24. Phase measurement

When one is interested in the phase relation between two signals REF and X, and these signals are not generated by the measurement system, one can use a derivative of the lock-in detection technique already described in chapter 23. In previous chapter the microcontroller generated the signal REF and is was simple to generate also the quadrature version of this signal (the 90-degree shifted version). Now, such signal must be derived from the REF signal using the Hilbert transform. The rest of the system remains the same.

24.1. The theory

The block diagram of the system to determine the phase φ between two harmonic signals is given in Fig. 24.1. Two harmonic signals *XR* and *XP* are available and the task of the measurement system is to determine the phase between them; both signals have the same frequency ω and adequate amplitudes A_R and A_P to be sampled by the ADC in the microcontroller.

$$X_R = A_R \sin \omega t \qquad X_P = A_P \sin(\omega t + \varphi)$$

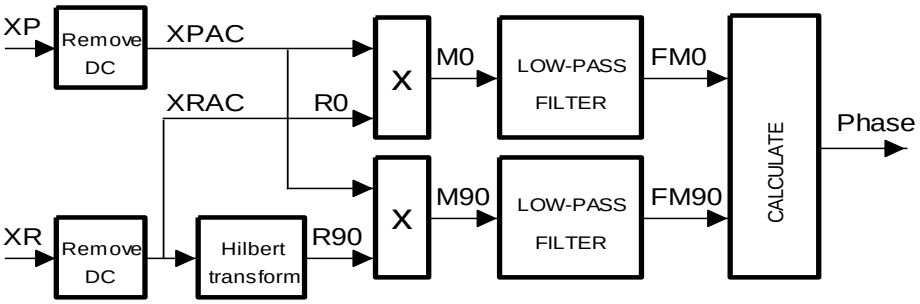


Figure 24.1: The phase measurement: block diagram

The block diagram introduces three new blocks compared to the diagram given in chapter 23, these are the "Remove DC" and the "Hilbert transform" blocks.

The ADC samples the input signals, and the result is between 0 and 4095 for 12 bit ADC; it is always a positive number. The sampled signal therefore rides on a steady DC component of roughly 2048, depending on the DC offset of the ADC. In this example we will strip the DC component off from the sampled signal by a simple DC-removal filter (the "Remove DC" blocks) to demonstrate the process. This operation is not mandatory or essential for the determination of the phase angle, we will simply introduce it here as it looks like a reasonable thing to do in many data processing applications.

Intuitively, the differential equation to strip the DC component off can be written simply as:

$$y_k = x_k - x_{k-1}$$

The y_k is the current output sample, and the x_k and x_{k-1} are the current input and previous input samples. The AC component is proportional to the change in the input value; with the fixed time

interval between samples the above equation returns an approximation of the derivative of the input signal which, of course, is the weighted AC component.

The same difference equation can be derived using knowledge about the z-transform and the influence of poles and zeroes in the z-plane. If we place a zero at the position where $\omega = 0$ in the z-plane, then the system implementing the corresponding transfer function will not let the DC component pass. The position of the zero is illustrated in Fig. 24.2. From here we can write the transfer function *H(z)* and the corresponding difference equation as (note a pole is added in the center of the z-plane; this does not affect the frequency characteristics):

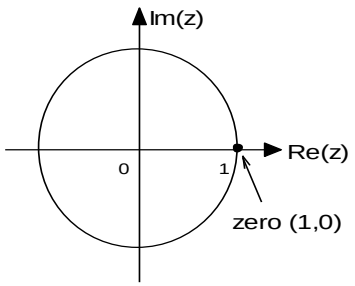
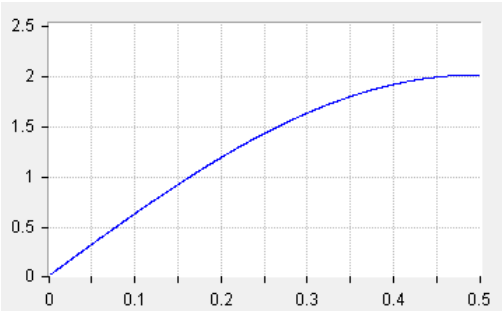


