Texas Instruments

Engine Knock Detection Using Spectral Analysis Techniques With a TMS320 DSP

Application Report

1995 Digital Signal Processing Products

[redacted]

Texas Instruments

Printed in U.S.A., August 1995

SPRA039

Engine Knock Detection Using Spectral Analysis Techniques With a TMS320 DSP

Thomas G. Horner, Member of Group Technical Staff
Digital Signal Processor Systems—Semiconductor Group

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Contents

INTRODUCTION 1
What Is Engine Knock? 1
Knock Sensors 1
Direct Measurements 1
Remote Measurements 1
Knock Detection Overview 2
Spectral Signature 2
Adaptation Requirements 2
Signal Conditioning 2
Detection Strategies 3
Control Strategies 4

DFT-BASED DETECTION METHOD 4
Signal Conditioning 5
Detection Strategy 8
Adaptation Strategy 12
Implementation Examples 13
Timing Considerations 13
Frequency Selection 15
Analog-to-Digital Converter (ADC) Resolution 16
TMS320C30 Implementation 16
Hardware Description 16
Software Description 17
Test Results 18
TMS320C25 Implementation 18
Hardware Description 19
Software Description 19
Test Results 20
Integration Road Map 20

MISFIRE DETECTION 21

REFERENCES 22

Appendixes

Appendix A: Sample-Rate and Block-Size Search Software 23

Appendix B: TMS320C30 Implementation Software 29
Description 30
Listing 30

Appendix C: TMS320C25 Implementation Software 40
Description 40
Listing 40

List of Illustrations

1 Algorithm Stages Associated With the Crankshaft Angle 14
2 TMS320C30-Based Knock Detection System Block Diagram 17
3 TMS320C30 Floating-Point DSP Knock Detection Program Structure 18
4 TMS320C25-Based Knock Detection System Block Diagram 19
5 TMS320C25 Fixed-Point DSP Knock Detection Program Structure 20
6 DSP Road Map From Knock Detection Subsystem to Full Engine Control 21

ABSTRACT

An efficient method of detecting engine knock using spectral analysis is presented. Multiple single-point DFTs are used to condition the measured knock signal. Using multiple frequencies in the detection algorithm provides a better signature of the combustion process and enhances the ability to detect low-level knock across the entire operating range of the engine. The detection strategy compares the DFT outputs to a variable reference to determine a knock intensity metric. Unlike currently used techniques, the algorithm adapts the reference (no-knock condition) to varying engine speeds and loads. An overview of the knock detection problem and current technology is presented. Implementation examples are included to aid in developing system-specific hardware and software. Two systems are presented. The first is based on the TMS320C30 floating-point DSP with software written in C. This system could be used for engine study and algorithm development. The second, which is more production oriented, is based on a TMS320C25 fixed-point DSP with software written in assembly language.

INTRODUCTION

What Is Engine Knock?

Modern engine control systems are designed to minimize exhaust emissions while maximizing power and fuel economy. The ability to maximize power and fuel economy by optimizing spark timing for a given air/fuel ratio is limited by engine knock. Detecting knock and controlling ignition timing to allow an engine to run at the knock threshold provides the best power and fuel economy. Normal combustion occurs when a gaseous mixture of air and fuel is ignited by the spark plug and burns smoothly from the point of ignition to the cylinder walls. Engine knock, or detonation, occurs when the temperature or pressure in the unburned air/fuel mixture (end gases) exceeds a critical level, causing autoignition of the end gases. This produces a shock wave that generates a rapid increase in cylinder pressure. The impulse caused by the shock wave excites a resonance in the cylinder at a characteristic frequency that is dependent primarily on cylinder bore diameter and combustion chamber temperature. Damage to pistons, rings, and exhaust valves can result if sustained heavy knock occurs. Additionally, most automotive customers find the sound of heavy engine knock objectionable.

Knock Sensors

Implementing a knock detection/control strategy requires sensors to monitor the combustion process and provide feedback to the engine controller. Knock sensors can be classified in two broad categories: direct and remote measurements.

Direct Measurements

Pressure sensors measure the pressure inside the combustion chamber of a running engine. This direct measurement of the combustion process provides the best signal to analyze to detect engine knock. However, each cylinder requires its own sensor, and individual sensor costs are still relatively high. As a result, pressure sensors are used primarily in research settings. Currently, Toyota is the only manufacturer that installs pressure sensors in production engines. Pressure sensor usage will increase in the future as sensor costs are reduced and automotive companies develop more sophisticated engine control strategies that monitor the combustion process.

Remote Measurements

Remote measurement sensors use vibrations transmitted through the structure of the engine to detect knock in the combustion chamber. The signal received by remote sensors can be contaminated by sources other

than engine knock, which increases the difficulty of signal detection. This is especially true at higher engine speeds in which background mechanical vibrations are much higher, effectively reducing the signal-to-noise ratio. One advantage of using remote sensors is that, with careful placement, only one or two sensors are required to monitor all cylinders. In addition, the sensors are less expensive, primarily due to a less harsh operating environment.

Two types of remote sensor are being used today: tuned and broadband. Tuned or resonant sensors are used in many low-end knock detection systems. Either mechanically or electronically, the sensor amplifies the magnitude of the signal in the frequency range of the knock-excited resonance (sometimes called the fundamental frequency). A limitation to this approach is that a different sensor can be required for each engine type, due to variations in the characteristic frequency. The resulting part number proliferation increases overall system costs for the manufacturer. To eliminate the cost penalty, sensor bandwidth can be made wide enough to encompass all expected variations in the fundamental frequency. However, doing so can possibly decrease system performance.

Broadband sensors have no resonant peaks below the 20-kHz operating range of the knock-detection system. One sensor works equally well for any engine configuration. Some type of postprocessing is required to identify the characteristic frequency, placing an additional burden on the signal conditioning part of the system. Since variations in the fundamental frequency can be expected for different engine configurations, a programmable solution provides the flexibility to easily modify the frequency range being monitored with minimal impact on system cost.

Knock Detection Overview

Spectral Signature

When engine knock occurs, a shock wave is generated inside the combustion chamber. The shock wave excites a characteristic frequency in the engine, which is typically in the 5 kHz–7 kHz range. Cylinder bore diameter and combustion chamber temperature are the main variables that affect this fundamental frequency. Variations in the fundamental frequency for a given engine configuration can be as much as ± 400 Hz. Larger diameters and/or lower temperatures result in a lower fundamental frequency.

Signals received by a remote sensor contain additional vibrational modes, which are structural resonances in the engine excited by the shock wave as it hits the cylinder wall. Typically, two to four additional frequency peaks are evident between the fundamental frequency and 20 kHz. Each engine structure can have different higher vibrational modes. Sensor mounting location can affect which modes are detectable and the amplitude of each with respect to the background mechanical noise.

Adaptation Requirements

An engine-knock detection algorithm must be able to adapt to a number of variables to enable the controller to generate optimum spark timing so that the engine can run at the knock threshold. As mentioned previously, the structural design of an engine and the mounting location of the knock sensor(s) affect which frequency modes are detectable by the sensor. Usually, the transfer function between the cylinder and the sensor is different for each cylinder. This causes both the relative and absolute magnitudes of the vibrational modes to be different for each cylinder. A good detection scheme should allow different calibrations for each cylinder.

Another variable that must be accounted for is changes in nonknocking (reference) signal amplitude due to the mechanical vibration of the engine at different RPMs. As the engine speed increases, the background vibration level increases. When a fixed reference is used, a compromise in performance must be made

because signal magnitudes that would indicate knock at lower engine RPMs are equal to or less than the background level at higher engine RPMs. The reference must be set low enough that knock can be detected at lower RPMs, which limits the algorithm's ability to function at higher speeds. For this reason, some knock detection systems are shut off above 4000 RPM, and very conservative spark timings are used to guarantee that knock will not occur. A good detection strategy should adapt to varying levels of background vibration levels to allow trace knock to be detected at all engine speeds.

Finally, an engine's operating characteristics change with time. As an engine wears, tolerances between components change, which could change the magnitudes of the vibrational modes detected by a remote sensor. Normal background vibrational levels could be higher for a given engine speed. The signal-to-noise ratio could decrease at higher engine speeds. A good detection strategy should adjust to changes in daily operating characteristics to allow reliable identification of trace knock without false triggers.

Signal Conditioning

Knock detection systems must perform some type of signal conditioning prior to executing the detection strategy. Information about the signal strength in the frequency range(s) excited by knock must be extracted from the measurement. If a tuned sensor with a very narrow resonant peak about the fundamental frequency is being used, no further signal conditioning is required. In all other situations, either a filtering technique (analog or digital) or a spectral estimation technique must be used.

Analog filtering is the predominant method used today, due to its low cost, ease of implementation, and lack of computational power of the engine controller CPU. The output of a simple analog filter tuned to the fundamental knock frequency is integrated and sent to the engine control unit (ECU) to execute the detection strategy. However, now that higher precision and/or multiple frequency ranges are desired, an analog implementation is becoming cost prohibitive.

Digital filters are starting to become practical as the computational performance of the ECU increases. Programmability allows the same hardware to be shared across a number of engine configurations. Reduction in part numbers can provide big cost savings to manufacturers in all steps in a product's life cycle. Enhancing filter performance or adding additional frequency ranges can be readily accomplished as long as the computational limits of the CPU are not exceeded.

Another digital signal conditioning technique is spectral analysis; for example, Fast Fourier Transform (FFT). An FFT provides a higher level of frequency resolving power than a digital filter. In addition, multiple frequency ranges are available as the basic output of the FFT. Limited computational throughput of the ECU and unfamiliarity with the technique have limited its use to research and development. No current production system uses spectral analysis techniques. The advent of cost-effective digital signal processors like TI's TMS320 fixed-point family is making the computational power available to bring spectral analysis techniques to prominence.

Detection Strategies

Knock detection strategies use the output of a signal conditioning stage to compare with a reference to determine the presence or absence of knock. Most systems today use windowing to isolate periods during the cylinder's firing cycle for analysis when knock is possible. There is a window from approximately 10° to 70° after top dead center (ATDC) of the piston's cycle when detonation is most likely to occur for the firing cylinder. The detection algorithm is run only during this window for that cylinder. By eliminating possible false trigger sources, such as valve closing, the detection algorithm is more robust. The time this window is active varies with engine speed from 20 ms at 500 RPM down to 1.25 ms at 8000 RPM. Tracking changing engine speed to calculate this time variation requires hardware or software overhead for implementation.

An indication of signal strength during the active window period is typically used by the detection algorithm. Integrating the output of a filter is a common method. The magnitude of the signal strength is compared with a reference level, which, in nonadaptive systems, must be predetermined during system development. If the reference level cannot adapt to changes in engine RPM, a compromise must be made. The reference level must be low enough to prevent sustained knock at low speed, yet high enough to prevent false triggers at higher engine speed. Today, at engine speeds above approximately 4000 RPM, a combination of very conservative spark timing maps and shutting off the control strategy is used to guarantee knock-free operation. This results in less than optimal performance and fuel efficiency at higher engine speeds, particularly for systems using only fundamental frequency detection. Even at lower engine speeds, some compromise is required to guarantee that knock is likely occur only during transient operation.

The tradeoffs between using only the fundamental frequency or the combination of fundamental and vibration mode frequencies concern the issues of false triggers vs. complexity, available CPU time, and cost. When multiple frequencies are used, a better signature is available to determine if knock is present. This effectively increases the signal-to-noise ratio of the system. As a result, either the RPM range for reliable detection can be extended or the baseline spark timing at lower engine speeds can be advanced to allow the engine to continuously run closer to the knock threshold.

Control Strategies

Knock control strategies today adjust spark timing to let an engine run at the knock threshold. Look-up tables are used to obtain a baseline setting for a given speed, load, and temperature. Based on the level of knock detected, timing can be advanced (no knock) or retarded (knock). The rate of advance or retardation can also be modified based on knock magnitude and/or offset from the baseline spark timing setting.

The strategies fall into two categories. The simplest strategy, global control, retards the spark timing of all cylinders by the same amount when any knock is detected. This approach has the advantage that only one knock value has to be tracked and only one timing control loop needs to be executed. The computational burden on the CPU is minimized, as are the memory requirements for both data and program. However, engine performance can be compromised if only one or two cylinders are more likely to knock at a given operating condition. Since all cylinders' spark timing is retarded equally, the cylinders not at their knock threshold are not providing optimal power and/or fuel efficiency.

A more sophisticated strategy is to control each cylinder individually so that all cylinders are running at the knock threshold and provide the best power and fuel efficiency. To implement this type of control strategy, knock computations and control updates must be performed individually for each cylinder. The computational burden and memory requirements are higher than with the global control strategy.

Both control strategies are used today. As automotive manufacturers attempt to improve the emissions, power, and fuel efficiency of their engines to meet the competition or government regulations, the individual cylinder control strategy will become predominant. Advanced engine control strategies are being developed that use torque and/or combustion feedback to optimize the operation of each cylinder independently for best spark timing, fuel delivery, and possibly valve timing. When these strategies are implemented, global control will no longer be a viable alternative.

DFT-BASED DETECTION METHOD

An individual cylinder knock detection method has been developed that offers superior performance compared with analog bandpass filtering, digital bandpass filtering, or FFT techniques. This new method uses multiple single-point Discrete Fourier Transforms (DFTs) for the signal conditioning step to monitor

the fundamental frequency plus the vibrational modes of an engine. The DFT algorithm provides better frequency discrimination than low-cost analog filters, provides better frequency discrimination and/or less computational burden on the CPU than digital filters, and is less computationally intensive than an FFT. In addition, because the computation of the DFT algorithm can be spread over all the samples in a block of data, it leaves more time to run the detection algorithm before the next cylinder's data must be processed.

The DFT algorithm can be cost effectively implemented on a TI fixed-point DSP, and that system can replace currently used lower-performance knock detection systems. As a result of the frequency discrimination capability of the DFT and the use of multiple frequency ranges, the effective signal-to-noise ratio is enhanced. In addition, the computational throughput of the DSP allows the reference signal to adapt in real time to changing operating conditions, such as engine speed. At lower speeds, spark timing can be advanced closer to the knock threshold because the reference is adjusted to the lower background vibration levels. It is also possible to resolve a knock signature at higher engine speeds, due to both the multiple frequency ranges, which give a more distinctive signal, and the adaptive reference, which reduces false triggers. This ability allows the variable spark timing of the engine to run closer to the knock threshold to obtain improved power and fuel efficiency.

The following sections present the details of the DFT algorithm, detection strategies, and an adaptation algorithm for engine speed. Two hardware and software implementation methods for TI TMS320 DSPs are discussed. One is a high-level-language version for the TMS320C30 written completely in C, which could be used for research and development. The other is an assembly-language version written for the TMS320C25, which could be used in a production system. Finally, an integration strategy is described that shows a road map, starting with the replacement of an existing analog knock detection system interfaced to an engine ECU with a DSP-based system. The strategy leads to a single-processor solution with increased features and performance.

Signal Conditioning

A lower-cost broadband knock sensor is used in place of one tuned only to the fundamental frequency to allow the DFT signal-conditioning algorithm to monitor multiple frequencies. The signal-conditioning algorithm starts with the basic definition of the single-point direct-form DFT shown in equation (1).

