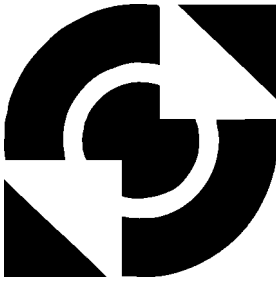
University of Twente



EEMCS / Electrical Engineering
Control Engineering

Universidad de Navarra
Nafarroako Unibertsitatea
Escuela Superior de Ingenieros
Ingeniarien Goi Mailako Eskola

Feasibility study of implementing a new algorithm to measure the frequency in a universal power measuring device.

Víctor Gutiérrez Pérez

MSc report

Supervisors
Prof.dr.ir. P.P. Regtien
Ing. R.M. Klomp
Dr. L. Fontán

January 2009

Report nr. 002CE2009
Control Engineering
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

This report discusses the possibility of implementing a new method to measure the frequency of a power supply. Starting point is a universal measuring transducer called EM4000AC produced by ELEQ, a company dedicated to the electrical sector, situated in Steenwijk (The Netherlands). The inside microprocessor responsible for all the calculations is a Digital Signal Processor (dsPIC30F6011A) fabricated by Microchip. The DSP functionality available in the current hardware is not used yet. The calculation of the frequency is an important parameter for the correct performance of the whole device.

Several algorithms to improve the correct operation are studied and evaluated. The possibility of changing the hardware using new faster micro-processors available in the current market is also investigated.

Preface

The frequency of a power system is an important operational parameter for the safety, stability, and efficiency of the power system. Reliable frequency measurement is prerequisite for effective power control, load shedding, load restoration, and generator protection. Sometimes it is necessary to know the apparent, active of reactive power in the network. These parameters can be measured by a transducer but it is necessary an accurate result. This is the objective of these pages. I hope my MSc-thesis will be helpful for ELEQ.

I would like to take this opportunity to thank all the people that I have had the opportunity to work with. First and foremost I want to thank my supervisor P. P. L. Regtien and the responsible in ELEQ ing. R. M. Klomp, not only for the technical help, but also the enthusiasm they have for the project. Further, I want to thank Iker for the help in some difficult moments. I would also thank my parents, Emilio and Lucia, for their continued support.

Victor Gutierrez
January 2009

List of abbreviations

ASIC	Application Specific Integrated Circuit
ASSP	Application Specific Standard Product
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DFT	Discrete Fourier Transform
DIF	Decimation-in-Frequency
DIT	Decimation-in-Time
DSP	Digital Signal Processing / Digital Signal Processor
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
MCU	Microcontroller Unit
MIPS	Million Instructions Per Second
PCB	Printed Circuit Board
PLD	Programmable Logic Device
PLL	Phase-Locked Loop
RAM	Random Access Memory
VHDL	VHSIC hardware description language
VHSIC	Very High Speed Integrated Circuit

Contents

List of Figures

List of Tables

1 Chapter 1 - Introduction

1.1 ELEQ

ELEQ is the name of the combination of FAGET and KWK. As of 2003, we make up an international co-operation so as to be of better service to the European market. Our programme comprises a wide range of transformers, transducers, meters and connection techniques. Quality products that find their way to electricity companies, the industry sector and technical wholesalers, among others.

ELEQ is a company dedicated to the electrical sector. It is situated in Steenwijk (Netherlands). ELEQ designs and manufactures a broad programme of transformers, transducers, meters and connection techniques.

FIGURE 1.1 - ELEQ, Mastering Electricity.

ELEQ works in three different areas: metering, lighting and protection.

Metering: Measuring equipment and energy management for electricity companies, marine and the industrial sector. Transformers, transducers and meters.

Lighting: Connection boxes and management software for public lighting.

Protection: Current/voltage transformer systems for manufacturers in the electricity transport and distribution market (power transformer-, switchgear-, and generator-builders).

1.1.1 Metering

They offer to electricity companies, the navy, the industrial sector and their suppliers a broad programme of products for measuring purposes and energy management:

- Special solutions in cast resin
- Standard range current transformers
- Special products
- Measuring transducers
- Measuring instruments

1.1.2 Ligthing

They offer municipalities, provinces, sub-contractors, lamppost manufacturers and airports the following specialist products:

- Lighting mast sets
- Public lighting management systems

1.1.3 Protection

For the manufacturers of power transformers, power-cutout switches and generators, among other companies, they offer the following programme:

- Solutions in cast resin
- Current transformers
- Current transformer systems

One of the products that ELEQ produces is a transducer denominated: “FAGET Universal measuring transducer EM4000” with the following general characteristics:

- Compact housing
- High accuracy
- Full galvanic separation
- MODbus (RS232/RS485)
- 3 Analogue + 1 Pulse output
- Direct measurement up to 690V
- 48 hour service EXW

This device is going to be studied in detail in the Section 1.2 of the Chapter 1. (6)

1.2 The EM4000AC

FIGURE 1.2 - EM4000AC

1.2.1 Introduction

At present, ELEQ has, among others, a stable product, a multifunction transducer denominated EM4000AC.

This device is able to measure all this following functions:

- Voltages L-L (select 2 phases) (V)
- Voltages L-N (V)
- Current (A)
- Frequency (Hz)
- Active power (Pw) (W)
- Reactive power (Pq) (VAr)
- Apparent power (Ps) (VA)
- Power factor

- cos φ
- sin φ
- φ
- Real energy consumption (Wh)
- Reactive energy consumption (Varh)
- Reactive energy consumption (VAh)

1.2.2 How EM4000AC works

This device is built in different hardware modules. The most important one is where all the calculations are processed. The "brain" of this module is a High-Performance 16-bit Digital Signal Controller denominated dsPIC30F6011A.

To be able to calculate all the parameters first of all it is necessary to calculate the frequency of the signal in a precise way because the rest of the parameters depend on the accuracy of the frequency.

It can be seen from Figure 1.3 and Figure 1.4 a small schematic of the hardware.

To be able to measure it, it is necessary to make some previous transformations to reduce the value of the input signal as can be seen in Figure 1.3.

