *phase_noise =  dpn ;

  // clean up
  if( phi )
    free( phi );

  if( phi_err )
    free( phi_err );

  if( idx_vec )
    free( idx_vec );

  return ( PSK_ERR_OK );
}

//--------------------------------------------------------------------------------
psk_int32 psk_isi_param( psk_complex*  sig_bb_int,
                         psk_uint32    sig_bb_len,
                         psk_double*   isi_m,
                         psk_double*   isi_d,
                         psk_double*   phase_range,
                         psk_complex** sig_out )
{
  psk_uint32 idx;

  // used in loop to downsample the input signal
  psk_uint32 n_taps_k  = 4; // number of symbols
  psk_uint32 start_idx = (psk_double)FS_INT / ( 16.0 * (psk_double)FC );
  psk_uint32 end_idx   = (psk_double)sig_bb_len
                         - 16.0 * (psk_double)start_idx * (psk_double)FC * ETU;

  psk_uint32 nyquist_len = 16;
  psk_uint32 sps_p       = round( 16.0 * (psk_double)FC * ETU );
  psk_uint32* idx_vec    = NULL;

  const psk_int32* soc_lut; // constant array depending on ORDER
```

```
if( ORDER == 8 )
  soc_lut = SOC_PSK8_deg;
else if( ORDER == 16)
  soc_lut = SOC_PSK16_deg;
else
  return(PSK_ERR_PARAMETER);

psk_complex* estimate_symbol_resp = calloc( sps_p * n_taps_k,
                                            sizeof(psk_complex) );

psk_complex* ref_sig = calloc( PSK_len, sizeof( psk_complex ) );
for( idx = 0; idx < PSK_len; idx++ )
{
  RE(ref_sig[idx]) = cos( (psk_double)soc_lut[idx]*(psk_double)M_PI / 180.0 );
  IM(ref_sig[idx]) = sin( (psk_double)soc_lut[idx]*(psk_double)M_PI / 180.0 );
}

// estimate symbol response
for( idx = 0; idx < sps_p; idx++ )
{
  psk_uint32 len = 0;
  idx_vec = psk_idx_linspace( start_idx + idx * start_idx,
                              start_idx * sps_p,
                              end_idx + ( idx + 1 ) * start_idx,
                              &len );
  psk_complex bb_out_grid_align_k[len];
  psk_uint32 cpy_idx;
  for( cpy_idx = 0; cpy_idx < len; cpy_idx++ )
    bb_out_grid_align_k[cpy_idx] = sig_bb_int[idx_vec[cpy_idx] - 1];


  psk_complex* hVec_k = psk_ff_lms( ref_sig,
                                    bb_out_grid_align_k,
                                    len,
                                    n_taps_k );
  // hVec_k = [a1 b1 c1 d1 ] if idx = 0
  // hVec_k = [a2 b2 c2 d2 ] if idx = 1 and so on
  // estimate_symbol_resp=[a1 a2 a3...][b1 b2 b3...][c1 c2 c3...][d1 d2 d3...]
  // because n_taps_k == 4 :
  estimate_symbol_resp[idx + (n_taps_k - 1) * sps_p] = hVec_k[(n_taps_k - 1)];
  estimate_symbol_resp[idx + (n_taps_k - 2) * sps_p] = hVec_k[(n_taps_k - 2)];
  estimate_symbol_resp[idx + (n_taps_k - 3) * sps_p] = hVec_k[(n_taps_k - 3)];
  estimate_symbol_resp[idx + (n_taps_k - 4) * sps_p] = hVec_k[(n_taps_k - 4)];

  if ( hVec_k )
    free( hVec_k );
}

psk_complex h[n_taps_k];
psk_complex tmp_sum;
RE( tmp_sum ) = 0.0;
IM( tmp_sum ) = 0.0;
psk_complex s0, s1;
RE( s0 ) = cosl( PR / 2.0 / 180.0 * M_PI );
IM( s0 ) = sinl( PR / 2.0 / 180.0 * M_PI );
s1 = psk_cmpl_conj( s0 );
psk_complex s_in[nyquist_len];

psk_double pr_comp; // stores current phase range
psk_complex s_in_pr[16];

for( idx = 0; idx < nyquist_len; idx++ )
  s_in[idx] = ( idx % 2 ) ? s0 : s1;

// s_in_pr = [s1 s1 s1 s1 s0 s0 s0 s0 s1 s1 s1 s1 s0 s0 s0 s0]
for( idx = 0; idx < 4; idx++ )
{
  s_in_pr[idx]      = s1;
  s_in_pr[idx + 8] = s1;
  s_in_pr[idx + 4] = s0;
  s_in_pr[idx + 12] = s0;
  }

  psk_complex* s_out = NULL;
  psk_complex* pr_out = NULL;
  psk_double L = 0.0;
  psk_double isi_angle_1 = 0.0;
  psk_double isi_angle_2 = 0.0;
  psk_double isi_angle = 0.0;
  psk_uint32 idx_max = 0;
  psk_double* isi_m_vec = calloc( nyquist_len+2, sizeof(psk_double) );
  psk_double* isi_d_vec = calloc( nyquist_len+2, sizeof(psk_double) );
  psk_double* phase_range_vec = calloc( nyquist_len+2, sizeof(psk_double) );
  psk_uint32 idx2=0;

  for( idx = sps_p; idx > (sps_p - nyquist_len - 1) ; idx-- )
  {
    RE( tmp_sum ) = 0.0;
    IM( tmp_sum ) = 0.0;