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi k[n] / N} \tag{1}$$

where: k = frequency index
n = sample time index
N = sample block length
X[k] = amplitude of sinusoid at frequency index k
x[n] = time domain representation of a signal

Three variables determine the effective center frequency and bandwidth (frequency discrimination) of a single-point DFT: sample rate, block size, and indexed frequency. The relationships are:

$$\text{Nyquist rate} : f_N = f_s/2$$

$$\text{Minimum frequency} : f_{min} = f_N/N$$

$$\text{Indexed frequency} : f[k] = k \cdot f_{min} \qquad k : 0, \ldots, N-1$$

The Nyquist rate (f_N) is one-half the sample rate (f_s), which determines the highest frequency that can be discriminated without aliasing. For instance, the most significant knock frequencies are typically between

5 kHz–20 kHz; the sample rate for the knock detection system needs to be at least 40 kHz to prevent aliasing. This results in a loop time for the signal-conditioning algorithm of no more than 50 µs. A high computational throughput in the CPU is required to calculate any sophisticated algorithms in such a short period of time.

The minimum frequency (f_{min}) that can be discriminated is determined by the sample rate and the block size. For a given sample rate, the larger the block size, the greater the frequency discrimination of the DFT. Frequency discrimination can be considered similar to the bandwidth of a filter.

The specific frequencies that can be monitored are determined by the indexed frequency (f[k]) once the sample rate and block size are set. One of the advantages of a DFT over an FFT is that the block size can be any length and is not limited to a power of 2. This makes the frequency ranges that can be set using a DFT more flexible than those set using an FFT. A digital filter can be more adjustable than the DFT, but it carries the penalty of higher computational overhead and a larger memory requirement than an equivalent single-point DFT.

The general expression of the DFT allows the input signal x[n] to be complex, but in sampled data systems like this one for knock detection, the input signal is real. This allows a simplification of the final expression. In addition, digital processors cannot easily represent complex exponentials, so a different representation of equation (1) is required. Euler's relationship between complex exponentials and sinusoids is the starting point for developing a set of expressions that can be implemented on a DSP:

$$e^{-j\Theta} = \cos\Theta - j\sin\Theta \tag{2}$$

Substituting (2) into (1) and allowing the input signal to be only a real value yields:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot \left(\cos\left(2\pi[n]k/N\right) - j\sin\left(2\pi[n]k/N\right)\right) \tag{3}$$

Expanding the expression into the real and imaginary parts of the computation gives:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot \cos(2\pi[n]k/N) - j\sum_{n=0}^{N-1} x[n] \cdot \sin\left(2\pi[n]k/N\right) \tag{4}$$

Finally, separating the real and imaginary parts of the computation into individual equations gives forms that can be easily programmed on a digital processor:

$$X_R[k] = \sum_{n=0}^{N-1} x[n] \cdot \cos\left(2\pi[n]k/N\right) \tag{5a}$$

$$X_I[k] = \sum_{n=0}^{N-1} x[n] \cdot \sin\left(2\pi[n]k/N\right) \tag{5b}$$

To minimize the computational burden on the DSP, the calculations are spread out over the entire sample block. Each time a sample is received, one step in the summation is computed, which is an advantage of

DFT over FFT implementation. All samples for an FFT must be stored in memory before the calculations can be performed. The DFT can be calculated one sample at a time because there is no linkage between samples as there is with the FFT. Even though the FFT is more efficient in total time required when calculating across all allowable frequency ranges, the DFT makes superior uses of CPU resources when only a few of the possible frequency ranges are being monitored. The actual form of the difference equation implemented on the DSP is:

$$X_R[k] = X_R[k] + x[n] \cdot \cos\left(2\pi[n]k/N\right) \tag{6a}$$

$$X_I[k] = X_I[k] + x[n] \cdot \sin\left(2\pi[n]k/N\right) \tag{6b}$$

These calculations are repeated once every sample period and require two multiplies and two adds per frequency range monitored, regardless of the sample block length used. Note that only a single sample must be stored to implement this algorithm. This is one of the advantages of the DFT over a digital filter, such as a finite impulse response (FIR) filter. Computing one N-tap FIR filter requires that N samples and N coefficients be stored in memory. N multiplies and N accumulates are required each sample period. The number of frequency ranges or the frequency discrimination in each range (the number of filter taps) is much more limited for a given CPU if a filter-based signal-conditioning technique is used, due to the increased computational burden.

The cosine and sine values used in the DFT computation are best implemented as look-up tables for this application. Computing the values requires too much time and the memory required to store the tables is not excessive for the block sizes used for knock detection. To further minimize memory requirements, a single table can be used with two pointers: one for cosine values and another for sine values. The table is constructed to contain N elements that define one period of the lowest frequency (f_{min}) sine wave. Using this single table, any of the allowable frequencies can be monitored. This is another advantage of DFT over digital filter implementation. A digital filter has a unique set of coefficients for each frequency monitored, while the single-point DFT has the “coefficients” as subsets of one N-length table of values for all frequencies.

When the sample block summation is completed, the magnitude of the signal strength for each frequency range must be computed. The standard equation for computing magnitude is:

$$X[k] = \sqrt{X_R[k]^2 + X_I[k]^2} \tag{7}$$

The detection algorithm develops the knock metric by comparing the current signal strength to the reference. To simplify the calculations, the square root is not computed, because it does not provide additional information. Using only the sum of the squares amplifies differences in magnitude, because the square root function tends to compress the number range. The equation used to compute the relative magnitude is:

$$X[k]^2 = X_R[k]^2 + X_I[k]^2 \tag{8}$$

The DFT signal-conditioning algorithm should run for as much of the variable time period from 10° to 70° ATDC (the knock window) as possible. To make the signal strength calculations consistent, a single block

size is used for all engine speeds. The highest engine speed determines the maximum block size that can be used for a given sample rate. At lower engine speeds, multiple sample blocks are required to span the knock window. When more than one sample block is used, an outer loop around the core DFT algorithm is required to select the sample block with the highest magnitude signals. A high-level pseudocode description of an algorithm to monitor five frequency ranges over multiple sample blocks is shown below:

```
for (i = 0;i < BLOCKS,i ++)
{
    for(n = 0;n ≤ N – 1;n ++)
    {
TEST
            if (NO SAMPLE), then go to TEST
            x0 = SAMPLE
            X_R[k1] = X_R[k1] + x0 · cos(2πk1[n]/N)
            X_I[k1] = X_I[k1] + x0 · sin(2πk1[n]/N)
                              .
                              .
                              .
            X_R[k5] = X_R[k5] + x0 · cos(2πk5[n]/N)
            X_I[k5] = X_I[k5] + x0 · sin(2πk5[n]/N)
    }
        X[k1]² = X_R[k1]² + X_I[k1]²
                              .
                              .
                              .
        X[k5]² = X_R[k5]² + X_I[k5]²
          X_sum = X[k1]² + · · · + X[k5]²
    if (X_sum > X_max), then
    {
        X_sum = X_max
        X[k1]_max = X[k1]²
                  .
                  .
                  .
        X[k5]_max = X[k5]²
    }
}
```

When the signal conditioning calculations are complete, the signal magnitudes $X[k]_{max}$ are passed to the detection algorithm.

More complex DFT algorithms can be implemented that use a running magnitude computation technique rather than the sequential block technique shown above. The sliding-mode DFT and the Goertzel algorithm

are two such techniques. Advantages of these algorithms are that they can be used to determine exactly where, during the ignition cycle, the signal reaches the maximum amplitude. This information can be used by the detection strategy to more accurately determine the knock severity. The sooner after spark ignition that engine knock occurs, the more severe the knock and the more aggressive the control response should be. The disadvantages of these techniques are that the computational overhead is slightly higher and the data memory requirements increase, because a sample-block-length number of samples (N) must be retained for the calculations. Only the sequential block technique is discussed in this paper. References [4] and [5] contain details of the more complex methods.

Detection Strategy

The detection strategy uses the output of the DFT-based signal conditioning software module to develop the knock intensity metric that will be used by the control strategy. There are several knock detection strategies that have varying levels of sophistication and computational overhead. A few examples are given here to demonstrate the range of complexity, starting with the simplest and increasing in complexity. The strategy of choice is dependent on the specific engine configurations and control strategy implementations.

The simplest strategy is to use only the DFT-generated frequency magnitudes for the detection algorithm. A more complicated approach is to include the rate of change of the magnitudes, which can be used in predictive algorithms. A more sophisticated approach is to implement a fuzzy logic control strategy. Since the combustion process is highly variable from cycle to cycle, statistical methods can be used to compute trends toward the knock threshold. The magnitude-only approach is presented in more detail.

Magnitude detection strategies can use a number of techniques. One is to count the number of frequencies from the current measurement that exceed the reference, then use the count to indicate knock intensity. This is the simplest technique and is a good starting point when first implementing the DFT-based signal conditioning algorithm. A high-level pseudocode description is shown below:

if $(X[k1]_{max} > X[k1]_{ref})$, then counter ++

.
.
.

if $(X[k5]_{max} > X[k5]_{ref})$, then counter ++

KNOCK = counter

Another approach is to weight the count according to which frequency range exceeds the reference. The weighting can also be varied with engine RPM. For instance, the fundamental frequency and one of the vibrational modes could carry a higher weight at lower engine speed and a lower weight at higher speed. In this example, it is assumed that the importance of the frequency ranges is assigned as follows:

Priority	Variable	Frequency	Name
5	range1	7250 Hz	Fundamental
2	range2	9475 Hz	Vibrational mode #1
4	range3	12325 Hz	Vibrational mode #2
3	range4	16000 Hz	Vibrational mode #3
1	range5	18775 Hz	Vibrational mode #4

A simple weighting scheme is shown below in which the fundamental frequency and vibrational mode #2 have weights that vary with engine speed. The pseudocode shows the initialization and the determination of the knock intensity metric.

```
// Initialization of weights
if (RPM ==LO), then
{
      range1 = 0 11 00 00 00 00b
      range3 = 0 00 11 00 00 00b
{
else
{
      range1 = 0 01 00 00 00 00b
      range3 = 0 00 01 00 00 00b
}
range2 = 0 00 00 00 11 00b
range4 = 0 00 00 10 00 00b
range5 = 0 00 00 00 00 01b
           .
           .
           .

// Knock intensity metric calculation
counter = 0
if (X[k1]max > X[k1]ref), then counter+ = range1
                  .
                  .
                  .
if (X[k5]max > X[k5]ref), then counter+ = range5
KNOCK = counter
```

The amount by which the frequency magnitude exceeds the reference can be used as additional information for determining a knock intensity metric and can be integrated into the weighting factor. This metric can be used to fine-tune the algorithm for detecting trace knock conditions. It requires additional computational resources to implement. The value of $\Delta[k]$ can be different for each frequency being monitored. An example using pseudocode is shown below. Here, the weights vary for both engine RPM and variation between the measurement and the reference value.

```
// Initialization of weights
if (RPM = = LO), then
{
      range1LO = 0 110 000 000 000 000b          // 06000h
      range1HI = 0 111 000 000 000 000b          // 07000h
      range3LO = 0 000 110 000 000 000b          // 00C00h
      range3HI = 0 000 111 000 000 000b          // 00E00h
}
else
{
      range1LO = 0 010 000 000 000 000b          // 02000h
      range1HI = 0 011 000 000 000 000b          // 03000h
      range3LO = 0 000 010 000 000 000b          // 00400h
      range3HI = 0 000 011 000 000 000b          // 00600h
}
      range2LO = 0 000 000 000 110 000b          // 00030h
```

$range2_{HI}$ = 0 000 000 000 111 000b // 00038h
$range4_{LO}$ = 0 000 100 000 000 000b // 00800h
$range4_{HI}$ = 0 000 101 000 000 000b // 00A00h
$range5_{LO}$ = 0 000 000 000 000 010b // 00002h
$range5_{HI}$ = 0 000 000 000 000 011b // 00003h
.
.
.

// Compute reference to measurement difference

$\Delta[k1] = X[k1]_{max} - X[k1]_{ref}$
.
.
.
$\Delta[k5] = X[k5]_{max} - X[k5]_{ref}$
.
.
.

// Calculate knock intensity metric
counter = 0
if ($\Delta[k1] > \Delta[k1]_{HI}$), then counter+ = $range1_{HI}$
else if ($\Delta[k1] > \Delta[k1]_{LO}$), then counter+ = $range1_{LO}$
.
.
.

if ($\Delta[k1] > \Delta[k5]_{HI}$), then counter+ = $range5_{HI}$
else if ($\Delta[k1] > \Delta[k5]_{LO}$), then counter+ = $range5_{LO}$
KNOCK = counter

In some instances, it may be desirable to predict when knock is about to occur and begin to make adjustments to the control strategy preemptively. The rate of change of the amplitude can be used for this prediction function, and this predictor can be implemented as a simple difference between the current and some previous measurement. The greater the time difference, the less effect cycle-by-cycle variations will have, but the more effect changing the engine speed will have on the result. This technique takes more computational resources and increases memory requirements, because previous values must be retained for the calculation. Normalizing the average rate of change over all the ranges can yield the weighting factor that would be used to adjust the knock intensity metric. The pseudocode for an 8-sample time difference example is shown below. A two-dimensional array is used to store the current and previously measured amplitudes for the frequencies being monitored.

// Compute rate of change

$\Delta_{rate}[k1] = \Delta[k1, m]_{max} - \Delta[k1, m-8]_{max}$
.
.
.

$\Delta_{rate}[k5] = \Delta[k5, m]_{max} - \Delta[k5, m-8]_{max}$

.
.
.

/ / Normalize the rate of change over frequencies
/ / (Δ_{max} is largest possible difference)
$\Delta_{norm} = 1 / \Delta_{max}$

$$\Delta_{ave} = 0.2 \cdot \sum_{n=1}^{5} \Delta_{norm} \cdot \Delta_{rate} \text{ [kn]}$$

.
.
.

/ / Adjust knock metric for amplitude rate of change
if ($\Delta_{rate} > 0$), then
KNOCK = Δ_{ave} · KNOCK

Using measurements too close together in time in highly variable systems can cause the control response to become unstable. However, with a carefully designed rate computation, the additional information provided can allow better control during transient operating conditions when speed and/or load are changing rapidly. This technique can even be extended to the rate of change of the knock intensity metric itself.

Adding sophistication to the detection strategy increases the computational burden on the CPU and can provide only a marginal increase in algorithm performance. Increasing sophistication also requires that the system designer have a high level of understanding of the knock process in the specific engine family on which the knock detection system is being implemented. The implementation examples given in a following section demonstrate one of the simple techniques that make it widely applicable.

Adaptation Strategy

The ability to adapt to changing engine operating conditions, primarily speed and load, greatly improves the performance of a knock detection system. Less compromise is required in the baseline spark timing, because the system is able to more accurately determine low-level knock at all engine speeds. Using a TI DSP provides the computational throughput required to execute both the knock detection and adaptation calculations in real time over the entire speed range of today's passenger car engines.