Figure 24.2: The position of a zero in z-plane for a simple DC-removal filter

$$H(z) = \frac{z-1}{z} = \frac{1-z^{-1}}{1} \qquad \rightarrow \qquad y_k = x_k - x_{k-1}$$

The output of this DC-removal filter depends strongly on the frequency of the incoming signal, the amplitude and phase plots are given in Fig. 24.3, and can be calculated using:

$$AMPLITUDE = \left| H(z)|_{z=e^{i\omega t}} \right| \qquad \text{and} \qquad PHASE = \tan^{-1} \frac{IM(H(z))}{RE(H(z))} \Bigg|_{z=e^{i\omega t}}$$



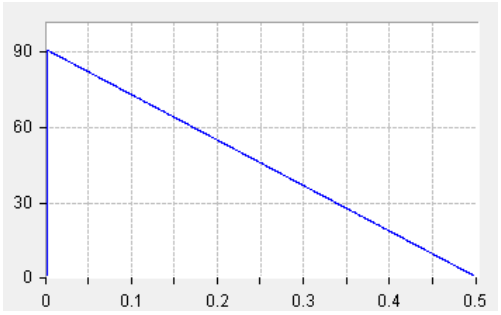


Figure 24.3: Amplitude and phase characteristics of a simple DC-removal filter; horizontal: normalized frequency f/fs, vertical: gain / phase (degrees)

In our experiment the phase characteristic of the filter has no influence on the measurement since equal filters are inserted in both X_R and X_P signals and affect both equally. The reduction of the amplitude at low frequencies might be more of a problem, and can be avoided by modifying a pole-zero diagram. When a pole is inserted close to a zero but inside the unity circle (Fig. 24.4) at a distance *r* from the center the transfer function and the difference equation change to:

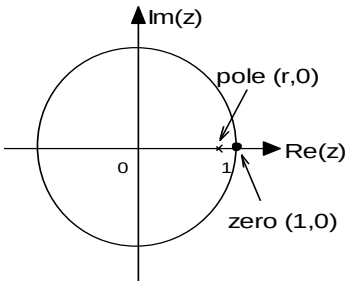


Figure 24.4: The position of a zero and a pole in z-plane for a better DC-removal filter

$$H(z) = \frac{z-1}{z-r} = \frac{1-z^{-1}}{1-rz^{-1}} \qquad \rightarrow \qquad y_k = ry_{k-1} + x_k - x_{k-1}$$

This changes the amplitude and phase plots to the ones given in Fig. 24.5. Note the gain characteristics which is mostly flat now. Both versions of difference equations are simple to implement. In this case the first, the simpler version of differential equation, suffices and will be implemented in software.

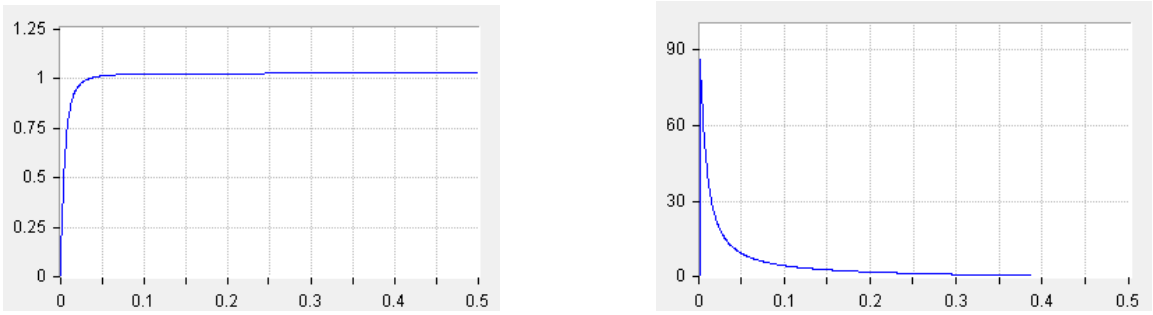


Figure 24.5: Amplitude and phase characteristics of a better DC-removal filter; horizontal: normalized frequency f/f_s, vertical: gain / phase (degrees)

