MOTOROLA

Order by AN1942/D
(Motorola Order Number)
Rev. 0, 3/02

DSP56F80x Resolver Driver and Hardware Interface

Describes the DSP56F80x Resolver Driver and Hardware Interface and Shows an Example of Driver Use

Martin Mienkina, Pavel Pekarek, Frantisek Dobes

Contents

1. Introduction

Resolver position sensors resemble small motors and are essentially rotary transformers so the coefficient of coupling between rotor and stator varies with shaft angle. A resolver is based on the concept of encoding the shaft angle into sine and co-sine signals.

This Application Note describes a solution for obtaining the estimations of actual angle and speed of the resolver. A theoretical analysis, proposal of the Resolver-to-Digital (R/D) hardware interface and a design of the DSP software driver are mentioned here. The software driver is written in C language using powerful intrinsic functions. It provides estimations of rotor angle and speed to be achieved to 10-bit accuracy at a CPU load of 7.5% (8KHz update rate). Finally, a brief description is given for both an example of driver use and some experimental results. It is demonstrated that the DSP56F80x is suitable for high-end vector control applications utilizing the resolver position sensor.

2. Features of Motorola DSPs

The Motorola DSP56F80x family is well suited for digital motor control and data processing, combining the DSP's calculation capability with MCU's controller features on a single chip. This arises from the high-speed DSP cores of the devices and a number of architectural features. Typically, Motorola DSPs offer many dedicated peripherals, such as Pulse Width Modulation (PWM) units, Analog-to-Digital Converters (ADC), Timers, communication peripherals (SCI, SPI, CAN), on-board Flash and RAM.

Several members of the family are available, including the DSP56F801, DSP56F803, DSP56F805 and DSP56F807, each with various peripheral sets and on-board memory configurations.

A typical member of the family, the DSP56F805, provides the following peripheral blocks:

- 12-bit Analog to Digital Converters (ADCs), supporting two simultaneous conversions with dual 4-pin multiplexed inputs, ADCs can be synchronized by the PWM modules
- Two Pulse Width Modulator modules (PWMA & PWMB), each with six PWM outputs, three Current Sense inputs, and four Fault inputs, fault tolerant design with deadtime insertion, supporting both Center- and Edge- aligned modes
- Two Quadrature Decoders (Quad Dec0 & Quad Dec1), each with four inputs, or two additional Quad Timers A & B
- Two dedicated General Purpose Quad Timers totalling 6 pins: Timer C with 2 pins and Timer D with 4 pins
- A CAN 2.0 A/B Module with 2-pin ports used to transmit and receive
- Two Serial Communication Interfaces (SCI0 & SCI1), each with two pins, or four additional GPIO lines
- A serial Peripheral Interface (SPI), with configurable 4-pin port, or four additional GPIO lines
- A computer Operating Properly (COP) Watchdog timer
- Two dedicated external interrupt pins
- 14 dedicated General Purpose I/O (GPIO) pins, 18 multiplexed GPIO pins
- An external reset pin for hardware reset
- A JTAG/On-Chip Emulation (OnCE)
- A software-programmable, Phase Lock Loop-based frequency synthesizer for the DSP core

Table 2-1. Memory Configuration

	DSP56F801	DSP56F803	DSP56F805	DSP56F807
Program Flash	8188 x 16-bit	32252 x 16-bit	32252 x 16-bit	61436 x 16-bit
Data Flash	2K x 16-bit	4K x 16-bit	4K x 16-bit	8K x 16-bit
Program RAM	1K x 16-bit	512 x 16-bit	512 x 16-bit	2K x 16-bit
Data RAM	1K x 16-bit	2K x 16-bit	2K x 16-bit	4K x 16-bit
Boot Flash	2K x 16-bit	2K x 16-bit	2K x 16-bit	2K x 16-bit

The resolver driver utilizes two ADC channels and one timer of the DSP56F80x. In this particular application, the ADC channels must be configured to sample both sine and co-sine signals simultaneously. The timer provides the generation of the square wave signal. This signal is further conditioned by external hardware to the form, which is convenient for excitation of the resolver. The DSP estimates the actual angle of the rotor shaft on the basis of the measured sine and co-sine signals of the resolver. However, the DSP is not only dedicated to realization of the R/D conversion, hence the software driver of the resolver has to be designed in a way to be able to link and operate within an existing application (e.g. a PMSM vector control application).

The accuracy of the rotor angle and speed estimations greatly depends on features of the ADC. Particularly, ADC accuracy, resolution and set of possible operation modes are crucial for achieving the higher accuracy estimations. To provide a comprehensive description of the R/D conversion and show its real implementation, a brief survey of features of the appropriate ADC is given here.

- 12-bit resolution with a sampling rate up to 1.66 million samples per second
- Maximum ADC Clock frequency of 5MHz with a 200 ns period
- Single conversion time of 8.5 ADC Clock cycles (8.5 x 200 ns = 1.7 us)
- Additional conversion time of 6 ADC clock cycles (6 x 200 ns = 1.2 us)
- Eight conversions in 26.5 ADC Clock cycles (26.5 x 200 ns = 5.3 us) using Simultaneous mode
- ADC can be synchronized to the PWM or via the SYNC signal
- Simultaneous or Sequential sampling
- Internal multiplexer to select two of eight inputs
- Ability to sequentially scan and store up to eight measurements and simultaneously sample and hold two inputs
- Optional interrupts at end of scan, if an out-of-range limit is exceeded, or at zero crossing
- Optional sample correction by subtracting a pre-programmed offset value
- Signed or unsigned result
- Single ended or differential inputs

There are two separate converters available on DSP56F801, DSP56F803 and DSP56F805, each associated with 4 analog inputs and providing two channels to be sampled simultaneously. The DSP56F807 has four separate converters each having 4 inputs and therefore achieving four channels to be sampled simultaneously. The results of conversions are stored for further post-processing in readily accessible registers. The conversion process may be either initiated by the SYNC signal or by setting the START bit of the ADC Control Register (ADCR).

3. Resolvers

Resolvers may be considered as inductive position sensors which have their own rotor and stator windings shifted by 90°.

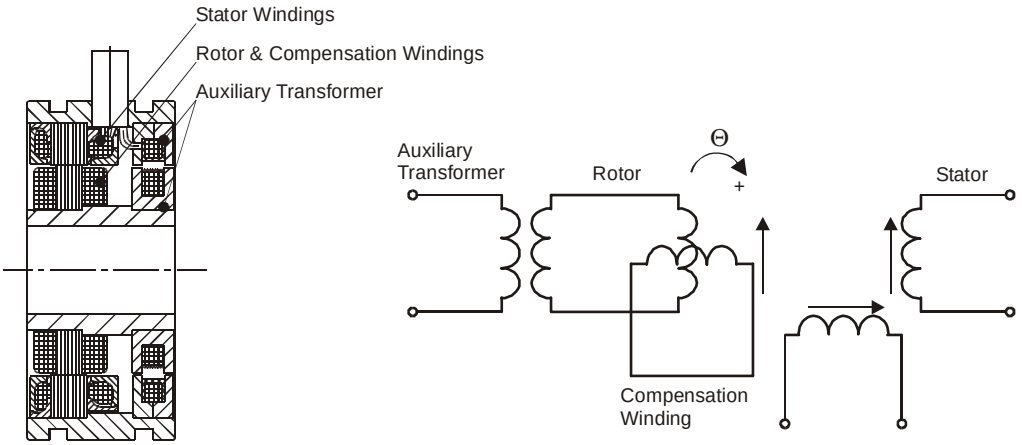

Figure 3-1. Block Scheme of the Hollow Shaft Resolver

The majority of resolvers used nowadays are referred to as *Hollow Shaft Resolvers*. They transfer energy from stator to rotor by means of an auxiliary rotary transformer (see **Figure 3-1**). The resolver rotor is directly mounted on the motor shaft and the resolver stator is fixed to the motor shield. Note that these resolver parts have to be concentrically fastened with the longitudinal axis of the motor shaft. Ordinarily, a concentricity of resolver rotor against stator up to 0.05 mm is needed, otherwise resolver parameters might deteriorate. Thanks to the absence of bearings and brushes the life-cycle of *Hollow Shaft Resolvers* is practically unlimited.

3.1 Resolver Parameters

This section lists the principal resolver parameters, which should be considered seriously during the development stage of the software driver and hardware interface. These parameters are the following:

1. **Electrical Error** - determines the accuracy of the measurement of rotor angle and frequently varies from $\pm 3'$ to $\pm 15'$. Standard *Hollow Shaft Resolvers* have an electrical error of up to $\pm 10'$.
2. **Transformation Ratio** - defines ratio between output and input voltage. This ratio is practically maintained in a range of $0.5 \pm 5\%$, however, it may be set at a wide range, e.g. 0.25-1.0.
3. **Input Voltage, Current and Frequency** - usually from 4 to 30 V_{rms}, from 20 to 100 mA and from 400Hz up to 10 kHz. It should be noted that *Electrical Error* and *Transformation Ratio* are independent parameters of the *Input Voltage* and *Current and Frequency*, at a wide range.
4. **Null Voltage** - denotes the content of disturbance and orthogonal voltage components of the resolver (commonly lower than 20 mV) at zero output voltage.
5. **Phase Shift** - refers to a phase shift between excitation of the resolver and its output signals. Regarding 2-Phase resolvers, it frequently varies within $\pm 10°$. Note that this parameter should be taken into account seriously during hardware and software implementation of the R/D conversion, as will be discussed in **Section 4.1** and **Section 5.**
6. **Stator and Rotor Resistance** - R_S, R_R
7. **Short-Circuit and No-Load Stator and Rotor Impedances** - Z_{SS}, Z_{SO}, Z_{RS}, Z_{RO}

The resolver output voltage amplitudes correspond to sine and co-sine of the rotor angle Θ, as shown in **Figure 3-2**. Note that these voltages can be expressed in terms of the actual rotor angle Θ:

$$U_{S1S3} = K\, U_{R2R4} \sin\Theta \qquad \text{(EQ 3-1)}$$

$$U_{S2S4} = K\, U_{R2R4} \cos\Theta \qquad \text{(EQ 3-2)}$$

where K is the *Transformation Ratio*, and $U_{R2R4} = U_{MAX}\sin(\omega t)$ is an instantaneous excitation voltage applied on the auxiliary transformer winding. Note that the instantaneous excitation voltage U_{R2R4} has an amplitude U_{MAX} and operates at angular frequency $\omega = 2\pi f$. The instantaneous excitation voltage U_{R2R4} will be further referred to as a reference voltage U_{ref}. The output resolver voltages U_{S1S3}, U_{S2S4} will be further referred to as U_{sin}, U_{cos}, respectively.

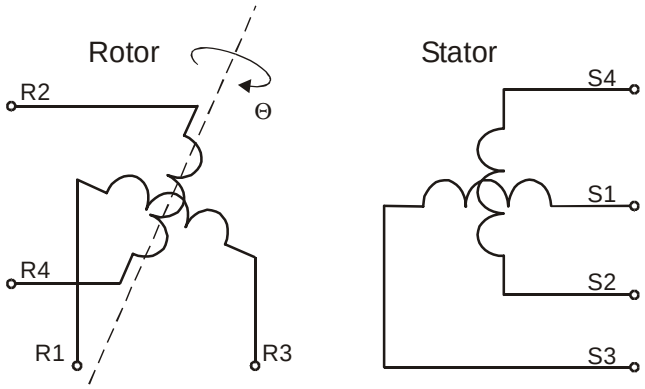

Figure 3-2. Block Scheme of the Resolver

3.2 Theory of Resolver Operation

The resolver is basically a rotary transformer with one rotating reference winding (supplied by U_{ref}) and two stator windings. The reference winding is fixed on the rotor, and therefore, it rotates jointly with the shaft passing the output windings, as is depicted in **Figure 3-3**. Two stator windings are placed in quadrature of one another and generate the sine and co-sine voltages U_{sin}, U_{cos}, respectively. Note that the sine winding is phase advanced by 90° with respect to co-sine winding. Both windings will be further referred to as output windings.

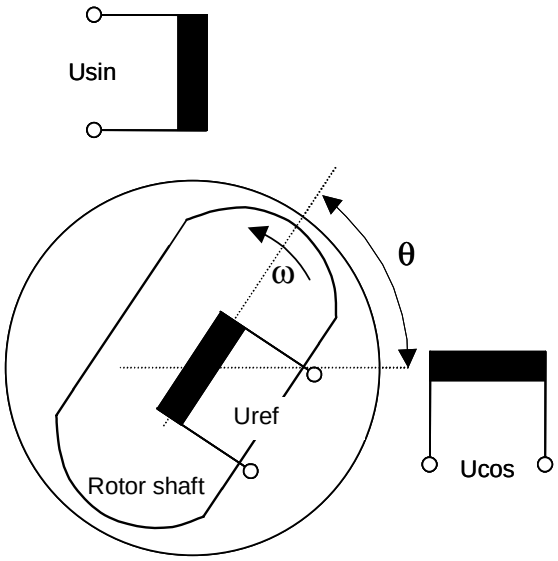

Figure 3-3. Resolver Basics

In consequence of the excitement applied on the reference winding U_{ref} and along with the angular movement of the motor shaft Θ, the respective voltages are generated by resolver output windings U_{sin}, U_{cos} (see **Figure 3-4**).

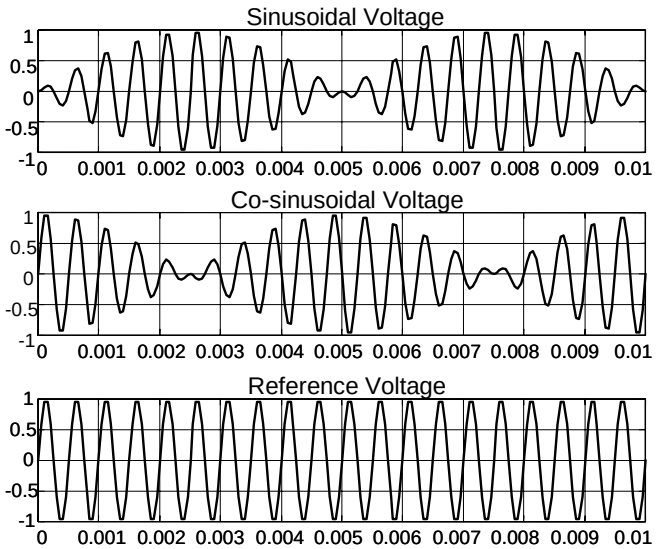

Figure 3-4. Excitation and Output Signal of the Resolver

The frequency of the generated voltages is identical to the reference voltage and their amplitudes vary according to the sine and co-sine of the shaft angle Θ. Considering that one of the output windings is aligned with the reference winding, then it is generated full voltage on that output winding and zero voltage on the other output winding and vice versa. The rotor angle Θ can be extracted from these voltages using a digital approach as will be discussed in the next section.

3.3 Basics of Angle Extraction

Formerly resolvers were used primarily in analog design in conjunction with a *Resolver Transmitter - Resolver Control Transformer* [1]. These systems were frequently employed in servomechanisms, e.g. in aircraft on board instrument systems. Modern systems, however, use the digital approach to extract rotor angle and speed from the resolver output signals. The most common solution is either a *Trigonometric* or *Angle Tracking Observer* method. It should be noted that both methods require fast and high accuracy measurement of the resolver output signals to be carried out.

3.3.1 Trigonometric

The shaft angle can be determined by an *Inverse Tangent* function of the quotient of the sampled resolver output voltages U_{sin}, U_{cos}. This determination can be expressed, in terms of resolver output voltages, as follows:

$$\Theta = \text{atan}\left(\frac{U_{sin}}{U_{cos}}\right) \qquad \text{(EQ 3-3)}$$

An indispensable precondition of the accurate rotor angle estimation is to sample the resolver output signals simultaneously and close to their period peaks (see **Figure 3-5**).

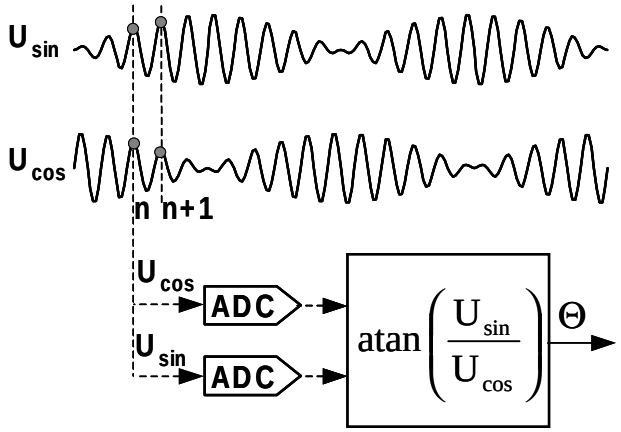

Figure 3-5. Angle Extraction Using the Inverse Tangent Method

Note that modern control algorithms for electric drives require knowledge of the rotor angle and the rotor speed. The *Trigonometric* method, however, only yields values of the unfiltered rotor angle without any speed information. Therefore, for a final application, it is often required that a speed calculation with smoothing capability be added. This drawback might readily be eliminated if a special *Angle Tracking Observer* is utilized. This method is discussed in the next section.