The adaptation technique used in this application report generates the reference signal from a sliding weighted-average computation. A weighted mean is computed individually for each cylinder and is updated for each firing cycle. The general form of the weighted mean is:

$$Y[kn] = \sum_{m=0}^{N-1} \frac{x[k(n-m)]}{N}$$

$$R[kn] = K \cdot Y[kn]$$

Where: Y[kn] = the mean value of a time history of inputs
x[k(n – m)] = the time history of inputs, where each input is a spectral magnitude
N = the number of elements in the time history
R[kn] = the weighted mean value
K = the weighting coefficient

However, this method for computing Y[kn] requires N multiply/accumulates for each average value. A more effective computational technique is to add the newest input to the running average and subtract the

oldest value. This method requires two multiply/accumulates to compute a new average regardless of the number of values in the time history. The memory storage requirement increases by 1 (to N + 1 values) due to the need to store the previous average value. This computational method has the form:

$$Y[kn] = Y[k(n-1)] + \frac{x[kn]}{N} - \frac{x[k(n-N)]}{N}$$

The weighting factor (K) is used in the computation of R[kn] to adjust the reference value so that cycle-to-cycle variation does not falsely trigger the detection algorithm. Empirical studies put the value for K at 1.5–2. These values can be different for each cylinder on the same engine due to the effects of sensor placement and structural dynamics. In addition, the weighting coefficient could be modified in real time to track changes in engine speed and/or load if cycle-to-cycle variations change. However, at the current level of control strategy sophistication, the value of investing the additional computational time is probably marginal. Future systems may require statistical analysis of the combustion signature to adjust for cycle-to-cycle variation, but the system implementation described in this report does not use them.

A number of conditions must be taken into account when choosing the averaging period. The averaging period must be long enough that the effects of cycle-to-cycle variations are minimized, but short enough that changing engine speed or load are not filtered out. Values in the range of 8–16 previous samples have been used in development systems with considerable success.

Another consideration when implementing an adaptation strategy is how to handle reference updates in the presence of engine knock. If the reference is updated when knock is occurring, the average value is contaminated by the higher magnitude signals generated by the knock. Assuming that corrective action by the ECU eliminates knock within a small number of firing cycles, a simple method of handling adaptation in the presence of knock can be used: do not update the reference value when knock is detected. This method is practical, because knock is eliminated and the reference updates can restart before any significant changes develop in the operating condition of the engine. The technique described in this section uses this assumption.

For this adaptation strategy to work, there must be a period of time during initialization when nonknocking operation can be guaranteed. Normally, the strategy initializes the adaptation algorithm during the short period of time following engine start when the likelihood of knock is very low. Reinitializing the adaptation algorithm each time the engine is started allows the reference to reflect slowly changing operating conditions.

Implementation Examples

In this section, two implementation examples are presented. The first is a floating-point solution based on the TMS320C30 with all the software written in C. This could be used as a research and development system because of the high computational bandwidth and ease of software development. Due to cost pressures, a production system most likely needs to be a fixed-point solution. The second implementation example is based on the fixed-point TMS320C25 with the software written in assembly language.

While presenting the implementation examples, this report explains the rationale for decisions made on specific structures or variables. This rationale shows how modifications can be made to adapt the concept to new systems. As background for the examples, a discussion on available computational time, sample rate, sample block size, and ADC resolution is presented first, followed by details on the specific target DSPs.

Timing Considerations

The amount of time available for the various stages of the algorithm can affect implementation decisions. The general form of the real-time code for the knock detection software shows the time slots required for the given modules as a function of crankshaft angle. Figure 1 shows the breakdown.

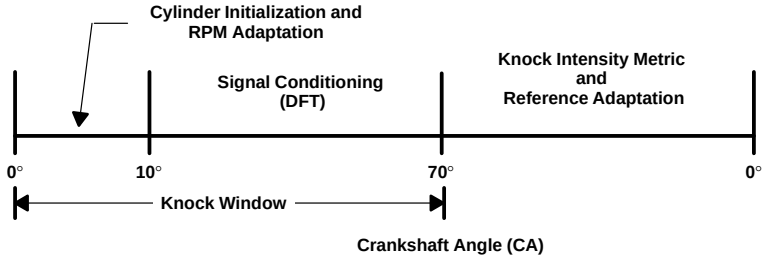

Figure 1. Algorithm Stages Associated With the Crankshaft Angle

A study of the number of cylinders, engine RPM range, sample rate, and sample data block size gives the information required to start making implementation decisions.

Engines today can have operating speeds from 450 RPM at idle up to almost 8000 RPM. Using an execution window comprising 60° of the crankshaft angle (10°–70° CA ATDC) for the monitoring period, the minimum amount of time available to perform the DFT-based signal-conditioning algorithm can be computed. The minimum period occurs when the engine speed reaches its maximum, or 8000 RPM.

$$\frac{min}{8000\ revs} \cdot \frac{rev}{360^\circ\ CA} \cdot \frac{60\ s}{min} = 20.8\overline{3}\ \circ \frac{\mu s}{CA}$$

$$\frac{60^\circ\ CA}{knock\ window} \cdot \frac{20.8\overline{3}\ \mu s}{^\circ CA} = 1.25\ \frac{ms}{knock\ window}$$

Even at high engine speeds, there is a significant amount of time available for the signal-conditioning calculations using a TMS320 DSP. The 40-MHz TMS320C30 has time for 25,000 instruction cycles, and the 50-MHz TMS320C25 is able to complete 15,625 cycles.

Next, the time available for the detection and adaptation tasks must be determined. This is the amount of time between 70° ATDC on the monitored cylinder and 0° ATDC on the next cylinder in the firing order, evaluated at maximum engine RPM. This time period is directly affected by the number of cylinders in the engine. Today, the majority of engines use a 4-cycle design, which goes through a complete firing sequence every 720° of crankshaft angle. Most engines have 4, 6, or 8 cylinders, though 3-, 5-, 10-, and 12-cylinder versions exist. Evaluating a number of engine configurations for the time available for the detection and reference adaptation algorithms is shown in the following equations:

$$\left[\left[\frac{720^\circ\ CA}{firing\ cycle} \times \frac{firing\ cycle}{12\ cylinders}\right] - \frac{70^\circ\ CA}{knock\ window}\right] \times \frac{20.8\overline{3}\ \mu s}{^\circ CA} = -208.3\ \mu s$$

$$\left[\left[\frac{720^\circ\ CA}{firing\ cycle} \times \frac{firing\ cycle}{10\ cylinders}\right] - \frac{70^\circ\ CA}{knock\ window}\right] \times \frac{20.8\overline{3}\ \mu s}{^\circ CA} = 41.2\ \mu s$$

$$\left[\left[\frac{720^\circ\ CA}{firing\ cycle} \times \frac{firing\ cycle}{8\ cylinders}\right] - \frac{70^\circ\ CA}{knock\ window}\right] \times \frac{20.8\overline{3}\ \mu s}{^\circ CA} = 416.7\ \mu s$$

$$\left[\left[\frac{720^\circ\ CA}{firing\ cycle} \times \frac{firing\ cycle}{6\ cylinders}\right] - \frac{70^\circ\ CA}{knock\ window}\right] \times \frac{20.8\overline{3}\ \mu s}{^\circ CA} = 1.04\ ms$$

$$\left[\left[\frac{720^\circ\ CA}{firing\ cycle} \times \frac{firing\ cycle}{4\ cylinders}\right] - \frac{70^\circ\ CA}{knock\ window}\right] \times \frac{20.8\overline{3}\ \mu s}{^\circ CA} = 2.29\ ms$$

These calculations show that for 12-cylinder and probably 10-cylinder engines there is computational overlap between the current cylinder's detection and adaptation tasks and the next cylinder's initialization. A method of interleaving the calculations must be implemented for these engine configurations, but such a method exceeds the scope of this report.

To simplify the example, the assumption is made that the knock detection algorithm is being implemented on a 6-cylinder engine. This assures enough time to complete all calculations before the next cylinder's monitoring period begins. The time budget is now established. There are at least 208.3 μs for the cylinder initialization and RPM adaptation, 1.25 ms for the DFT-based signal-conditioning routine, and 1.04 ms for the detection and reference adaptation routines.

Frequency Selection

Next, the sample rate and block size must be selected. These two parameters determine which center frequencies and bandwidths are possible using the individual single-point DFTs. After defining the fundamental and primary vibrational mode frequencies, the sample rate and block size are varied until a good match is obtained.

A simple C program that searches over a specified range of sample rates and block sizes finds the best matches for a given set of center frequencies. Best match is defined as the combination of sample rate (f_s) and block size (N) that produces the lowest average and lowest difference in standard deviation between the desired and attainable center frequencies. The program listing is contained in Appendix A. When the program runs, it queries the user to define the search parameters and then generates a list that shows the ten best matches.

Tradeoffs on block size, sample rate, and frequency match can then be made, taking into account computation time (the higher the f_s, the less time available between samples) and frequency selectivity (the larger the N, the narrower the effective bandwidth). The effective bandwidth (BW) can be estimated by:

$$BW \approx \frac{f_s}{4 \cdot N}$$

Once the sample rate and block size have been determined, the frequency index numbers that most closely match the desired center frequencies must be determined. The following C code can be used to list the available center frequencies versus index number:

$$f_{min} = \frac{f_s}{2 \cdot N};$$

for(k = 0; k < N; k++)

$$f(k) = k \cdot f_{min};$$

Choose the values of k that give the closest matches between f(k) and the desired center frequencies.

Analog-to-Digital Converter (ADC) Resolution

The dynamic range of the external sensor must cover several orders of magnitude. Over the operating range of an engine, values can vary from less than 40 mV at low RPM without knock to full scale of the sensor at high RPM and high knock. Full scale ranges typically available are ±10 V, ±5 V, and 0 V–5 V. To cover this dynamic range, an ADC input system with an effective resolution of 8 bits (±5 V and 0 V–5 V) to 10 bits (±10 V) is required. Higher resolution is beneficial, because it provides a guard band at either extreme, which can be used for system diagnostics. However, the added resolution increases system cost. Each system must be evaluated to determine the required resolution for that particular implementation.

An 8-bit ADC can be used, even in systems that require higher resolution, by employing external hardware gain switching combined with software control and compensation. A software routine can be incorporated into the input signal-conditioning algorithm to monitor the input signal magnitude and adjust an external gain stage at the input to the ADC. It also must adjust the signal gain to compensate for the external gain changes prior to use in the DFT calculations. The internal compensation is inversely proportional to the external gain adjustment to make the effective input sample dynamic range constant. The implementations presented here assume that the ADC has adequate dynamic range for the sensor that is used, so no gain adjustments are required.

TMS320C30 Implementation

A ’C30-based system is ideal for research and development to develop algorithms and determine analog input system requirements. It is also an excellent platform to use as the sole processor in a high-performance engine control system. The ’C30 is a 32-bit floating-point processor with two on-chip timers and two serial ports. The timers are used to determine engine speed and to trigger execution of the signal-conditioning algorithm. Because of the architectural features and the efficiency of the optimizing C compiler, all the application software can be written in C.

Hardware Description

A high-level block diagram of a ’C30-based system is shown in Figure 2. The system consists of an analog input system (containing analog signal conditioning, a multiplexer, and an A/D converter), the TMS320C30, EPROM, SRAM, processor interface circuitry, and an engine controller (with the assumption that initial tests simply replace an existing knock detection subsystem).

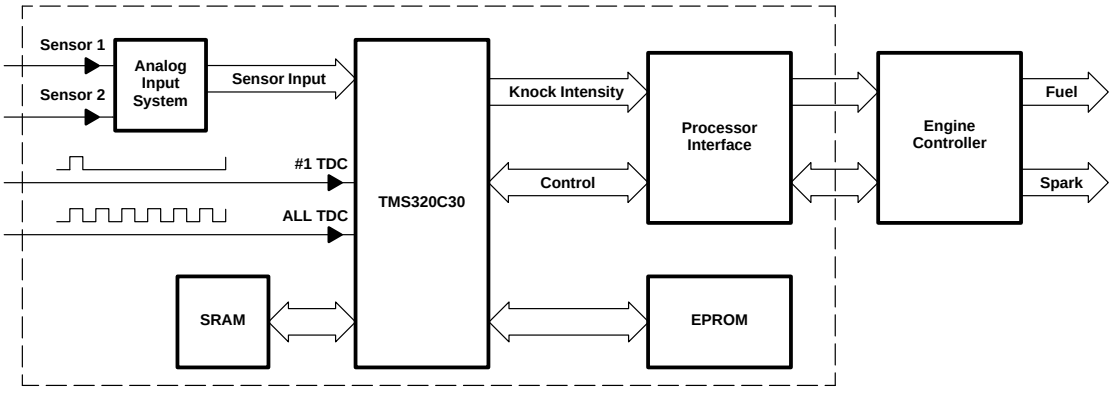

Figure 2. TMS320C30-Based Knock Detection System Block Diagram

Four input signals are required for this configuration: two for knock measurements and two for timing. The two sensor signals are from the engine-mounted accelerometers that monitor the combustion process. Two sensors are used so that one sensor can be put on each bank of a V-configured engine. An in-line engine would require only a single knock sensor. This is standard practice today for the purpose of obtaining a satisfactory signal-to-noise ratio. Single-sensor systems for V-bank engines can also be investigated with this system, but this is not done in this report.

The timing signals are used to compute engine RPM (#1 TDC) and to control when the algorithm runs (ALL TDC) for each cylinder. The #1 TDC signal occurs every 720° of crankshaft rotation. In addition to being used to compute engine RPM, this signal is also used to synchronize the algorithm to the cylinder firing order. The ALL TDC signal occurs every 120° (in the 6-cylinder example) and triggers the algorithm to run for the next cylinder. It is possible to perform both functions using only the #1 TDC signal, but doing so requires more computational overhead.

The memory configuration is set up to allow easy changes to the knock detection system software. Slow EPROM (or EEPROM/Flash) memory is programmed with the current version of the software to be run. During initialization, the real-time code and variables are transferred from EPROM to zero-wait-state SRAM or on-chip RAM.

Software Description

Details of the hardware implementation affect how the initialization and input/output code is written. Therefore, the software in this example is not a complete knock detection system implementation, but is limited to the core algorithm. The assumption is made that no hardware or software input scaling is required since such scaling is also implementation specific. Also, a simple indication of knock intensity is passed to the engine controller in a single word. The algorithm is structured to monitor five frequencies and uses the simple comparison method (number of monitored frequencies exceeding the reference) for detection. Tradeoff decisions between speed and code size in this example favor speed to maximize computational throughput.

Both ’C30 timers are required for this implementation. One is used to determine the engine RPM, which is required to adjust the delay period from top dead center (TDC) to 10° ATDC and the number of sample blocks analyzed in the 10°–70° ATDC knock window. Execution of the code to compute engine RPM computation occurs once per combustion cycle (Cylinder #1 TDC). During the ignition cycles of the remaining cylinders, this code is bypassed. The other timer is used to control when the DFT algorithm begins execution at 10° ATDC.

Figure 3 shows the structure of the program used with the ’C30 floating-point DSP. The first three modules are written in assembler. They are used to define the interrupt vectors, initialize the C environment, and transfer real-time code from EPROM to SRAM or on-chip RAM memory. All remaining modules that pertain to the algorithm are written in C. Except for the initialization module, all functions reside in interrupt service routines (ISRs). An external interrupt triggered by the ALL TDC signal drives the cylinder initialization ISR. This ISR also contains the RPM adaptation code. One of the onboard timers is used to measure the time to rotate through 720° CA, which is directly proportional to RPM. The other timer is used to trigger the signal-conditioning and knock detection ISR to run after the adjustable delay time between 0° to 10°ATDC has elapsed. The comments in the program listing in Appendix B discuss more details of the program’s functionality.