The implementation of the Hilbert transform in a FIR filter was discussed in chapter 19. It was stated that the coefficients of the filter core for a Hilbert transform are given as:

$$h_m = \frac{-1 + (-1)^m}{\pi m}$$

Due to the limited time available between two neighboring samples to calculate the convolution for FIR filter, the number of coefficients in filter core is always restricted, and this affects the frequency characteristics of such a filter; the characteristics can be determined by calculating a Fourier transform of the filter core, and the result for $|m| \leq 32$ is given in Fig. 24.6, blue line. The characteristics is not ideal:

- the amplitude of the filter response changes a lot with the frequency, and
- the amplitude of the response approaches zero at DC and at 0.5 of the sampling frequency.

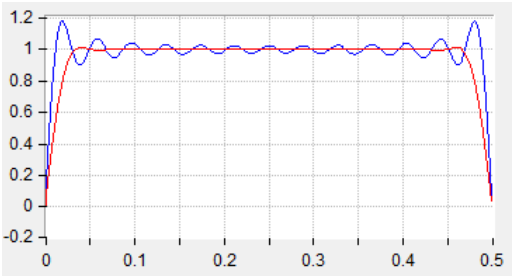


Figure 24.6: Amplitude characteristics for a Hilbert transform with m=32; vertical: normalized amplitude, horizontal: normalized frequency f/f_s

The first obstacle can be significantly reduced by implementing a windowing function, as already mentioned in chapter 19; the effect of windowing is shown by the red line in the same figure.

The second obstacle can be reduced by increasing the number of coefficients in the filter core (you do not want to do this since this increases the time to calculate the convolution) or by carefully selecting the sampling frequency. Suppose the sampling frequency is four times the frequency of the input signals, then we are working at $0.25 \cdot f_s$, and the amplitude characteristics (the red one, windowed coefficients) from Fig. 24.6 is flat in the region. Actually, there is a broad range of frequencies we can use to sample the incoming signal; we must only stay within the flat region of the amplitude characteristics (red) from Fig. 24.6. We could even reduce the number of coefficients in the filter core for the Hilbert transform, get flat response of the filter in smaller central region, and select the sampling frequency to work within this flat region.

It would not be wise to select the sampling frequency that is exactly four times the frequency of the input signals though. The result of multiplication of two signals with the same frequency is composed of two components: one is a DC signal, the other is a signal with a double frequency; $f_s/2$ in this case. According to the Nyquist criteria a signal with frequency of $f_s/2$ behaves like a DC signal, and this comes straight through the low-pass filter! This is not what we want! The same is true also for input signals with frequencies between 0.25 and 0.30 times the sampling frequency; the product contains a component that comes through the low-pass filter and corrupts the measurement.

We have learned that the sampling frequency should be matched to the frequency of the input signal; not precisely, a wide range of sampling frequencies is acceptable for this experiment, but there are also combinations to be avoided.

The rest of the theory was already described in chapter 23 and will not be repeated here.

24.2. The implementation of the lock-in detection

The phase measurement as described above will be tested using an external sine-wave generator and a simple RC passive network to shift the sine-wave. The schematic diagram is given in Fig. 24.7. The signal from the function generator, being a sine-wave, is connected to the network and also to the

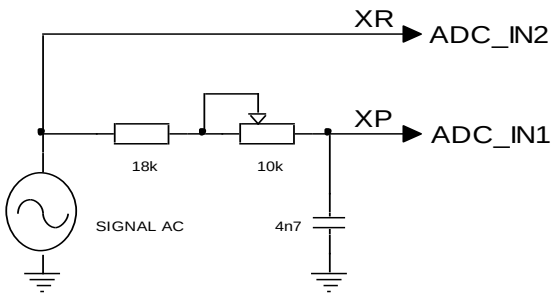


Figure 24.7: The external circuit to the BaseBoard for the phase measurement experiment

ADC input ADC_IN2 at the microcontroller. The values of components in the network are calculated to give a phase shift of about 45 degrees for sinusoidal signals with a frequency of 1500 Hz. The phase shift can be altered by the rotation of the potentiometer.