3.3.2 Angle Tracking Observer

The second method (algorithm), widely used for estimation of the rotor angle and speed, is generally known as an *Angle Tracking Observer* (see **Figure 3-6**).

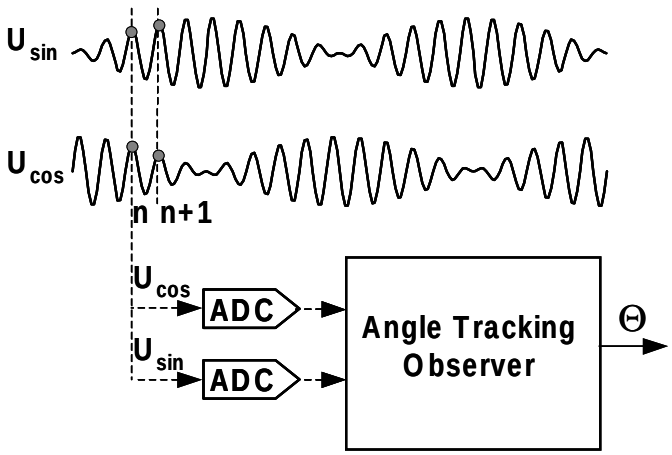

Figure 3-6. Angle Extraction Using the Angle Tracking Observer

A great advantage of the *Angle Tracking Observer* method, compared to the *Trigonometric* method, is that it yields smooth and accurate estimations of both the rotor angle and rotor speed [2].

The *Angle Tracking Observer* compares values of the resolver output signals U_{sin}, U_{cos} with their corresponding estimations $\hat{U}_{sin}$, $\hat{U}_{cos}$. As in any common closed-loop systems, the intent is to minimize observer error. The observer error is given here by subtraction of the estimated resolver rotor angle $\hat{\Theta}$ from the actual rotor angle Θ (see **Figure 3-7**).

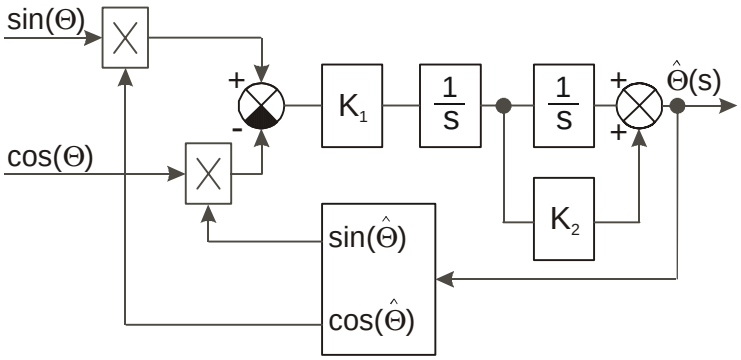

Figure 3-7. Block Scheme of the Angle Tracking Observer

Note that mathematical expression of observer error is known as the formula of the difference of two angles:

$$\sin(\Theta - \hat{\Theta}) = \sin(\Theta)\cos(\hat{\Theta}) - \cos(\Theta)\sin(\hat{\Theta}) \qquad \text{(EQ 3-4)}$$

where $\sin(\Theta - \hat{\Theta})$ denotes observer error, Θ is the actual rotor angle and $\hat{\Theta}$ is its corresponding estimation.

In case of small deviations of the estimated rotor angle compared to the actual rotor angle, the observer error may be considered in the form $\Theta - \hat{\Theta}$.

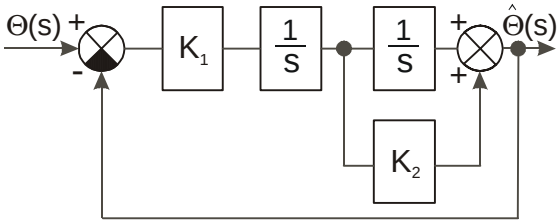

Figure 3-8. Simplified Block Scheme of the Angle Tracking Observer

The main benefit of the *Angle Tracking Observer* utilization, in comparison with the *Trigonometric* method, is its smoothing capability. Smoothing is achieved by the integrator and PI controller, which are connected in series and closed by a unit feedback loop, see the block diagram in **Figure 3-8**. This block diagram nicely tracks actual rotor angle and speed and continuously updates their estimations.

The *Angle Tracking Observer* transfer function is expressed, with the help of its simplified block scheme in **Figure 3-8**, as follows:

$$F(s) = \frac{\hat{\Theta}(s)}{\Theta(s)} = \frac{K_1(1 + K_2 s)}{s^2 + K_1 K_2 s + K_1} \qquad \text{(EQ 3-5)}$$

The characteristic polynomial of the *Angle Tracking Observer* corresponds to the denominator of transfer function **EQ 3-5**:

$$s^2 + K_1 K_2 s + K_1 \quad \text{(EQ 3-6)}$$

Appropriate dynamic behavior of the *Angle Tracking Observer* may be achieved by placement of the poles of the characteristic polynomial. This general method is based on matching the coefficients of the characteristic polynomial with the coefficients of the general second-order system $G(s)$:

$$G(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \quad \text{(EQ 3-7)}$$

where ω_n is the *Natural Frequency* $[\text{rad}\,\text{s}^{-1}]$ and ζ is the *Damping Factor* $[-]$. Once the desired response of the general second-order system $G(s)$ is found, the *Angle Tracking Observer* coefficients K_1, K_2 can be calculated using these expressions:

$$K_1 = \omega_n^2 \quad \text{(EQ 3-8)}$$

$$K_2 = \frac{2\zeta}{\omega_n} \quad \text{(EQ 3-9)}$$

The *Angle Tracking Observer* transfer function is a second order and has one zero. This real-axis zero affects the residue, or amplitude, of a response component but does not affect the nature of the response - exponential damped sine. It can be proven that the closer the zero is to the dominant poles, the greater its effect is on the transient response. As the zero moves away from the dominant poles the transient response approaches that of the two-pole system [3], [4].

The transient responses of the *Angle Tracking Observer* to a $180°$ step change of the actual rotor angle, given by simulation of the transfer function **EQ 3-5**, are discussed thereinafter. These simulations have been carried out for two sets of *Angle Tracking Observer* coefficients. It is shown that both coefficient sets have enabled the *Angle Tracking Observer* to reach a steady state value within a *Settling Time*[1] < 15 ms and with a *Peak Overshoot*[2] < 17 %. It should be noted that such large step changes could never happen in a real application due to the continuous minimization of observer error. In reality, possible step changes are much smaller (compared to $180°$) so a steady state is reached much faster.

3.3.3 Selecting Optimal Observer Coefficients

This application note discusses two coefficient sets. The first set of coefficients was chosen to perform optimal smoothing of the rotor angle and speed estimations. On the contrary, the second set of coefficients was selected to minimize the *Settling Time*. The next paragraph gives an overview of the utilized observer parameters and shows some results of the *Angle Tracking Observer* simulations - all simulations were pursued using Matlab software. The simulation models were solved using "Euler-Forward Integration" with an integration step $62.5\mu s$.

1. The time taken for the response to reach and remain within a specified range of its final value. An allowable tolerance of ±20′ (electrical) has been considered. This tolerance roughly matches an estimation error of ±1 LSB when a 10-bit resolution is used for angle measurement.
2. The amplitude of the first peak normally expressed as a percentage of the final (steady-state) value.

In the first case, the coefficients of the *Angle Tracking Observer* have been originated on the parameters of the general second-order system $\omega_n = 500\,\text{rad}\,\text{s}^{-1}$, $\zeta = 0.4, 0.84, 1.2$ and 1.6. On the basis of these parameters, the coefficients of the *Angle Tracking Observer* $K_1 = 250000$, $K_2 = 0.0016, 0.00336, 0.0048$ and 0.0064 were calculated using **EQ 3-8** and **EQ 3-9**. The corresponding simulation results are shown in **Figure 3-9**.

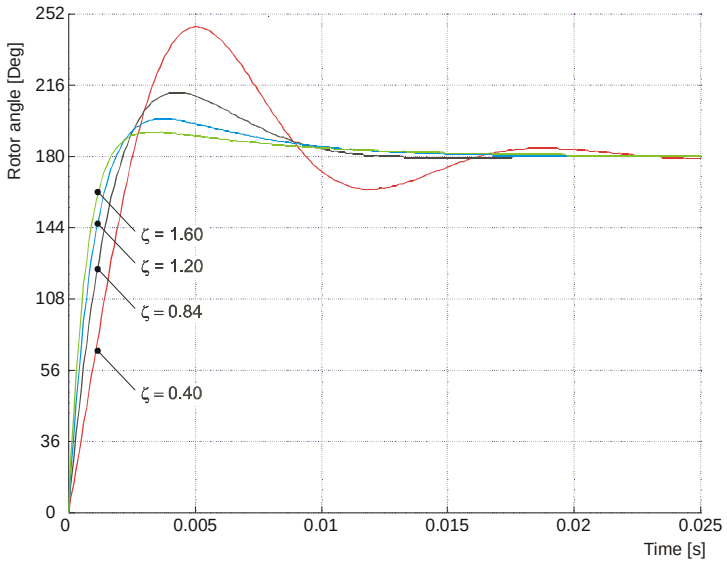

Figure 3-9. Dynamic Responses of the Rotor Angle Estimations - $\omega_n = 500\,\text{rad}\,\text{s}^{-1}$, $\zeta = 0.4, 0.84, 1.2$ and 1.6

This figure clearly illustrates that the minimal possible *Settling Time*, $t_S = 0.013\text{s}$, is achieved with the *Natural Frequency* $\omega_n = 500\,\text{rad}\,\text{s}^{-1}$ and the *Damping Factor* $\zeta = 0.84$. Note that for this setting the *Angle Tracking Observer* generates the *Peak Overshoot* $\text{OV} = 17\ \%$ in the transient response of the estimated rotor angle. This selection of the *Natural Frequency* and the *Damping Factor* yields a good smoothing capability of the *Angle Tracking Observer*, see **Section 6.3** for more details.

In the second case, the coefficients of the *Angle Tracking Observer* were based on the parameters of the general second-order system $\omega_n = 1200\,\text{rad}\,\text{s}^{-1}$, $\zeta = 0.4, 0.84, 1.2$ and 1.6. The resulting coefficients of the *Angle Tracking Observer* $K_1 = 1440000$ and $K_2 = 0.0006, 0.0014, 0.002$ and 0.0026 were found with the help of equation **EQ 3-8** and **EQ 3-9**, respectively. The simulation results that correspond to this particular setting of the *Angle Tracking Observer* are shown in **Figure 3-10**.

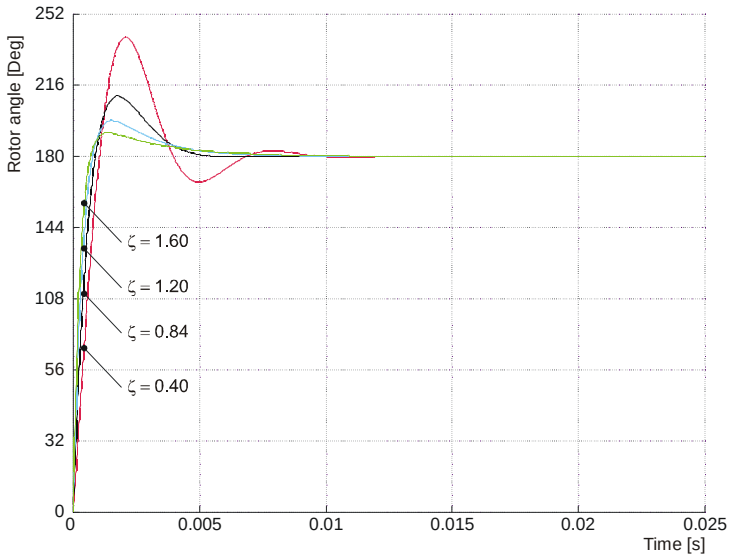

Figure 3-10. Dynamic Responses of the Rotor Angle Estimations - $\omega_n = 1200 \text{rad s}^{-1}$, $\zeta = 0.4, 0.84, 1.2$ and 1.6

Note that the minimal *Settling Time*, $t_S = 0.0053\text{s}$, is achieved with the *Natural Frequency* $\omega_n = 1200 \text{ rad s}^{-1}$ and the *Damping Factor* $\zeta = 0.84$. The *Angle Tracking Observer*, having been based on the dynamic parameters of the general second-order system, generates the *Peak Overshoot* $\text{OV} = 17\%$. This selection of the *Natural Frequency* reasonably shortens the *Settling Time*, however it also decreases the smoothing capability of the *Angle Tracking Observer*.

By considering **Figure 3-9** and **Figure 3-10**, the following conclusions may be reached relating to the selection of the coefficients of the *Angle Tracking Observer*:

- The *Settling Time* of the transient response greatly depends on the *Natural Frequency*; i.e., any higher values of the *Natural Frequency* shorten the *Settling Time* and vice versa. Note that the *Peak Overshoot* does not depend on the *Natural Frequency*.
- The *Peak Overshoot* only depends on the *Damping Factor*; i.e., any lower values of the *Damping Factor* increase the *Peak Overshoots*. Note that considerable changes in the *Damping Factor* only slightly influence the *Settling Time*.

The impact of the *Natural Frequency* and *Damping Factor* on the dynamic behavior of the *Angle Tracking Observer* is clearly summarized in **Figure 3-11**, **Figure 3-12**.

Figure 3-11 graphically illustrates variations in the *Peak Overshoot* (z-axis) in terms of changes made to the *Natural Frequency* and the *Damping Factor*. In this figure, the x-axis represents the *Natural Frequency*, expressed in $\text{rad}\,\text{s}^{-1}$, and the y-axis the dimensionless *Damping Factor*. The *Natural Frequency* and *Damping Factor* vary in the ranges $300\ \text{rad}\,\text{s}^{-1} \le \omega_n \le 1500\ \text{rad}\,\text{s}^{-1}$ and $0.5 \le \zeta \le 2.5$, respectively.

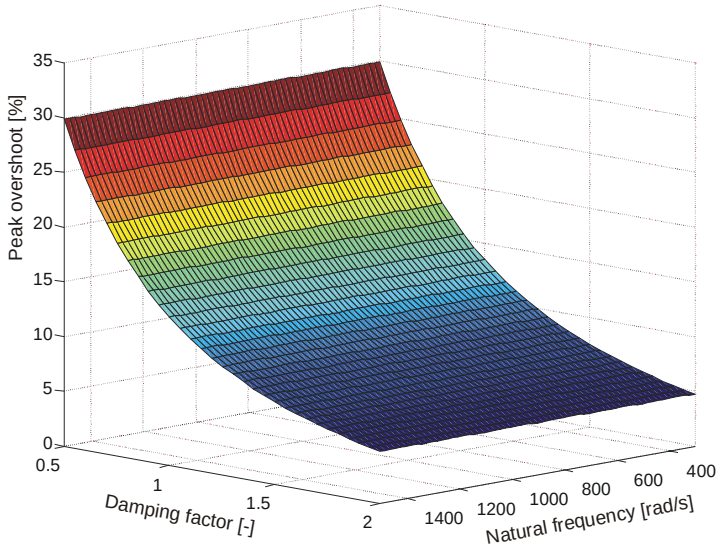

Figure 3-11. Peak Overshoot as a Function of Natural Frequency and Damping Factor

Figure 3-12 graphically illustrates variations in the *Settling Time* (z-axis) in terms of changes of the *Natural Frequency* and the *Damping Factor*. The x and y-axes represent the variations of the *Natural Frequency* and *Damping Factor*, respectively.

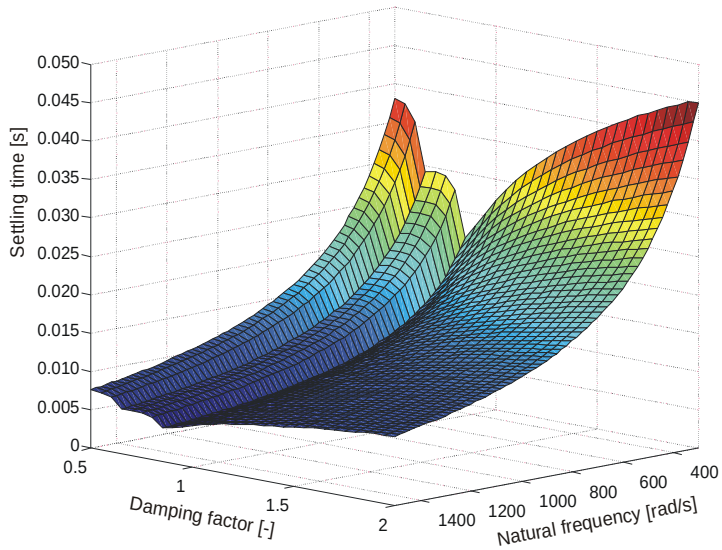

Figure 3-12. Settling Time as a Function of Natural Frequency and Damping Factor

This figure clearly shows the *Damping Factor* $\zeta = 0.84$, at which the *Settling Time* of the *Angle Tracking Observer* is minimized.