    // estimate_symbol_resp./sum(estimate_symbol_resp)
    tmp_sum = psk_add( tmp_sum, estimate_symbol_resp[ idx + 0 * sps_p - 1] );
    tmp_sum = psk_add( tmp_sum, estimate_symbol_resp[ idx + 1 * sps_p - 1] );
    tmp_sum = psk_add( tmp_sum, estimate_symbol_resp[ idx + 2 * sps_p - 1] );
    tmp_sum = psk_add( tmp_sum, estimate_symbol_resp[ idx + 3 * sps_p - 1] );
    h[0] = psk_cmpl_div( estimate_symbol_resp[ idx + 0 * sps_p - 1], tmp_sum );
    h[1] = psk_cmpl_div( estimate_symbol_resp[ idx + 1 * sps_p - 1], tmp_sum );
    h[2] = psk_cmpl_div( estimate_symbol_resp[ idx + 2 * sps_p - 1], tmp_sum );
    h[3] = psk_cmpl_div( estimate_symbol_resp[ idx + 3 * sps_p - 1], tmp_sum );

    // fir filter(h, s_in)
    s_out = psk_FIR_filter_cmpl( h, n_taps_k, s_in, nyquist_len );
    if( !s_out )
      return( PSK_ERR_OUT_OF_MEM );

    // phase range computation
    pr_out = psk_FIR_filter_cmpl( h, n_taps_k, s_in_pr, 16 );
    if ( !pr_out )
      return( PSK_ERR_OUT_OF_MEM );

    // compute phase range
    psk_complex pr_mean_high;
    RE( pr_mean_high ) = ( RE( pr_out[6] ) + RE( pr_out[7] ) + RE( pr_out[14] ) + RE( pr_
out[15] ) ) / 4.0;
    IM( pr_mean_high ) = ( IM( pr_out[6] ) + IM( pr_out[7] ) + IM( pr_out[14] ) + IM( pr_
out[15] ) ) / 4.0;
    psk_double pr_angle1 = atan2l( IM( pr_mean_high ), RE( pr_mean_high ) );

    psk_complex pr_mean_low;
    RE( pr_mean_low ) = ( RE( pr_out[2] ) + RE( pr_out[3] ) + RE( pr_out[10] ) + RE( pr_
out[11] ) ) / 4.0;
    IM( pr_mean_low ) = ( IM( pr_out[2] ) + IM( pr_out[3] ) + IM( pr_out[10] ) + IM( pr_
out[11] ) ) / 4.0;
    psk_double pr_angle2 = atan2l( IM( pr_mean_low ), RE( pr_mean_low ) );

    pr_comp = ( pr_angle1 - pr_angle2 ) * 180.0 / (psk_double)M_PI;


    psk_complex diff;
    psk_double tmp, max_val1, max_val2;
    tmp = max_val1 = max_val2 = 0.0;
    for( idx2 = 0; idx2 < nyquist_len; idx2++ )
    {
      // calculate isi_angle_1
      diff = psk_sub( s_out[idx2], s_in[idx2] );
      tmp = psk_abs( diff );
      if( tmp > max_val1 )
        max_val1 = tmp;

      //calculate isi_angle_2
      tmp  = atan2( IM( s_out[idx2] ), RE( s_out[idx2] ) );
      tmp -= atan2( IM( s_in[idx2] ),  RE( s_in[idx2] ) );
      tmp = ABS( tmp );
      if( tmp > max_val2 )
        max_val2 = tmp;
    }
    isi_angle_2 = max_val2;

    L = ( max_val1 > 0.99999 ) ? 0.99999 : max_val1;
    isi_angle_1 = asin( L );
    isi_angle = ( isi_angle_1 > isi_angle_2 ) ? isi_angle_1 : isi_angle_2;

    // find max isi_m value
    isi_m_vec[ sps_p - idx ] = isi_angle * 180.0 / M_PI / PR * ( ORDER - 1);
    isi_d_vec[ sps_p -idx ] = -( atan2( IM( h[1] ), RE( h[1] ) ) -
                             atan2( IM( h[0] ), RE( h[0] ) ) ) * 180.0 / M_PI;
    phase_range_vec[ sps_p - idx ] = pr_comp;

    if( s_out )
      free( s_out );
    s_out = NULL;

    if( pr_out )
      free( pr_out );
    pr_out = NULL;
  }

  for( idx = 0; idx < 3; idx++)
  {
    idx_max = 0;
    for( idx2 = 1; idx2 < nyquist_len+1; idx2 ++)
    {
      if( isi_m_vec[idx2] > isi_m_vec[idx_max] )
      {
        idx_max = idx2;

      }
    }
    *isi_m = isi_m_vec[idx_max];
    isi_m_vec[idx_max] = -1e302;
  }
  *isi_d = isi_d_vec[idx_max];
  *phase_range = phase_range_vec[idx_max];

  // isi_m, isi_d computation done. downsample sig_in
  psk_uint32 ds_start = (start_idx + sps_p - nyquist_len + idx_max) * start_idx;
  psk_uint32 ds_end   = (sig_bb_len / (start_idx * sps_p) ) * start_idx * sps_p;
  psk_uint32 ds_out_len = 0;
  psk_uint32* out_idx_vec = psk_idx_linspace( ds_start - 1,
                                              start_idx * sps_p,
                                              ds_end,
                                              &ds_out_len );

  psk_complex* sig_out_ = calloc( ds_out_len, sizeof(psk_complex) );
  for( idx = 0; idx < ds_out_len; idx++ )
    sig_out_[idx] =  sig_bb_int[out_idx_vec[idx] - 1] ;

  *sig_out = sig_out_;

  if( idx_vec )
    free( idx_vec );
  idx_vec = NULL;

  if( out_idx_vec )
    free( out_idx_vec );
  out_idx_vec = NULL;

  if( estimate_symbol_resp )
    free( estimate_symbol_resp );
  estimate_symbol_resp = NULL;

  return( PSK_ERR_OK );
}
```

```
//--------------------------------------------------------------------------------
psk_int32 psk_re_align_symbol_grid( psk_complex* v_in,
                                    psk_uint32 len,
                                    psk_uint32* strt_idx,
                                    psk_uint32* end_idx )
{
  psk_uint32 idx, end_idx_1, strt_idx_1, sig_len, N_symb, sps;
  idx = end_idx_1 = strt_idx_1 = sig_len = N_symb = sps = 0;
  sps = round( (psk_double)FS_INT * ETU );
  psk_uint32 L_start = 50; // 50 symbols to remove beginning