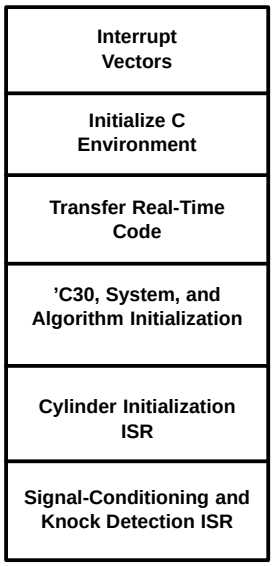

Figure 3. TMS320C30 Floating-Point DSP Knock Detection Program Structure

Test Results

Limited hardware testing has been performed on a ’C30-based platform. Assuming a V6 engine with a maximum speed of 8000 RPM, a 64-sample block size, and 16 frequencies to monitor, initial indications are that sample rates of 75 kHz are achievable. Sample rates below 60 kHz will probably be used when the desired effective bandwidths of the DFTs are taken into consideration. This indicates that a ’C30 with software written in a high-level language like C is an excellent platform for algorithm development and system testing.

TMS320C25 Implementation

To put a DSP-based knock detection system into production, a fixed-point DSP would most likely be used to keep system costs down. The TMS320C25 is the processor chosen for this example. It is the midpoint of TI’s fixed-point DSP family. This device is a 16-bit fixed-point processor with a single onboard timer. Depending on the sample rate and the number of frequencies to be monitored in a specific implementation, a member of the TMS320C1x family might be used. To maximize system efficiency, the software is written in assembler, which allows the user to best take advantage of the Harvard architecture used in TI’s fixed-point DSP family.

Hardware Description

A high-level block diagram of a possible ’C25-based system is shown in Figure 4. The system consists of an analog input system (containing analog signal conditioning, a multiplexer, and an A/D converter), the ’C25, the engine RPM timer, processor interface circuitry, and an engine controller (with the assumption that the initial implementation retains the existing microprocessor).

The primary differences between the ’C30-based system and the ’C25-based system are twofold. First, no external memory is used, which reduces system cost. Second, a 22–24-bit timer is required to compute engine RPM (down to 100 RPM). The onboard timer on the ’C25 is used to control the DFT algorithm execution (see *Software Description*, page 17, for details). The functionality in the ’C25-based knock detection hardware subsystem can be integrated into a cDSP (customized DSP) device if the specifics of a given implementation warrant doing so.

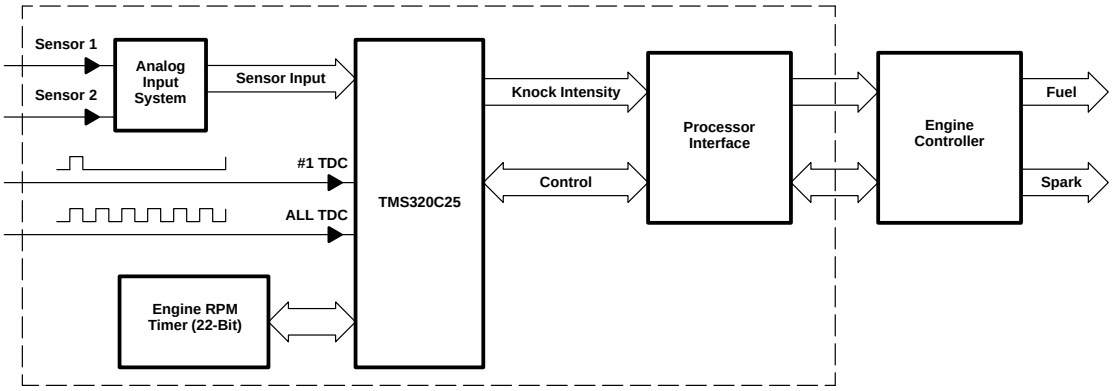

Figure 4. TMS320C25-Based Knock Detection System Block Diagram

Software Description

As in the ’C30 example, the software presented in this example is not a complete knock detection system implementation but is limited to the core algorithm code. The assumptions made for this implementation are the same as those made for the ’C30 example — that no hardware or software input scaling is required, that a simple indication of knock intensity is passed to the engine controller in a single word, that the algorithm monitors five frequencies, and that a simple frequency comparison method is used for detection.

In the case of the ’C25, tradeoff decisions between speed and code size favor code size to reduce memory usage. Specifically, the cosine and sine coefficient tables used to compute the DFTs are not precomputed for the entire sample block. Instead, a single sine wave period one sample block long is used. This saves $(2 \times \text{number of frequencies} - 1) \times N$ memory locations, but costs five instruction cycles per frequency monitored.

Figure 5 shows the structure of the program used with the ’C25 fixed-point DSP. All modules shown are written in assembler. An external interrupt triggered by the ALL TDC signal drives the cylinder initialization ISR. The onboard timer triggers the signal-conditioning and knock detection ISR to run after the adjustable delay time from 0° to 10°ATDC has elapsed. The comments in the program listing in Appendix C contain more details of the program’s functionality.

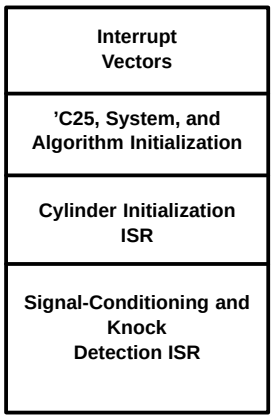

Figure 5. TMS320C25 Fixed-Point DSP Knock Detection Program Structure

Test Results

The fixed-point implementation is still being developed. Preliminary testing to date has been restricted to simulation. Again, assuming a V6 engine with a maximum speed of 8000 RPM, a 64-sample block size, and 16 frequencies to monitor, a sample rate of approximately 50 kHz should be achievable. Reducing the number of frequencies to ten or fewer should increase the sample rate to over 60 kHz.

The most significant differences in performance between the 'C30 and the 'C25 result from the need to eliminate external memory on the fixed-point DSP system. Therefore, the large interleaved cosine and sine coefficient tables constructed in the 'C30 example are not used. Instead of precalculating all coefficients during initialization, the fixed-point implementation must carry the overhead of coefficient-table pointer manipulation in the real-time code. This adds at least five instruction cycles for each frequency monitored to the real-time code that runs between each sample input. If only three frequencies were monitored, the coefficient-table pointers could be kept permanently in six of the eight auxiliary registers. This would eliminate four of the five additional instruction cycles of overhead penalty for the 'C25, boosting performance significantly.

Integration Road Map

The knock detection system based on a TI DSP can be used as a building block to incorporate advanced capabilities in an engine control system. An implementation road map is shown in Figure 6. The DSP-based knock detection system is initially used as a coprocessor to the current engine control microprocessor. This allows new features to be added cost effectively to the ECU with minimal change to existing hardware. The microprocessor can be retained in the system to perform spark, fuel, idle speed, and evaporative recovery system actuation while the DSP incorporates new, advanced analysis and control strategy computations. If additional cost savings are desired, the microprocessor functions can be integrated into the DSP and the microprocessor can be eliminated from the system.

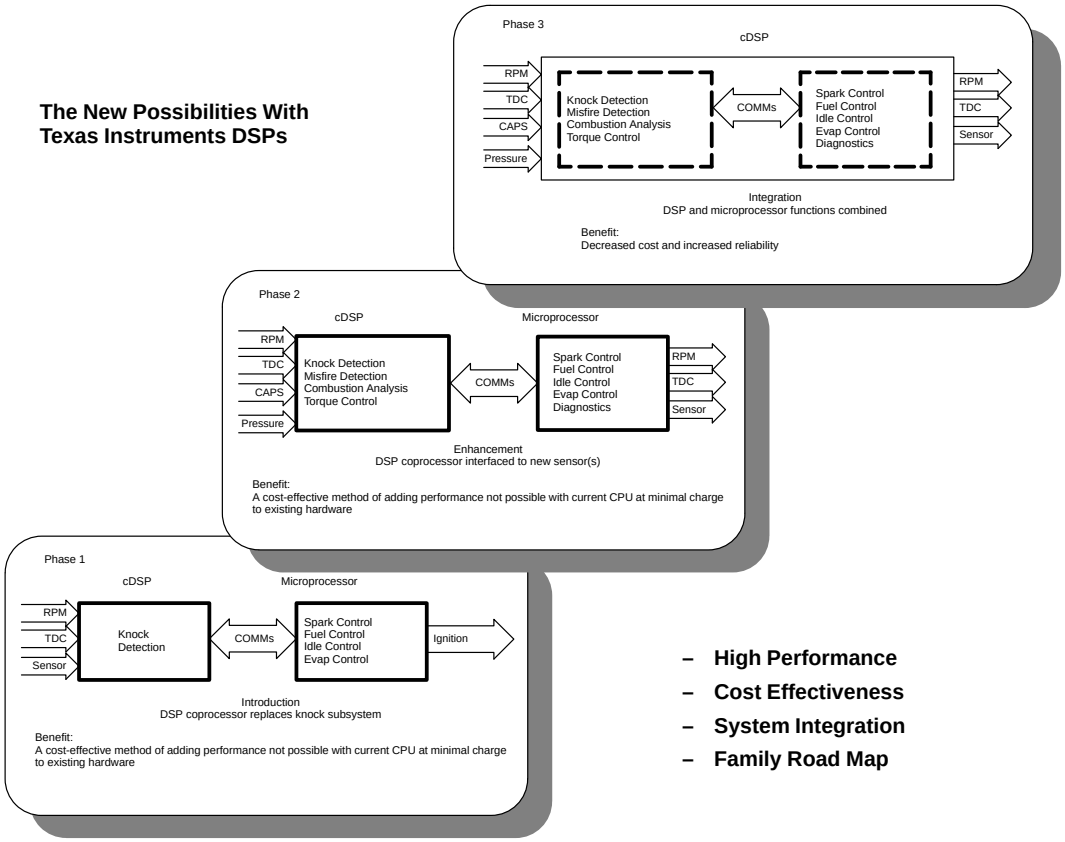

Figure 6. DSP Road Map From Knock Detection Subsystem to Full Engine Control

Texas Instruments has the capability to achieve system integration of digital signal processors, microcontrollers, memory, digital logic, and analog functions on a single chip. The TI integration road map offers customers the benefits of reduced board space, fewer packages, lower power consumption, and reduced total cost of ownership for their unique application.

MISFIRE DETECTION

Misfire detection may be possible using a DFT-based algorithm, which would require only minimal change to the detection part of the system. Misfire is defined as the lack of combustion. When the air/fuel mixture in the firing cylinder does not ignite, a reduction in excitation of the engine structure occurs, and the signal received by an external sensor shows a reduction in signal strength in the fundamental or vibrational mode frequencies. Comparing the signature of a nonfiring cylinder to the computed reference signal may indicate misfire.

The signal-conditioning algorithm using the DFT is the same for knock or misfire detection; only the detection part of the algorithm is different. When misfire occurs, the amplitude of one or more of the monitored frequency ranges is reduced relative to the reference. The change in signal strength when the cylinder misfires must be determined experimentally. Once this is done, the detection algorithm for knock just needs to be modified to trigger when the current signal strength is less than, rather than greater than, the reference magnitude.

Unlike knock detection in which corrective action is taken immediately, California's Onboard Diagnostics (OBDII) regulations are written to detect the occurrence of misfire as a percentage of 200 crankshaft revolutions (for system damage) or a percentage of 1000 revolutions (for emission control performance). This places the additional burden on the system of tracking misfire over a relatively long period before any action is taken.

Using a signal-strength analysis technique like the DFT algorithm described in the preceding sections requires only a small additional computational resource to implement both knock and misfire detection. Two other techniques are widely discussed in the literature. One is accurately measuring crankshaft speed fluctuation during a cylinder's ignition cycle and then using a software model to compute engine torque. Significant computational effort is required to compute the torque using an engine model to correct for torsional and vibrational dynamics. The other method is to obtain direct torque measurements using a special sensor (not yet commercially available), which significantly adds to system cost.

Incorporating a system that is able to detect both knock and misfire provides a cost-effective means of improving performance and fuel economy (knock detection) while complying with government emissions system monitoring regulations (misfire detection).

REFERENCES

1. C.F. Taylor; *The Internal Combustion Engine in Theory and Practice*; Vol. 2, pp. 34–85; MIT Press; 1968.
2. *Automotive Handbook*, second edition; pp. 289–290, 408–409; Robert Bosch GmbH, publisher; 1986.
3. S.M. Dues, J.M. Adams, G.A. Shinkle; "Combustion Knock Sensing: Sensor Selection and Application Issues"; SAE Paper 900488.
4. C.S. Burrus, T.W. Parks; *DFT/FFT and Convolution Algorithms: Theory and Implementation*; pp. 21–36; John Wiley & Sons; 1985.
5. J.G. Proakis, D.G. Manolakis; *Introduction to Digital Signal Processing*; pp. 682–726; Macmillan Publishing; 1988.