This section has given a detailed description of the methods of angle extraction. Two methods, *Trigonometric* and *Angle Tracking Observer*, have been addressed and theoretically analyzed. Theoretical analysis shows that the *Angle Tracking Observer* algorithm performs a better estimation of the rotor angle and speed than the *Trigonometric* method. Because of this fact, the final resolver software driver exploits the *Angle Tracking Observer* algorithm.

4. DSP56F80x - Resolver Software

All software is written in ANSI C with the usage of intrinsic functions to better utilize the processor instruction set and to be able to easily work with fractional arithmetic. The usage of C language without any assembly language instructions is better in terms of code portability and readability and is more user friendly.

The software needed to perform Resolver-to-Digital conversion, can be divided into two parts:

1. **Application Software** - this part includes configuration of the on-chip peripheral modules needed for R/D conversion, synchronization of reference signal and A/D conversion, reading resolver sine and co-sine samples and passing these samples to the driver functions.
2. **Driver Software** (resolver driver) - functions implementing estimation of the actual rotor angle from resolver sine and co-sine samples (this part is independent of specific application settings).

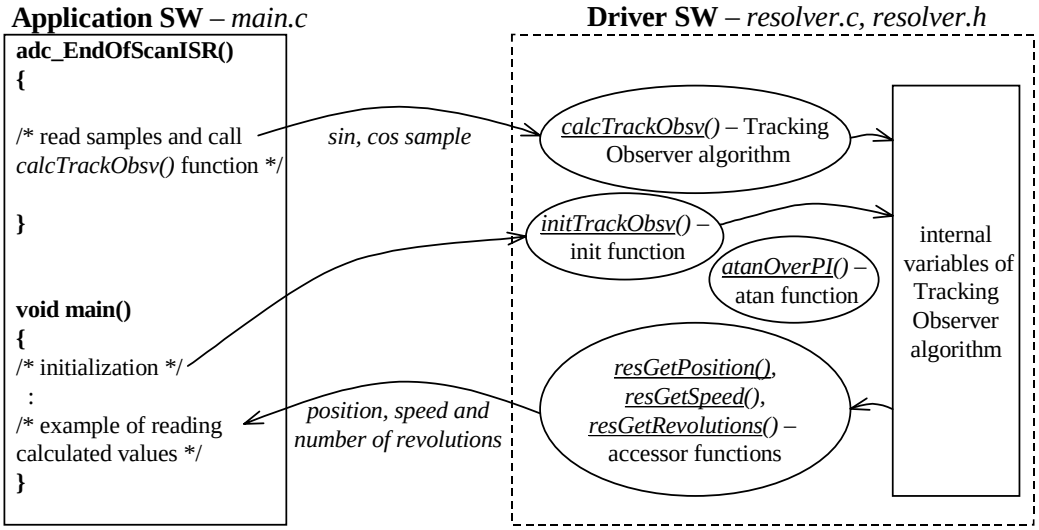

Figure 4-13. Software Structure

Figure 4-13 depicts usage of resolver driver functions in the user application. First of all, necessary initialization is performed in *main()* function. Consecutively, all processing takes place in the *adc_EndOfScanISR()* interrupt service routine (ISR). In this routine, the samples read from the ADC are scaled to full range -1 to 1 and are then passed to the *calcTrackObsv()* function. This function estimates the actual rotor angle, speed and number of revolutions and stores the results in internal variables of the *Angle Tracking Observer* algorithm. The information stored in these internal variables can be accessed from user code by the accessor functions *resGetPosition(), resGetSpeed()* and *resGetRevolutions()*.

Figure 4-13 clearly shows that the Driver SW operates almost independently - it only requires that initialization and samples of the resolver sine and co-sine waveforms be provided by the user application. Due to this driver design, it can be very easily incorporated into various user applications.

4.1 Application Software

In order to implement appropriate R/D conversion, the processor must execute the following tasks:

- Initialization of *Angle Tracking Observer* algorithm + initialization of on-chip peripheral modules (PWM A, timer C2, ADC A, timer D1) at the beginning of the user application
- Generation of the reference signal (timer D1)
- A/D conversion in synchronization with the reference signal (ADC A) and PWM
- Estimation of actual rotor angle, speed and number of revolutions (*calcTrackObsv()* function)
- Accessing estimated rotor angle, speed and number of revolutions using accessor functions *getResPosition()*, *getResSpeed()* and *getResRevolutions()*

It should be noted that all tasks except of initialization must be executed concurrently with the main motor control application. Therefore the design of the R/D conversion was done with respect to easy integration into the motor control applications.

4.1.1 Synchronizing ADC with PWM on DSP56F80x

The motor control applications use an ADC peripheral module to sample motor phase currents. The phase currents are passed as additional inputs into a motor control algorithm that calculates PWM duty cycles. It is advantageous if the A/D conversion is performed synchronously with generated PWM signals because of the elimination of noise and because of the fact that phase currents are synchronous with respect to PWM.

The DSP56F80x processors provide very effective hardware synchronization[1] of the PWM peripheral module and the ADC peripheral module. The main benefit of the hardware synchronization is very high sampling accuracy in the time domain. Because the hardware synchronization ADC->PWM is automatically performed on our DSPs, it is described here in more details.

The timing diagram of the synchronization is shown in **Figure 4-14** (top part). In this mode the synchronization base is provided by the PWM A counter. Each time a PWM A reload event occurs the SYNC pulse is generated. The SYNC signal is internally connected to one channel of the Quad Timer module (C). Then timer channel (C) acts as a programmable delay line. It starts counting after the SYNC pulse appears, then counts up to a pre-programmed compare value, and when the compare event occurs, it asserts its output. The assertion of the timer channel C2 output triggers an ADC A conversion. By pre-programming the compare value, the sampling point can be shifted anywhere within two PWM reload events.

1. Synchronization mode is fully autonomous and does not require any software interference. Therefore it is often used in a wide range of motor control applications.

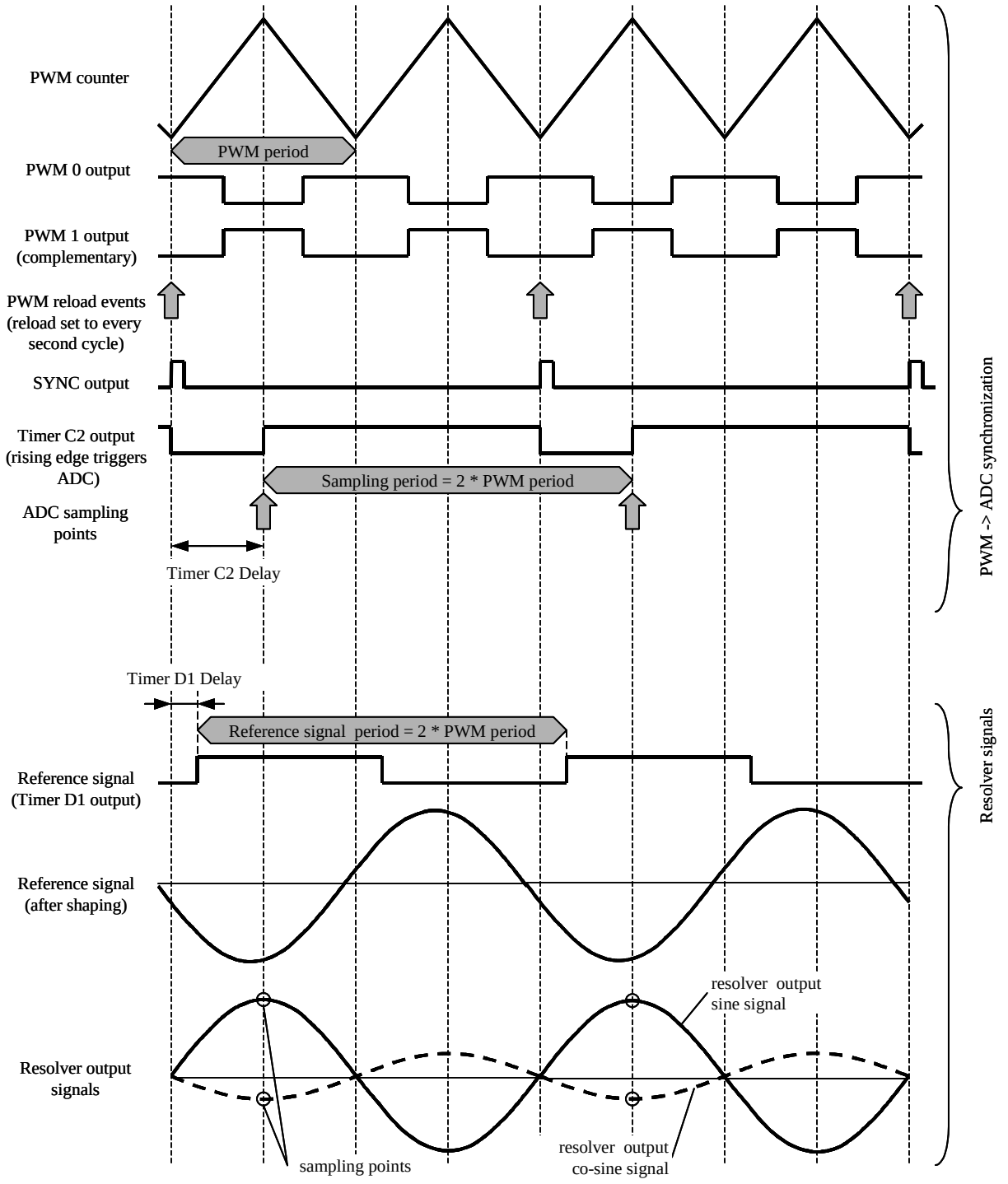

Figure 4-14. Timing Diagram

4.1.2 Initialization of the Resolver-to-Digital Conversion

The initialization includes the setup of the internal variables of the *Angle Tracking Observer* algorithm and configuration of used on-chip peripheral modules. The configuration of the following on-chip peripheral modules must be performed:

- PWM A
- timer C2
- ADC A
- timer D1.

The PWM A, C2 timer and ADC A are used in a way described in the previous section; i.e., ADC A sampling is in synchronization with the PWM A peripheral module. In addition to sampling the application specific signals (e.g. phase currents), two channels of ADC A are reserved for simultaneous sampling of resolver output sine and cosine signals.

These parameters of PWM A -> ADC A synchronization are set:

- PWM frequency 16 kHz
- sampling frequency 8 kHz (reload every second cycle -> therefore sampling frequency is half of PWM frequency)
- sampling points are shifted to the middle of two PWM reload opportunities
- both resolver output sine and cosine signals are sampled simultaneously

The D1 timer generates a resolver reference signal with frequency 8 kHz (equal to the sampling frequency). The generated signal is a square wave one with 1:1 duty ratio. These parameters of the D1 timer are set:

- Count rising edges of IPBus clock
- Toggle output on successful compare
- Count until compare, then re-initialize
- Output enabled
- Compare Value stored in Compare Register 1 is set to

$$\frac{\text{reference signal period}}{2} - 1 \quad \text{, [timer ticks]}^{1} \qquad \text{(EQ 4-1)}$$

- Load Register is set to 0

The timer channel periodically counts up until a compare, and whenever a compare event occurs, the output of the timer channel is toggled (square-wave generation) and the value stored in the Load Register is automatically loaded into the Counter Register - timer is automatically re-initialized.

A special triggering sequence is executed to start D1 timer in order to synchronize the timer with respect to ADC A with a required phase shift. See **Section 4.1.3** where synchronization of the reference signal (generated by D1 timer) and ADC A is discussed in detail.

1. The subtraction of 1 is included in the equation because, for a value of 0 in the Compare Register, the timer produces a delay of 1 clock cycle.

4.1.3 Synchronizing ADC with the Reference Signal

As was stated in **Section 3.3.1**, an indispensable precondition of accurate rotor angle estimation is to sample the resolver output signals simultaneously and as close as possible to their period peaks. **Figure 4-14** shows that there are two possibilities: to sample in period peaks in the first or in the second half of the period of the resolver output signals. Both possibilities can be used because both give correct results.

The sampling in period peaks is required for two reasons. The first reason is that with this preposition the sampling will never occur at a moment when the signal crosses the zero level and therefore the sampled value is equal to zero only when the amplitude of the signal is zero. The second reason is to utilize the full range of the ADC A.

The precondition of sampling in period peaks means that the ADC and reference signal must be in synchronization with a certain phase shift. In order to guarantee this synchronization, a reference signal is automatically generated by timer channel D1 with the frequency equal to the sampling frequency. The sampling points are required to be as close as possible to the period peaks of the resolver output signals. Because in motor control applications the sampling points are usually placed in the middle of two PWM reloads. The reference signal generated by timer D1 is shifted with respect to PWM reload events by $T_{RefSignalDelay}$. This shift compensates phase shift introduced by external resolver circuits and the resolver itself. As seen in **Figure 4-14**, the formula for the delay is:

$$T_{RefSignalDelay} = T_{SampleDelay} - T_{ResPhaseShift}, \quad \text{(EQ 4-2)}$$

where:

- $T_{RefSignalDelay}$ is delay between PWM reload event and rising edge of rectangular reference signal generated by timer D1 [s]
- $T_{SampleDelay}$ is delay between PWM reload event and resolver output signals sampling points [s]
- $T_{ResPhaseShift}$ is delay caused by phase shift of external resolver circuits and resolver itself (or in other words, the delay between the rising edge of rectangular reference signal generated by timer channel D1 and peaks of sampled resolver output signals) [s].

The phase shift of the signal generated by timer D1 channel with respect to the PWM reload events is performed by executing a special triggering sequence in the application initialization phase. The flowchart of this sequence is shown in **Figure 4-15**.

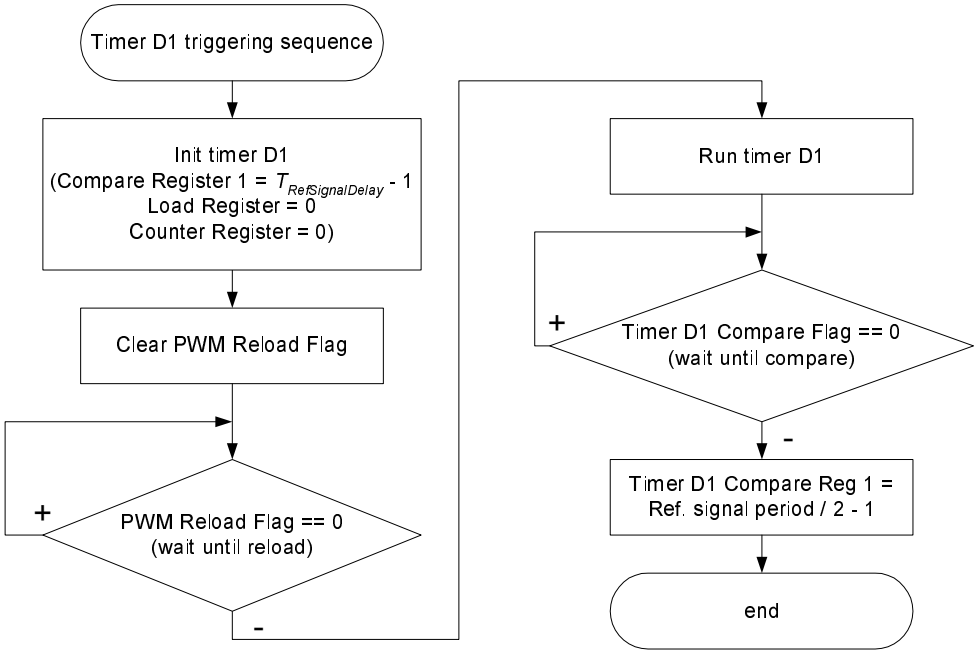

Figure 4-15. Flowchart of Timer D1 Triggering Sequence

The triggering sequence starts with the initialization of D1 timer. Then after detecting the PWM reload event (i.e. PWM Reload Flag is set), the timer is started and counts until the compare. This causes the delay of $T_{RefSignalDelay}$ after the PWM reload event (see **Figure 4-14**). When the compare event occurs, the output of timer is toggled, which means the rising edge of the reference signal is shifted by $T_{RefSignalDelay}$.

The compare event is detected in software by testing the Timer Compare Flag. Finally, the compare value in Timer Compare Register 1 is changed to a value corresponding to half of the reference signal period. Timer D1 then automatically generates a reference signal by toggling its output at every compare event and with the required phase shift with respect to the PWM reload event (and therefore also with respect to the sampling points).

The $T_{RefSignalDelay}$ delay has to be correctly set in order to sample in maximums of the resolver output signals. The formula for $T_{RefSignalDelay}$ delay is given in **EQ 4-2**. The phase shift $T_{ResPhaseShift}$ in this formula can be either measured by an oscilloscope or it can be automatically determined by software in the application initialization phase.