  // -------------------------------------------------------------------------
  // (Step A) Realign on a symbol level:
  // Input              : y_ref,                       Reference symbols
  // Input              : in_bb_m                          resampled (to int fc)
  //                    demod data (scope into complex Baseband), LONGER than REF
  // Output(internal)   : BasebandRx_SymbAlign_m       ALIGNED to REF ± 2
  //                                Symbols (trimmed signal)             fs = N*fc
  // -------------------------------------------------------------------------

  // find approx start and end points of phase modulated signal
  psk_double offset = (24.0 - 8.0 ) * (psk_double)M_PI / 180.0;
  psk_double* phi = psk_vec_angle( v_in, len);


  //---------------------------------------------
  // (1) find and remove silence at the beginning, keep N etu's silence before
  // first modulation
  //---------------------------------------------

  // find beginning
  while( phi[idx] <= offset )
    idx++;

  strt_idx_1 = idx;

  // find end
  idx = 0;
  while( phi[ len - (1 + idx) ] <= offset )
    idx++;

  end_idx_1 = idx;


  if( strt_idx_1 > 25 * sps )
    strt_idx_1 = strt_idx_1 - 25 * sps + 1;
  else
    strt_idx_1 =0;

  if( end_idx_1 > 10 * sps )
    end_idx_1 = len - end_idx_1 - 2 + 10 * sps;
  else
    end_idx_1 = len - 1;

  sig_len = end_idx_1 - strt_idx_1 + 1;

  N_symb = (psk_double)sig_len / (psk_double)sps;
  if( len < PSK_len * sps )
    return( PSK_ERR_SIGNAL_TOO_SHORT );


  // normalize input signal
  psk_complex* v_in_norm = psk_cmpl_normalize( v_in, len );

  //---------------------------------------------
  // (2) take a initial random sampling phase
  //---------------------------------------------

  psk_uint32 t_idx_len = 0;
  psk_uint32* t_idx_vec = psk_idx_linspace( (strt_idx_1+sps) , sps, end_idx_1, &t_idx_
len);
```

```
psk_complex* symb_rx = calloc( t_idx_len, sizeof(psk_complex) );
for( idx = 0; idx < t_idx_len; idx++ )
  symb_rx[idx] = v_in_norm[ t_idx_vec[idx] ];


// compute complex signal of SOC
psk_complex* y_ref = calloc( PSK_len, sizeof(psk_complex) );
psk_complex* y_ref_cut = calloc( PSK_len - L_start, sizeof(psk_complex) );
const psk_int32* soc_lut;
if( ORDER == 8 )
  soc_lut = SOC_PSK8_deg;
else if( ORDER == 16)
  soc_lut = SOC_PSK16_deg;
else
  return(PSK_ERR_PARAMETER);


for( idx = 0; idx < PSK_len; idx++ )
{
  RE(y_ref[idx]) = cosl( (psk_double)soc_lut[idx]*(psk_double)M_PI / 180.0 );
  IM(y_ref[idx]) = sinl( (psk_double)soc_lut[idx]*(psk_double)M_PI / 180.0 );
  if( idx > L_start - 1 )
  {
    RE( y_ref_cut[idx - L_start] ) = RE( y_ref[idx] );
    IM( y_ref_cut[idx - L_start] ) = IM( y_ref[idx] );
  }
}


psk_uint32 max_shift = N_symb - (psk_uint32)PSK_len+L_start;

//--------------------------------------------
// (3) Align on a symbol level
//--------------------------------------------
psk_uint32 delay_1 = 0;

psk_int32 err = psk_find_approx_delay( y_ref_cut,
                                       PSK_len-L_start,
                                       symb_rx,
                                       t_idx_len,
                                       max_shift,
                                       &delay_1);

if( err < 0)
  return(err);


delay_1 -= (L_start-1); // corr. by L_start
psk_uint32 idxBegin = strt_idx_1 + delay_1*sps - 1;
psk_uint32 idxEnd   = idxBegin + (psk_uint32)PSK_len * sps - 1;


if( y_ref_cut )
{
  free( y_ref_cut );
  y_ref_cut = NULL;
}

if( symb_rx )
{
  free( symb_rx );
  symb_rx = NULL;
}

// ----------------------------------------------------------------------------
// (Step B) Realign on a sample level:
//  Input: y_ref REF TX symbols  fSymb
// Input: BasebandRx_SymbAlign_m: ALIGNED to REF ± 2 Symbol  fsOut = N*fc
// ----------------------------------------------------------------------------
```

```
  psk_uint32 MAX_MISALIGN        = 2; //upper bound to input misalignment
  psk_uint32 FLAT_REGION_MIN     = 1; //algo specific, MIN 1, 2 safer; get at
  // least a full etu leading the beginning of symbol response
  psk_uint32 GuardTimeInterval   = 0; // This param depends loosely on below
  // def of MaxDev and on input SNR. These vals are taken for SNR_dB_IQ=30dB
  // (very noisy), which tend to increase guard interval.

  psk_uint32 Ns_m = idxEnd-idxBegin+1; // signal length
  psk_uint32 gridVals = ( 2 * MAX_MISALIGN + FLAT_REGION_MIN + 1 ) * sps; // idx length
  psk_uint32 N_samples = ( Ns_m - gridVals ) / sps; // number of samples used for
cross-covariance compuation
  psk_complex* futureXC = calloc( gridVals, sizeof(psk_complex) );
  psk_double* DelaySignature = calloc( gridVals, sizeof(psk_double) );

  // ----------------------------------------------
  // (1) Get a pseudo symbol-response (signature1)
  // ----------------------------------------------

  // start/stop index definition for y_ref signal
  // future  tx symbols
  psk_uint32 strt_idx_y_ref = FLAT_REGION_MIN + MAX_MISALIGN;
  psk_uint32 end_idx_y_ref = N_samples + 1;