Appendix A: Sample-Rate and Block-Size Search Software

```
/* Sample Rate and Block Size Search Routine                              */
/*                                                                       */
/*   File: FINDFBIN.C              Rev: 1.0                              */
/*   Start Date:  5/7/93           Last Change: 6/9/93                   */
/*   Written by: Thomas G. Horner                                        */
/*               Texas Instruments                                       */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <time.h>

/*        PROTOTYPES         */
/*                           */
void   define(void);
void   file_open(void);
void   make_space(void);
void   search(void);

/*        DEFINITIONS        */
/*                           */

#define   list_length  10

/*      GLOBAL VARIABLES     */
/*                           */
FILE      *fptr;

float *delta;
float *fbin;
float *freq_mean;
float *freq_stddev;
float *fs_list;
float *N_list;

float freq[16];

float dif;
float fmin;
float freqn;
float fs;
float fs_max;
float fs_min;
float fs_range;
float fs_step;
float mean;
float std_dev;
float sum;
float var_sum;

int fs_i;
int nfreq;

int N;
int Ni;
int Nmax;
int Nmin;
int Nover2;
int Nrange;
int Nstep;

/*        MAIN FUNCTION     */
/*                          */

/* main() is used for overall system control.  */
/* Functions are called which do all the work. */
```

```
void main(void)
{
   file_open();
   define();
   make_space();
   search();
}

/*      FUNCTION DECLARATIONS        */
/*                                   */

/*     Search Parameter Definition   */
/*                                   */
/* define() is used to define the search    */
/* parameters by questioning the user.      */

void   define(void)
{
   int i;

   clrscr();

/***** Define search by questioning user.  *****/

   printf("Please answer the following questions to ");
   printf("define search parameters.\n");

     /* Define the number of frequencies to  */
     /* search                               */
   printf("\nHow many frequencies do you want to monitor (1–16)? ");
   scanf("%d",&nfreq);

     /* Define the frequencies         */
   printf("\nNow enter the frequencies to be \n");
   printf("monitored one at a time.\n");
   printf("Enter a number between 2000–20000.\n");
   for(i=0 ; i<nfreq ;   i++)
   {
     printf("\nEnter frequency No.%2d  ",i+1);
     scanf("%f",&freq[i]);
   }

     /* Define the block size min/max     */
   printf("\n\nNext enter the sample block size \n");
   printf("minimum and maximum values. \n");
   printf("Use a number from 32 to 128.\n");
   printf("\nEnter the minimum block size.  ");
   scanf("%d",&Nmin);
   printf("\nEnter the maximum block size.  ");
   scanf("%d",&Nmax);

     /* Define block size step size      */
   printf("\n\nNow enter the step size to be used ");
   printf("to go from the minimum");
   printf("\nto the maximum value of block size. ");
   scanf("%d",&Nstep);

     /*Define the sample rate min/max     */
   printf("\n\nNext you are asked to enter the sample rate ");
   printf("minimum and maximum values.");
   printf("\n Use a number between 40000 and 60000.\n");
   printf("\nEnter the minimum sample rate.  ");
   scanf("%f",&fs_min);
   printf("\nEnter the maximum sample rate.  ");
   scanf("%f",&fs_max);
```

```
     /* Define sample rate step size      */
   printf("\n\nNow enter the step size to be used ");
   printf("for the sample rate ");
   printf("sweep\nfrom the minimum to the maximum value. ");
   scanf("%f",&fs_step);

/***** Initialize parameters for search   *****/

     /* Compute block size range        */
   Nrange = Nmax–Nmin;

     /* Compute block size search repeats   */
   Ni = Nrange/Nstep + 1;

     /* Compute sample rate range       */
   fs_range = fs_max–fs_min;

     /* Compute sample rate search repeats   */
   fs_i = (int)(fs_range/fs_step) + 1;

     /* Write search definition to output file */
   fprintf(fptr,"\nNmax  = %d \n",Nmax);
   fprintf(fptr,"Nmin  = %d \n",Nmin);
   fprintf(fptr,"Nrange = %d \n",Nrange);
   fprintf(fptr,"Nstep  = %d \n",Nstep);
   fprintf(fptr,"Ni   = %d \n",Ni);

   fprintf(fptr,"\nfs_max  = %5.0f \n",fs_max);
   fprintf(fptr,"fs_min  = %5.0f \n",fs_min);
   fprintf(fptr,"fs_range = %5.0f \n",fs_range);
   fprintf(fptr,"fs_step = %5.0f \n",fs_step);
   fprintf(fptr,"fs_i   = %d \n\n",fs_i);
}

/*   File Allocation   */
/*                     */
/* file_open() is used to open any files which */
/* might needed during program execution.   */

void file_open(void)
{
   if ((fptr = fopen("findfbin.dat", "w")) == NULL)
   {
     printf("error in opening output file #1.");
     exit(0);
   }
}

/*   Array Allocation */
/*                    */
/* make_space() is used to allocate memory   */
/* space for arrays whose length is determined */
/* at runtime. Checks are made to verify that */
/* there is enough memory available. Some of */
/* the arrays are initialized to start values. */

void   make_space(void)
{
   int i;

   fbin = (float*)malloc(sizeof(float)*(Nmax/2+1));
   if (fbin==NULL)
   {
     printf("fbin[] allocation error \n");
     exit(0);
   }

   delta = (float*)malloc(sizeof(float)*nfreq);
   if (delta==NULL)
```

```
   {
     printf("delta[] allocation error \n");
     exit(0);
   }
   fs_list = (float*)malloc(sizeof(float)*list_length);
   if (fs_list==NULL)
   {
     printf("fs_list[] allocation error \n");
     exit(0);
   }
   for(i=0;i<list_length;i++)
     fs_list[i]=0.;
   N_list = (float*)malloc(sizeof(float)*list_length);
   if (N_list==NULL)
   {
     printf("N_list[] allocation error \n");
     exit(0);
   }
   for(i=0;i<list_length;i++)
     N_list[i]=0;
   freq_mean = (float*)malloc(sizeof(float)*list_length);
   if (freq_mean==NULL)
   {
     printf("freq_mean[] allocation error \n");
     exit(0);
   }
   for(i=0;i<list_length;i++)
     freq_mean[i]=1e+04;
   freq_stddev = (float*)malloc(sizeof(float)*list_length);
   if (freq_stddev==NULL)
   {
     printf("freq_stddev[] allocation error \n");
     exit(0);
   }
   for(i=0;i<list_length;i++)
     freq_stddev[i]=1e+04;
}

/* Frequency Bin Search */
/*                      */
/* search() carries out a search to find the  */
/* sample rate and block size that gives      */
/* center frequencies closest to the user's   */
/* specifications. The top 10 matches are     */
/* listed. The average error and deviation    */
/* are used to determine the best match.      */

void search(void)
{
   int i,j,k,l,m,n;
   int TESTING;

/* The frequency bin search strategy is to have */
/* an outer loop for sampling frequency and an  */
/* inner loop for block size. For a given set   */
/* of fs and N:                                 */
/* 1. Compute the frequency bins.               */
/* 2. Compare against desired frequencies and   */
/*   find deltas for closest matches.           */
/* 3. Compute delta mean and std. deviation.    */
/* 4. Compare against top 10 matches and        */
/*   update match array if necessary.           */
```

```
    l=0;
    /* Search over frequency range and   */
    /* block size range.                 */
    /*                                   */
    for(i=0;i<fs_i;i++)
    {
        fs = (float)(fs_min+i*fs_step);
        for(j=0;j<Ni;j++)
        {
            l+=1;
            N = Nmin+j*Nstep;
            Nover2 = N/2;
            for(m=0;m<nfreq;m++)
            delta[m]=1e+10;
        /* Compute min frequency for current    */
        /* sample rate and block size and       */
        /* compute frequency bins values.       */
            fmin = (float)(fs/N);
            for(k=0;k<=Nover2;k++)
            {
                fbin[k] = k*fmin;
                /* Check if new frequency range less    */
                /* than previous smallest difference    */
                for(m=0;m<nfreq;m++)
                {
                    dif=abs(fbin[k]-freq[m]);
                    if(dif<delta[m])
                        delta[m]=dif;
                }
            }
        }
            /* Compute average and std deviation    */
        sum=0.;
        var_sum=0.;
        freqn = (float)nfreq;
        for(m=0;m<nfreq;m++)
            sum+=delta[m];
        mean=sum/freqn;
        for(m=0;m<nfreq;m++)
            var_sum+=(delta[m]-mean)*(delta[m]-mean);
        std_dev=sqrt(var_sum/freqn);
            /* Update results if better matches     */
        for(n=0,TESTING=1;(n<list_length)&&TESTING;n++)
        {
            if(mean<freq_mean[n] && std_dev<freq_stddev[n])
            {
                TESTING=0;
                for(m=list_length-2;m>=n;m--)
                {
                    fs_list[m+1]=fs_list[m];
                    N_list[m+1]=N_list[m];
                    freq_mean[m+1]=freq_mean[m];
                    freq_stddev[m+1]=freq_stddev[m];
                }
                fs_list[n]=fs;
                N_list[n]=N;
                freq_mean[n]=mean;
                freq_stddev[n]=std_dev;
            }
        }
}
```

```
      /* Write results to output file      */
   fprintf(fptr,"\n");
   for(n=0;n<nfreq;n++)
   {
      fprintf(fptr,"freq%2d=%5.0f \n",n+1,freq[n]);
   }
   fprintf(fptr,"\n");
   for(n=0;n<list_length;n++)
   {
      fprintf(fptr,"N[%2d]=%4d \t",n,(int)N_list[n]);
      fprintf(fptr,"fs[%2d]=%6.0f \t",n,fs_list[n]);
      fprintf(fptr,"mean[%2d]=%6.0f \t",n,freq_mean[n]);
      fprintf(fptr,"stddev[%2d]=%6.0f \n",n,freq_stddev[n]);
   }
}
```

Appendix B: TMS320C30 Implementation Software

This appendix contains software to implement the core elements of the knock detection algorithm for a TI TMS320C30 DSP. It is not intended to be a complete description of the code required to implement a knock detection system.

Time-critical code is loaded to on-chip memory block B0. To minimize pipeline conflicts, variables accessed during the same instruction are located in different memory spaces. Variables required for the real-time code are split between the heap (dynamically allocated), which is placed in on-chip memory block B1, and external zero-wait-state SRAM.

The layout of the code is grouped into four sections:

- Definitions
  - Function prototypes
  - Definitions of constants
  - Global variables/pointers defined
- Main
- Interrupt Service Routines (ISR)
  - External interrupt #0: c_int0( )
  - Timer interrupt: c_int09( )
- Function Declarations
  - Global variable initialization: define( )
  - Dynamic memory allocation: make_space( )

Description

Dynamic Memory Initialization	The function make_space( ) contains the code to dynamically allocate structures and arrays.
RPM Adaptation	The ISR c_int01() contains the RPM adaptation code.
Cylinder Initialization	The ISR c_int01() contains the cylinder init code.
DFT Signal Conditioning	The ISR c_int09() contains the DFT algorithm.
Knock Intensity Metric	The ISR c_int09() contains the knock intensity metric calculation code.
Reference Adaptation	The ISR c_int09() contains the reference adaptation code.