The device is supplied by the voltage and current to be measured (VR, VS, VT, IR, IS and IT) and also with an external power supply to work. These voltages and currents are reduced their value in three voltage and current transformers. After this transformation, the signal (UL1', UL2', UL3', IL1', IL2'and IL3',) is carried to the principal PCB (Figure 1.4) and processed in it. In this PCB it is filtered and amplified to introduce the new signal in the Digital Signal Processor dsPIC30F6011A which will calculate all the parameters as can be seen in Figure 1.4.

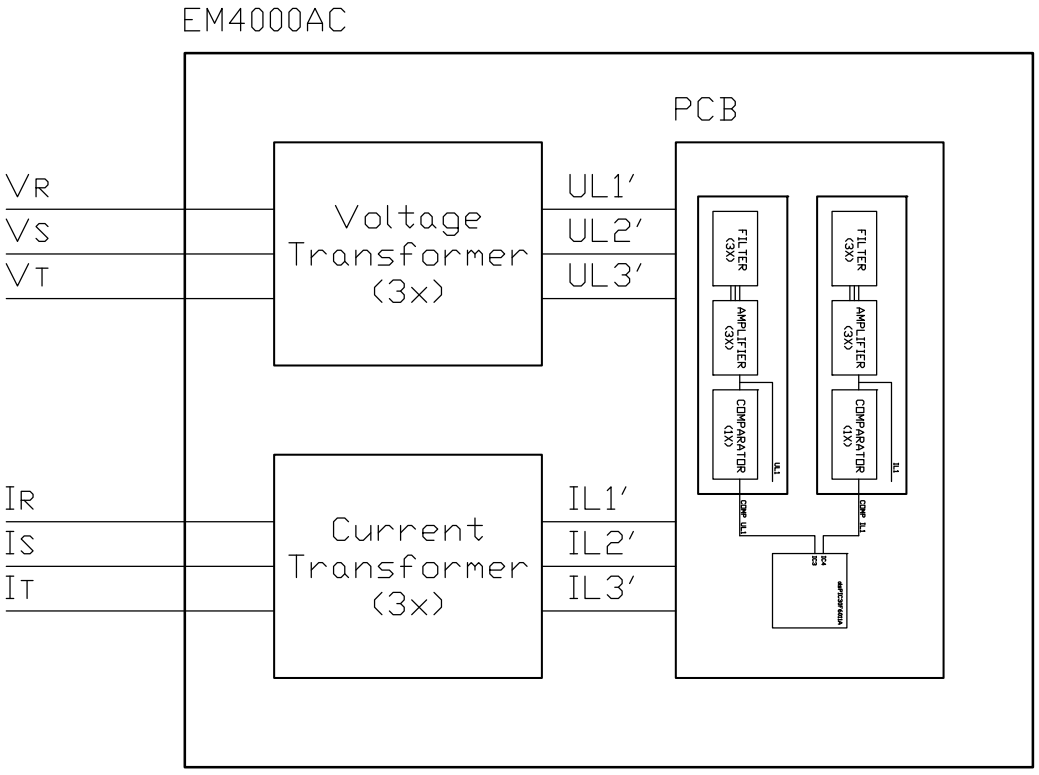


FIGURE 1.3 - Small schematic of the hardware in EM4000AC.

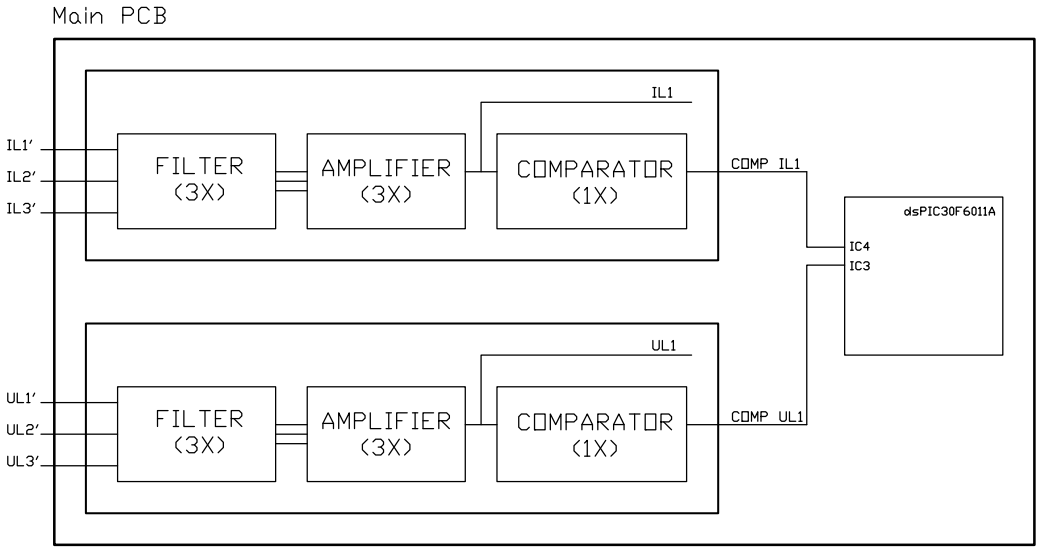


FIGURE 1.4 - Small schematic of the main PCB in EM4000AC.

This new signal is introduced in the DSP through two pins. The pins 44 and 45 correspond to two hardware interrupts (IC3 and IC4) depending on if the frequency is measured with the voltage or with the current.

Now, the calculation of the frequency is carried by the software of the Digital Signal Processor.

To be able to explain how it works it is necessary to know some other previous configurations like TIMER1 configuration.

In the 16-bit Timer mode, the timer increments on every instruction cycle up to a value loaded into the Period register PR1, then resets to '0' and continues to count. The timer also has the ability to generate an interrupt on every period. When the timer count matches the Period register, the T1IF bit is asserted and an interrupt will be generated if enabled. The T1IF bit must be cleared in software. The timer interrupt flag, T1IF, is located in the IFS0 Control register in the interrupt controller.