The program is a modified version of the program from the previous chapter. It is implemented as an interrupt routine, which is driven by timer TIM5, the time interval between successive interrupts is fixed to 100 μs, giving the sampling frequency which is about 6 times the frequency of the input signal.

The sampling of the input signal is implemented the same way as described in chapters 12 and 13. The timer overflow triggers the ADC to start the conversion, and the end-of-conversion signal from the ADC is used to interrupt the microcontroller and start the interrupt routine ADC_IRQhandler.

As in the previous chapter the calculation of a complex function (the arc tangent) is not implemented within the interrupt routine. Instead, the consecutive results are accumulated and then the sum of 128 consecutive results is passed to the main program, where the calculation is performed.

The filter implemented could be the same as used in the previous chapter, but here the IIR version is used, as described in chapter 20. The same filter coefficients are used.

The ADC interrupt function for the lock-in detection is presented in the listing below.

```
void ADC_IRQHandler(void)       {
  GPIOE->BSRRL = BIT_8;                  // signal start, execution time is about 6 us !       // 2

  XR[k] = ADC1->DR;                      // to clear ADC IRQ flag                              // 4
  XP[k] = (ADC2->DR & 0xfff);            // acquire                                            // 5
  XRAC[k] = XR[k] - XR[(k-1) & 127];    // remove DC component                                // 6
  XPAC[k] = XP[k] - XP[(k-1) & 127];    // remove DC component                                // 7

  // derive cos component using Hilbert transform
  float conv = (float)XRAC[(k - 32) & 127] * w[0];                                           // 10
  for (short m = 1; m < 32; m += 2)                                                          // 11
    conv += w[m] *(XRAC[(k - 32 + m) & 127] - XRAC[(k - 32 - m) & 127]);                     // 12
  R0[k] = (float)XRAC[(k - 32) & 127];                                                       // 13
  R90[k] = conv;                                                                             // 14
```

```
  M0[k]  = (float)XPAC[(k - 32) & 127] * R0[k];    // multiply with sin                    // 16
  M90[k] = (float)XPAC[(k - 32) & 127] * R90[k];   // multiply with cos                    // 17

  // IIR low-pass filter for sine component
  conv = 0;                                                                               // 20
  for (short m = 0; m<5; m++)   conv += a[m] * M0[(k - m) & 127];                         // 21
  for (short n = 1; n<5; n++)   conv += b[n] * FM0[(k - n) & 127];                        // 22
  FM0[k] = conv;       FM0acc += conv;                                                    // 23

  // IIR low-pass filter for cosine component
  conv = 0;                                                                               // 26
  for (short m = 0; m<5; m++)   conv += a[m] * M90[(k - m) & 127];                        // 27
  for (short n = 1; n<5; n++)   conv += b[n] * FM90[(k - n) & 127];                       // 28
  FM90[k] = conv;      FM90acc += conv;                                                   // 29

  // average and report result every 128-th time
  if (k == 0)   {                                                                         // 32
    FM0ave  = FM0acc;  FM0acc  = 0;                                                       // 33
    FM90ave = FM90acc; FM90acc = 0;                                                       // 34
    PhaseDisplay = 1;                                                                     // 35
  };                                                                                      // 36

  k++; k &= 127;                                                                          // 38
  GPIOE->BSRRH = BIT_8;               // signal end                                       // 39
}
```

The function commences with setting pin 8, port E, high to signal the start of execution, line 2; the same pin is reset when the function ends, line 39.

Next the result is read from the two ADCs in lines 4 and 5. This returns signals *XR* and *XP*, which are placed into two circular buffers at position *k*. The position pointer *k* increments on every execution of the interrupt routine, line 38. This circular buffer has 128 elements.