Listing

```
/*-------------------------------------------------------------*/
/*-------------------------------------------------------------*/
/*                FUNCTION PROTOTYPES                          */
/*                                                             */
void   c_int01(void);            /* External Interrupt #1 */
void   c_int09(void);            /* Timer #1 Interrupt    */
       :
       :
void   comp_exp(void);          /* Cosine/sine tables     */
void   define(void);            /* Variable Definition    */
void   make_space(void);        /* Memory Allocation      */
       :
       :


/*-------------------------------------------------------------*/
/*-------------------------------------------------------------*/
/*                   C LANGUAGE MACROS                         */
/*                                                             */

#define   MAX(a,b)(((a) > (b))?(a):(b))
#define   MIN(a,b)(((a) < (b))?(a):(b))


/*-------------------------------------------------------------*/
/*-------------------------------------------------------------*/
/*                      DEFINITIONS                            */
/*                                                             */

#define pi   3.14159265358979323846

#define   clkin         32000000     /* DSP CLKIN rate (Hz)      */
#define   clk2cnt       4            /* No. clkins / timer count */
#define   clk2inst      2            /* No. clkins / instruction */
#define   cyl_number    6            /* No. cylinders            */
#define   fs            55555        /* Sample rate              */
#define   history       8            /* DFT ave filter length    */
#define   knockdeg      60           /* Knock window (°CA)       */
#define   knock_levels  4            /* No. knock metric levels  */
#define   nfreq         5            /* No. frequencies monitored */
#define   nrange        5            /* No. frequency ranges     */
#define   revprd2deg    1/720        /* Combustion cycles (CC )  */
                                     /* per °CA                  */
#define   tdc2knock     10           /* TDC to DFT calc (°CA)    */
#define   DFT_calc_cnts 1/125        /* tDFT+DFT time (CLKs)     */
#define   N             45           /* Sample block size        */

#define   cnt2sec   clk2cnt/clkin    /* Timer cnt period (sec)   */
#define   inst2sec  clk2isnt/clkin   /* Inst cycle time (sec)    */
#define   rev2delay tdc2knock/720    /* DFT delay per CC         */

/*

DFT frequency bin numbers for fs=47 kHz and N=45. These numbers are used to
generate the sine/cosine tables needed for the DFT calculations. The bin
numbers are different for different block sizes, sample rates, and desired
frequencies. If any of these values change, then new bin numbers must be
determined and entered here.

*/
#define nfreq1   6    /* f1 = 6.14 KHz      */
#define nfreq2   8    /* f2 = 8.18 KHz      */
#define nfreq3   11    /* f3 = 11.25 KHz      */
#define nfreq4   14    /* f4 = 14.32 KHz      */
#define nfreq5   19    /* f5 = 19.43 KHz      */
```

```
/*

The structure defined here contains the cylinder-specific information which
is used by the time-critical part of the DFT calculations. The definition of
the elements of the structure are:

   MAG[]  = computed DFT magnitude history array
   REF[] b = scaled/averaged DFT array
   knock[] = knock metric histogram (diagnostics)

*/

typedef struct cylinder
         {
          float MAG[nrange*(history+1)];
          float REF[nrange];
          float knock[knock_levels];
         } cyl_type, *cyl_pntr;

typedef unsigned short UINT; /* Shorthand definition      */

/*-----------------------------------------------------------*/
/*-----------------------------------------------------------*/
/*                   GLOBAL VARIABLES                        */
/*                                                           */

/* Create array of structures (1/cyl)        */
cyl_pntr cyl_list[cyl_number];

/* Create pointers to elements of structure  */
cyl_pntr   cp;
float     *Mag_p;
float     *Mag_p_src;
float     *Mag_p_dst;
float     *REFp;
float     *kp;

/* Create pointers to arrays in dynamic memory (heap)                */
float  *tDFT_p;               /* Intermediate |DFT| array            */
float  *DFT_p;                /* Final |DFT| array                   */
float  *real_p;               /* DFT[real] sum                       */
float  *imag_p;               /* DFT[imag] sum                       */

/* Allocate arrays for external SRAM memory                          */
float  real_exp[N];              /* Base cosine table array          */
float  imag_exp[N];              /* Base sine table array            */
float  real_table[(nfreq)*N];    /* Interleaved cosine table array   */
float  imag_table[(nfreq)*N];    /* Interleaved sine table array     */
float  K[cyl_number][nrange];    /* Gain array used for REF[] calc   */

/* Allocate global variables*/
float  delay;                    /* Delay period in Timer counts     */
float  fs;                       /* Sample rate (Hz)                 */
float  k_nblocks;                /* Conversion of Timer cnts to      */
                                 /* sample blocks / cylinder         */
float  rev2knock;                /* CC degrees / knock degrees       */
float  DFT_loop;                 /* Number Timer cnts / DFT calc loop*/
float  DFT_calc;                 /* Number Timer cnts / max DFT      */
                                 /* update loop                      */

/*

Offset pntr into base sine/cosine tables, which is used to construct the
interleaved sine/cosine tables. Each frequency being monitored requires
separate pointer into base tables.
/
int    offset1;
```

```
int    offset2;
int    offset3;
int    offset4;
int    offset5;

int    level_cntr;         /* Counter of DFT>REF for cyl */
int    nrpm;               /* Number RPM from RESET      */
int    ncyl;               /* Current cylinder pointer   */

/*-------------------------------------------------------------*/
/*-------------------------------------------------------------*/
/*         MAIN FUNCTION                */
/*                                      */
/*

Function main() is used for overall system control. It calls the functions
that do the initialization and setup tasks.

*/
void main(void)
{
       :
       :
   comp_exp(void);
   define();
   makespace();
       :
       :
   for (;;)
   ;       /* Infinite WAIT loop. All real-time    */
           /* code in interrupt routines.          */
}

/*-----------------------------------------------------*/
/*-----------------------------------------------------*/
/*         INTERRUPT SERVICE ROUTINES                   */
/*                                                      */
                :
                :
/*-----------------------------------------------------*/
/*    INT0 ISR                                          */
/*
Function c_int01() is the INT0 ISR, which is triggered by the ALL TDC signal.
The software always initializes Timer 0 Interrupt after the engine RPM calc
has stabilized (delay period needs to be known before DFT calcs started). If
CYL=1 (ncly=0), then delay period and nblocks are updated, otherwise those
blocks of code are bypassed.
*/
void   c_int01(void)
{
   UINT   *mem_pntr;          /* Pntr to memory      */
   float  *sum_p;             /* Pntr to |DFT| array */
   int    i;                  /* Counter             */

/*
Determine if CYL #1 by checking #1 TDC signal (LOW = yes). If yes, then read
720° CA period and run RPM adaptation routine.

*/
   mem_pntr = (UINT *) READ_#1_TDC;
   temp = *mem_pntr;
   if (temp == 0)
   {
      mem_pntr = (UINT *) T1_CNTR;         /*  Read 720° period */
      T1cnts = *mem_pntr;
      mem_pntr = (UINT *) T1_CNTL;         /*  Reset Timer 1 &  */
```

```
      *mem_pntr = TIMER_RUN;              /* restart counter  */
      nblocks = (int) T1cnts*k_nblocks;   /* Update nblocks   */
      if(nblocks==0) nblocks = 1;         /* nblocks  ≠ zero  */
      delay = T1cnts*rev2delay;           /* Update delay and */
      mem_pntr = (UINT *) T0_PRD;         /* Timer 0 PRD      */
      *mem_pntr = delay;                  /* (for next cyl)   */
      ncyl = 0;                           /* Reset cyl cntr   */
   }
/* Cylinder initialization */
   cp = cyl_list[ncyl];             /* Create pointers to      */
   Mag_p = cp –> MAG;               /* structure for current   */
   REFp = cp –> REF;                /* cylinder                */
   kp = cp –> knock;

   sum_p = DFT_p;                   /* Init final |DFT| array  */
   for (i=0 ; i<nfreq ; i++)        /* to zero                 */
        *sum_p++ = 0;
}

/*-------------------------------------------------------------*/
/*                        TIMER 0 ISR                          */
/*
Function c_int09() is the TIMER 0 ISR which is triggered at the end of Timer
0 count down period (10 °CA). All DFT algorithm routines are run in this ISR.
Communications of knock intensity to ECU also occurs in this ISR.
*/
void     c_int09(void)
{
   UINT *mem_pntr;         /* Pntr to C30 memory         */
   float *rtable_p;        /* Pntr to cos table          */
   float *itable_p;        /* Pntr to sine table         */
   float *rvar_p;          /* Pntr to DFT real sum       */
   float *ivar_p;          /* Pntr to DFT imag sum       */
   float *tsum_p;          /* Pntr to intermediate |DFT| */
   float *sum_p;           /* Pntr to final |DFT|        */
   float temp;             /* Storage for ADC input      */
   int i,j;                /* Counters                   */

/* Init ISR */
   reset_T0_ie();               /* Reset T0 enable bit in IE  */
   mem_pntr = (UINT *) Ain_DATA; /*Point to ADC I/O Port       */
/*
```

```
Start of outer loop of DFT real-time calculations. This loop is run once for
each sample block that is computed. It initializes inner loop variables, and
computes intermediate and final |DFT|s.
*/
   for (j=0 ; j<nblocks ; j++)
   {
      rvar_p = real_p;           /* Zero DFT real/imag      */
      ivar_p = imag_p;           /* sums. There is one      */
      for (i=0 ; i<nfreq ; i++)  /* real/imag pair for      */
      {                          /* each frequency being    */
         *rvar_p++ = 0;             /* monitored.              */
         *ivar_p++ = 0;
      }
      rtable_p = real_table;     /* Init pointers to        */
      itable_p = imag_table;     /* interleaved sine and    */
                                 /* cosine tables.          */
      tsum_p = tDFT_p;           /* Init pointers to        */
      sum_p = DFT_p;             /* intermediate and        */
                                 /* final |DFT| arrays.     */
      clr_int2();                /* Clear INT2 bit in IF    */
                                 /* to init ADC polling     */
/*
Start of inner loop of DFT real-time calculations. This loop is run once per
sample and computes the real/imaginary DFT running sums. It is reinitialized
for every sample block. The ADC sampling is polled rather than
interrupt-driven to save interrupt latency time.
*/
      for (i=0 ; i<N ; i++)
      {
         wait_int2();               /* Wait for INT2 flag      */
         ADC_in = *mem_pntr;        /* Read ADC                */
         clr_int2();                /* Clear INT2 IF           */
         temp = (float) ADC_in;     /* Convert to float        */
         rvar_p = real_p;           /* Init pntrs to DFT       */
         ivar_p = imag_p;           /* real/imag running       */
                                    /* sums.                   */
/*
This section computes the real and imaginary parts of the DFT. Inline code is
used to increase execution speed. Perform real/imag 2 mults and 2 adds for
each frequency being monitored.
*/
         *rvar_p++ += temp * *rtable_p++;  /* Freq #1 */
         *ivar_p++ += temp * *itable_p++;
         *rvar_p++ += temp * *rtable_p++;  /* Freq #2 */
         *ivar_p++ += temp * *itable_p++;
         *rvar_p++ += temp * *rtable_p++;  /* Freq #3 */
         *ivar_p++ += temp * *itable_p++;
         *rvar_p++ += temp * *rtable_p++;  /* Freq #4 */
         *ivar_p++ += temp * *itable_p++;
         *rvar_p += temp * *rtable_p++;    /* Freq #5 */
         *ivar_p += temp * *itable_p++;
      }
/* Completed inner loop    */
/*
This section computes the intermediate |DFT|^2 for the current sample block
and determines final |DFT|^2 for all sample blocks for current cylinder
firing. The |DFT|^2 is used instead of |DFT| to save CPU processing time. The
tests made to determine knock intensity are looking for changes, so do not
require mathematical definition of |DFT| to function correctly.
*/
      rvar_p = real_p;    /* Reinit pointers to  */
      ivar_p = imag_p;    /* real/imag DFT sums. */
/* Intermediate |DFT|^2 */
```

```
      *tsum_p++ = *rvar_p * *rvar_p++ + *ivar_p * *ivar_p++;
      *tsum_p++ = *rvar_p * *rvar_p++ + *ivar_p * *ivar_p++;
      *tsum_p++ = *rvar_p * *rvar_p++ + *ivar_p * *ivar_p++;
      *tsum_p++ = *rvar_p * *rvar_p++ + *ivar_p * *ivar_p++;
      *tsum_p   = *rvar_p * *rvar_p +   *ivar_p * *ivar_p;
/* Final |DFT|^2 */
      tsum_p = tDFT_p;          /* Init pntr to int |DFT| */
      *sum_p++ = MAX(*sum_p,tsum_p[0]);
      *sum_p++ = MAX(*sum_p,tsum_p[1]);
      *sum_p++ = MAX(*sum_p,tsum_p[2]);
      *sum_p++ = MAX(*sum_p,tsum_p[3]);
      *sum_p   = MAX(*sum_p,tsum_p[4]);

   }
/* Completed outer loop   */

/*
Compute knock metric. This is simple direct comparison method. Each monitored
frequency range is compared against the adaptive reference. The knock metric
is the number of frequency ranges that the measured magnitude is higher than
the reference magnitude.
*/
    level_cntr = 0;                      /* Init cntr for knock tests     */
   if(DFT_p[0] > REFp[0]) level_cntr++; /* Test DFTn>REFn for all ranges */
   if(DFT_p[1] > REFp[1]) level_cntr++;
   if(DFT_p[2] > REFp[2]) level_cntr++;
   if(DFT_p[3] > REFp[3]) level_cntr++;
   if(DFT_p[4] > REFp[4]) level_cntr++;
   kp = level_cntr;

/*
If revs > rev_count then, the knock intensity metric is valid so output to
ECU is enabled.
*/
   if(knock_out_flag)
   {
     mem_pntr = (UINT *) WRITE_PA1;
     *mem_pntr = kp;
   }
/*
This section tests level_cntr to determine if knock has occurred. If no, then
new scaled average magnitudes (REF) are computed and the arrays in structure
holding past values (MAG) are updated.
*/
   if (level_cntr == 0)
   {
/* Copy measured magnitudes into history array in structure */
     Mag_p[0] = DFT_p[0];
     Mag_p[1] = DFT_p[1];
     Mag_p[2] = DFT_p[2];
     Mag_p[3] = DFT_p[3];
     Mag_p[4] = DFT_p[4];
/* Move Mag[n]’s back one time step in array */
     for (i=0 ; i<history ; i++)
     {
        Mag_p_dst = Mag_p + ((history-i)*nrange);
        Mag_p_src = Mag_p_dst – nrange;
        for (j=0 ; j<nrange ; j++)
          Mag_p_dst[j] = Mag_p_src[j];
     }
/*
Compute new scaled average magnitude. The reference is the scaled average of
the last 8 non-knocking events. A unique scaling factor is used for each
frequency range of each cylinder.
*/
```

```
   for (i=0 ; i<nrange ; i++)    /* Zero REF array            */
      REFp[i] = 0;
   for (i=0 ; i<nrange ; i++)    /* Compute new REF for all   */
   {                             /* the frequency ranges.     */
      for (j=1 ; j<=history ; j++)
      {
         REFp[i] += Mag_p[(j*nrange)+i];
      }
      REFp[i] = 0.125 * REFp[i];
      REFp[i] = K[ncyl][i] * REFp[i];
   }
   ncyl++;                       /* Increment cylinder counter */
}
/*-------------------------------------------------------------*/
/*-------------------------------------------------------------*/
/*                FUNCTION DECLARATIONS                        */
/*                                                             */
                :
                :
/*-------------------------------------------------------------*/
/*              Define Complex Exponential                     */
/*                                                             */
/*
Function comp_exp() computes the complex cos (real) and sin (imag) table
arrays and then constructs the interleaved tables used for the DFT
calculations. Block size determines the table length.
*/
void   comp_exp(void)
{
   int j,k;   /* Counters   */
/* This section computes the base sine/cosine arrays which are N long. */
   for (j=0 ; j<N ; j++)
   {
     real_exp[j]=cos(2*pi*j/N);
     imag_exp[j]=sin(2*pi*j/N);
   }
/*
This section constructs the interleaved sine/cosine arrays from the base
arrays. This was done to save the time required to manage numerous circular
buffers during the real-time code execution. It is feasible in an
experimental system because of the extra memory available on the C30 board.
The interleaved arrays are each nfreq*N long. The method used to manage the
pointer to the base arrays works with any block length N.
*/
   for (j=0 ; j<N ; j++)
   {
     k=j*nfreq;          /* Init offset pointer   */
     real_table[k] = real_exp[offset1];
     imag_table[k] = imag_exp[offset1];
     offset1 += nfreq1;
     if (offset1 >= N)
        offset1 -= N;
     k ++;
     real_table[k] = real_exp[offset2];
     imag_table[k] = imag_exp[offset2];
     offset2 += nfreq2;
     if (offset2 >= N)
        offset2 -= N;
     k ++;
     real_table[k] = real_exp[offset3];
     imag_table[k] = imag_exp[offset3];
```

```
     offset3 += nfreq3;
     if (offset3 >= N)
        offset3 –= N;
     k ++;
     real_table[k] = real_exp[offset4];
     imag_table[k] = imag_exp[offset4];
     offset4 += nfreq4;
     if (offset4 >= N)
        offset4 –= N;
     k ++;
     real_table[k] = real_exp[offset5];
     imag_table[k] = imag_exp[offset5];
     offset5 += nfreq5;
     if (offset5 >= N)
        offset5 –= N;
   }
}
                :
                :
/*-------------------------------------------------------------*/
/*             Global Variable Initialization                  */
/*                                                             */
/*
Function define() is used to define the initial values of global variables.
*/
void   define(void)
{
   ADC_in = 0;
   knock_out_flag = 0;
   level_cntr = 0;
   ncyl = 0;
   nrpm = 0;
       :
       :
/*
NOTE: This section initializes the Kn's for each cylinder separately. If the
number of cylinders for the engine is not 6 (current value of cyl_number),
then the number of the Kn sets below has to be modified to match.
*/
   K[0][0] = 2.;    /* [Cylinder #1] [Range #1]   */
   K[0][1] = 2.;    /*               [Range #2]   */
   K[0][2] = 2.;    /*               [Range #3]   */
   K[0][3] = 2.;    /*               [Range #4]   */
   K[0][4] = 2.;    /*               [Range #5]   */
   K[1][0] = 2.;    /* Cylinder #2                */
   K[1][1] = 2.;
   K[1][2] = 2.;
   K[1][3] = 2.;
   K[1][4] = 2.;
   K[2][0] = 2.;    /* Cylinder #3                */
   K[2][1] = 2.;
   K[2][2] = 2.;
   K[2][3] = 2.;
   K[2][4] = 2.;
   K[3][0] = 2.;    /* Cylinder #4                */
   K[3][1] = 2.;
   K[3][2] = 2.;
   K[3][3] = 2.;
   K[3][4] = 2.;
   K[4][0] = 2.;    /* Cylinder #5   */
   K[4][1] = 2.;
```

```
   K[4][2] = 2.;
   K[4][3] = 2.;
   K[4][4] = 2.;
   K[5][0] = 2.;    /* Cylinder #6   */
   K[5][1] = 2.;
   K[5][2] = 2.;
   K[5][3] = 2.;
   K[5][4] = 2.;
      :
      :
/* This section computes a coefficient "k_nblocks" which is multiplied  */
/* the variable "T1cnts" (720 crank angle degrees period) to determine  */
/* the number of sample blocks to use for the current engine RPM. The   */
/* calculation is defined below:                                        */
/*                                                                      */
/*                                                                      */
/*                                                                      */
/*                                    CNT2SEC                           */
/*   K_NBLOCKS = REV2KNOCK * ----------------------                     */
/*                           (DFT_LOOP + DFT_CALC)                      */
/*                                                                      */
/*                                                                      */
/*                                                                      */
/* WHERE:                                                               */
/*                                                                      */
/*                                                                      */
/* REV2KNOCK: Fraction of Combustion Cycle (720 deg) that knock window  */
/*      exists for a single cylinder.                                   */
/*                                                                      */
/*  revprd   60 deg                                                     */
/*  ------- * --------                                                 */
/*  720 deg  knockprd                                                   */
/*                                                                      */
/*                                                                      */
/*                                                                      */
/*                                                                      */
/* CNT2SEC: Number of seconds per Timer count.                          */
/*                                                                      */
/*  4 clkins/T1cnt                                                      */
/*  --------------                                                      */
/*  32M clkins/sec                                                      */
/*                                                                      */
/*                                                                      */
/*                                                                      */
/*                                                                      */
/* DFT_LOOP: Number of seconds required for inner loop for given sample */
/*      rate and sample block size.                                     */
/*                                                                      */
/*  N samples    1  sec                                                 */
/*  --------- * ----------                                             */
/*   1 block    fs samples                                              */
/*                                                                      */
/*                                                                      */
/*                                                                      */
/*                                                                      */
/* DFT_CALC: Number of seconds required to compute |DFT|^2 and select   */
/*           the largest value. This number is approximate and is       */
/*           determined experimentally. If the technique to calculate   */
/*           magnitude or select max value is changed the time required */
/*           for this section should be checked and modified.           */
/*                                                                      */
/*   340 inst   2 clkins/inst                                             */
/*   -------- * --------------                                          */
```

```
/*    block     32M clkins/sec                                          */
/*                                                                      */
/*                                                                      */

  rev2knock = revprd2deg*knockdeg;
  DFT_loop = N/fs
  DFT_calc = DFT_calc_cnts*inst2sec;
  k_nblocks = rev2knock*(cnt2sec/(DFT_loop+DFT_calc));

}

/*--------------------------------------------------------*/
/*                    Array Allocation                    */
/*                                                        */
/*
Function make_space() is used to allocate memory space for arrays whose
length is determined at runtime. The values of the variable arrays in the
structure are initialized also. The dynamic memory allocation stores the
variables on the heap (ON-CHIP RAM BLK1) and accesses them using pointers.
This method is used to speed execution time of knock intensity metric.
*/

void   make_space(void)
{
   int i,j,k;

/* Allocate structure */
   for (i=0 ; i<cyl_number ; i++)
      cyl_list[i] = (cyl_pntr)malloc(sizeof(cyl_type));
/* Initialize structure */
   for (i=0 ; i<cyl_number ; i++)
   {
      cp = cyl_list[i];   /* Create pointers        */
      Mag_p = cp –> D;
      REFp = cp –> REF;
      kp = cp –> knock;
      for (j=0 ; j<knock_levels ; j++)
         kp[j] = 0.;      /* Init knock histogram    */
      for (j=0 ; j<nrange ; j++)
      {
         /* Init Mag's & REF's large pos. number.       */
         for (k=0 ; k<=history ; k++)
         {
            Mag_p[(k*nrange)+j] = 1.0e+37;
         }
         REFp[j] = 2.0*Mag_p[j];
      }
   }

/* Allocate dynamic arrays */
   tDFT_p = (float*)malloc(sizeof(float)*nfreq);
   DFT_p = (float*)malloc(sizeof(float)*nfreq);
   real_p = (float*)malloc(sizeof(float)*nfreq);
   imag_p = (float*)malloc(sizeof(float)*nfreq);
}
```

Appendix C: TMS320C25 Implementation Software

This appendix contains software to implement the core elements of the knock detection algorithm for a TI TMS320C25 DSP. It is not intended to be a complete description of the code required to implement a knock detection system.