  // prepare reference signal for cross-correlation operation
  psk_uint32 N_y_ref_cut = (end_idx_y_ref) - (strt_idx_y_ref + L_start) + 1;
  y_ref_cut = calloc( N_y_ref_cut, sizeof( psk_complex ) );
  psk_complex* y_ref_cut_conj = calloc( N_y_ref_cut, sizeof( psk_complex ) );

  for( idx = strt_idx_y_ref + L_start-1; idx< end_idx_y_ref; idx++ )
  {
    RE( y_ref_cut[ idx - (strt_idx_y_ref + L_start-1) ] ) = RE( y_ref[idx] );
    IM( y_ref_cut[ idx - (strt_idx_y_ref + L_start-1) ] ) = IM( y_ref[idx] );
  }

  psk_complex mu_x2 = psk_cmpl_mean( y_ref_cut, N_y_ref_cut );
  for( idx = 0; idx < N_y_ref_cut; idx++ )
  {
    y_ref_cut[ idx ] = psk_sub( y_ref_cut[ idx ], mu_x2 );
    y_ref_cut_conj[ idx ] = psk_cmpl_conj( y_ref_cut[ idx ] );
  }

  // length index for signal at fs
  psk_uint32 const_end_offset_m = ( N_samples - MAX_MISALIGN - FLAT_REGION_MIN ) * sps;
  psk_uint32 Lstart_m = L_start * sps; // length of periodic part in SOC at fs

  psk_uint32 start, stop,iGrid;
  start = stop = 0;
  psk_uint32 len_vec_idx = 0;
  psk_complex prod_tmp;
  psk_uint32 *idx_vec = NULL;

  for( iGrid = 0; iGrid < gridVals; iGrid++ )
  {
    // compute start/stop indices
    start =  idxBegin + iGrid - 1 + Lstart_m - sps;
    stop =   idxBegin + iGrid - 1 + const_end_offset_m;
    // get index vector
    idx_vec = psk_idx_linspace(start, sps, stop, &len_vec_idx);

    // cut out signal acc. to index vector
    symb_rx = calloc( len_vec_idx, sizeof(psk_complex) );
    for( idx = 0; idx < len_vec_idx; idx++ )
      symb_rx[idx] = v_in_norm[ idx_vec[idx] ];

    //compute and remove mean
    psk_complex mu_x1 = psk_cmpl_mean( symb_rx, len_vec_idx );
    for( idx = 0; idx < len_vec_idx; idx++ )
    {
      symb_rx[idx] = psk_sub( symb_rx[idx], mu_x1 );
      prod_tmp = psk_cmpl_mult( symb_rx[idx],  y_ref_cut_conj[idx] );
      futureXC[iGrid] = psk_add( futureXC[iGrid], prod_tmp );
    }
    DelaySignature[iGrid] = RE( futureXC[iGrid] );
    if( symb_rx )
    {
      free( symb_rx );
      symb_rx = NULL;
    }
    if( idx_vec )
    {
      free( idx_vec );
      idx_vec = NULL;
    }
  }

  // perform quality checks
  psk_complex *ref_corr_0;
  psk_int32 *ref_lag_0;
  psk_double max_corr_val, peak_ref_0 ;
  psk_uint32 max_corr_idx;

  psk_int32 err1 = psk_cross_covariance( y_ref,
                                         PSK_len,
                                         y_ref,
                                         PSK_len,
                                         0,
                                         &ref_corr_0,
                                         &ref_lag_0 );

  if( err1 != PSK_ERR_OK )
    return( err1 );

  peak_ref_0 = psk_abs( ref_corr_0[0] );
  // ----------------------------------------------
  // (2) Find the peak in signature1
  // ----------------------------------------------

  psk_max( DelaySignature, gridVals, &max_corr_val, &max_corr_idx );

  if( max_corr_val < ( peak_ref_0 / 4.0 ) )
  {
    printf("\nSymbol Grid Alignment Failed, no reliable xcorr peak found.\n");
    return( PSK_SYMBOL_GRID_ALIGNMENT_FAIL );
  }

  if( futureXC )
    free( futureXC );

  // ----------------------------------------------------------------------
  // (3) Consider signature2 the slope (derivative) of signature1.
  // ----------------------------------------------------------------------

  psk_uint32 num_coef = 6;
  psk_double slope_coef[] = {1.0 / 12.0,
                             3.0 / 12.0,
                             2.0 / 12.0,
                            -2.0 / 12.0,
                            -3.0 / 12.0,
                            -1.0 / 12.0}; // normalized by sum(abs(nominator))

  psk_uint32 strt_idx_diff = num_coef +1;


  psk_double *DelaySignatureDiff = psk_FIR_filter( slope_coef,
                                                   num_coef,
                                                   DelaySignature,
                                                   gridVals );

  // ----------------------------------------------
  // (4) identify flat region in signature 2, to the left of the peak in signature1
  // flat region first guess based on signature 1 peak
  // ----------------------------------------------

  //point P1, earliest corner position
  psk_uint32 flatRegionCornerPosMin = max_corr_idx - (psk_uint32)( 1.3 * (psk_double)sps );
  psk_uint32 idx1 = MAX( num_coef, (flatRegionCornerPosMin - sps) );

  // update start index:
  psk_uint32 end_flatregion = flatRegionCornerPosMin;
  psk_uint32 strt_idx_diff_idx = strt_idx_diff + idx1 - 1;

  // If needed adjust flat region by 1/2 etu to the right (if we detect we are too early
  // (EMC case), this enables calculating MaxDev on a more significant portion of the flat
region)
  psk_uint32 strt_idx_diff_emc = strt_idx_diff_idx + sps / 2;
  psk_uint32 end_flatregion_emc = end_flatregion + sps / 2;