Note that the PWM A generator must be running when this initialization sequence is executed because the D1 timer is synchronized with respect to the PWM reload event. The PWM outputs are disabled so no PWM signals are generated. The interrupts also have to be disabled, otherwise the timing of the timer D1 initialization could be corrupted.

The implemented solution provides easy integration of R/D conversion into motor control applications. Since the synchronization of the generation of the reference signal is independent from application specific ADC -> PWM synchronization, the incorporation of R/D conversion is easy. It only requires configuring a timer for generating the reference signal and two ADC channels to sample the resolver output signals.

4.1.4 Angle Tracking Observer Call

After initialization, the processing of the R/D conversion takes place every sampling period, e.g. in the ADC End of Scan Interrupt Service Routine (ISR). The ADC End of Scan ISR is called whenever the ADC conversion is finished. The sine and co-sine samples read from ADC A are scaled to full 16-bit fractional range -1 to 1. After the scaling, the function *calcTrackObsv()* is called. This function estimates actual rotor position and speed using the *Angle Tracking Observer* algorithm and updates number of revolutions. All calculated results are stored in the internal variables of the resolver driver.

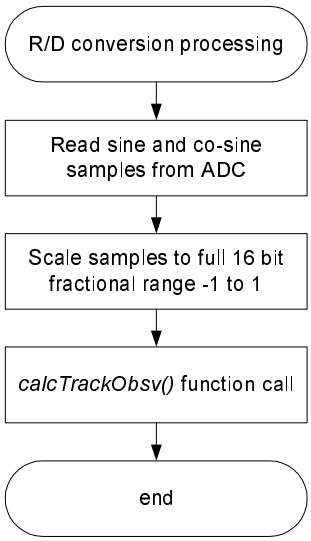

Figure 4-16. R/D conversion processing

The scaling of sine and co-sine samples to full 16-bit fractional range -1 to 1 is required by the *Angle Tracking Observer* algorithm. The observer error signal $\sin(\Theta - \hat{\Theta})$ is calculated as

$$\sin(\Theta - \hat{\Theta}) = \sin(\Theta)\cos(\hat{\Theta}) - \cos(\Theta)\sin(\hat{\Theta}), \qquad \text{(EQ 4-3)}$$

where Θ is the actual rotor angle and $\hat{\Theta}$ is its corresponding estimation (see **Figure 3-7**). The functions $\sin(\Theta)$ and $\cos(\Theta)$ are represented by sine and co-sine samples. Therefore static scaling of samples to the range -1 to 1 is mandatory. Otherwise the dynamic behavior of the Angle *Tracking Observer* algorithm would be different (slower).

The scaling coefficients for both sine and co-sine samples are expressed in the form SCALE_MANT*2^{SCALE_EXP}, where SCALE_MANT is in the range 0 to 1. Scaling is then performed by multiplying with fractional value SCALE_MANT and shifting by SCALE_EXP bits to the left.

4.2 Driver Software Implementation

The driver software contains functions that perform an estimation of the actual rotor position, speed and number of revolutions using the *Angle Tracking Observer* algorithm. The data flow of the *Angle Tracking Observer* is showed in **Figure 4-17**. For a given sine and co-sine sample the rotor position, speed and number of revolutions are calculated.

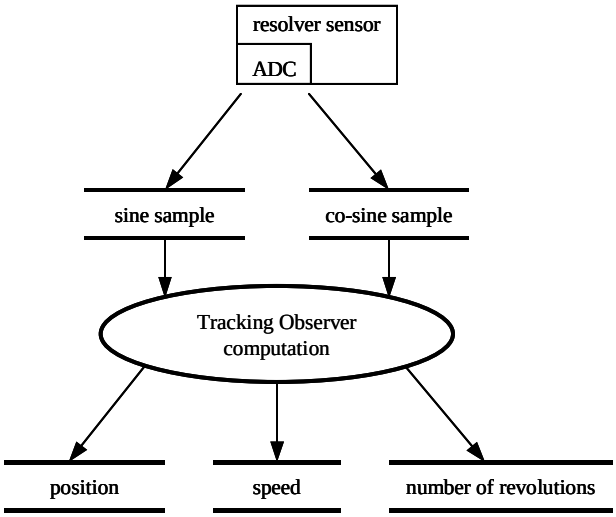

Figure 4-17. Tracking Observer Data Flow

All functions of the resolver driver are defined and declared in module *resolver.c* and include file *resolver.h*, respectively. The detailed description of driver functions is given in **Section 4.2.1**. Implementation of the *Angle Tracking Observer* algorithm is discussed in **Section 4.2.2**.

4.2.1 Driver Functions

This section describes in detail the resolver driver functions.

- *void initTrackObsv(void);*

This function initializes internal variables of the *Angle Tracking Observer*. It should be called in the initialization part of the user's application software.

- *void calcTrackObsv(Frac16 sinA, Frac16 cosA);*

This function calculates the *Angle Tracking Observer* algorithm. This function is called every sampling period, e.g. in the ADC End of Scan interrupt service routine. It requires two input arguments: the sine sample *sinA* and co-sine sample *cosA*. Note that those samples must be scaled to the range -1 to 1, before passing to that function. The implementation of the *Angle Tracking Observer* algorithm is described in detail in **Section 4.2.2**.

The function returns an estimation of the actual rotor angle, speed and number of revolutions in internal variables of the *Angle Tracking Observer* algorithm. Then these internal variables can be read by accessory functions *getResPosition()*, *getResSpeed()*, and *getRevolutions()* respectively, which are described below.

- *Frac16 getResPosition(void)*

This function returns an estimate of the actual rotor angle. Note that function *calcTrackObsv()* must be called prior to the call of this driver function. The returned value is the 16 bit signed fractional value in the range -1 to 1 corresponding $-\pi$ to π.

- *Frac32 getResSpeed(void)*

This function returns the estimate of the actual rotor speed read from an internal variable of the *Angle Tracking Observer* algorithm. Note that the function *calcTrackObsv()* must be called prior to the call of this driver function. The function returns a 32-bit signed fractional value in the range -1 to 1. The relation between the returned digital rotor speed in step *k+1*, marked as $\Omega_d(k+1)$, and the actual rotor speed in step *k+1*, $\Omega(k+1)$ in [rad/s] is

$$\Omega_d(k+1) = \frac{\Omega(k+1) \cdot T_s}{\pi}, \qquad \text{(EQ 4-4)}$$

where T_s [s] is the sampling period.

- *int getResRevolutions(void)*

This function returns the actual number of rotor revolutions. The returned value is taken as a 16-bit signed integer (range -32768 to 32767).

- *void setResPosition(Frac16 newPosition)*

This function is used to set a new angle to the current angle. The function is supposed to initialize the instantaneous rotor angle to zero or any other value which might be useful at the start-up of the application. The passed argument *newPosition* is in the 16-bit fractional range -1 to 1 corresponding $-\pi$ to π.

- *void setResRevolutions(int newRevolutions)*

This function sets the number of revolutions that is stored in the internal variable of the *Angle Tracking Observer* algorithm. It is usually used to set the number of revolutions to zero in the application initialization phase.

- *Frac16 atan2OverPI(Frac16 y, Frac16 x)*

This routine computes $\text{atan}(y/x)/\pi$, where the atan function is approximated by the fifth order Taylor series.

The simple code example demonstrating proper use of the resolver driver functions is shown in **Appendix A: Basic Code Example**.

4.2.2 Implementation of the Angle Tracking Observer

This section discusses the way the *Angle Tracking Observer* algorithm is practically implemented on the DSP56F80x fixed-point digital signal processor.

The analog integrators in **Figure 3-7**, marked as 1/s, are replaced by an equivalent of the discrete-time integrator (see **Figure 4-18**), using the *Forward Euler* integration method.

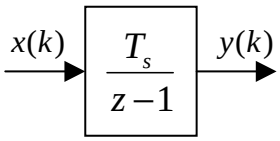

Figure 4-18. Discrete-Time Integrator (Forward Euler)

From the definition of this method, the analog integrator is approximated by a difference equation:

$$y(k+1) = y(k) + x(k) \cdot T_s, \quad \text{(EQ 4-5)}$$

where $x(k)$ and $y(k)$ are input and output values in step k and T_s [s] is the sampling period. Index k represents the previous (old) value and index $k + 1$ the current (new) value. The transfer function corresponding to this difference equation is:

$$H(z) = \frac{T_s}{z-1}. \quad \text{(EQ 4-6)}$$

The discrete-time block diagram of the *Angle Tracking Observer* is shown in **Figure 4-19**.

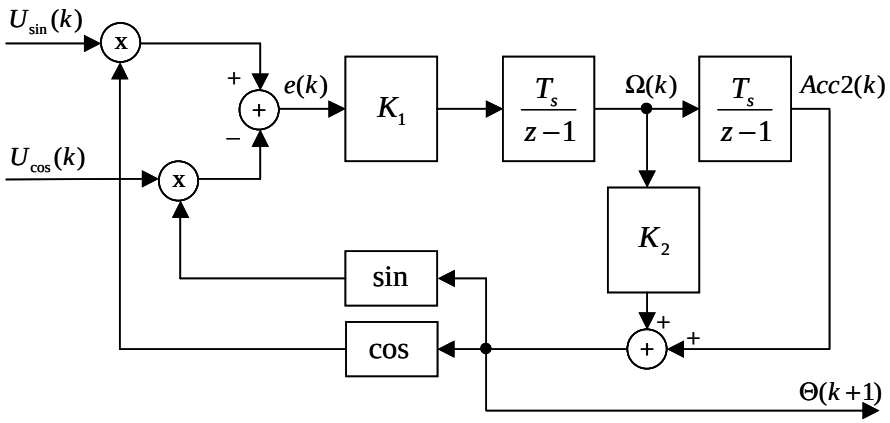

Figure 4-19. Block Scheme of Discrete-Time Tracking Observer

It should be noted that the loop with gain K_2 is predictive and together with *Acc2(k)* it provides for the estimation of the actual rotor angle in step *k+1*. The essential equations for implementation of the *Angle Tracking Observer*, according to block scheme in **Figure 4-19**, are as follows:

$$\Omega(k+1) = \Omega(k) + T_s \cdot e(k) \cdot K_1 \quad \text{(EQ 4-7)}$$

$$Acc2(k+1) = Acc2(k) + T_s \cdot \Omega(k) \quad \text{(EQ 4-8)}$$

$$\Theta(k+1) = K_2 \cdot \Omega(k+1) + Acc2(k+1) \quad \text{(EQ 4-9)}$$

$$e(k+1) = U_{sin}(k+1) \cdot \cos\Theta(k+1) - (U_{cos}(k+1) \cdot \sin\Theta(k+1)) \quad \text{(EQ 4-10)}$$

where: $K_1 = \omega_n^2$ and $K_2 = 2\zeta/\omega_n$ are coefficients of the *Angle Tracking Observer*, (EQ 4-11)

$e(k)$ is observer error in step k,

ω_n and ζ are the natural frequency [rad s^{-1}] and damping factor [-],

T_s is the sampling period [s],

$\Omega(k+1)$ is the actual rotor speed [rad s^{-1}] in step $k + 1$,

$Acc2(k+1)$ is the actual rotor angle [rad] without scaled addition of speed in step $k + 1$,

$\Theta(k+1)$ is the actual rotor angle [rad] in step $k + 1$,

$U_{sin}(k+1)$ and $U_{cos}(k+1)$ are sine and co-sine samples in step $k + 1$.

In equations **EQ 4-7** to **EQ 4-10**, there are coefficients and quantities that are greater than one (for example, the actual rotor speed $\Omega(k+1)$) or that are too small to be precisely represented by a 16 bit fractional value. Due to this fact a special transformation of equations **EQ 4-7** to **EQ 4-10** must be carried out in order to be successfully implemented using fractional arithmetic. This transformation is based on several steps.

Firstly, the actual rotor angle in the digital representation $\Theta_d(k+1)$[1] is scaled by π to fit into the range -1 to 1:

$$\Theta_d(k+1) = \frac{\Theta(k+1)}{\pi}. \qquad \text{(EQ 4-12)}$$

Secondly, the discrete-time integrators are replaced by accumulators; i.e., the integrators are computed only as summations without multiplying the input value by sampling period T_s. In comparison with **EQ 4-5**, the accumulator is defined as $y(k+1) = y(k) + x(k)$, where $x(k)$ and $y(k)$ are input and output values in step k.

The last step of the transformation is that the coefficients of *Angle Tracking Observer* K_1 in equation **EQ 4-7** and K_2 in equation **EQ 4-9** are replaced by their scaled equivalents K_{1d} and K_{2d} to reflect the scaling of position by π and the elimination of sampling period T_s in the integrator computation.

Finally, after the transformation, the equations suitable for implementation on the DSP56800 core are as follows:

$$\Omega_d(k+1) = \Omega_d(k) + e(k) \cdot K_{1d} \qquad \text{(EQ 4-13)}$$

$$Acc2_d(k+1) = Acc2_d(k) + \Omega_d(k) \qquad \text{(EQ 4-14)}$$

$$\Theta_d(k+1) = K_{2d} \cdot \Omega_d(k+1) + Acc2_d(k+1) \qquad \text{(EQ 4-15)}$$

$$e(k+1) = U_{\sin}(k+1) \cdot \cos(\pi \cdot \Theta_d(k+1)) - (U_{\cos}(k+1) \cdot \sin(\pi \cdot \Theta_d(k+1))), \qquad \text{(EQ 4-16)}$$

where the scaled coefficients K_{1d} and K_{2d} can be expressed after the derivation by the formulas:

$$K_{1d} = \frac{1}{\pi} \cdot T_s^2 \cdot K_1 = \frac{1}{\pi} \cdot \omega_n^2 \cdot T_s^2 \qquad \text{(EQ 4-17)}$$

$$K_{2d} = \frac{K_2}{T_s} = \frac{2 \cdot \zeta}{\omega_n \cdot T_s} \qquad \text{(EQ 4-18)}$$

where: ω_n is the natural frequency [rad s^{-1}], ζ is the damping factor [-] and T_s is the sampling period [s].

There is a $1/\pi$ included in the K_{1d} coefficient as a result of scaling the rotor position by π and the sampling period T_s in K_{1d} and K_{2d} coefficients as a result of replacing discrete-time integrators by accumulators.

1. Subscript ***d*** denotes a digital representation of the corresponding constant/variable.

The relation between the digital rotor speed $\Omega_d(k+1)$ in the range –1 to 1 and the actual rotor speed $\Omega(k+1)$ in [rad/s] is:

$$\Omega_d(k+1) = \frac{\Omega(k+1) \cdot T_s}{\pi}. \qquad \text{(EQ 4-19)}$$

Note that this expression can be directly derived from the comparison of equations **EQ 4-7** and **EQ 4-13**. **Table 4-2** shows the maximal and minimal rotor speed that corresponds to the digital rotor speed of 1 and 2^{-31}, respectively.

Table 4-2. Maximal and Minimal Rotor Speed

Sampling Frequency [kHz]	Maximal Rotor Speed [r.p.m.]	Minimal Rotor[1] Speed [r.p.m.]
16	480000	0.00022
8	240000	0.00011
4	120000	0.00005

[1] The minimal measurable rotor speed is very low. In certain application, however, it is typically limited by noise conditions to 0.1 r.p.m.

The functionality of the *Angle Tracking Observer* can also be explained using an example of the constant rotor speed. If the observer error $e(k)$ is zero then the first accumulator, representing speed $\Omega_d(k+1)$, remains constant. At every sampling period a constant value (first accumulator) - angular difference passed during T_s - is added to the second accumulator, representing position $Acc2_d(k+1)$. Note that implementation must reflect angular position overflow at the π/-π boundary.

4.2.3 Angle Tracking Observer Coefficients

Before the resolver driver functions are used, the user is required to define the *Angle Tracking Observer* coefficients K1_D, K1_SCALE and K2_D, K2_SCALE in the include file *resolver.h*. These coefficients can be calculated using expressions:

$$K1\_D = K_{1d} \cdot 2^{K1\_SCALE} \qquad \text{(EQ 4-20)}$$

$$K2\_D = K_{2d} \cdot 2^{-K2\_SCALE} \qquad \text{(EQ 4-21)}$$

where: K_{1d} and K_{2d} are coefficients given by equation **EQ 4-17** and **EQ 4-18** and

$K1\_SCALE$, $K2\_SCALE$ are chosen in such a way that $K1\_D,\ K2\_D \in \langle 0.5, 1.0)$.

Both coefficients K_{1d} and K_{2d} are normalized using introduced transformations to fit in a 16-bit fractional format. Having assigned scaling coefficients, the multiplication by coefficient K_{1d} (K_{2d}) can be easily performed by multiplication with its normalized value K1_D (K2_D) and then by shifting the result right (left) accordingly to the number of bits given by K1_SCALE (K2_SCALE).