The fixed-point implementation is intended to present code that is more appropriate for a production environment than the code in the ’C30 example. It is not as easily configurable because it is assumed that most constants are finalized and don’t need to be changed as often as in a development system. Many of the constants and variables computed in the ’C30 example must be defined by the user in this one.

The layout of the code is grouped into three sections:

- Definitions
  - Definitions of constants
  - Global variables/pointers defined
- Main
  - Global variable initialization
- Interrupt Service Routines (ISR)
  - External interrupt #0: int0
  - Timer interrupt: tint

Description

Algorithm Initialization	The main routine contains the algorithm initialization code.
RPM Adaptation	The INT0 ISR contains the RPM adaptation code.
Cylinder Initialization	The INT0 ISR contains the cylinder init code.
DFT Signal Conditioning	The TIMER ISR contains the DFT algorithm
Knock Intensity Metric	The TIMER ISR contains the knock intensity metric calculation code.
Reference Adaptation	The TIMER ISR contains the reference adaptation code.

Listing

```
*-------------------------------------------------------
*-------------------------------------------------------
*                        DEFINITIONS
*
ENABLE_int0   .set 01h         ;OR value w/ IMR to enable ext interrupt #0
DISABLE_int0  .set 0fffeh      ;AND value w/ IMR to disable ext interrupt #0
ENABLE_tint   .set 08h         ;OR value w/ IMR to enable timer interrupt
DISABLE_tint  .set 0fff7h      ;AND value w/ IMR to disable timer

N             .set 44          ;Sample block size
c_offset      .set N/4         ;Offset to start of cosine wave
nfreqs        .set 5           ;Number of monitored frequencies
cyl_num       .set 6           ;Number of cylinders
history       .set 8           ;DFT average filter length
power         .set 3           ;2^power = history
shift         .set 16-power    ;scaling for ave mag calculation
```

```
* The constant array_size defines the amount of memory required to
* store the individual cylinder data. The total amount of space
* available on-chip is 384 words. Each cylinder contains the
* following arrays:
*  K[nfreqs]
*  REF[nfreqs]
*  MAG[nfreqs*history]
* The constant cyl_size defines the number of memory locations for
* each cylinder. It is used to compute the constant array_size and
* to generate pointers into the array.
*
* The total number of memory locations required for this example is
* shown here:
*  array_size = 6*(2*5+5*8) = 300
*
cyl_size     .set 2*nfreqs+nfreqs*history
array_size   .set cyl_num*cyl_size

* Gain factors for K[] arrays. This example assumes the gain for each
* frequency is the same for each cylinder across the entire operating
* range. Might not be true in all cases.
*
K0     .set 6
K1     .set 2
K2     .set 4
K3     .set 5
K4     .set 4

* DFT frequency bin numbers for fs=47 KHz and N=45. These numbers
* are used to modify the sine/cosine table pointers needed for the
* DFT calculations. The bin numbers are different for various
* block sizes, sample rates, and desired frequencies. If any of
* these values change, then new bin numbers must be determined and
* entered here.
*
nfreq0    .set 6       ; f1 = 6.14 KHz
nfreq1    .set 8       ; f2 = 8.18 KHz
nfreq2    .set 11      ; f3 = 11.25 KHz
nfreq3    .set 14      ; f4 = 14.32 KHz
nfreq4    .set 19      ; f5 = 19.43 KHz

* This section shows how to compute the coefficient “k_nblocks” which is
* multiplied by the 24 bit variable “Timer” (time for 720° crank angle)
* to determine the number of sample blocks to use for the current engine
* RPM. The calculation is defined below using example numbers:
*
*
*
*                                  CNT2SEC            24
*   K_NBLOCKS = REV2KNOCK * ---------------------- * 2
*                           (DFT_LOOP + DFT_CALC)
*
*
* WHERE:
*
* REV2KNOCK: Fraction of Combustion Cycle (720 deg) that knock window
*            exists for a single cylinder.
*
*  revprd     60 deg             -2
*  ------- * -------- = 8.33*10
*  720 deg   knockprd
*

*
* CNT2SEC: Number of seconds per internal Timer count.
```

```
*
*   4 clkins/Tcnt          -7
*  -------------- = 1*10
*  40M clkins/sec
*
*
* DFT_LOOP: Number of seconds required for inner loop for given sample
*           rate and sample block size.
*
*  44 samples     1  sec               -4
*  ---------- * ----------- = 9.362*10
*   1 block     47K samples
*
*
*
*
* DFT_CALC: Number of seconds required to compute |DFT|^2 and select
*           the largest value. This number is approximate and is
*           determined experimentally. If the technique to calculate
*           magnitude or select max value is changed the time required
*           for this section should be checked and modified.
*
*   18 inst   5 freq   4 clkins/inst        -6
*   ------- * ------ * -------------- = 9*10
*     freq    block    40M clkins/sec
*
*
*
*
* WHICH GIVES:
*
*                                          -7
*                        -2              1*10              24
*  K_NBLOCKS =   8.33*10    *   -------------------- *   2   = 147.9
*                                        -4     -6
*                               9.362*10 + 9*10
*
*
KNBLOCK      .set 148
* This section shows how to compute the coefficient ”delay_frac” which is
* multiplied by the 24 bit variable “Timer” to determine the number of
* internal timer counts required for the specified delay period between
* TDC and the beginning of the knock window. The example below is for a
* desired delay period of 9°:
*
*                     9 deg    15
*   DELAY_FRACT = ------- * 2    = 409.6
*                     720 deg
DELAYFRACT   .set 410
*-------------------------------------------------------
*-------------------------------------------------------
*         DATA TABLE
* The sine table is a multiple of four long. When this
* configuration is used, the values at 0°, 90°, 180°, and 270°
* are included which gives more accuracy in the calculations
*
sintabl
   .word   00h
   .word   01238h
   .word   02410h
   .word   0352ch
   .word   04534h
```

```
   .word   053d2h
   .word   060bch
   .word   06baeh
   .word   0746eh
   .word   07ad0h
   .word   07eb2h
   .word   07fffh
   .word   07eb2h
   .word   07ad0h
   .word   0746eh
   .word   06baeh
   .word   060bch
   .word   053d2h
   .word   04534h
   .word   0352ch
   .word   02410h
   .word   01238h
   .word   01h
   .word   0edc8h
   .word   0dbf0h
   .word   0cad4h
   .word   0bacch
   .word   0ac2eh
   .word   09f44h
   .word   09452h
   .word   08b92h
   .word   08530h
   .word   0814eh
   .word   08000h
   .word   0814eh
   .word   08530h
   .word   08b92h
   .word   09452h
   .word   09f44h
   .word   0ac2eh
   .word   0bacch
   .word   0cad4h
   .word   0dbf0h
   .word   0edc8h
endtable

*-------------------------------------------------------
*-------------------------------------------------------
*         GLOBAL VARIABLES
*
; DFT algorithm variables
; Knock detection algorithm variables
; Cylinder history variables

;  BLOCK B2
;  --------
; These variables are used by the DFT algorithm and apply to all cylinders.
delay_frac    .usect ″dft_vars″, 1
delay         .usect ″dft_vars″, 1
flags         .usect ″dft_vars″, 1
k_nblocks     .usect ″dft_vars″, 1
metric        .usect ″dft_vars″, 1
nblocks       .usect ″dft_vars″, 1
ncyl          .usect ″dft_vars″, 1
tablend       .usect ″dft_vars″, 1
tdc#1         .usect ″dft_vars″, 1
temp          .usect ″dft_vars″, 1
timer_hi      .usect ″dft_vars″, 1
timer_lo      .usect ″dft_vars″, 1
```

```
ADC_in       .usect ”dft_vars”, 1
K            .usect ”dft_vars”, 1
MAG          .usect ”dft_vars”, 1
REF          .usect ”dft_vars”, 1

;  BLOCK B0 (DP=4)
;  ---------------
;
; sinewave table
   .bss   sinewave, N

; Pointers for magnitude and detection algorithms
   .bss   c_pntr,   nfreqs
   .bss   s_pntr,   nfreqs
   .bss   r_sum,    nfreqs
   .bss   i_sum,    nfreqs
   .bss   tDFT,     nfreqs
   .bss   DFT,      nfreqs

;  BLOCK B0 (DP=5) and BLOCK B1 (DP=6/7)
;  -------------------------------------
;
; This memory section starts at 0280h and contains arrays
; for individual cylinder variables. Each cylinder requires
; the following arrays:
;  K[nfreqs]
;  REF[nfreqs]
;  MAG[nfreqs*history]
; The magnitude array is organized as:
;  +----------------------
;  |      |history -->
;  |------+---------------
;  |nfreqs|
;  | |    |
;  | v    |
;  |

cylinder  .usect ”dft_array”, array_size

*-------------------------------------------------------------
*-------------------------------------------------------------
*         MAIN FUNCTION
*

   .mmregs
Reset
; Disable all interrupts
   ldpk   0      ;DP = mmregs data page
   lack   0      ;Reset IMR bits to disable interrupts
   sacl   IMR

; Disable OVERFLOW mode (ACC won’t saturate)
   rovm

; Setup P register shift mode
   spm    1      ;Shift left 1 on PREG ==> ACC

; Enable sign extention mode
   ssxm

; Init variables
   ldpk      delay_frac
   lalk      DELAYFRACT
   sacl      delay_frac
   lalk      KNBLOCK
   sacl      k_nblock
   zac
   sacl      ncyl
```

```
; Init sine table and tablend pointer
   ldpk      sinewave
   larp      AR1
   lrlk      AR1, sinewave
   rptk      endtable-sintabl-1
   blkp      sintabl,*+
   lalk      sinewave
   addk      N–1
   ldpk      tablend
   sacl      tablend

; Init K[] array. This example uses same freq gain for each cylinder.
; In actual implementation all gains might be different.
;
   lark   AR0, nfreq–1     ;Init counter
   lrlk   AR1, cylinder    ;Init cylinder pointer
repeat
   larp   AR1
   lack   K0               ;Init K[0]
   sacl   *+
   lack   K1               ;Init K[1]
   sacl   *+
   lack   K2               ;Init K[2]
   sacl   *+
   lack   K3               ;Init K[3]
   sacl   *+
   lack   K4               ;Init K[4]
   sacl   *+
   adrk   cyl_size-nfreqs,AR0 ;Point to K[] for next cylinder
   banz   repeat, AR1
                :
                :
Wait
   b      Wait             ; Endless loop

*-------------------------------------------------------
*-------------------------------------------------------
*         INTERRUPT SERVICE ROUTINES
*
*
*-------------------------------------------------------
*    INT0 ISR
*
* ISR INT0 is triggered by the ALL TDC signal. The software
* always initializes Timer 0 Interrupt after the engine
* RPM calc has stabilized (delay period needs to be known
* before DFT calcs started. If CYL=1 (ncly=0), then delay
* period and nblocks are updated, otherwise those blocks of
* code are bypassed.

; Determine if cylinder #1 by checking #1 TDC signal (LOW = yes).
; If yes, then read external timer to determine 720° period and
; run RPM adaptation routine.
   in     tdc#1, PA2       ; Read #1 TDC signal
   bit    tdc#1, 0fh       ; Test for cylinder #1
   bbnz   not_#1           ; If no, skip delay/nblock calc

; Read 24 bit external timer to obtain combustion cycle period.
   in     timer_hi, PA3    ; Read upper byte of timer
   in     timer_lo, PA4    ; Read lower word of timer

; Delay count calculation
   lt     delay_frac
   mpyu   timer_lo
```

```
   mpya   timer_hi
   sach   temp
   pac
   add    temp
   sacl   delay
; NBLOCKS calculation
   spm    0
   lt     k_nblock
   mpyu   timer_lo
   mpya   timer_hi
   sach   temp
   pac
   add    temp
   sacl   temp
   lac    temp, 8
   sach   nblocks
   spm    1
   zac                    ;Set ncyl=0
   ldpk   ncyl
   sacl   ncyl
not_#1
; Initialize pointers to cylinder arrays K[], REF[], and MAG[]
   lalk   cylinder        ;ACC = cylinder address
   spm    2               ;Setup for Q27 multiply
   ldpk   temp
   lack   nfreqs
   sacl   temp
   lt     ncyl            ;Calculate offset for
   mpyk   cyl_size        ; current cylinder
   apac                   ;Compute K[] address
   sach   K
   addh   temp            ;Compute REF[] address
   sach   REF
   addh   temp            ;Compute MAG[] address
   sach   MAG
   spm    1               ;Restore PM to Q30 multiply
; Zero DFT array for use in knock detection algorithm
   lrlk   AR0, DFT
   larp   AR0
   zac
   rptk   nfreq–1
   sacl   *+
                :
                :
*-------------------------------------------------------------
*    TIMER ISR
*
The TIMER ISR is triggered at the end of the on-chip timer count down period
(10 °CA). All DFT algorithm routines are run in this ISR. Communications of
knock intensity to ECU also occurs here.
*
tint
; disable Timer interrupt
   ldpk   0
   lac    IMR
   andk   DISABLE_tint
   sacl   IMR
   in     ADC_in, PA0         ;Init ADC conversions
;START BLOCK PROCESSING
```

```
    ldpk   nblocks
    lar    AR4, nblocks        ;Init block counter

blocks

;Zero real and imag sums
    lrlk   AR1, r_sum          ;AR1 = real sum array
    lrlk   AR2, i_sum          ;AR2 = imag sum array
    lark   AR3, nfreqs–1       ;AR3 = real/imag sum array counter
    larp   AR1
    zac
zero
    sacl   *+, AR2          ;Zero real sum element
    sacl   *+, AR3          ;Zero imag sum element
    banz   zero, AR1        ;Check if done

;Init cosine/sine table pointers
    lrlk   AR1, c_pntr      ;AR1 = cosine pointer array
    lrlk   AR2, s_pntr      ;AR2 = sine pointer array
    lark   AR3, nfreqs–1    ;AR3 = cos/sin pntr array counter
    larp   AR2
setup
    lalk   sinewave         ;ACC = start of sine table
    sacl   *+, AR1          ;Init sine pointer element
    addk   c_offset         ;ACC = start of cosine table
    sacl   *+, AR3          ;Init cosine pointer element
    banz   setup, AR2       ;Check if done

    lark   AR3, N–1         ;Init sample counter
    larp   AR1              ;ARP = AR1 (cos table pointer)

;START DFT PROCESSING

dft
; The BIO pin is polled to determine when ADC conversion is complete.
; This can be used to eliminate interrupt latency. If knock detection
; is not implemented on dedicated processor, an interrupt could be used
; for this section of the calculations.
; backward
    bioz   forward          ; If ADC complete continue
    b      backward         ; else wait
forward

;Read latest measurement from Port 0. ADC input is left justified.
    ldpk   ADC_in
    in     ADC_in, PA0
    lt     ADC_in           ;Load T reg with latest measurement

; Compute running DFT calculation. A different index is loaded into AR0
; for each frequency being monitored.

; Frequency 0
    lark   AR0, nfreq0      ;AR0 = freq0 index
    lar    AR1, c_pntr      ;AR1 = freq0 cos table pntr
    lar    AR2, s_pntr      ;AR2 = freq0 sin table pntr
    mpy    *0+, AR2         ;P = x*cos(freq0)
    zalh   r_sum            ;ACC = real sum0
    mpya   *0+, AR1         ;P = x*sin(freq0) & accum real product
    sach   r_sum            ;store updated real sum0
    zalh   i_sum            ;ACC = imag sum0
    sar    AR1, c_pntr      ;store freq0 cos pntr
    sar    AR2, s_pntr      ;store freq0 sin pntr

; Frequency 1
    lark   AR0, nfreq1      ;AR0 = freq1 index
    lar    AR1, c_pntr+1    ;AR1 = freq1 cos table pntr
    lar    AR2, s_pntr+1    ;AR2 = freq1 sin table pntr
```

```
   mpya   *0+, AR2         ;P = x*cos(freq1) & accum imag product
   sach   i_sum+0          ;store updated imag sum0
   zalh   r_sum+1          ;ACC = real sum1
   mpya   *0+, AR1         ;P = x*sin(freq1) & accum real product
   sach   r_sum+1          ;store updated real sum1
   zalh   i_sum+1          ;ACC = imag sum1
   Sar    AR1, c_pntr+1    ;store freq1 cos pntr
   sar    AR2, s_pntr+1    ;store freq1 sin pntr

; Frequency 2
   lark   AR0, nfreq2      ;AR0 = freq2 index
   lar    AR1, c_pntr+2    ;AR1 = freq2 cos table pntr
   lar    AR2, s_pntr+2    ;AR2 = freq2 sin table pntr
   mpya   *0+, AR2         ;P = x*cos(freq2) & accum imag product
   sach   i_sum+1          ;store updated imag sum1
   zalh   r_sum+2          ;ACC = real sum2
   mpya   *0+, AR1         ;P = x*sin(freq2) & accum real product
   sach   r_sum+2          ;store updated real sum2
   zalh   i_sum+2          ;ACC = imag sum2
   sar    AR1, c_pntr+2    ;store freq2 cos pntr
   sar    AR2, s_pntr+2    ;store freq2 sin pntr

; Frequency 3
   lark   AR0, nfreq3      ;AR0 = freq3 index
   lar    AR1, c_pntr+3    ;AR1 = freq3 cos table pntr
   lar    AR2, s_pntr+3    ;AR2 = freq3 sin table pntr
   mpya   *0+, AR2         ;P = x*cos(freq3) & accum imag product
   sach   i_sum+2          ;store updated imag sum2
   zalh   r_sum+3          ;ACC = real sum3
   mpya   *0+, AR1         ;P = x*sin(freq3) & accum real product
   sach   r_sum+3          ;store updated real sum3
   zalh   i_sum+3          ;ACC = imag sum3
   sar    AR1, c_pntr+3    ;store freq3 cos pntr
   sar    AR2, s_pntr+3    ;store freq3 sin pntr

; Frequency 4
   lark   AR0, nfreq4      ;AR0 = freq4 index
   lar    AR1, c_pntr+4    ;AR1 = freq4 cos table pntr
   lar    AR2, s_pntr+4    ;AR2 = freq4 sin table pntr
   mpya   *0+, AR2         ;P = x*cos(freq4) & accum imag product
   sach   i_sum+3          ;store updated imag sum3
   zalh   r_sum+4          ;ACC = real sum4
   mpya   *0+, AR1         ;P = x*sin(freq4) & accum real product
   sach   r_sum+4          ;store updated real sum4
   zalh   i_sum+4          ;ACC = imag sum4
   apac                    ;acc imag product
   sach   i_sum+4          ;store updated imag sum4
   sar    AR1, c_pntr+4    ;store freq4 cos pntr
   sar    AR2, s_pntr+4    ;store freq4 sin pntr

; Check for cosine/sine table pointer overrun
   lac    c_pntr           ; Check freq0 cosine pointer
   sub    tablend
   blez   spntr0
   adlk   sinewave         ; Adjust pointer if overrun occurs
   sacl   c_pntr
spntr0
   lac    s_pntr           ; Check freq0 sine pointer
   sub    tablend
   blez   rpntr1
   adlk   sinewave         ; Adjust pointer if overrun occurs
   sacl   s_pntr

rpntr1
   lac    c_pntr+1         ; Check freq1 cosine pointer
   sub    tablend
```

```
    blez   spntr1
    adlk   sinewave        ; Adjust pointer if overrun occurs
    sacl   c_pntr+1
spntr1
    lac    s_pntr+1        ; Check freq1 sine pointer
    sub    tablend
    blez   rpntr2
    adlk   sinewave        ; Adjust pointer if overrun occurs
    sacl   s_pntr+1

rpntr2
    lac    c_pntr+2        ; Check freq2 cosine pointer
    sub    tablend
    blez   spntr2
    adlk   sinewave        ; Adjust pointer if overrun occurs
    sacl   c_pntr+2
spntr2
    lac    s_pntr+2        ; Check freq2 sine pointer
    sub    tablend
    blez   rpntr3
    adlk   sinewave        ; Adjust pointer if overrun occurs
    sacl   s_pntr+2

rpntr3
    lac    c_pntr+3        ; Check freq3 cosine pointer
    sub    tablend
    blez   spntr3
    adlk   sinewave        ; Adjust pointer if overrun occurs
    sacl   c_pntr+3
spntr3
    lac    s_pntr+3        ; Check freq3 sine pointer
    sub    tablend
    blez   rpntr4
    adlk   sinewave        ; Adjust pointer if overrun occurs
    sacl   s_pntr+3

rpntr4
    lac    c_pntr+4        ; Check freq4 cosine pointer
    sub    tablend
    blez   spntr4
    adlk   sinewave        ; Adjust pointer if overrun occurs
    sacl   c_pntr+4
spntr4
    lac    s_pntr+4        ; Check freq4 sine pointer
    sub    tablend
    blez   pntr_end
    adlk   sinewave        ; Adjust pointer if overrun occurs
    sacl   s_pntr+4

pntr_end
    banz   dft,AR1         ; Go back if more samples
; end inner loop

;Compute magnitudes: tDFT
    zac                    ;Clear ACC
    mpyk   0               ;Clear P reg
    sqra   r_sum           ;P = real0^2
    sqra   i_sum           ;P = imag0^2 / ACC = real0^2

    sqra   r_sum+1         ;P = real1^2 / ACC = real0^2+imag0^2
    sach   tDFT            ;Store |DFT0|
    zac                    ;Clear ACC
    sqra   i_sum+1         ;P = imag1^2 / ACC = real1^2

    sqra   r_sum+2         ;P = real2^2 / ACC = real1^2+imag1^2
    sach   tDFT+1          ;Store |DFT1|
```

```
    zac                     ;Clear ACC
    sqra   i_sum+2          ;P = imag2^2 / ACC = real2^2

    sqra   r_sum+3          ;P = real3^2 / ACC = real2^2+imag2^2
    sach   tDFT+2           ;Store |DFT2|
    zac                     ;Clear ACC
    sqra   i_sum+3          ;P = imag3^2 / ACC = real3^2

    sqra   r_sum+4          ;P = real4^2 / ACC = real3^2+imag3^2
    sach   tDFT+3           ;Store |DFT3|
    zac                     ;Clear ACC
    sqra   i_sum+4          ;P = imag4^2 / ACC = real4^2
    apac                    ;ACC = real4^2+imag4^2
    sach   tDFT+4           ;Store |DFT4|

;Determine max magnitude: MAX(DFT,tDFT)
    ldpk   DFT
    lac    DFT              ;ACC = DFT0
    sub    tDFT             ;ACC = DFT0 – tDFT0
    bgz    max1             ;If DFT > tDFT, no change
    lac    tDFT             ;Else DFT = tDFT
    sacl   DFT
max1
    lac    DFT+1            ;ACC = DFT1
    sub    tDFT+1           ;ACC = DFT1 – tDFT1
    bgz    max2             ;If DFT > tDFT, no change
    lac    tDFT+1           ;Else DFT = tDFT
    sacl   DFT+1
max2
    lac    DFT+2            ;ACC = DFT2
    sub    tDFT+2           ;ACC = DFT2 – tDFT2
    bgz    max3             ;If DFT > tDFT, no change
    lac    tDFT+2           ;Else DFT = tDFT
    sacl   DFT+2
max3
    lac    DFT+3            ;ACC = DFT3
    sub    tDFT+3           ;ACC = DFT3 – tDFT3
    bgz    max4             ;If DFT > tDFT, no change
    lac    tDFT+3           ;Else DFT = tDFT
    sacl   DFT+3
max4
    lac    DFT+4            ;ACC = DFT4
    sub    tDFT+4           ;ACC = DFT4 – tDFT4
    bgz    max_end          ;If DFT > tDFT, no change
    lac    tDFT+4           ;Else DFT = tDFT
    sacl   DFT+4
max_end
    larp   AR4              ;Set ARP
    banz   blocks           ;Go back if more blocks
; end outer loop

;Knock detection algorithm
    lark   AR0, 0           ; Init counter
    ldpk   DFT
    lar    AR1, DFT         ; Point to magnitude array
    ldpk   REF
    lar    AR2, REF         ; Point to reference array
    larp   AR1              ; Activate AR1

    lac    *+, AR2          ; ACC = DFT
    sub    *+, AR1          ; ACC = DFT–REF
    blez   dif1             ; Is DFT>REF?
    larp   AR0              ; Yes – increment counter
    mar    *+, AR1
dif1                         ; No – test next frequency range
    lac    *+, AR2          ; ACC = DFT
```

```
    sub    *+, AR1          ; ACC = DFT–REF
    blez   dif2             ; Is DFT>REF?
    larp   AR0              ; Yes – increment counter
    mar    *+, AR1
dif2                        ; No – test next frequency range
    lac    *+, AR2          ; ACC = DFT
    sub    *+, AR1          ; ACC = DFT–REF
    blez   dif3             ; Is DFT>REF?
    larp   AR0              ; Yes – increment counter
    mar    *+, AR1
dif3                        ; No – test next frequency range
    lac    *+, AR2          ; ACC = DFT
    sub    *+, AR1          ; ACC = DFT–REF
    blez   dif4             ; Is DFT>REF?
    larp   AR0              ; Yes – increment counter
    mar    *+, AR1
dif4                        ; No – test next frequency range
    lac    *+, AR2          ; ACC = DFT
    sub    *+, AR1          ; ACC = DFT–REF
    blez   dif_end          ; Is DFT>REF?
    larp   AR0              ; Yes – increment counter
    mar    *+, AR1
dif_end                     ; No – save test results
    sar    AR0, metric

; If knock_flag set (revs > rev_count) then, the knock intensity metric
; is valid so output to ECU is enabled.
    bit    flags, 0fh       ; Test knock_flag
    bbz    update           ; Skip output if flag=0
    out    PA1, metric      ; Output to ECU

; Reference adaptation if no knock occurred
update
    lac    metric           ; Test for knock
    bez    done             ; Skip reference update if knock occurred

; Copy measured magnitudes into history array in structure
    larp   AR1
    lark   AR0, history
    lrlk   AR1, MAG
    rptk   nfreqs–1
    blkd   DFT,*0+

; Move Mag[] elements back one time step in array for each frequency
; (must be onchip to use DMOV)
    lark   AR0, nfreq–1
    lrlk   AR1, MAG+(nfreqs*history–2)
move
    rptk   history–1        ;Move current frequency history
    dmov   *–
    mar    *–               ;Setup for next frequency
    mar    *–
    larp   AR0
    banz   move, AR1        ;Check if done

; Compute new average magnitude (use power of two length: 4,8,16)
    lrlk   AR1, MAG         ; Point to start of magnitude array
    lrlk   AR2, REF         ; Point to start of reference array
    lark   AR3, nfreqs–1    ; Init frequency counter
    larp   AR1
average
    zac                     ; Zero ACC
    rptk   history–1        ; Sum history of magnitudes
    add    *+, shift, AR2   ; w/ 1/history scaling for average
    sach   *+, AR3          ; Store average to REF array
    banz   average, AR1
```

```
; Scale average magnitudes to create new REF array. This technique
; assumes that K*REF<32767. If this condition is not met, then additional
; scaling must be done.
   ldpk   K
   lar    AR1, K           ; Point to start of scale factor array
   lar    AR2, REF         ; Point to start of reference array
   lark   AR3, nfreqs–1    ; Init frequency counter
scale
   lt     *+, AR2          ; T = K (scale factor)
   mpy    *                ; P = K*REF (scaled ave)
   spl    *+, AR3          ; Store back to REF array
   banz   scale, AR1       ; Check for more

done
   ldpk   ncyl
   lac    ncyl             ; Increment cylinder counter
   addk   1
   sacl   nycl

   eint                    ; and exit
   ret
```