The AC-removal filter is implemented in lines 6 and 7 for each signal separately. A simpler version of the filter is used, and results are stored in two circular buffers *XRAC* and *XPAC*.

Next the 90-degree delayed signal is derived out of the signal *XRAC*. The derivation is based on the Hilbert transform filter. The convolution is calculated in lines 10 to 12, and the original version of the signal *XRAC* and its 90 degree delayed version are then stored in circular buffers named R0 and R90 in lines 13 and 14.

The multiplication of the phase shifted signal *XPAC* with the reference is given in lines 16 and 17 to obtain signals *M0* and *M90*. These are then filtered using the IIR version of the filter in lines 20 to 23 and 26 to 29. Finally, lines 32 to 36 implement the passing of the accumulated results to the main part of the program.

The listing of the rest of the program is given below.

```
short int XR[128], XP[128], XRAC[128], XPAC[128], k = 0;
float w[32];
float R0[128], R90[128], M0[128], M90[128], FM0[128], FM90[128];
float FM0ave, FM90ave, FM0acc, FM90acc;
short PhaseDisplay = 0;

// declare and init IIR weights: 4th order, Chebishew, Low Pass, reference [1]
// +/- 0.5%, -3dB at 0.025 (250Hz) of sampling frequency (10kHz)
float a[5] = {1.504626e-5, 6.018503e-5, 9.027754e-5, 6.018503e-5, 1.504626e-5};
float b[5] = {0          , 3.725385e0, -5.226004e0 , 3.270902e0, -7.705239e-1};
```

```
int main ()  {                                                                          // 12

  // calculate Hilbert core for FIR filter
  w[0] = 0;                                    // central weight: 2 x fc / fs         // 15
  for (short m = 1; m < 32; m++)                                                       // 16
    w[m] = (-1.0 + cos(pi * m)) / (pi * m);    // other weights                       // 17
  for (short m = 1; m < 32; m++)                                                       // 18
    w[m] = w[m] * cos(pi / 2 * m / 31.0);      // windowing                           // 19

  LCD_init ();         LCD_string("Phase=", 0x40);                                     // 21

  GPIOEinit ();        ADCinit_T5_CC1_IRQ();          DACinit();                       // 23
  TIM5init_TimeBase_CC1(8400);          // 8400 == 100us == 10kHz                     // 24

  while (1) {                           // endless loop                              // 26
    if (PhaseDisplay) {                                                               // 27
      PhaseDisplay = 0;                                                               // 28
      float Phase = (atan (FM90ave / FM0ave)) / 3.14159 * 180;                        // 29
      if (Phase < 0)    {LCD_string("-", 0x47);    Phase = -Phase; }                  // 30
      else              {LCD_string(" ", 0x47);                    };                 // 31
      short PhaseInt = (int)(Phase);                                                  // 32
      short PhaseFrc = (int)((Phase - PhaseInt) * 10);                                // 33
      LCD_sInt3DG((int)PhaseInt, 0x48, 1);                                            // 34
      LCD_char('.');    LCD_char(PhaseFrc + '0'); LCD_char(0xdf);                     // 35
      for (int j = 0; j<2000000; j++){};      // waste some time                      // 36
    };                                                                                // 37
  };                                                                                  // 38
}                                                                                     // 39
```

This listing starts with the inclusion of CMSIS files (not shown) and the declaration of the variables used. The function “main()” starts in line 12 with the initialization of the filter coefficients for the Hilbert transform; the initialization is copied from Chapter 19, the FIR filtering. The program continues to the configuration functions for LCD, port E, ADC, DAC and timer TIM5; all configuration functions were already described in previous chapters.

The endless while loop, lines 26 to 38, is used to calculate the phase angle between signals *XP* and *XR* from results passed by the ADC interrupt routine, and to display the calculated value at the LCD display. The value of a global variable *PhaseDisplay* signals the availability of a new result. When one, the calculation commences.

The phase is represented by a floating point number. This must be converted to integer and fractional component as in previous chapter, and then displayed. The program is given in lines 29 to 35. Some delay is added in line 36 to reduce the number of screen updates.