  // cut out signal acc. to index vector
  len_vec_idx = end_flatregion - strt_idx_diff_idx + 1;
  psk_double *tmp_vec = calloc( len_vec_idx, sizeof(psk_double) );
  for( idx = strt_idx_diff_idx; idx <= end_flatregion; idx++ )
      tmp_vec[ idx-strt_idx_diff_idx ] = DelaySignatureDiff[ idx ];

  psk_double std_delSigDiff = psk_std(tmp_vec, len_vec_idx);
  std_delSigDiff *= 2.0;

  if(tmp_vec)
  {
    free(tmp_vec);
    tmp_vec = NULL;
  }

  // cut out signal acc. to index vector EMC based part
  len_vec_idx = end_flatregion_emc - strt_idx_diff_emc+1;
  tmp_vec = calloc( len_vec_idx, sizeof(psk_double) );
  for( idx = strt_idx_diff_emc; idx <= end_flatregion_emc; idx++)
      tmp_vec[ idx - strt_idx_diff_emc ] = DelaySignatureDiff[ idx ];

  psk_double std_delSigDiff_emc = psk_std(tmp_vec, len_vec_idx);

  if ( std_delSigDiff_emc < std_delSigDiff )
  {
    printf("\nOvershooting Channel suspected, improving flat region location\n");

    flatRegionCornerPosMin  = flatRegionCornerPosMin + sps / 4;
    idx1                    = MAX( num_coef, ( flatRegionCornerPosMin - sps ) );
    strt_idx_diff_idx = strt_idx_diff_emc;
    end_flatregion = end_flatregion_emc;

  }

  if( tmp_vec )
  {
    free( tmp_vec );
    tmp_vec = NULL;
  }

  // ----------------------------------------------
  // (5) define "end of flat region", or corner point of signature 2
  // define "end of flat region", by a vertical interval within which flat
  // region TO THE RIGHT P1 should be contained
  // ----------------------------------------------

  // cut out signal acc. to index vector EMC based part
  len_vec_idx = end_flatregion - strt_idx_diff + 1;
  tmp_vec = calloc( len_vec_idx, sizeof(psk_double) );
  for( idx = strt_idx_diff; idx < end_flatregion; idx++)
      tmp_vec[ idx - strt_idx_diff ] = DelaySignatureDiff[ idx ];

  psk_double MeanVal = psk_mean( tmp_vec, len_vec_idx );
  psk_double MaxDev = psk_std(tmp_vec, len_vec_idx);
  MaxDev *= 6.0;

  if( tmp_vec )
  {
    free( tmp_vec );
    tmp_vec = NULL;
  }

  // ----------------------------------------------
  //(6) Reference Timing is @ the end of this flat region, corrected for the
  //    derivative delay and a guard interval.
  //---------------------------------------------------------

  idx = strt_idx_diff + flatRegionCornerPosMin;
  psk_uint32 FLG = 0;
  while( DelaySignatureDiff[ idx ] <= ( MeanVal + MaxDev ) )
  {
    idx++;
    if( idx == gridVals )
    {
      break;
      FLG = 1;
    }
  }

  psk_uint32 idxPosFirstGuess = 0;
  if( FLG == 1 )
  {
    idxPosFirstGuess = flatRegionCornerPosMin;
  }
  else
  {
    idxPosFirstGuess = idx - strt_idx_diff + 1;
  }

  psk_uint32 backOff = GuardTimeInterval + 2; // (2..4) (==> slopeDelay: factor
                                              // two due to filter delay
                                              // max(1, floor((num_coef-1)/2)))
  psk_uint32 idxSymbolBegin = idxPosFirstGuess - backOff + strt_idx_diff - 1;

  // extract (re-Index) the sample aligned within input signal
  idxBegin -= (FLAT_REGION_MIN + MAX_MISALIGN) * sps;
  idxBegin += ( idxSymbolBegin );
  *strt_idx = idxBegin;
  *end_idx  = idxBegin + (psk_uint32)PSK_len * sps - 1;

  // clean up memory
  if( DelaySignature )
  {
    free( DelaySignature );
    DelaySignature = NULL;
  }

  if( DelaySignatureDiff )
  {
    free( DelaySignatureDiff );
    DelaySignatureDiff = NULL;
  }
  if( y_ref_cut )
  {
    free( y_ref_cut );
    y_ref_cut = NULL;
  }
  if( y_ref_cut_conj )
  {
    free( y_ref_cut_conj );
    y_ref_cut_conj = NULL;
  }
  return( PSK_ERR_OK );
}
```

```
//---------------------------------------------------------------------------
psk_int32 psk_find_approx_delay( psk_complex* v_ref_in,
                                 psk_uint32   ref_len,
                                 psk_complex* v_in,
                                 psk_uint32   len,
                                 psk_uint32   max_shift,
                                 psk_uint32*  delay )
{
  psk_uint32 delay_int, idx;
  psk_complex* v_corr;
  psk_int32* v_lag_idx;
  psk_double max_val = 0.0;
  psk_uint32 max_idx = 0;
  psk_int32 err = PSK_ERR_OK;
  psk_complex* v_in_aligned = calloc( ref_len, sizeof(psk_complex) );
  if( v_in_aligned == NULL )
    return( PSK_ERR_OUT_OF_MEM );

  if( len - ref_len < max_shift )
    printf("psk_find_approx_delay: v_in signal too short!");

  err = psk_cross_covariance( v_ref_in,
                              ref_len,
                              v_in,
                              len,
                              max_shift,
                              &v_corr,
                              &v_lag_idx );

  if (err != PSK_ERR_OK )
    return( err );

  psk_double* v_abs_corr = psk_vec_abs( v_corr, max_shift );

  psk_max( v_abs_corr, max_shift, &max_val, &max_idx );

  if( v_lag_idx[max_idx]> 1)
    printf("psk_find_approx_delay: akausal peak detected (peak > 1)");

  delay_int = -1 * v_lag_idx[max_idx];
  // signal alignment
  if( delay_int > 1 )
  {
    for( idx = delay_int; idx < delay_int+ref_len; idx++ )
    {
      RE( v_in_aligned[idx-delay_int] ) = RE( v_in[idx] );
      IM( v_in_aligned[idx-delay_int] ) = IM( v_in[idx] );
    }
  }