It follows an example of calculation of the *Angle Tracking Observer* coefficients:

$\omega_n = 2\ \pi\ 100$ rad s^{-1} (F_n = 100 Hz)

$\zeta = 1.5$

$F_s = 1/T_s$ = 8000 Hz

K_{1d} = 1.963e-3

K_{2d} = 38.197

K1_D = 0.5026548, K1_SCALE = 8

K2_D = 0.5968310, K2_SCALE = 6

Note that these coefficients must be defined in the *resolver.h* header file:

```
#define K1_D          FRAC16(0.5026548)  1
#define K2_D          FRAC16(0.5968310)
#define K1_SCALE      8
#define K2_SCALE      6
```

4.2.4 DSP Processing Load

Table 4-3 displays clock cycles for all functions - CodeWarrior 4.0 compiler was used.

Table 4-3. DSP Usage of R/D conversion (CW 4.0)

Function	Execution Time [clock cycles]
calcTrackObsv()	638
getResPosition()	32
getResSpeed()	36
getResRevolutions()	26

For the maximum 80 MHz DSP core frequency (clock cycle = 12.5 ns) and the 8 kHz reference signal frequency the total processor loading regarding R/D conversion is 7.5 %.

Table 4-4 shows the memory requirements[2] of R/D conversion.

Table 4-4. Ram and Flash Memory Usage of R/D conversion (CW 4.0)

Memory	Used Memory (in 16-bit words)
Program FLASH	319
Data RAM	10 + 8 stack
Data FLASH	257

1. FRAC16 is macro which transforms a fractional value in the range -1 to 1 into a signed integer value in the range -32768 to 32767.
2. The data FLASH is used for the sine table storage.

5. Hardware Interface

An interface for direct connection of the resolver position sensor with the DSP56F805EVM is discussed here. This interface circuit generates/shapes the signal for the resolver reference winding and conditions signals from sin/cos windings for measurement by the on chip ADC module.

The interface circuit in **Figure 5-20** consists of two main parts:

- Resolver driving circuitry
- Resolver sin/cos signals conditioning circuitry

The resolver driving circuitry shapes a rectangular reference signal from DSP Quad Timer (channel TD1) output to a sinusoidal waveform. U1A creates an integrator which transforms the rectangular signal into a triangle. The remaining higher harmonic component is filtered out by the following stage U1B that drives the resolver reference winding. The U1B stage is, in fact, an integrator too. The ratios of R1/R3 and R2/R4 resistors control the integrator's linearity; the higher the ratio the better the sine curve generated at the output. However, if the ratio is too high the circuitry is sensitive to noise pickups from the power stages and also to changes in the reference signal duty cycle. Therefore, the reference signal is automatically generated by the Quad Timer module (channel TD1) to achieve a precise 50% duty cycle for a quality reference signal.

The resolver reference winding used in our application circuit has a resistance of 27 ohm, which leads to a 120mA peak current. The amplifier TCA0372 was chosen as driving stage, because it is capable of driving up to a 1A output current. The R5,C8 creates Boucherot circuitry that suppresses output ringing when driving inductance load. In the other type of resolver being used, it may be necessary to modify the values of R5,C8 to limit possible oscillations. Output capacitor C4 decouples the output signal dc component. Both amplifiers operate on single supply so a virtual ground is created by resistor divider R6,R7. Supply voltage VCC should be at least 2V higher than the required peak-to-peak output swing.

Driving circuitry introduces a phase shift between timer output signal and resulting resolver reference waveform. This phase shift together with resolver phase shift and signal conditioning circuitry phase shift are corrected in software by advancing the phase of the reference signal (channel TD1) relative to the ADC sampling point, which is in most motor control applications synchronized to PWM, refer to **Figure 4-14**. In this way, the sampling at peaks of the sin/cos signals is ensured, resulting in better achieved resolution.

The resolver sin/cos signals conditioning circuitry adjusts voltage levels from resolver sin/cos signals to the range acceptable by the on-chip ADC module. It also carries out level shifting, which places a zero level of the signals to the middle of the ADC range. U2A, U2B amplifiers act as differential unity amplifiers with output level referenced to virtual ground (middle of the VCCA 3.3V). U2A,B amplifiers are rail-to-rail (MC33202, MC33502) or similar ones capable of 3.3V single supply operation. The capacitors C9,C12, C14,C16 add low pass filtering to suppress unwanted high frequency noise, which is often present in systems with power electronics.

The cutoff frequency of the U2A and U2B amplifiers is set according to the resolver reference frequency and should be well above it not to affect resolver signals. U2A,B amplifiers should be placed as close as possible to the ADC inputs to avoid noise crosstalk from other components.

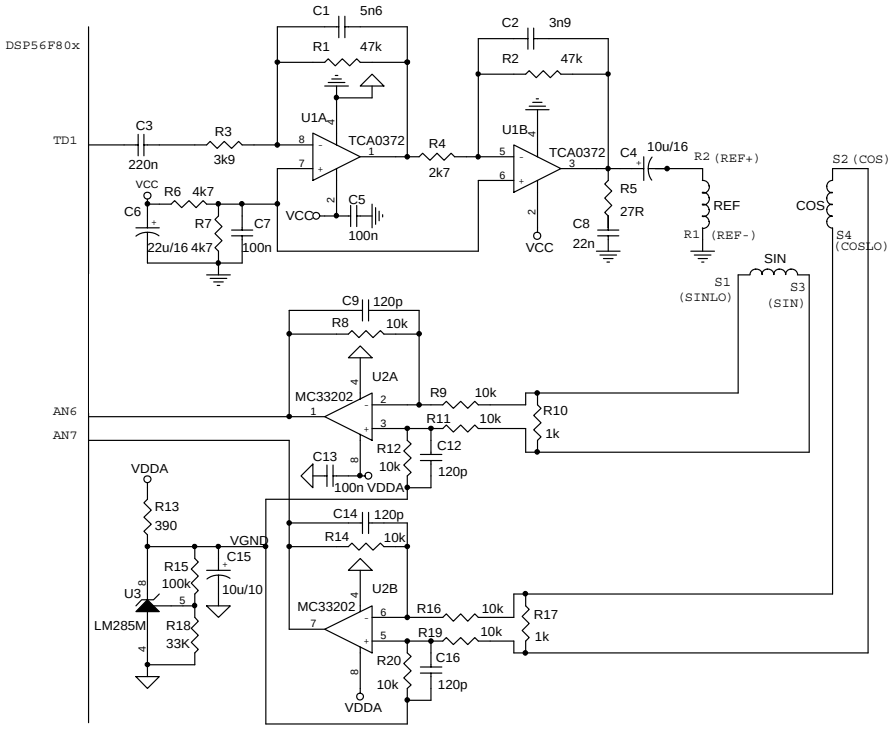

Figure 5-20. Resolver Interface Board Schematic

All values of the schematic component given in **Figure 5-20** are designed for 8 kHz resolver reference frequency (half of the usual motor control PWM frequency), resolver ratio 2:1. This interface might be adjusted in cases when reference signal frequency or the resolver transformation ratio is different. The gain of the resolver signal conditioning circuitry is unity and therefore the levels on the sin/cos signal inputs must have peak-to-peak amplitude up to the ADC reference voltage (with small headroom to avoid limiting).

In case the reference frequency is different than the values of C1,R3 and C2,R4 should be adjusted to get a proper sine waveform with the needed resolver driving level. Note the ratios R1/R3 and R2/R4 have little effect on the driving level, they mainly affect the noise sensitivity of the circuit. The other possibility for changing the driving level is to use a resistor divider connected between the TD1 output and the amplifier input. The schematic explained here is suitable for direct connection to DSP56F805EVM boards.

6. Experimental Tests

Three sets of tests were carried out using ideal, emulated noise and real resolver signals on the DSP56F805 evaluation module (DSP56F805EVM).

- Firstly, the resolver was tested to demonstrate that the *Angle Tracking Observer*, running on the DSP56F805EVM, is competent enough to produce fast estimations of rotor angle and rotor speed; refer to **Section 6.1**.
- Secondly, the smoothing feature of the *Angle Tracking Observer* was demonstrated; refer to **Section 6.2**.
- The third set of tests was a study of the dynamic behavior and smoothing of the *Angle Tracking Observer* in the whole application; i.e., the observer was driven by real output signals of the resolver; refer to **Section 6.3**.

6.1 Dynamic parameters of the Angle Tracking Observer

It is clear that dynamic parameters of the observer could never be obtained using the signals of the real resolver because of certain mechanical and electrical inertia of such devices. For that reason, the real resolver signals were replaced here with their ideal equivalents calculated on the DSP prior to observer calculation. In fact, the main goal of these tests was to find the ideal dynamic responses of the observer algorithm on step changes of the rotor angle and speed.

The dynamic responses of the *Angle Tracking Observer* were materialized using special software running on the DSP56F805EVM. This software was completely written in C language using the CodeWarrior DSP56800 development tool. The software performs the following tasks:

- It generates step changes of the resolver angle and calculates corresponding sine and co-sine resolver signals.
- It calculates the *Angle Tracking Observer* and sends calculated data (waveforms) to personal computer for printing and further post-processing.

The transient responses of the angle estimations on the step changes of the actual rotor angle are shown in **Figure 6-21**. In this case, the coefficients of the *Angle Tracking Observer* were based on the parameters of the second order system $\omega_n = 500\,\mathrm{rad\,s^{-1}}$, $\zeta = 0.84$, refer to **Section 3.3.3** for more details about selection of optimal observer coefficients.

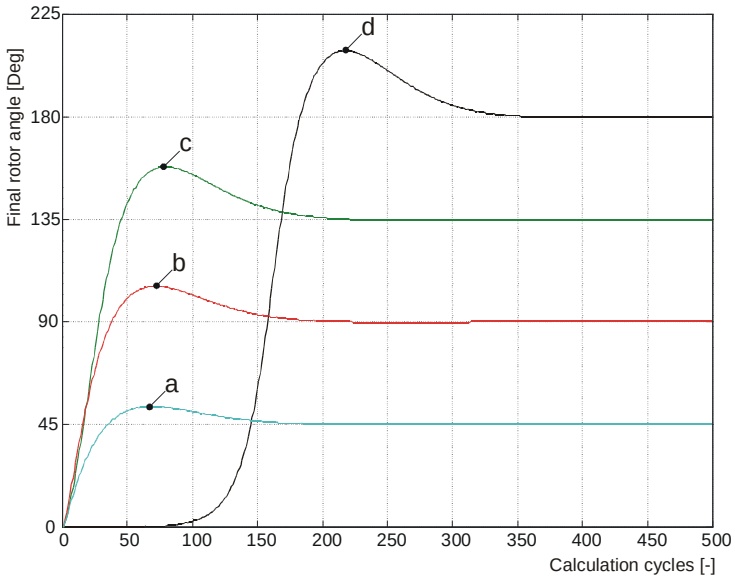

Figure 6-21. Responses of the Estimated Angle of the Angle Tracking Observer - $\omega_n = 500\,\mathrm{rad\,s^{-1}}$, $\zeta = 0.84$.

The estimated rotor angle (y-axis) are illustrated versus calculation cycles (x-axis) of the *Angle Tracking Observer* algorithm.

As is known from theory of control systems, a well-designed observer tries to minimize its estimation error at every calculation cycle. In other words, observers require a certain number of calculation cycles to produce a flawless estimation. This flawless estimation means that the observer outputs reach and remain within a specified tolerance of its steady state value. This tolerance is given by the designer and considerably depends on the requirements of the particular application. Of course, the larger the tolerance the less computational cycles are required to reach a steady state.

An allowable tolerance of ±20′ (electrical) was considered. The considered tolerance corresponds to an estimation error of ±1 LSB in the case that the rotor angle is measured with 10-bit accuracy. The measured *Settling Times* and *Peak Overshoots* in terms of magnitude of the step change of the rotor angle are summarized in **Table 6-5**.

Table 6-5. Numerical Representation of Responses - Figure 6-21.

Response Waveform	Step Change	Peak Overshoot	Settling in Calculation Cycles	Settling Time [s]
d	180^o	17%	352	0.022
c	135^o	17%	208	0.013
b	90^o	17%	192	0.012
a	45^o	17%	176	0.011

The *Settling Times* are calculated considering the timing between two subsequent calculation cycles $62.5\mu s$. **Figure 6-22** shows transient responses of the rotor angle estimations on the step changes of the actual rotor angle. In this case, the coefficients of the *Angle Tracking Observer* were based on the parameters of the general second order system $\omega_n = 1200 \text{rad s}^{-1}$, $\zeta = 0.84$.

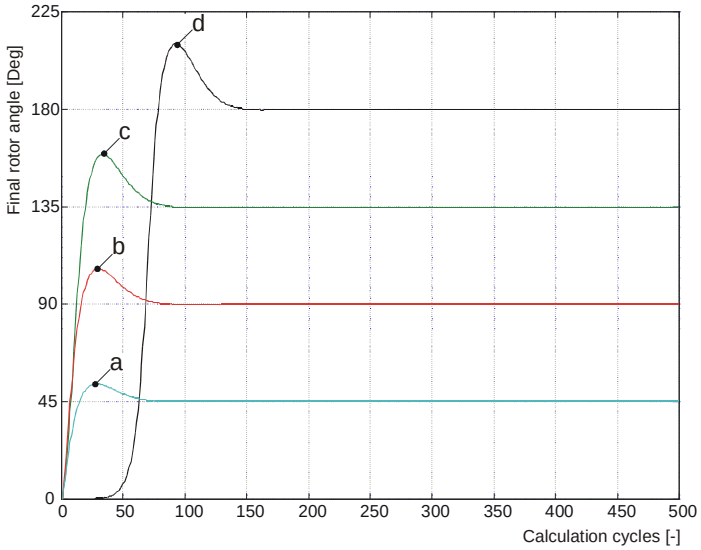

Figure 6-22. Responses of the Estimated Angle of the Angle Tracking Observer - $\omega_n = 1200 \text{rad s}^{-1}$, $\zeta = 0.84$.

The estimation of rotor angle (y-axis) is illustrated versus the calculation cycles (x-axis) of the *Angle Tracking Observer* algorithm. The *Settling Times* and *Peak Overshoots* in terms of magnitudes of the step changes of the rotor angle are summarized in **Table 6-6**.

Table 6-6. Numerical Representation of Responses - Figure 6-22.

Response waveform	Step Change	Peak Overshoot	Settling in calculation cycles	Settling Time [s]
d	180^o	17%	130	0.0081
c	135^o	17%	90	0.0056
b	90^o	17%	80	0.0050
a	45^o	17%	68	0.0043

The question may arise, why the *Settling Times* t_S, expressed here for the step changes of the rotor angle $180°$, differ from those captured in **Figure 3-9** and **Figure 3-10**, despite the fact that identical parameters of the *Angle Tracking Observer* algorithm were used. This is because a different expression for calculating the estimation errors was considered. In the previous sections, the expression $\Theta - \hat{\Theta}$ was used for calculating the estimation error - see responses for $\zeta = 0.84$ in **Figure 3-9** and **Figure 3-10**. In this case, however, the expression $\sin(\Theta - \hat{\Theta})$, reflecting the natural implementation of the *Angle Tracking Observer* algorithm, is considered (see **EQ 3-4**).

The *Settling Times* (y-axis) as a function of steady state angles (x-axis) are summarized in **Figure 6-23**. An allowable tolerance of ±20′ (electrical) was considered, which means that if the transient response of the observer lay within the tolerance, then the pending experiment was automatically stopped and the number of performed calculation cycles was recorded. This graph describes the behavior of the *Angle Tracking Observer* algorithm likewise in a real DSP application - observer estimation error is calculated using the expression $\sin(\Theta - \Theta)$.

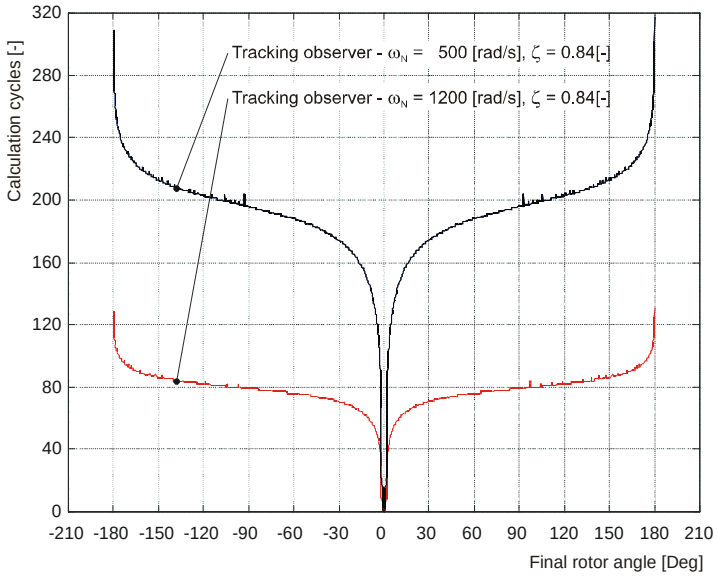

Figure 6-23. Settling Time of the Angle Tracking Observer.