The configuration of the hardware interrupts IC3 and IC4 interrupt is the next:

- Operate in sleep mode.
- Interrupt on first capture.
- Interrupt every rise edge.

Explanation:

In the first positive edge of the signal introduced in IC3 or IC4 a flag is activated (IC3IF=1 or IC4IF=1). In this moment the TIMER1 is reinitialized and begins to count until 16,000 (the value of this constant can be changed modifying PR1SETTING in *globals.h*) which mean 1 ms. When the TIMER1 counts 1 ms, a flag is activated (T1IF=1), and also it is counted the number of times (*uFreqTimer*) that this flag is activated between two positive edges of the signal introduced in IC3 or IC4. When the second positive edge of the signal arrives, the flag IC3IF is activated again and in this moment the value of TMR1 is stored in a variable (*iTickCurrent*). It can be seen in the Figure 1.5. The *CPU_CYCLE_FREQ* is 16 MHz which means that the ticks are generated with this frequency.

With the Equation 1.1 can be calculated the number of ticks elapsed between two consecutive flags of IC3 or IC4.

$$\text{uFrequencyTicks} = \text{iTickCurrent} + \left(\text{uFreqTimer} \times \text{PR1SETTING}\right) \tag{1.1}$$

where

iTickCurrent = Current value of TIMER1 = Ticks elapsed between the last TIMER1 active flag (T1IF=1) and the second positive edge of the input signal in IC3 or IC4 (IC3IF=1 or IC4IF=1).

uFreqTimer = Integer number of times that T1IF is activated between two positive edges of the signal introduced in IC3 or IC4, in other words integer number of milliseconds between two positive edges.

$PR1SETTING = \frac{\text{CPU\_CYCLE\_FREQ}}{1000} \cdot \frac{\text{uUsec}}{1000} = 16000$

$\frac{\text{CPU\_CYCLE\_FREQ}}{1000}$ = Number of samples needed to count 1 ms. Depends on the internal frequency of the CPU (16 MHz).

$\frac{\text{uUsec}}{1000}$ = number of ms to count. *uUsec* in μs.

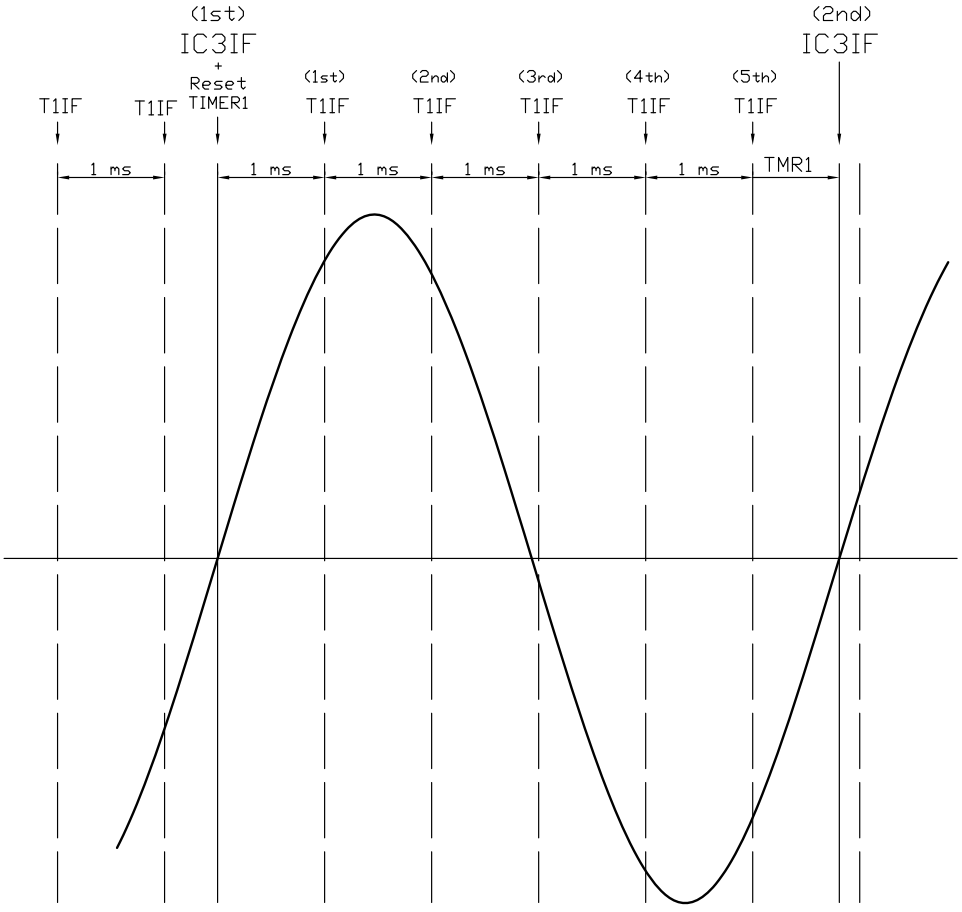


FIGURE 1.5 - Schematic.

$$\text{uFrequencyTicks} = \text{iTickCurrent} + \left(\text{uFreqTimer} \times \text{PR1SETTING}\right)$$

After this step we have the number of ticks elapsed between two positives edges of the input signal.

The number of Ticks stored in *uFrequencyTicks* has to be converted to time (in *frequency.c*). If *uFrequencyTicks* is divided by 16, the period (T) of the signal to measure is obtained in μs. The

frequency of the signal introduced is easily calculated as:

$$frequency = \frac{1}{T}$$

At this moment, the time needed to calculate the frequency is one period of the input signal (20ms) and the resolution obtained in the measure is 0.01Hz.

The brain of all this calculations is a Digital Signal Processor (DSP). In the Section 1.2.3 will be shown all the information about this DSP.

1.2.3 The Digital Signal Processor

The device EM4000AC is built with a DSP made by Microchip denominated dsPIC30F6011A.