  // perform quality checks
  psk_complex *ref_corr_0;
  psk_int32 *ref_lag_0;
  psk_complex *corr_0;
  psk_int32 *lag_0;
  psk_double peak_ref_0, peak_0;

  err = psk_cross_covariance( v_ref_in,
                              ref_len,
                              v_ref_in,
                              ref_len,
                              0,
                              &ref_corr_0,
                              &ref_lag_0 );
  peak_ref_0 = psk_abs( ref_corr_0[0] );

  err = psk_cross_covariance( v_in_aligned,
                              ref_len,
                              v_in_aligned,
                              ref_len,
                              0,
                            &corr_0,
                            &lag_0 );
  peak_0 = psk_abs( corr_0[0] );

  if( max_val < 0.25 * sqrt( peak_ref_0 * peak_0) )
    printf("WARNING: psk_find_approx_delay -- x-corr peak low! \
            Signals do not match well.\n");

  // clean up
  if( v_abs_corr )
    free( v_abs_corr );
  if( v_in_aligned )
  free( v_in_aligned );
  if( v_corr )
    free( v_corr );
  if( v_lag_idx )
    free( v_lag_idx );

  *delay = delay_int;
  return( PSK_ERR_OK );
}


/*****************************************************************************/
/*** psk_analysis.c                                                       ***/
/***  DESCIRPTION:                                                        ***/
/***    Main file of the VHBR wave shape analysis tool                   ***/
/*****************************************************************************/

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

#include "psk_types.h"
#include "psk_defines.h"
#include "psk_math.h"
#include "psk_dsp.h"

/*****************************************************************************/
/* function predeclaration */

/*****************************************************************************/
/** readcsv
 * Read a ascii coded file of type csv. First column time, second column
 * amplitude.
 * @param in_filename Name of input file. Not longer than 256 characters
 * @param samples Array of amplitude values from input file
 * @param len Number of amplitude and time values
 * @param times Array of time values from input file
 * @return error code
 */
psk_int32 readcsv( char* in_filename,    /*[in]*/
                   psk_double** samples, /*[out]*/
                   psk_uint32* len,      /*[out]*/
                   psk_double** times);  /*[out]*/

psk_uint32 ORDER = 0;
psk_double BIT_RATE = 0.0;
psk_uint32 PR= 0;
psk_uint32 EPI = 0;
psk_double ETU = 0.0;

/*****************************************************************************/
/** main
 * ISO/IEC 10373-6 VHBR PSK ANALYSIS TOOL
 * @param argc 2 parameters are expected
 * @param argv[1] expected to be a csv file
 * @param argv[2] expected to be a bit rate code [0=3fc/4, 1=fc, 2=3fc/2, 3=2fc]
 * @return error code as defined in psk_defines.h
 */
```

```
int main( int argc, char *argv[] )
{
  char in_filename[256];
  psk_uint32 samples_len = 0; // number of samples in input file
  psk_uint32 bitrate_code = -1;


  psk_double* samples = NULL;
  psk_double* times   = NULL;

  int status = PSK_ERR_OK;

  //
  // READ INPUT
  //
  printf("\n**********************************************");
  printf("\n****                                      ****");
  printf("\n**** ISO/IEC 10373-6 VHBR PSK ANALYSIS TOOL ****");
  printf("\n**** Version: 1.0 August 2013             ****");
  printf("\n****                                      ****");
  printf("\n**********************************************\n");

  if( argc == 1 )
  {
    printf("USAGE: \n");
    printf("1st parameter: file name, *.csv \n(1st column time vector, 2nd column amplitude
vector)\n\n");
    printf("2nd parameter: bit rate code [0=3fc/4, 1=fc, 2=3fc/2 or 3=2fc]\n\n");

    printf( "\nCSV File name :" );
    scanf( "%s", in_filename );

    printf( "\nBit rate code [0=3fc/4, 1=fc, 2=3fc/2 or 3=2fc] :" );
    scanf( "%d", &bitrate_code );
  }
  else
  {
    strcpy( in_filename, argv[1] );
    if( !strchr( in_filename, '.' ) )
      strcat( in_filename, ".csv" );

    if( argc > 2 )
    {
      char *br_code = argv[2];
      if( br_code[0] < '0' || br_code[0] > '3' )
      {
        printf( "\nERROR: valid bit rate codes are: [0=3fc/4, 1=fc, 2=3fc/2 or 3=2fc]\n"
);
        return( PSK_ERR_PARAMETER );
      }
      else
        bitrate_code = atoi( br_code );
    }
    else
    {
      printf( "\nBit rate code [0=3fc/4, 1=fc, 2=3fc/2 or 3=2fc] :" );
      scanf( "%d", &bitrate_code );
      if( bitrate_code < 0 || bitrate_code > 3 )
      {
        printf( "\nERROR: valid bit rate codes are: [0=3fc/4, 1=fc, 2=3fc/2 or 3=2fc]\n"
);
        return( PSK_ERR_PARAMETER );
      }
    }
  }

  if( !strchr( in_filename, '.' ) )
    strcat( in_filename, ".csv" );

  status = readcsv( in_filename,
                    &samples,
                        &samples_len,
                        &times);
  if( status != PSK_ERR_OK )
    return( status );


  if ( samples_len < MIN_NUM_SAMPLES )
    return( PSK_ERR_PARAMETER );

  // BIT RATE DEPENDENT PARAMETER
  switch( bitrate_code )
  {
  case 0:
    {
    ETU = 4.0 / (psk_double)FC;
    ORDER = 8;
    BIT_RATE = 3.0 * (psk_double)FC / 4.0;
    PR = 56; // degree
    EPI = 8; // degree
    break;
    }
  case 1:
    {
    ETU = 4.0 / (psk_double)FC;
    ORDER = 16;
    BIT_RATE = (psk_double)FC;
    PR = 60;
    EPI = 4;
    break;
    }
  case 2:
    {
    ETU = 2.0 / (psk_double)FC;
    ORDER = 8;
    BIT_RATE = 3.0 * (psk_double)FC / 2.0;
    PR = 56;
    EPI = 8;
    break;
    }
  case 3:
    {
    ETU = 2.0 / (psk_double)FC;
    ORDER = 16;
    BIT_RATE = 2.0*(psk_double)FC;
    PR = 60;
    EPI = 4;
    break;
    }
  default:
    {
      printf( "ERROR: No valid bit rate selected: %d\n", bitrate_code );
      printf( "\nBit rate code must be one of the following: [0=3fc/4, 1=fc, 2=3fc/2 or
3=2fc]. \n");
      return(PSK_ERR_PARAMETER);
    }
  }
  printf("Bit rate for evaluation is: %2.2f Mbps\n", BIT_RATE*1.0e-6 );