The experiments discussed so far have been focused on the study of the dynamic parameters of the *Angle Tracking Observer*. The objective has been to show some effects of the parameters of the general second-order system ω_n and ζ on the dynamic behavior of the angle and speed estimations.

It has been testified that the *Settling Time* t_S varies with changes of the *Natural Frequency* ω_n and the *Damping Factor* ζ, whereas the *Peak Overshoot* OV varies solely with changes in the *Damping Factor* ζ.

6.2 Smoothing Feature of the Angle Tracking Observer

This section discusses an additional important feature of the *Angle Tracking Observer*, which is known as smoothing (filtering). It is shown that smoothing remarkably depends on the proper selection of coefficients of the *Angle Tracking Observer*.

The theory of control systems defines the hypothesis; the faster the response of the estimated variables is, the less effective their smoothing is and vice versa. This hypothesis is clearly demonstrated in **Figure 6-24**.

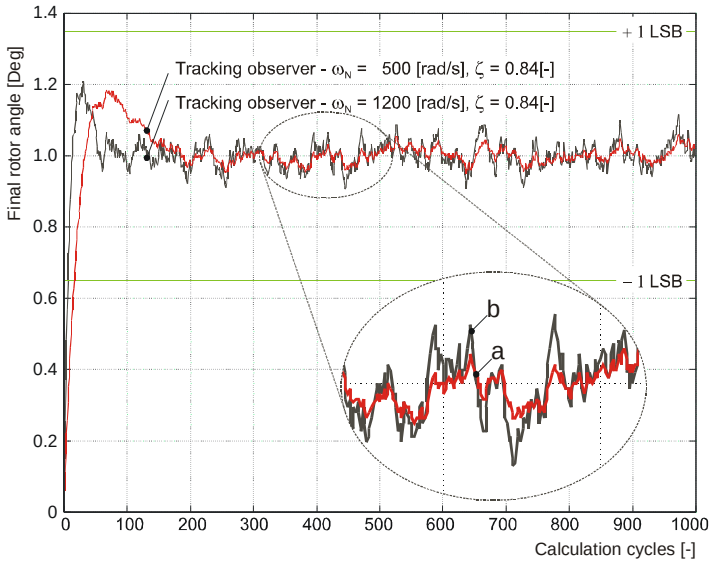

Figure 6-24. Effect of 8-bit ADC accuracy on the Accuracy of the Rotor Angle Estimation.

The figure shows responses of the estimated rotor angle for $1°$ unit-step change of the actual rotor angle. Note that the transient responses denoted in the graph as **a** and **b**, are based on the parameters $\omega_n = 500 \text{rad s}^{-1}$, $\zeta = 0.84$ and $\omega_n = 1200 \text{rad s}^{-1}$, $\zeta = 0.84$, respectively.

Generation of the resolver output signals was performed by special software. The software also adds error into generated signals. This software feature enabled us to simulate the observer algorithm in a mode similar to its normal operation; i.e., a mode with noisy resolver output signals measured using ADC with finite accuracy. First, we tried to show the smoothing feature using simulated sin/cos signals that correspond to the ADC accuracy of the DSP56F80x; however, obtained waveforms did not evidently demonstrate smoothing capability due to the higher accuracy of the simulated sin/cos signals. Consequently, we decided to present more convincing waveforms. Note that introduced error is in the rank of 8-bits signal accuracy.

Figure 6-24 clearly shows that the *Angle Tracking Observer* is capable of accurate estimations even if inaccurate measurement (8-bit ADC) of the resolver output signals is carried out. Note that in both cases, the resulting final estimation error is smaller than ±20'.

We have aimed so far to study the behavior of the *Angle Tracking Observer* in terms of rotor angle estimation. Advice was given for the selection of observer parameters, and discussed the dynamic and smoothing features of the angle estimation.

However, many electric drives require a precise measurement of the instantaneous rotor speed to be made. Generally, this information is obtained by differentiation of the estimated rotor angle or may be given by the *Angle Tracking Observer*. The following is the description of the dynamic behavior and smoothing features of the *Angle Tracking Observer* in terms of speed estimation.

The transient responses of estimated speed and estimation error, generated by the *Angle Tracking Observer* on the step changes of the rotor speed, are graphically illustrated in **Figure 6-25**...**Figure 6-28**.

Figure 6-25 shows the transient responses of the estimated speed $\hat{n}$ of the *Angle Tracking Observer*, whose coefficients have been calculated on the base of parameters $\omega_n = 500\,\mathrm{rad\,s^{-1}}$, $\zeta = 0.84$.

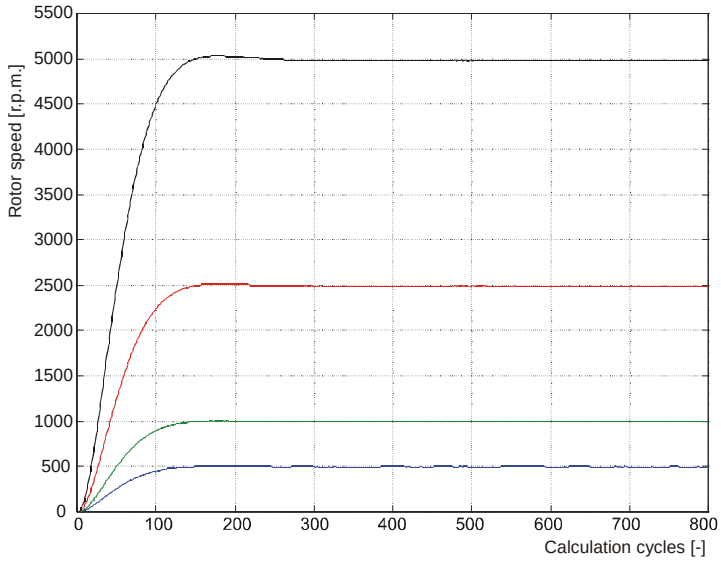

Figure 6-25. Transient Response of Rotor Speed Estimations - $\omega_n = 500\,\mathrm{rad\,s^{-1}}$, $\zeta = 0.84$.

Note that the estimated speed (y-axis) is expressed as a function of the calculation cycles (x-axis) of the *Angle Tracking Observer* algorithm. The depicted transient responses have settled in 160 calculation cycles, which gives - considering the time between two cycles, $62.5\,\mu s$ - a *Settling Time*, $t_S = 0.01\,s$. The *Peak Overshoot* of the transient responses is $OV < 1\%$.

Figure 6-26 expresses errors of the angle estimation $\Theta - \hat{\Theta}$ (y-axis) as a function of step changes of the rotor speed. The x-axis is in calculation cycles. The simulated Observer is based on parameters $\omega_n = 500 \text{rad s}^{-1}$, $\zeta = 0.84$.

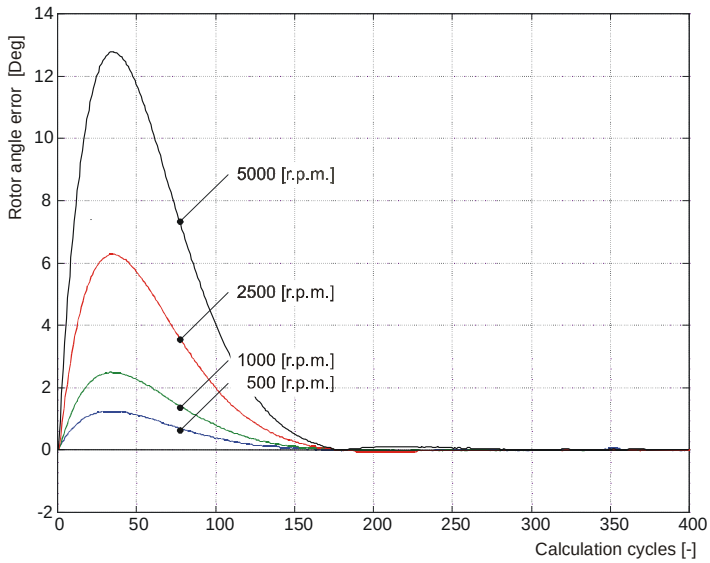

Figure 6-26. Transient Response of Rotor Angle Estimation Error - $\omega_n = 500 \text{rad s}^{-1}$, $\zeta = 0.84$.

Figure 6-27 shows transient responses of the estimated speed $\hat{n}$ that have been generated by the Observer based on parameters $\omega_n = 1200 \text{rad s}^{-1}$, $\zeta = 0.84$. These transient responses reach the steady state in 65 calculation cycles, which results in a *Settling Time*, $t_S = 0.0041 \text{s}$. The *Peak Overshoot* of the responses is $OV < 1\%$.

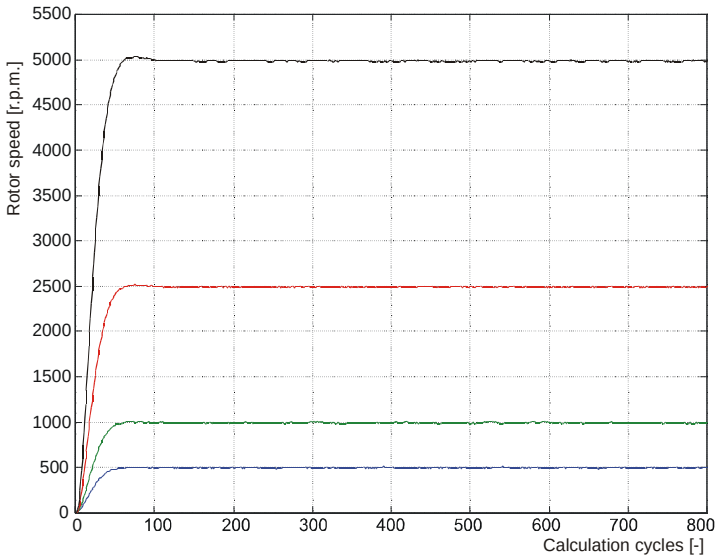

Figure 6-27. Transient Response of Rotor Speed Estimations - $\omega_n = 1200 \text{rad s}^{-1}$, $\zeta = 0.84$.

Figure 6-28 expresses errors of the angle estimation $\Theta - \hat{\Theta}$ during step changes of rotor speed. Here, the *Angle Tracking Observer* is based on parameters of the general second-order system $\omega_n = 1200 \text{rad s}^{-1}$, $\zeta = 0.84$.

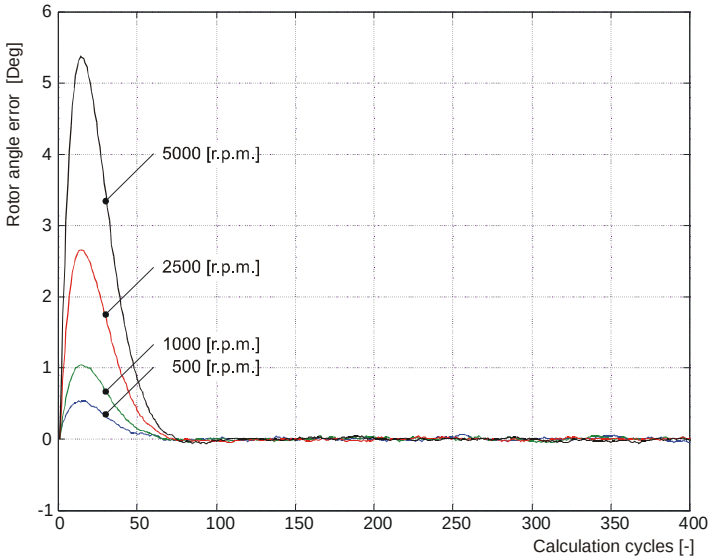

Figure 6-28. Transient Response of Rotor Angle Estimation Error - $\omega_n = 1200 \text{rad s}^{-1}$, $\zeta = 0.84$.

The effect of signal noise and limited accuracy of the signal measurement on the final accuracy of the speed estimation is shown in **Figure 6-29**.

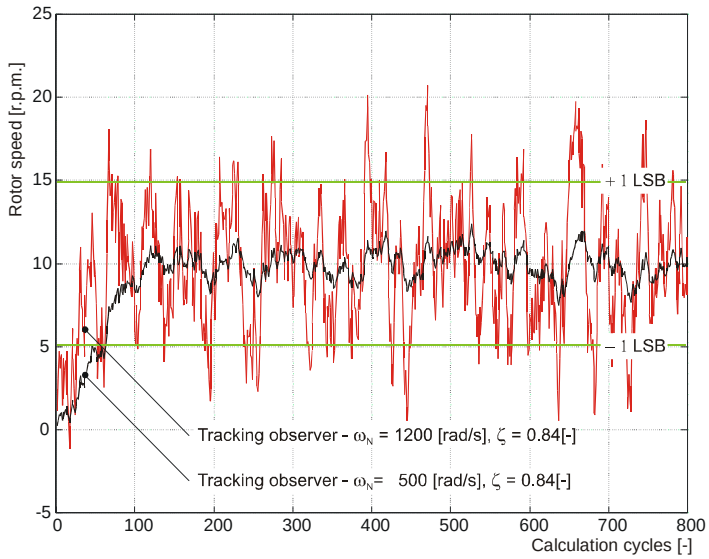

Figure 6-29. Effect of the 8-bit ADC Accuracy on the Accuracy of Rotor Speed Estimation.

This figure demonstrates the smoothing capability of the Observer algorithm. Note that Observer based on parameters of the general second-order system $\omega_n = 500 \text{rad s}^{-1}$, $\zeta = 0.84$ lead to speed estimations within allowable tolerance ± 1 LSB $(\pm 0.1\%)$. Note that this tolerance is equal to the speed measurement with 10-bit resolution performed in the speed range $0 < n < 5000$ r.p.m.

While in contrast, the Observer based on parameters $\omega_n = 1200 \text{rad}\,\text{s}^{-1}$, $\zeta = 0.84$ provides speed estimations exceeding these limits.

The following section focuses on the demonstration of the dynamic behavior and accuracy of the *Angle Tracking Observer* driven by real resolver output signals. It will demonstrate the DSP56F805EVM capability of driving motor with concurrent estimation of rotor angle and speed.

6.3 Test of the Resolver Driver in a Motor Control Application

The resolver driver and hardware interface were tested in a real *Permanent Magnet Synchronous Motor*[1] (PMSM) vector control application. The hollow shaft resolver[2] and incremental encoder[3] were both mounted on the PMSM shaft, which gave us the unique capability to perform measurements of the rotor angle and speed using two independent sensors. The PMSM application was created using the Embedded SDK (Software Developer Kit) - a set of powerful libraries supporting design of DSP applications.

The various tasks which are generally needed to run motor control application were executed in this application. The application handled the following tasks:

1. Calculation of the motor control algorithm, which provided for the generation of a stator voltage vector independently and in quadrature to the vector of the rotor magnetic flux.
2. Generating the switching pulses for the 3-Phase power stage. The software exploits a powerful PWM module of the DSP. This module is used here to produce three complementary, individually programable, PWM signal outputs. Complementary operation permits programmable dead-time insertion, distortion correction through current sensing by software and separate top and bottom output polarity control.
3. Measurement and evaluation of encoder signals using the internal Quadrature Decoder and Quad Timer module. These modules provides for accurate measurement of the angular position, speed and number of revolutions. The measurement of angular speed and number of revolutions is carried out in 16-bit resolution.
4. The excitement of the resolver rotor winding is achieved through the Timer output. The timer automatically generates a square-wave signal that is synchronized with the motor PWM pulses (see **Section 4.1.3**). This signal is then fed through an external filter, which filters out higher harmonics and produces a sine waveform suitable for resolver excitation.
5. Measurement of the resolver output signals and estimation of the rotor angle and speed. The measurements were carried out using the ADC module that was synchronized with the generation of the PWM pulses (see **Section 4.1.1**). After completing the measurements, the DSP calculates new estimations of the rotor angle and speed on the base of measured resolver signals.
6. PC master software application[4] communication support.

1. TGDrives, Type SBL3-0065-30-310/T0PS2X, nominal torque 0.65 Nm, nominal speed 300 r.p.m., nominal voltage 190 V, nominal current 1.05 A.
2. ATAS, Type ER5Kd 286, a two pole resolver with an electrical error ±10′, transformation ratio 0.5 ± 10%, supply voltage 7 V, max. current 50mA.
3. INDUcoder, Type ES 38-6-1024-05-D-S/I, resolution 4096 edges per shaft turn, 5 VDC, RS 422 line driver outputs.
4. PC master software application is the debugging tool delivered together with Motorola's SDK.

Two types of experimental tests were carried out:

- First, the noise of the rotor angle estimation was measured and post-processed in order to reveal some dependencies.
- Second, dependence of the rotor angle estimations on the instantaneous rotor position (resolver's sine and co-sine output waveforms) was analyzed.