What a DSP is

A *Digital Signal Processor* (DSP) is a type of specialised microprocessors with architectures designed specifically for the types of operations required in digital signal processing. It is a special-purpose CPU (Central Processing Unit) that provides ultra-fast instruction sequences, such as shift and add, and multiply and add, which are commonly used in math-intensive signal processing applications.

Features of dsPIC30F6011A

(13)

- **High-Performance Modified RISC CPU:**
  - + Modified Harvard architecture
  - + C compiler optimized instruction set architecture
  - + 84 base instructions with flexible addressing modes
  - + 24-bit wide instructions, 16-bit wide data path
  - + 16 x 16-bit working register array
  - + Up to 30 MIPs operation:
    - DC to 40 MHz external clock input
    - Internal FRC input with PLL active (4x, 8x, 16x)
    - 4 MHz-10 MHz oscillator input with PLL active (4x, 8x, 16x)
    - 10 MHz - 20 MHz oscillator input in HS/2 or HS/3 with PLL active (4x, 8x, 16x)
  - + Peripheral and External interrupt sources
  - + 8 user selectable priority levels for each interrupt
  - + 4 processor exceptions and software traps
  - + Primary and Alternate interrupt Vector Tables
- **DSP Engine Features:**
  - + Modulo and Bit-Reversed Addressing modes
  - + Two, 40-bit wide accumulators with optional saturation logic
  - + 17-bit x 17-bit single cycle hardware fractional/ integer multiplier
  - + Single cycle Multiply-Accumulate (MAC) operation
  - + 40-stage Barrel Shifter
  - + Dual data fetch
- **Peripheral Features:**
  - + High current sink/source I/O pins: 25 mA/25 mA
  - + Optionally pair up 16-bit timers into 32-bit timer modules
  - + 3-wire SPI modules (supports 4 Frame modes)
  - + I2C module supports Multi-Master/Slave mode and 7-bit/10-bit addressing
  - + Addressable UART modules with FIFO buffers and selectable pins
- **Analog Features:**
  - + 12-bit 200 Ksps Analog-to-Digital Converter (A/D)
  - + A/D Conversion available during Sleep and Idle

  + 1 Sample/Hold
  + Multiple Conversion Sequencing Options
- **Special Microcontroller Features:**
  + Enhanced Flash program memory:
    - 10,000 erase/write cycle (min.) for industrial temperature range, 100K (typical)
  + Data EEPROM memory:
    - 100,000 erase/write cycle (min.) for industrial temperature range, 1M (typical)
  + Self-reprogrammable under software control
  + Power-on Reset (POR), Power-up Timer (PWRT) and Oscillator Start-up Timer (OST)
  + Flexible Watchdog Timer (WDT) with on-chip low power RC oscillator for reliable operation
  + Fail-Safe clock monitor operation
  + Detects clock failure and switches to on-chip low power RC oscillator
  + Programmable code protection
  + In-Circuit Serial Programming (ICSP)
  + Programmable Brown-out Detection and Reset generation
  + Selectable Power Management modes
    - Sleep, Idle and Alternate Clock modes

- **CMOS Technology:**
  + Low power, high speed Flash technology
  + Wide operating voltage range (2.5V to 5.5V)
  + Industrial and Extended temperature ranges
  + Low power consumption

DSP functionalities
A DSP engine has been included to significantly enhance the core arithmetic capability and throughput. It features a high-speed 17-bit by 17-bit multiplier, a 40-bit ALU, two 40-bit saturating accumulators and a 40-bit bidirectional barrel shifter. Data in the accumulator or any working register can be shifted up to 16 bits right, or 16 bits left in a single cycle. The DSP instructions operate seamlessly with all other instructions and have been designed for optimal real-time performance. The MAC class of instructions can concurrently fetch two data operands from memory while multiplying two W registers. To enable this concurrent fetching of data operands, the data space has been split for these instructions and linear for all others. This has been achieved in a transparent and flexible manner, by dedicating certain working registers to each address space for the MAC class of instructions.

The dsPIC30F is a single-cycle instruction flow architecture; therefore, concurrent operation of the DSP engine with MCU (Microcontroller Unit) instruction flow is not possible. However, some MCU ALU (Arithmetic Logic Unit) and DSP engine resources may be used concurrently by the same instruction.

The DSP engine also has the capability to perform inherent accumulator-to-accumulator operations, which require no additional data. These instructions are ADD, SUB and NEG.

In the Figure B.2 (Appendix B) it can be seen a small schematic of DSP engine block diagram.

These are the functionalities that make special a DSP.

Modulo Addressing mode
Modulo Addressing is a method of providing an automated means to support circular data buffers using hardware. The objective is to remove the need for software to perform data address boundary checks when executing tightly looped code, as is typical in many DSP algorithms. Modulo Addressing can operate in either data or program space.

To fully appreciate the utility of modulo addressing, we should first understand the concept

of a Circular Buffer. A Circular Buffer is basically a region in memory which is used to store a fixed-size data array, such that after reaching the end of the array one needs to "wrap around" and continue accessing data from the beginning of the array. The data access is thus required to be always bounded within the limits of the buffer.

Three types of modulo buffers: (a) Incrementing, (b) Decrementing, and (c) Bi-directional as can be seen in the Figure 1.6.

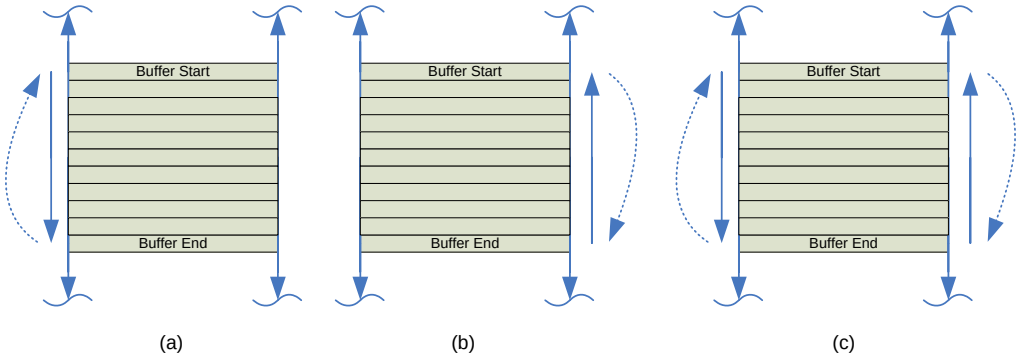


FIGURE 1.6 - Different types of modulo buffers.