  //
  // PREPROCESSING AND CONDITIONING
  // STEP 1 - ANTIALIASING FILTER
  //
  psk_double* samples_aa = NULL;
  psk_uint32 aa_len = 0;
 // psk_double fs_osc = (psk_double)samples_len /
  //                      ( times[samples_len - 1] - times[0] );
  psk_double fs_osc = 1.0 /( times[1] - times[0] );

  if( FS_INT > fs_osc )
```

```
{
  printf( "Target sampling rate must" );
  printf( " be equal to or smaller than initial sampling rate. " );
  printf( "Minimum initial sampling rate requirement: 500MSps!\n" );
  return( PSK_ERR_INVALID_SAMPLE_RATE );
}
status = psk_antialiasing( samples,
                           samples_len,
                           &samples_aa,
                           &aa_len,
                           fs_osc );
if ( status != PSK_ERR_OK )
  return( status );

//
// PREPROCESSING AND CONDITIONING
// STEP 2 - RESAMPLE
//
psk_uint32 len_int = 0; // length of resampled signal (integer number of fc)
psk_double* times_int = NULL;
psk_double* resampled_sig = NULL;
status = psk_downsample( samples_aa,
                         times,
                         aa_len,
                         &resampled_sig,
                         &times_int,
                         &len_int );
if ( status != PSK_ERR_OK )
  return( status );

//
// PREPROCESSING AND CONDITIONING
// STEP 3 - DEMODULATE
//
psk_complex* sig_bb = NULL;
psk_demodulation( resampled_sig, times_int, len_int, &sig_bb );

// clean up unused variables and arrays
if( samples )
{
  free( samples );
  samples = NULL;
}
if( times )
{
  free( times );
  times = NULL;
}
if( samples_aa )
{
  free( samples_aa );
  samples_aa = NULL;
}
if( resampled_sig )
{
  free( resampled_sig );
  resampled_sig = NULL;
}

//
// PREPROCESSING AND CONDITIONING
// STEP 4 - DEROTATION
//
psk_complex* sig_bb_derot = NULL;
status = psk_derotation( sig_bb, times_int, len_int, &sig_bb_derot );

//
// PREPROCESSING AND CONDITIONING
// STEP 5 - LP-FILTER
//
psk_uint32 i;
```

```
psk_uint32 ma_filt_len = (FS_INT/FC)/2;
psk_uint32 len_out = 0;

psk_complex* sig_bb_lp = calloc( len_out, sizeof(psk_complex) );

status = psk_ma_filter_cmpl( sig_bb_derot,
                             len_int,
                             &sig_bb_lp,
                             &len_out,
                             ma_filt_len );
if( status != PSK_ERR_OK )
  return( status );

//
// PREPROCESSING AND CONDITIONING
// STEP 6 - SILENCE RE-ADJUSTMENT
//
psk_complex mean = psk_cmpl_mean( sig_bb_lp, IDX_UNMOD );
psk_double phi_err = atan2( IM( mean ), RE( mean ) );
psk_complex exp_phi_err;
RE( exp_phi_err ) = cos( phi_err );
IM( exp_phi_err ) = -sin( phi_err );

for( i = 0; i < len_out; i++ )
  sig_bb_lp[i] = psk_cmpl_mult( sig_bb_lp[i], exp_phi_err );

// clean up
if( sig_bb )
  free( sig_bb );
sig_bb = NULL;

//
// SYMBOL GRID ALIGNMENT
// STEP 7 -
//
psk_uint32 strt_idx = 0;
psk_uint32 end_idx = 0;

status = psk_re_align_symbol_grid( sig_bb_lp, len_out,&strt_idx,&end_idx);

if( status != PSK_ERR_OK)
  return( status );

psk_uint32 sig_len_cutout = end_idx-strt_idx+1;
psk_complex* sig_bb_aligned = calloc( sig_len_cutout, sizeof( psk_complex ) );

for( i = strt_idx; i <= end_idx; i++ )
  sig_bb_aligned[i - strt_idx] = sig_bb_lp[ i ];


//
// ISI COMPUTATION
// STEP 8 -
//
psk_double isi_m = 0.0;
psk_double isi_d = 0.0;
psk_double phase_range = 0.0;
psk_complex* sig_out;
status = psk_isi_param( sig_bb_aligned,
                        sig_len_cutout,
                         &isi_m,
                         &isi_d,
                         &phase_range,
                         &sig_out );
if( status != PSK_ERR_OK )
  return( status );

//
// PHASE NOISE COMPUTATION
// STEP 9 -
//
```

```
  psk_double phase_noise = 0.0;
  status = psk_get_phase_noise( sig_bb_lp, strt_idx, &phase_noise);
  if( status != PSK_ERR_OK )
    return( status );


  printf("\n---------------------- STATISTICS --------------------------\n\n");
  printf("Input Sampling Rate: %d MHz\n", (psk_uint32)(fs_osc*1.0e-6 + 0.5) );
  printf("Number of Samples: %d \n\n", samples_len );

  printf("---------------------- RESULT --------------------------\n\n");
  printf("ISIm: %2.2E\n", isi_m);
  printf("ISId: %3.2f degrees\n", isi_d);
  printf("Phase range: %3.2f degrees\n", phase_range);
  printf("Normalized differential phase noise: %2.3E\n", phase_noise);

  return( status );
}

//------------------------------------------------------------------------------
psk_int32 readcsv( char* in_filename,
                   psk_double** samples,
                   psk_uint32* len,
                   psk_double** times)
{
  psk_int32 num_samples = 1;
  psk_double* vt = calloc( num_samples, sizeof(psk_double) );
  psk_double* va = calloc( num_samples, sizeof(psk_double) );
  FILE *sample_file;