Figure 6-30 shows how the noise magnitude of the rotor angle estimations depends on the coefficients of the *Angle Tracking Observer*. Note that lower values of the *Natural Frequency* suppress noise and increase the smoothing capability of the observer. This figure clearly demonstrates that the *Angle Tracking Observer* approach provides for smoother estimations than the *Trigonometric* approach (red curve).

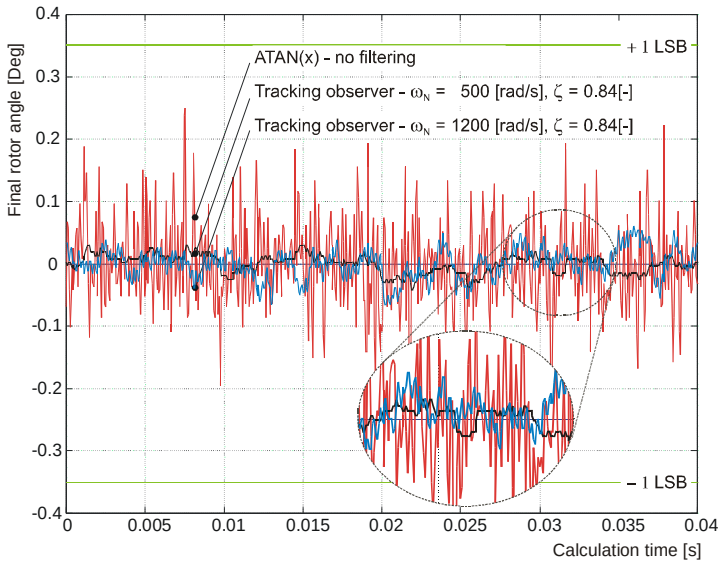

Figure 6-30. Noise of the Rotor Angle Estimation - Effect of Angle Tracking Observer Coefficients.

Figure 6-31 depicts the dependence of the estimation error of the rotor angle, denoted in degrees, on the instantaneous rotor position (resolver's sine and co-sine output waveforms). These curves were captured for two sets of the observer coefficients at motor speed 2500 r.p.m. It is evident that estimation errors are practically independent of the coefficients of the *Angle Tracking Observer*.

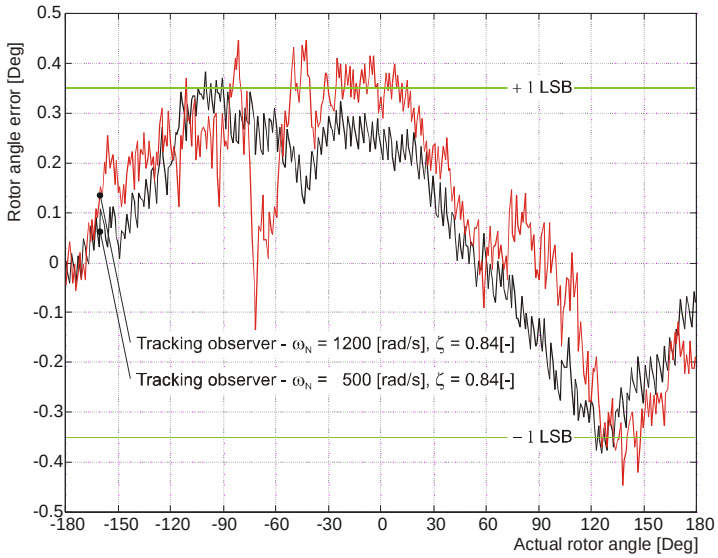

Figure 6-31. Error and Noise of the Rotor Angle Estimation - Effect of the Actual Rotor Angle.

Note that estimation errors vary somewhat with the rotor angle. These imperfections are mainly formed from two sources.

- The first source are undoubtedly imperfections in the two-pole resolver. The accuracy of the exploited resolver, guaranteed by the manufacturer, is ± 0.15 electrical degree.
- The second source of errors is imperfections in the calibration of the external conditioning hardware (see **Section 5.**) and ADC integral nonlinearity.

The smoothing feature of the *Angle Tracking Observer* in terms of speed estimation is shown in **Figure 6-32**. The figure shows waveforms of speed estimation, carried out at standstill mode for two sets of the observer coefficients. Note that the *Angle Tracking Observer*, whose coefficients were designed using $\omega_n = 500 \text{rad s}^{-1}$, $\zeta = 0.84$, performed an adequate smoothing of the estimated speed.

On the contrary, the *Angle Tracking Observer* based on parameters $\omega_n = 1200 \text{rad s}^{-1}$, $\zeta = 0.84$ performed inadequate smoothing; some values lie outside the allowable range. The term adequate smoothing means here that all estimations of the rotor speed lay within the allowable range ± 1 LSB $(\pm 0.1\%)$.

As shown in the magnified part of the **Figure 6-32**, the observer can estimate satisfactory speed waveforms in terms of accuracy. Nevertheless, it is also apparent that estimated waveforms contain higher harmonics, which would never occur in reality due to the certain mechanical time constants of the real rotor. These harmonic components arise because of the substantial sensitivity of the Observer speed estimation loop to the inaccuracies in the signal measurement. This sensitivity, however, must naturally be high in order to ensure fast convergence of the position estimation process. Note that too high sensitivity of the speed loop (pole is too far in the left half plane) results in a large observer coefficient which may cause saturation problems, instability and observer bandwidth increases. The bandwidth increases can cause a noise problem. Hence, proper judgement has to be used by the designer [5].

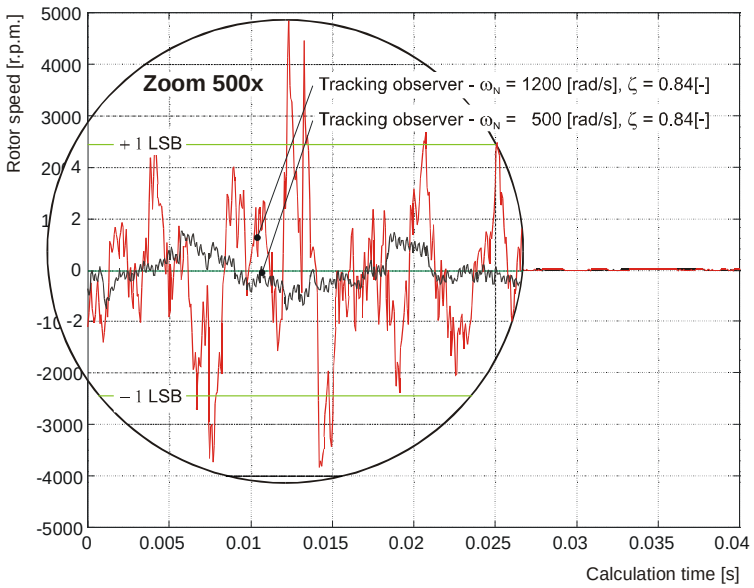

Figure 6-32. Noise of the Rotor Speed Estimation - Effect of Angle Tracking Observer Coefficients.

Note that ordinarily an additional filtering of the rotor speed estimates is mostly incorporated in the real application whenever achieving smoother waveforms is crucial without compromise on the dynamic of the position estimation.

The accuracy and smoothing of estimated variables may further be refined utilizing automatic calibration of inaccuracies in the external hardware interface, a more accurate resolver sensor and tailoring observer coefficients. The approach of selecting optimal observer coefficients is given in **Section 3.3.3**. Even more, **Figure 3-11** and **Figure 3-12** graphically illustrate variations of the *Peak Overshoot* and *Settling Time* in terms of the *Natural Frequency* and *Damping Factor* of the observer.

7. Conclusion

Hollow Shaft Resolvers are position sensors with high resistance to distortion, mechanical shocks and vibrations, deviations in operating temperature and dust in a wide range with practically unlimited durability. Among further advantages are low price, simple assembly and easy maintenance. In the case of two pole resolvers the absolute position is given immediately after starting up. Thanks to resolvers' advantages and their mass production, they are presently used in numerous electric drive applications. Their popularity is demonstrated by the remarkable growth of their worldwide application, starting from thousands of pieces in 1990 to millions pieces at the present.

It was demonstrated that Motorola DSP56F805, together with the resolver hardware and software interface, allows users to fully utilize resolver features and also permits cost reductions of final applications by achieving a single chip solution. The system design is simplified utilizing the versatile functionality of the Quad Timer and ADC on-chip modules together with a powerful DSP core.

The Motorola 40 Mips DSP56F80x family provides enough computational power to perform Resolver-to-Digital conversion in parallel to sophisticated digital control algorithms, like the demonstrated PMSM vector control application. The computational load by the resolver driver is approximately 7.5%; the rest is fully available for the user application.

Besides the presented features, the DSP56F80x family offers functions that satisfy a variety of motor control applications in terms of computational power, PWM generation, ADC, timers, communication modules, etc. The reliability and safety is maintained at a high degree by features like PWM fault protection and detection of Low-Voltage, Loss of PLL Lock and Loss of PLL Clock.

8. References

[1.] Petr Kohl, Tomas Holomek, Hollow Shaft Resolvers - Modern Position Sensors, ATAS Elektromotory a.s. Nachod, project report, 2000

[2.] Don Morgan - Tracking Demodulation - Embedded Systems Programming, n. 1, January 2001, pp. 115-120.

[3.] Norman S. Nise. - Control System Engineering, The Benjamin Icummings Publishing Company, California State Polytechnic University, 1995

[4.] Mohammed S. Santina, Allen R. Stubberud, Gene H. Hostetter - Digital Control System Design - Chapter 5 “Digital Observer and Regulator Design”, International Thomson Publishing, 1994

[5.] Bahram Shahian, Nichael Hassul - Control System Design Using Matlab - Chapter 8.3 “Observer Design”, Prentice Hall, Englewood Cliffs, 1993

Appendix A: Basic Code Example

The C programming language is the language of modern embedded system design. In this appendix, we present the C source code of the simple application that distinctly demonstrates proper use of the resolver driver functions. It also shows a static and dynamic initialization of the DSP on-chip peripherals using SDK 2.4 and CodeWarrior 4.0.

A.1 Module - MAIN.C

```
/*********************************************************************
 *                                                                   *
 * Motorola Inc.                                                     *
 * (c) Copyright 2001 Motorola, Inc.                                 *
 * ALL RIGHTS RESERVED.                                              *
 *                                                                   *
 *********************************************************************
 *                                                                   *
 * File Name: MAIN.C                                                 *
 *                                                                   *
 * Description: Main module of the resolver simple example           *
 *                                                                   *
 *********************************************************************/
#include "port.h"
#include "bsp.h"
#include "dspfunc.h"
#include "io.h"
#include "pwm.h"
#include "resolver.h"
#include "hwinit.h"
#include "arch.h"

/* exported function prototype(s)                                    */
void pwmCallbackFcn  (void);
void adcCallbackFcn  (void);

/* static variable definition(s)                                     */
static pwm_sCallback            pwmCB = {{pwmCallbackFcn},{NULL}};
static pwm_sComplementaryValues pwmVal; /* reload values of PWM     */
static Frac16   speed;                  /* resolver speed           */
static Frac16   position;               /* resolver position        */
static Frac16   turns;                  /* number of revolutions    */
static Frac16   sinSample;              /* scaled sine sample       */
static Frac16   cosSample;              /* scaled co-sine sample    */

/* main function of the module                                       */
void main (void)
{
    /* initialize DSP on-chip peripherals                            */
    Init_PWM (&pwmCB, &pwmVal);
    Init_ADC ();
    Init_TC2 ();
    Init_TD1 ();

    /* initialize resolver driver & enable all interrupts            */
    initTrackObsv ();
    archEnableInt ();

    /* main loop of the programme                                    */
    while (true){
        position = getResPosition    ();
        turns    = getResRevolutions ();

        /* angular speed returned by getResSpeed() function is       */
        /* 32-bit fractional value in range <-1; 1) that equals      */
        /* to the actual angular speed <-Fs*30; Fs*30)[RPM], where    */
        /* Fs is sampling frequency.                                 */
        /* In this example Fs = 8kHz; i.e., the angular speed        */
        /* varies from -240000 to 240000[RPM]. To transform this     */
        /* value to 16-bit range <-1, 1), corresponding to -5000 to  */
        /* 5000[RPM], the software must perform multiplication of    */
        /* the returned 32-bit value by: 240000/5000 => 0.75*2^6     */
        speed = round (L_shl (L_mult_ls (getResSpeed (), FRAC16(0.75)), 6));
    }
}
```

```
/*********************************************************************
 *                                                                   *
 * Module:      pwmCallbackFcn()                                     *
 *                                                                   *
 * Description: Callback fucntion of the PWM A unit - called at      *
 *              every second reload occurs                           *
 *                                                                   *
 * Returns:     none                                                 *
 *                                                                   *
 * Arguments:                                                        *
 *              inp(s): none                                         *
 *                                                                   *
 *                                                                   *
 *              out(s): none                                         *
 *                                                                   *
 * Range Issues:    none                                             *
 *                                                                   *
 * Special Issues:  none                                             *
 *                                                                   *
 *********************************************************************/
void pwmCallbackFcn (void)
{
    /* update pwm reload registers by new values                     */
    pwmVal.pwmChannel_0_Value = 0x7e00;
    pwmVal.pwmChannel_2_Value = 0x7e00;
    pwmVal.pwmChannel_4_Value = 0x7e00;

    /* update the actual contents of the value registers, fills      */
    /* these registers and sets Load OK bit at the end               */
    pwmIoctl (pwmFD, PWM_UPDATE_VALUE_REGS_COMPL, &pwmVal, BSP_DEVICE_NAME_PWM_A);

    /* clears reload interrupt flag                                  */
    pwmIoctl (pwmFD, PWM_CLEAR_RELOAD_FLAG, NULL, BSP_DEVICE_NAME_PWM_A);
}

/*********************************************************************
 *                                                                   *
 * Module:      adcCallbackFcn()                                     *
 *                                                                   *
 * Description: Callback fucntion of the ADC A unit - called at      *
 *              every end of scan interrupt occurs                   *
 *                                                                   *
 * Returns:     none                                                 *
 *                                                                   *
 * Arguments:                                                        *
 *              inp(s): none                                         *
 *                                                                   *
 *                                                                   *
 *              out(s): none                                         *
 *                                                                   *
 * Range Issues:    none                                             *
 *                                                                   *
 * Special Issues:  none                                             *
 *                                                                   *
 *********************************************************************/
void adcCallbackFcn (void)
{
    Frac16  sin, cos;

    /* read analog sine and co-sine signals                          */
    sin = periphMemRead (&ArchIO.AdcA.ResultReg[0]);
    cos = periphMemRead (&ArchIO.AdcA.ResultReg[4]);

    /* scaling of sin and cos samples to full range -1 to 1          */
    /* sin = sin * sinScaleMant * 2^sinScaleExp                      */
    /* cos = cos * cosScaleMant * 2^cosScaleExp                      */
    sinSample = mult_r (sin, SIN_SCALE_MANT) << SIN_SCALE_EXP;
    cosSample = mult_r (cos, COS_SCALE_MANT) << COS_SCALE_EXP;

    /* calculate actual position and speed                           */
    calcTrackObsv (sinSample, cosSample);
}
/*********************************************************************
 *End of module                                                      *
 *********************************************************************/
```