In general, any particular circular buffer can only be configured to operate in one direction, as there are certain restrictions on the buffer start address (for incrementing buffers), or end address (for decrementing buffers) based upon the direction of the buffer. The only exception to the usage restrictions is for buffers which have a power-of-2 length. As these buffers satisfy the start and end address criteria, they may operate in a Bidirectional mode (i.e., address boundary checks will be performed on both the lower and upper address boundaries). (31)

Bit-Reversed Addressing mode

Bit-Reversed is a specialized addressing mode that is beneficial for many DSP applications, particularly those that require analysis of the frequency components of signals using a Fast Fourier Transform (FFT).

Bit-reversed addressing is a special feature provided in the dsPIC30F architecture to support efficient implementation of radix-2 FFT algorithms.

Given the address of a particular element in the array, the dsPIC30F hardware automatically computes the address of the next element in the bit-reversed sequence.

A Radix-2 FFT algorithm implicitly rearranges the data array being processed. Therefore, bit-reversed reordering of data is typically done either at the beginning or at the end of a Radix-2 FFT algorithm, so that in the end we obtain the data arranged in sequential order. Bit-reversed reordering can be performed by copying data words from a sequentially addressed array into a bit-reverse addressed array. The DSP Library from Microchip contains easy-to-use and optimized functions for reordering FFT data using bit-reversed addressing. (31)

40-stage Barrel Shifter

The barrel shifter is capable of performing up to 16-bit arithmetic or logic right shifts, or up to 16-bit left shifts in a single cycle. The source can be either of the two DSP accumulators, or the X bus (to support multi-bit shifts of register or memory data).

The barrel shifter is 40 bits wide, thereby obtaining a 40-bit result for DSP shift operations and a 16-bit result for MCU shift operations. (13)

Dual data fetch
The MAC class of instructions can concurrently fetch two data operands from memory while multiplying two W registers. (13)

Two 40-bit wide accumulators with optional saturation logic
The data accumulator consists of a 40-bit adder/subtracter with automatic sign extension logic. It can select one of two accumulators (A or B) as its preaccumulation source and postaccumulation destination. For the ADD and LAC instructions, the data to be accumulated or loaded can be optionally scaled via the barrel shifter, prior to accumulation.

The adder/subtracter generates overflow status bits SA/SB and OA/OB, which are latched and reflected in the STATUS register.

The adder has an additional saturation block which controls accumulator data saturation. (13)

The device supports three saturation and overflow modes:

- Bit 39 Overflow and Saturation
- Bit 31 Overflow and Saturation
- Bit 39 Catastrophic Overflow

17-bit x 17-bit single cycle hardware fractional/ integer multiplier
The 17 x 17-bit multiplier is capable of signed or unsigned operation and can multiplex its output using a scaler to support either 1.31 fractional (Q31) or 32-bit integer results. Unsigned operands are zero-extended into the 17th bit of the multiplier input value. Signed operands are sign-extended into the 17th bit of the multiplier input value. The output of the 17 x 17-bit multiplier/scaler is a 33-bit value which is sign-extended to 40 bits. Integer data is inherently represented as a signed two's complement value, where the MSB is defined as a sign bit. Generally speaking, the range of an N-bit two's complement integer is -2^{N-1} to $2^{N-1}-1$. For a 16-bit integer, the data range is -32768 (0x8000) to 32767 (0x7FFF) including '0'. For a 32-bit integer, the data range is -2,147,483,648 (0x8000 0000) to 2,147,483,647 (0x7FFF FFFF).

The same multiplier is used to support the MCU multiply instructions which include integer 16-bit signed, unsigned and mixed sign multiplies.

The MUL instruction may be directed to use byte or word sized operands. Byte operands will direct a 16-bit result, and word operands will direct a 32-bit result to the specified register(s) in the W array. (13)

Single cycle Multiply-Accumulate (MAC) operation
Most instructions need one cycle to execute it as can be seen in (13).

1.2.4 Initial problem
The current problem in this device is the time used to calculate the frequency.

1.2.5 Objectives
The objective of this part is study the way of speed up the calculations. On the one hand, the implementation of other alternative method of measuring the frequency to be able to increase the speed of calculations and sampling frequency and, on the other hand, try to use the DSP functionality to be able to calculate the voltage, current, phase and power.

2 Chapter 2 - Methods of measuring

2.1 Introduction

The frequency of a power system is an important operational parameter for the safety, stability, and efficiency of the power system. Reliable frequency measurement is prerequisite for effective power control, load shedding, load restoration, and generator protection. Therefore, the problem is that of fast and accurate estimation of the frequency of the power system using voltage waveforms which may be corrupted by noise and harmonics components.

In this chapter we are going to show the different available options to calculate the frequency.

2.2 Different options

Usually, frequency or period measurement of periodic waveforms is achieved by means of **counter-timer** instruments (14), (25) by counting the pulses within a specific time interval. The hardware of these instruments is not simple. Other methods are based on **digital signal processing** (DSP) techniques, (2), (19). They require a common data acquisition board and a software algorithm. Among DSP techniques, some allow the measurement of frequency in a narrow range, for instance the frequency of a power system (30), (19). Other techniques allow the measurement in a wide range (2), (7), but the range is limited at half the sampling frequency (Nyquist limit). Counting the **zero crossings** is a simple and well known technique (2), (7).

Several methods have been proposed for determining the frequency of a sinusoid in noise. Among these are the periodogram (26), backward prediction with singular value decomposition (18), or using the mean value of the zero crossings (16).