  // open input file
  if( !strchr( in_filename, '.' ) ) strcat( in_filename, ".csv" );
  if( ( sample_file = fopen( in_filename, "r" ) ) == NULL )
  {
    fprintf( stderr, "Cannot open input file %s.\n", in_filename );
    return( PSK_ERR_READ_FILE );
  }

  // read input values into array
  while( !feof( sample_file ) )
  {
    vt = (psk_double*)realloc( vt, sizeof(psk_double) * ( num_samples ) );
    va = (psk_double*)realloc( va, sizeof(psk_double) * ( num_samples ) );
    if( num_samples >= MAX_NUM_SAMPLES )
    {
      fprintf(stderr, "Too many samples in input file: only %d samples read\n",
        num_samples );
      break;
    }

    fscanf( sample_file, "%lf,%lf\n", &vt[num_samples-1], &va[num_samples-1] );
    num_samples++;
  }

  if( sample_file )
    fclose( sample_file );

  *len = num_samples - 1;
  *times = vt;
  *samples = va;

  return( PSK_ERR_OK );
}
```

J.3.13 Example test report (informative)

Below, an example test report for one test position in the operating volume and the bit rate of *fc* is shown.

EXAMPLE 1 PASS case

```
PCD transmission test
------------
General Setting:
Bit Rate: 13,56 Mbit/s
Position (x,y,z):  (0,00 mm, 0,00 mm, 37,50 mm)
------------
Setup Description [optionally provide additional information]:

------------
Measurement Results
Phase range (PR)                       = 60,23 °

ISI Measures
ISI magnitude (ISIm)                   = 0,46
ISI rotation (ISId)                    = -57,61 °

Phase Noise Measure
Normalized differential phase noise    = 0,0298
------------
PASS-FAIL summary report
Phase range (PR)                       : PASS
ISI magnitude (ISIm)                   : PASS
Normalized differential phase noise    : PASS
------------
Overall test result                    : PASS
```

EXAMPLE 2 FAIL case (due to noise)

```
PCD transmission test
------------
General Setting
Bit Rate: 13,56 Mbit/s
Position (x,y,z):  (0,00 mm, 0,00 mm, 37,50 mm)
------------
Setup Description [optionally provide additional information]:

------------
Measurement Results
Phase range (PR) (measured)            = 60,99 °

ISI Measures
ISI magnitude (ISIm)                   = 0,46
ISI rotation (ISId)                    = -58,05 °

Phase Noise Measure
Normalized differential phase noise    = 0,0785
------------
PASS-FAIL summary report
Phase range (PR)                       : PASS
ISI magnitude (ISIm)                   : PASS
Normalized differential phase noise    : FAIL
------------
Overall test result                    : FAIL
```

EXAMPLE 3 FAIL case (due to ISI$_m$)

```
PCD transmission test
------------
General Setting
Bit Rate: 13,56 Mbit/s
Position (x,y,z):  (0,00 mm, 0,00 mm, 37,50 mm)
------------
Setup Description [optionally provide additional information]:

------------
Measurement Results
Phase range (PR)                      = 60,22 °

ISI Measures
ISI magnitude (ISIm)                  = 1,70
ISI rotation (ISId)                   = -50,03 °

Phase Noise Measure
```

```
Normalized differential phase noise   = 0,0001
------------
PASS-FAIL summary report
Phase range (PR)                      : PASS
ISI magnitude (ISIm)                  : FAIL
Normalized differential phase noise   : PASS
------------
Overall test result                   : FAIL
```

J.4 PCD signal creation for PICC reception tests (informative)

J.4.1 Introduction

The following subclauses describe how to create test signals for PICC reception tests as required for conditions 1 to 4 of J.2.2.1.2. Test signals are digitally pre-conditioned before transmission.

J.4.2 ISI_m and ISI_d test signal creation

The test signals are created using the baseband model of the test PCD antenna with impedance matching network for bit rates higher than *fc*/128. This baseband model can be used to derive a transfer function H_{bb} which describes the physical antenna resonator. The test signal is created by the following steps:

a) the sequence of NPVs in its complex representation is filtered by H_{bb};

b) the carrier signal is phase modulated by the filtered NPVs.

The resulting digital signal is equivalent to the signal observed at the air interface. Therefore, the transmission of this signal would result in slightly different ISI_m and ISI_d parameters due to the additional filter-effect of the test PCD antenna used for transmission.

This signal description does not take into account the additional filter-effect of the test PCD antenna.

A time-discrete baseband filter that creates the desired inter-symbol interference signal (with given ISI_m and ISI_d) is described in the z-domain by:

$$H_{bb}(z) = (1-p) \Big/ \left(1 - p \cdot z^{-1}\right),$$

with p the complex pole.

The complex pole position p can be computed for desired ISI_m and ISI_d parameters by:

$$p = \left(\frac{\frac{1}{2}\sin\left(ISI_m \cdot EPI\right) \cdot \exp\left(j \cdot ISI_d\right)}{\sin\left(\frac{1}{2} \cdot PR\right)} \right)^{\left(\frac{T_{sr}}{etu - 1/fc}\right)},$$

where j is the imaginary unit and T_{sr} is the sample duration (the inverse of the sample rate). This equation is only valid if $T_{sr} \leq 1/fc$.

J.4.3 Normalized differential phase noise test signal creation

The defined test signal is created by adding a low-pass filtered pseudo random white noise to the sequence of NPVs in its complex representation. A second order, Butterworth type low-pass filter with 3-dB cut-off frequency of 100 kHz is used to filter frequency components above the cut-off frequency. The filter characteristic is illustrated in Figure J.7.