Index

A

adaptation strategy 12
adapting to operating conditions 12
analog filtering 3
analog-to-digital converter (ADC) resolution 16

B

block-size search software listing 23
broadband sensors 2

C

contamination of received signal 1, 13
control strategies 4

D

detection strategies 3
  DFT detection strategy 4, 9
detonation 1, 3
DFT 4
  advantages 5–7
  sliding mode 8
DFT detection strategy 9
DFT signal conditioning algorithm 5
  variables affecting 5
DFT versus digital filter 7
DFT versus FFT 6, 7
DFT versus FIR filter 7
digital filtering 3
direct measurement of engine knock 1
discrete Fourier transform 4
  advantages 5–7

E

engine knock, definition 1

F

fast Fourier transform 3, 7
FFT 3, 7
finite impulse response (FIR) filter 7
fixed-point (TMS320C25) implementation 18
  block diagram 19
  software listing 40
floating-point (TMS320C30) implementation 16
  block diagram 17
  software listing 29
f_N (Nyquist rate) 5
frequency selection in implementation 15
fundamental frequency, variables affecting 2

G

global knock control
  advantage 4
  disadvantage 4
Goertzel algorithm 8

I

implementation examples 13
indirect measurement of engine knock. *See* remote measurement of engine knock
individual cylinder control
  advantage 4
  disadvantage 4

K

knock detection
  algorithm adaptation 2
  control strategies 4
    global control 4
    individual cylinder control 4
  overview 2
  strategies 3
knock detection algorithms, variables affecting 2–3
knock sensors, types 1
  pressure sensors 1
  remote sensors 1
knock window 3

M

magnitude detection strategies 9
misfire detection 21

N

Nyquist rate (f_N) 5

P

performance differences between floating- and fixed-point implementations 20

R

references 22

remote measurement of engine knock 1

remote measurement of knock detection, advantages 2

remote sensors
- broadband 2
- tuned 2

resonant sensors 2

S

sample-rate search software listing 23

signal conditioning 3
- with a broadband sensor 3
- with a tuned sensor 3
- with broadband sensor 5
  - analog filtering 3
  - digital filtering 3
  - spectral analysis 3

signal strength magnitude 7

spectral analysis 3

T

timing considerations in implementation 14

tuned sensors 2
- limitation 2

V

variables affecting fundamental frequency 2

W

weighting coefficient in adaptive DFT computation 12

*This template is for the "*See*" and "*See also*" references in your index. Since these entries do not have a page number associated with them, it's extremely difficult to locate one if you need to modify or delete it and you don't remember which chapter it's in. By using this template, you can alphabetize your entries according to the first letter of the first level entry.*

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.