Use of the zero crossing detection and calculation of the number of cycles that occur in a predetermined time interval (24) is a simple and well-known methodology. The **Discrete Fourier Transformation**, **least error squares**, and **Kalman filter** are known signal processing techniques, used for the frequency measurement (8), (28), (10), (15), (5). As shown in (17), the bilinear form approach seems to be a very efficient method for both small frequency deviation and off-nominal frequency estimation. An adaptive algorithm for measuring power system frequency over a wide range is suggested in (23). A technique for frequency estimation that provides accurate estimation in about 25 ms and requires modest computations is presented in paper (30). This algorithm shows advantages in accuracy and convergence rate in computation to the previous methods. In these papers, orthogonal FIR digital filters are used to extract the real and imaginary parts of the fundamental frequency component of the signal. These filters are designed so that they minimize the noise effect and are not affected by the presence of harmonics. An iterative procedure for frequency estimation and filter adjusting is presented in these papers.

Frequency of a power system remains constant if sum of all the loads plus losses equals total generation in the system. However, the frequency starts to decrease if total generation is less than the sum of loads and losses. On the other hand, the system frequency increases if total generation exceeds the sum of loads and losses. It is essential that the frequency of a power system be maintained very close to its nominal frequency, i.e., 60 Hz in North America and 50 Hz in Europe. The normal frequency levels should, therefore, be maintained as quickly as possible. A decrease in system frequency also reduces the reactive power supplied by capacitors, transmission lines, etc., and, therefore, upsets the reactive power balance and affects the voltage levels.

The above discussion indicates that the system frequency should be maintained at its nominal frequency. If it deviates from nominal value, corrective actions should be taken. Underfrequency and overfrequency relays are provided in a power system. Underfrequency relays are

used to automatically shed blocks of loads for restoring the frequency to its nominal value. When generation in a system exceeds the load and losses, the frequency increases above the nominal frequency.

Overfrequency relays are used to detect this condition and shed some generation. Also, when a generating unit is suddenly separated from the system, its speed increases and the unit can be damaged. Overfrequency relays are used to detect this situation, and take corrective measures. Frequency relays having a measurement range from 40 Hz to 70 Hz are adequate for over- and under-frequency relaying. Frequency relays presently used are of electromagnetic, solid-state and microprocessor types. Accuracy of electromagnetic relays range from 0.1 to 0.2 Hz of the set frequency. Solid-state relays measure time duration between zero crossings and are adversely affected by presence of distortion and noise which shift the zero-crossings and create multiple zero-crossings. Microprocessor relays use algorithms that process the sampled and digitized values of the system voltage to estimate the frequency (24), (27), (9), (35), (33). These techniques are adversely affected by the presence of harmonics in the signals. Also, they may take from 50 ms to few seconds to provide accurate estimates of the frequency. It is necessary to detect underfrequency and overfrequency condition very quickly so that necessary corrective actions can be taken.

Quick estimation of frequency would provide extra time at the hands of operators to take corrective actions. On the other hand, accurate estimates aid in determining the correct loads and generation that has to be shed. Therefore, the problem is that of fast and accurate determination of the frequency of the power system using voltage waveforms which may be corrupted by noise and harmonic components. Recently, adaptive algorithms which may provide faster estimates of the system frequency have been proposed (36). However the algorithm proposed in (36) requires considerable computational resources and, therefore, can not be economically implemented with the presently available technology.

2.2.1 Frequency to digital conversion

Another method could be "**Frequency to digital conversion**".

There are two main methods of converting a variable frequency sinusoidal signal into a parallel digital output signal. The sine wave must first be converted into a square wave signal with sharp edges using a Schmitt trigger circuit.

In the first method the frequency fs of the signal is measured by counting the number of pulses during a fixed time interval T. In the second method the period Ts of the signal is measured by counting the number of clock pulses within Ts.

The principle of the first method is shown in Figure 2.1. The number Ns of positive-going edges during T is counted, giving:

$$f_s = \frac{N_s}{T}$$

Figure 2.2 shows one possible system for implementing this method. Both counters are initially reset to zero. The number Nc is then loaded into the clock counter; this sets the counting interval T to be:

$$T = \frac{N_C}{f_c}$$

where fc is the clock frequency. Clock pulses are input to the clock counter, which then counts down to zero. Signal pulses are input to the signal counter; this counts up until the clock counter reaches zero, when the count is stopped. The signal count is then:

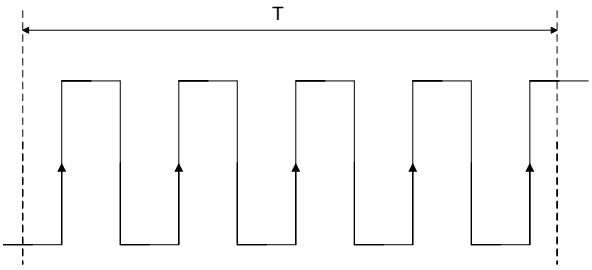


FIGURE 2.1 - Frequency measurement - Principle

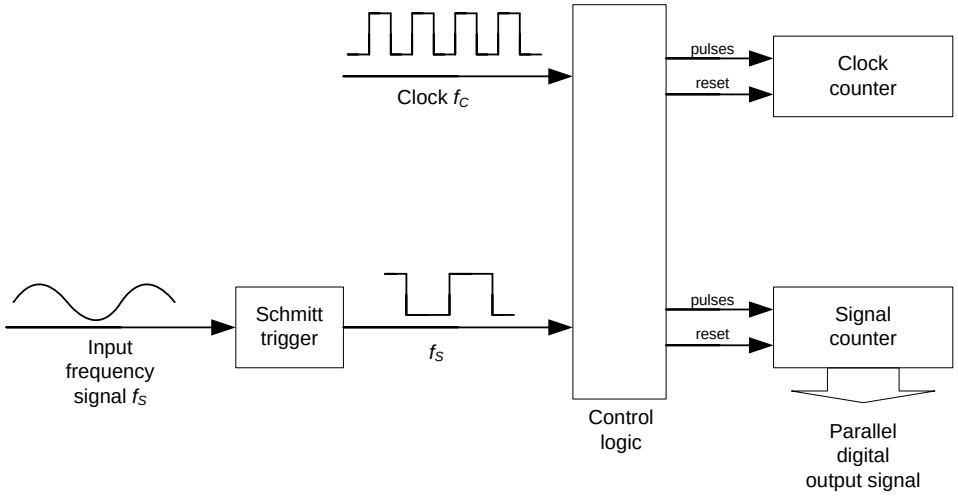


FIGURE 2.2 - Frequency measurement - System

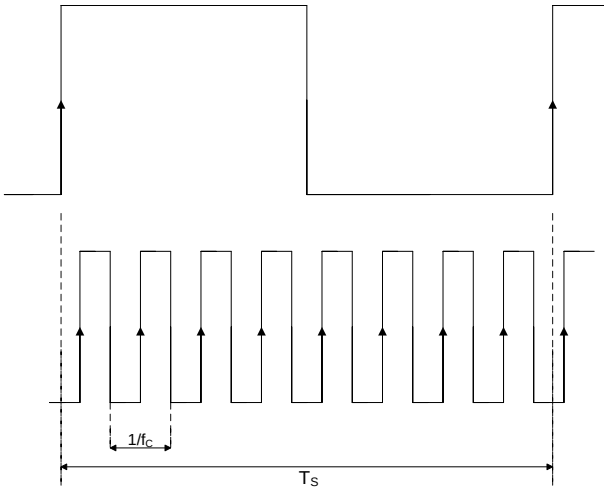


FIGURE 2.3 - Period measurement - Principle

$$N_S = \frac{N_C}{f_C} \cdot f_S$$

The parallel digital output signal corresponds to signal count Ns, which is proportional to input

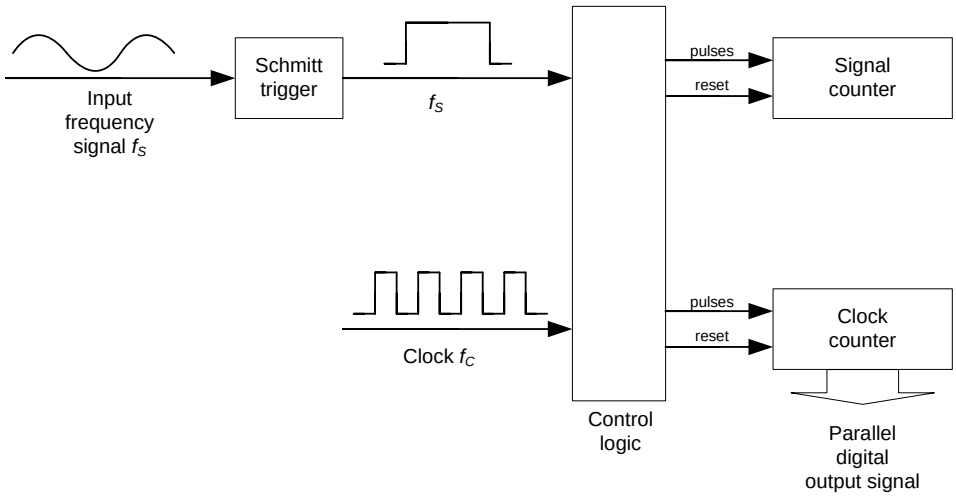


FIGURE 2.4 - Period measurement - System

signal frequency fs.

The resolution of this method is limited to ±1 signal count: this can mean poor percentage resolution at low frequencies. If we consider the example when $fs = 20Hz$ and $fc = 10KHz$, then to get $T = 1s$, $Nc = 10000$, giving $Ns = 20 \pm 1$, i.e. a percentage resolution of ±5%. Percentage resolution can be increased by increasing the counting interval, but this is only possible if T is small compared with the time scale of dynamic variations in the measurement signal.

This problem can be solved using the second method. Here the period Ts of the signal is measured by counting the number of clock pulses within Ts. The principle is shown in Figure 2.3. The number Nc of positive-going edges is counted, giving:

$$T_s = \frac{N_C}{f_C} \tag{2.1}$$

where 1/fc is the clock period. Figure 2.4 shows one possible system for implementing this method. Both counters are initially set to zero; clock pulses are input to the clock counter and signal pulses to the signal counter. The clock counter counts up until a count of 1 is registered in the signal counter; the count then stops. The clock count Nc is then given by Equation 2.1 and proportional to Ts. The parallel digital output signal is proportional to this clock count.

The resolution of this method is ±1 clock count. Using the above example with fs = 20 Hz, Ts = 0.05 and $fc = 10KHz$, $Nc = 500 \pm 1$, i.e. a percentage resolution of ±0.002%.

However, some frequency signals are subject to random fluctuations which cause the signal period to vary, even thoug the input true value of the measured variable is constant. For example, the period of a nominal 20hz signal from a vortex flow meter may vary randomly between 0.04 s and 0.06 s about a mean value of 0.05 s. It is therefore essential to measure the mean frequency or mean period of the signal. The first method gives the mean frequency of 20 cycles measured over 1 second; the second method gives the period of one cycle only. This problem is solved using the second method but now the clock counter counts up until a count of 20 is registered in the signal counter. The mean clock count Nc for one signal cycle is then the total clock count divides by 20; the mean signal period Ts is then given by Equation 2.1. (3)

2.2.2 Counter-timer methods

Frequency meter for low frequencies with known nominal values.

The conventional frequency measurement techniques, which are based on counting the number of cycles during a fixed time interval, become unsuitable at such low frequencies because of the long time interval needed for counting. For the instantaneous measurement of such frequencies, different techniques are devised, using both analog and digital circuits. Most of these methods are based on measuring the time period of the frequency to be measured, then the corresponding frequency is found by different methods such as computation using microprocessors, or generating a voltage inversely proportional to the time period, etc., (12), (21), (20), (37).

Fortunately, most of the low frequencies to be measured have known nominal values and their deviations from these nominal values are very small.