A.2 Module - HWINIT.C

```
/*****************************************************************
 *                                                               *
 * Motorola Inc.                                                 *
 * (c) Copyright 2001 Motorola, Inc.                             *
 * ALL RIGHTS RESERVED.                                          *
 *                                                               *
 *****************************************************************
 *                                                               *
 * File Name: HWINIT.C                                           *
 *                                                               *
 * Description: Module with initialization function of the on-chip  *
 *              resources (hardware modules)                     *
 *                                                               *
 *****************************************************************/
#include "port.h"
#include "bsp.h"
#include "dspfunc.h"
#include "io.h"
#include "pwm.h"
#include "adc.h"
#include "quadraturetimer.h"
#include "resolver.h"
#include "hwinit.h"

/* exported variable definition(s)                               */
int pwmFD;          /* descriptor of the PWM                     */
int adcFD[2];       /* descriptor of the ADC                     */
int tc2FD;          /* descriptor of the Quad Timer C2           */
int td1FD;          /* descriptor of the Quad Timer D1           */

/*****************************************************************
 *                                                               *
 * Module:      Init_PWM()                                       *
 *                                                               *
 * Description: Function initializes pwmA on-chip peripheral.    *
 *                                                               *
 * Returns:     none                                             *
 *                                                               *
 * Arguments:                                                    *
 *              inp(s): pSC - pointer to the initialized         *
 *                      pwm_sCallback data structure that is     *
 *                      defined in the pwm.h header file         *
 *                                                               *
 *                      pVal- pointer to the                     *
 *                      pwm_sComplementaryValues data structure  *
 *                      that is defined in the pwm.h header file *
 *                                                               *
 *              out(s): none                                     *
 *                                                               *
 * Range Issues:    none                                         *
 *                                                               *
 * Special Issues:  none                                         *
 *                                                               *
 *****************************************************************/
void Init_PWM (pwm_sCallback *pSC, pwm_sComplementaryValues *pVal)
{
    /* open & install PWM driver - more info about current setting  */
    /* is given in appconfig.h, section dedicated to PWM support    */
    pwmFD = open (BSP_DEVICE_NAME_PWM_A, 0);

    /* install user callback function                               */
    pwmIoctl (pwmFD, PWM_SET_RELOAD_CALLBACK, pSC, BSP_DEVICE_NAME_PWM_A);


    /* enables reload interrupt                                     */
    pwmIoctl (pwmFD, PWM_RELOAD_INTERRUPT, PWM_ENABLE, BSP_DEVICE_NAME_PWM_A);

    /* sets 50% duty cycle for all channels                         */
    pVal->pwmChannel_0_Value = 0x4000;
    pVal->pwmChannel_2_Value = 0x4000;
    pVal->pwmChannel_4_Value = 0x4000;

    /* update the actual contents of the value registers, fills     */
    /* these registers and sets Load OK bit at the end              */
    pwmIoctl (pwmFD, PWM_UPDATE_VALUE_REGS_COMPL, pVal, BSP_DEVICE_NAME_PWM_A);
}
```

```
/*********************************************************************
 *                                                                   *
 * Module:      Init_ADC()                                           *
 *                                                                   *
 * Description: Function initializes ADC on-chip peripheral.         *
 *                                                                   *
 * Returns:     none                                                 *
 *                                                                   *
 * Arguments:                                                        *
 *              inp(s): none                                         *
 *                                                                   *
 *              out(s): none                                         *
 *                                                                   *
 * Range Issues:    none                                             *
 *                                                                   *
 * Special Issues:  none                                             *
 *                                                                   *
 *********************************************************************/
void Init_ADC (void)
{
  static const adc_sState sAdc[2] =
    {
        {
            /* AnalogChannel    = */  ADC_CHANNEL_6,    /* SINE     */
            /* SampleMask       = */  0x0001,           /* Sample 0 */
            /* OffsetRegister   = */  0x3ffc,
            /* LowLimitRegister = */  0,
            /* HighLimitRegister= */  0,
            /* ZeroCrossing     = */  0,
        },
        {
            /* AnalogChannel    = */  ADC_CHANNEL_7,    /* CO-SINE  */
            /* SampleMask       = */  0x0010,           /* Sample 4 */
            /* OffsetRegister   = */  0x4090,
            /* LowLimitRegister = */  0,
            /* HighLimitRegister= */  0,
            /* ZeroCrossing     = */  0,
        }
    };

    /* open respective adc channels                                  */
    adcFD[0] = open (BSP_DEVICE_NAME_ADC_0, 0, &sAdc[0]);
    adcFD[1] = open (BSP_DEVICE_NAME_ADC_0, 0, &sAdc[1]);
}

/*********************************************************************
 *                                                                   *
 * Module:      Init_TC2()                                           *
 *                                                                   *
 * Description: Function initializes TC2 timer (on-chip peripheral) *
 *              that is used for the synchronization of ADC with     *
 *              PWMA unit.                                           *
 *                                                                   *
 * Returns:     none                                                 *
 *                                                                   *
 * Arguments:                                                        *
 *              inp(s): none                                         *
 *                                                                   *
 *              out(s): none                                         *
 *                                                                   *
 * Range Issues:    none                                             *
 *                                                                   *
 * Special Issues:  none                                             *
 *                                                                   *
 *********************************************************************/
void Init_TC2 (void)
{
    static const qt_sState paramTC2 =
    {
        /* Mode = */                    qtTriggeredCount,
        /* InputSource = */             qtPrescalerDiv1,
        /* InputPolarity = */           qtNormal,
        /* SecondaryInputSource = */    qtSISCounter2Input,

        /* CountFrequency = */          qtRepeatedly,
        /* CountLength = */             qtUntilCompare,
        /* CountDirection = */          qtUp,

        /* OutputMode = */              qtDeassertOnSecondary,
        /* OutputPolarity = */          qtNormal,
        /* OutputDisabled = */          false,

        /* Master = */                  false,
        /* OutputOnMaster = */          false,
        /* CoChannelInitialize = */     false,
        /* AssertWhenForced = */        false,
```

```
        /* CaptureMode = */             qtDisabled,

        /* CompareValue1 = */           C2_DELAY,
        /* CompareValue2 = */           0,
        /* InitialLoadValue = */        0x0000,

        /* CallbackOnCompare = */       { NULL, NULL },
        /* CallbackOnOverflow = */      { NULL, NULL },
        /* CallbackOnInputEdge = */     { NULL, NULL }
    };

    /* open and install Quad Timer C2 - synchronization PWM to ADC  */
    tc2FD = open (BSP_DEVICE_NAME_QUAD_TIMER_C_2, 0, NULL);

    /* start of the Quad Timer C2                                   */
    ioctl (tc2FD, QT_ENABLE, (void*)&paramTC2);
}

/*********************************************************************
 *                                                                   *
 * Module:      Init_TD1()                                           *
 *                                                                   *
 * Description: Function initializes TD1 timer (on-chip peripheral)  *
 *              that is used for the generation of the reference     *
 *              voltage for the resolver reference winding.          *
 *                                                                   *
 * Returns:     none                                                 *
 *                                                                   *
 * Arguments:                                                        *
 *              inp(s): none                                         *
 *                                                                   *
 *              out(s): none                                         *
 *                                                                   *
 * Range Issues:    none                                            *
 *                                                                   *
 * Special Issues:  none                                            *
 *                                                                   *
 *********************************************************************/
void Init_TD1 (void)
{
    static const qt_sState paramTD1 =
    {
        /* Mode = */                    qtCount,
        /* InputSource = */             qtPrescalerDiv1,
        /* InputPolarity = */           qtNormal,
        /* SecondaryInputSource = */    qtSISCounter0Input,

        /* CountFrequency = */          qtRepeatedly,
        /* CountLength = */             qtUntilCompare,
        /* CountDirection = */          qtUp,

        /* OutputMode = */              qtToggleOnCompare,
        /* OutputPolarity = */          qtNormal,
        /* OutputDisabled = */          false,

        /* Master = */                  false,
        /* OutputOnMaster = */          false,
        /* CoChannelInitialize = */     false,
        /* AssertWhenForced = */        false,

        /* CaptureMode = */             qtDisabled,

        /* CompareValue1 = */           D1_DELAY,
        /* CompareValue2 = */           0,
        /* InitialLoadValue = */        0,

        /* CallbackOnCompare = */       { NULL, NULL },
        /* CallbackOnOverflow = */      { NULL, NULL },
        /* CallbackOnInputEdge = */     { NULL, NULL }
    };

    /* open and install Quad Timer D1 - gneration of reference      */
    /* signal for the resolver                                      */
    td1FD = open (BSP_DEVICE_NAME_QUAD_TIMER_D_1, 0, NULL);

    /* connect timer output to the external pin                     */
    ioctl (td1FD, QT_ENABLE_OUTPUT, (void *)&paramTD1);

    /* Sequence to synchronize PWM with timer:                      */
    /* - first clear reload flag and wait until reload              */
    /* - second run timer and wait until compare                    */
    /* - set timer compare register 1 for generation of the square  */
    /*   wave reference signal                                      */
    pwmIoctl (pwmFD, PWM_CLEAR_RELOAD_FLAG, NULL, BSP_DEVICE_NAME_PWM_A);
    while (!(pwmIoctl (pwmFD, PWM_READ_CONTROL_REG, NULL,
        BSP_DEVICE_NAME_PWM_A) & 0x0010));
```

```
    ioctl (td1FD, QT_ENABLE, (void *)&paramTD1);
    while (!(ioctl (td1FD, QT_GET_STATUS, (void *)&paramTD1) & 0x8000));

    ioctl (td1FD, QT_WRITE_COMPARE_VALUE1, D1_PERIOD-1);
}
/*********************************************************************
 *End of module                                                      *
 *********************************************************************/
```

A.3 Module - HWINIT.H

```
/*********************************************************************
 *                                                                   *
 * Motorola Inc.                                                     *
 * (c) Copyright 2001 Motorola, Inc.                                 *
 * ALL RIGHTS RESERVED.                                              *
 *                                                                   *
 *********************************************************************
 *                                                                   *
 * File Name: HWINIT.H                                               *
 *                                                                   *
 * Description: Header file of the hwinit.c module                   *
 *                                                                   *
 *********************************************************************/
#ifndef __HWINIT_H__
    #define __HWINIT_H__

    /* exported data prototypes - descriptors of opened on-chip     */
    /* hardware modules                                             */
    extern int  pwmFD, adcFD[2], tc2FD, td1FD;

    /* exported function prototypes                                 */
    extern void Init_PWM (pwm_sCallback *pSC, pwm_sComplementaryValues *pVal);
    extern void Init_ADC (void);
    extern void Init_TC2 (void);
    extern void Init_TD1 (void);

#endif
```

A.4 Module - APPCONFIG.H

```
/*********************************************************************
 *                                                                   *
 * Motorola Inc.                                                     *
 * (c) Copyright 2001 Motorola, Inc.                                 *
 * ALL RIGHTS RESERVED.                                              *
 *                                                                   *
 *********************************************************************
 *                                                                   *
 * File Name: APPCONFIG.H                                            *
 *                                                                   *
 * Description: Module with the definition of on-chip resources and *
 *              constants for their static initialization            *
 *                                                                   *
 *********************************************************************/
#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*
*********************************************************************
Include needed SDK components below by changing #undef to #define.
Refer to ../config/config.h for complete list of all components and
component default initialization.
*********************************************************************
*/
#define INCLUDE_BSP           /* BSP support - includes SIM,COP...  */
#undef  INCLUDE_3DES          /* 3des library                       */
#define  INCLUDE_ADC          /* ADC support                        */
#undef  INCLUDE_AEC           /* AEC library                        */
#undef  INCLUDE_BLDC          /* BLDC library                       */
#undef  INCLUDE_BUTTON        /* Button support                     */
#undef  INCLUDE_CALLER_ID     /* CallerID library                   */
#undef  INCLUDE_CAN           /* CAN support                        */
#undef  INCLUDE_CAS_DETECT    /* CAS_detect library                 */
#undef  INCLUDE_COP           /* COP support (subset of BSP)        */
#undef  INCLUDE_CORE          /* CORE support (subset of BSP)       */
#undef  INCLUDE_CPT           /* CPT library                        */
#undef  INCLUDE_DAC           /* DAC support                        */
#undef  INCLUDE_DECODER       /* Quadrature Decoder support         */
#undef  INCLUDE_DES           /* DES library                        */
#define  INCLUDE_DSPFUNC      /* DSP Function library               */
#undef  INCLUDE_DTMF_DET      /* DTMF detect library                */
#undef  INCLUDE_DTMF_GEN      /* DTMF generation library            */
#undef  INCLUDE_FILEIO        /* File I/O support                   */
```

```
#undef  INCLUDE_FLASH         /* Flash support                        */
#undef  INCLUDE_G165          /* G165 vocoder library                 */
#undef  INCLUDE_G711          /* G711 vocoder library                 */
#undef  INCLUDE_G726          /* G726 vocoder library                 */
#undef  INCLUDE_GPIO          /* General Purpose I/O support          */
#define  INCLUDE_IO           /* I/O support                          */
#undef  INCLUDE_ITCN          /* ITCN support (subset of BSP)         */
#undef  INCLUDE_LED           /* LED support for target board         */
#undef  INCLUDE_MCFUNC        /* Motor Control functional library     */
#undef  INCLUDE_MEMORY        /* Memory support                       */
#define  INCLUDE_PCMASTER     /* PC Master application support        */
#define  INCLUDE_PLL          /* PLL support (subset of BSP)          */
#define  INCLUDE_PWM          /* PWM support                          */
#define  INCLUDE_QUAD_TIMER   /* Quadrature timer support             */
#undef  INCLUDE_SCI           /* SCI support                          */
#undef  INCLUDE_SIM           /* SIM support (subset of BSP)          */
#undef  INCLUDE_SPI           /* SPI support                          */
#undef  INCLUDE_SRM           /* Switch Relactance library            */
#undef  INCLUDE_STACK_CHECK   /* Stack utilization routines           */
#undef  INCLUDE_SWITCH        /* Switch support                       */
#undef  INCLUDE_TIMER         /* Timer support                        */
#undef  INCLUDE_VAD           /* VAD library                          */
#undef  INCLUDE_V8BIS         /* V8bis library                        */
#undef  INCLUDE_V22           /* V22 library                          */
#undef  INCLUDE_V42BIS        /* V42bis library                       */


/*
**************************************************************************
 PLL support
**************************************************************************
*/
/* defines ZCLK = 80MHz, IPBus clock = 40MHz                            */
#define BSP_OSCILLATOR_FREQ     8000000L
#define PLL_MUL                 20

/*
**************************************************************************
 PWM support
**************************************************************************
*/
#define PWM_EXCLUDE_PWM_B   /* PWM B is not used                        */

/* PWM period = 16kHz - IPBus @40MHz                                    */
/* Dead time 1.0us, center-aligned PWM selected                         */
/* Sets complementary PWM operation                                     */
/* Reload PWM at every two oportunities                                 */
/* Half cycle reload disabled                                           */
/* PWM value register 0 selected for PWM0-1 in the next PWM cycle       */
/* PWM value register 2 selected for PWM2-3 in the next PWM cycle       */
/* PWM value register 4 selected for PWM4-5 in the next PWM cycle       */
/* PWM clock frequency = IP_BUS_CLOCK/1                                 */
/* PWMF CPU interrupt request disabled                                  */
/* PWM generator and PWM pins enabled                                   */
/* Interrupt priority is set to default value (1)                       */
/* Output pads enabled                                                  */
#define PWM_A_CONTROL_REG               0x1001
#define PWM_A_COUNTER_MODULO_REG        1250
#define PWM_A_DEAD_TIME_REG             0x0029
#define PWM_A_FAULT_CONTROL_REG         0
#define PWM_A_FAULT_STATUS_REG          0
#define PWM_A_OUTPUT_CONTROL_REG        0x8000
#define PWM_A_DISABLE_MAPPING_1_REG     0
#define PWM_A_DISABLE_MAPPING_2_REG     0
#define PWM_A_CONFIG_REG                0
#define PWM_A_CHANNEL_CONTROL_REG       0

/*
**************************************************************************
 ADC support
**************************************************************************
*/
/* ADC_CLK = 5MHz - IPBus @40MHz                                        */
/* Channels AN0 and AN1 work in signle ended mode                       */
/* AN0 and AN1 measure sine and co-sine waveforms, respectively         */
/* Both channels work in single ended mode - default setting            */
/* Interrupt priority assigned to 2                                     */
#define ADC_SCANMODE                    ADC_SIMULTANEOUS_TRIGGERED
#define ADC_INITIATE_SCAN               ADC_INITIATE_SCAN_ON_SYNC
#define ADC_CLOCK_DIVISOR               3
#define INCLUDE_ADCA_SAMPLE_0           true
#define INCLUDE_ADCA_SAMPLE_4           true
#define GPR_INT_PRIORITY_55             2
#define ADC_RAW_CONVERSION_COMPLETE_CALLBACK   adcCallbackFcn
```

```
/*
*********************************************************************
 Quad Timer support
*********************************************************************
*/
#define INCLUDE_USER_TIMER_C_2          0
#define INCLUDE_USER_TIMER_D_1          0

/*
*********************************************************************
PC MASTER application support
*********************************************************************
*/
#define PC_MASTER_REC_BUFF_LEN          800
#define PC_MASTER_RECORDER_TIME_BASE    0x8032


/*
*********************************************************************
Application constants (IPBusFrequency = 40MHz)
*********************************************************************
*/
/* delay between PWM reload and ADC scan [ticks], 31.25us           */
#define   C2_DELAY      1250

/* delay between PWM reload and edge of ref. signal [ticks], 54.8us */
#define   D1_DELAY      2194

/* PWM frequency = 16 kHz                                           */
#define   PWM_MODULO    1250

/* reference signal period divided by 2 [ticks], 1/8000Hz/2 = 62.5us*/
#define   D1_PERIOD     PWM_MODULO+PWM_MODULO

/* Sine/Co-sine waveform scaling coefficients - depends on resolver
type and particular hardware interface                           */
#define SIN_SCALE_MANT  FRAC16(0.490120)
#define SIN_SCALE_EXP   2
#define COS_SCALE_MANT  FRAC16(0.499516)
#define COS_SCALE_EXP   2

#endif
```


How to reach us:
USA/EUROPE/Locations Not Listed: Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado 80217. 1–303–675–2140 or 1–800–441–2447

JAPAN: Motorola Japan Ltd.; SPS, Technical Information Center, 3–20–1, Minami–Azabu. Minato–ku, Tokyo 106–8573 Japan. 81–3–3440–3569

ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd.; Silicon Harbour Centre, 2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong. 852–26668334

Technical Information Center: 1–800–521–6274

HOME PAGE: http://www.motorola.com/semiconductors/