Principle of operation
For a periodic waveform, the frequency f and the period T are related by the following inverse operation:

$$f = \frac{1}{T} \tag{2.2}$$

For the instantaneous measurement of very low frequencies, one has to measure the time period T, and then find the frequency by the inverse operation given in (1). The implementation of the inverse operation given in Equation 2.2 using analog or digital techniques is not a simple task, and therefore different approximation methods are used in the literature to overcome this difficulty.

For the class of periodic waveforms having frequencies with known nominal values, and whose actual frequencies are not deviating far from these nominal values, the period T can be written as

$$T = T_O \pm \Delta T \tag{2.3}$$

where T_O is the nominal period and ΔT is the absolute deviation of the actual period T from its nominal value T_O.

By substituting Equation 2.3 in Equation 2.2 , it becomes

$$f = 1/(T_O \pm \Delta T) \tag{2.4}$$

Equation 2.4 can be put in the following form

$$f = \frac{f_O}{T_O \pm \frac{\Delta T}{T_O}} \tag{2.5}$$

where $f_O = 1/T_O$, is the nominal frequency.

If the deviation of the actual frequency from its nominal value is small, i.e., $\Delta f << f_O$ or $\Delta T << T_O$, then Equation 2.5 can be approximated as

$$f_m \cong f_O \cdot \left(1 \mp \frac{\Delta T}{T_O}\right) \tag{2.6}$$

where f_m is the measured frequency. Eq. 2.6 can be rewritten in terms of f_O and ΔT as

$$f_m \cong f_O \mp f_O^2 \Delta T \tag{2.7}$$

Equation 2.7 is used in (11) to measure the deviation of a frequency from its nominal value. Equation 2.7 is modified to get a simple programmable frequency meter as follows.

Since $f_O \cdot T_O = 1$, then Eq. 2.7 becomes

$$f_m \cong f_O^2 T_O \mp f_O^2 \Delta T \tag{2.8}$$

The measured frequency f_m can be written in terms of its period T by adding and subtracting the term $f_O^2 T_O$ to the right-hand side of Eq. 2.8. By noticing that $T = T_O \pm \Delta T$ it becomes

$$f_m = 2 f_O - f_O^2 T \tag{2.9}$$

It is obvious from Eq. 2.9 that the frequency to be measured is linearly related to its period and the accuracy of this relation depends on how large the deviation of T from its nominal value T_O. Equation 2.9 can be easily implemented using an up/down counter operating in the down mode only as follows: first, the up/down counter is loaded by a number equaling twice the nominal frequency $2 f_O$, secondly, the counter is enabled to count down from this loaded value at a clock rate of f_O^2 Hz for T seconds. At the end of the counting period T, the contents of the counter are latched and displayed. This process is repeated in every cycle for the instantaneous frequency measurement.

The accuracy of this proposed method depends on the amount of deviation of the actual frequency from its nominal value. The normalized error is given by

$$Error = \left(f - f_m\right) / f \tag{2.10}$$

where f_m is the measured frequency given by Eq. 2.9, and f is the exact frequency given by Eq. 2.4. By substituting Eq. 2.4 and Eq. 2.9 in Eq. 2.10, we get

$$Error = -\left(f_O / f - 1\right)^2 \tag{2.11}$$

As long as the deviation of the frequency f from its nominal value fo is small, the error is acceptable for many applications. As an example the error will not exceed 1% for f lying in the range $0.909 f_O < f < 1.1 f_O$, and 5% for $0.817 f_O < f < 1.224 f_O$. In power systems, the absolute deviation of the line frequency from its nominal value is generally less than 2 Hz . Therefore, the percentage error will not exceed 0.174% for 50 Hz and 0.12% for 60 Hz.

In Figure 2.5 and Figure 2.6 can be seen a block Diagram and a timing diagram of the proposed frequency meter in (14).

2.2.3 Digital signal processing

There are different techniques for measuring the frequency using the DSP functionality. We are going to explain the method of using an FFT to be able to calculate the frequency.

In complex notation, the time and frequency domains each contain one signal made up of N complex points. Each of these complex points is composed of two numbers, the real part and the imaginary part. Each complex variable holds two numbers. When two complex variables are multiplied, the four individual components must be combined to form the two components of the product.

The FFT operates by decomposing an N point time domain signal into N time domain signals each composed of a single point. The second step is to calculate the N frequency spectra corresponding to these N time domain signals. Lastly, the N spectra are synthesized into a single frequency spectrum.

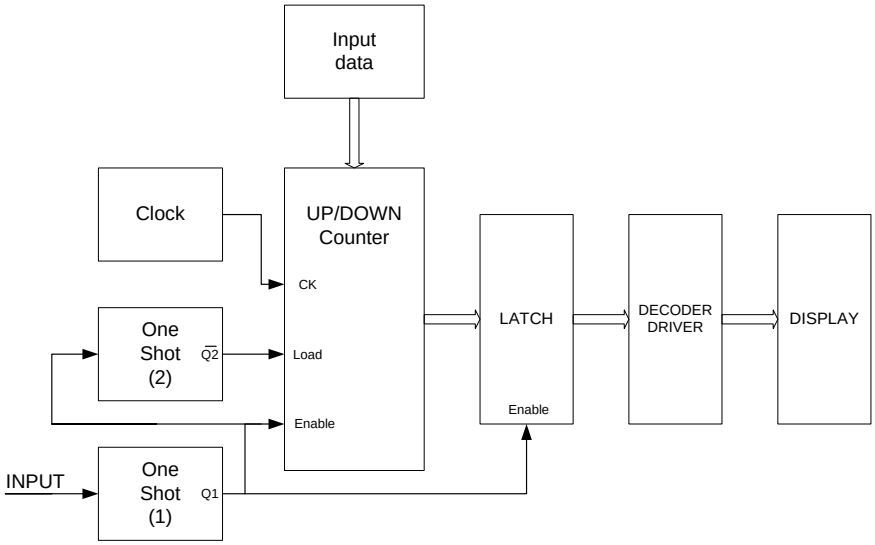


FIGURE 2.5 - Block Diagram of the proposed programmable frequency meter.

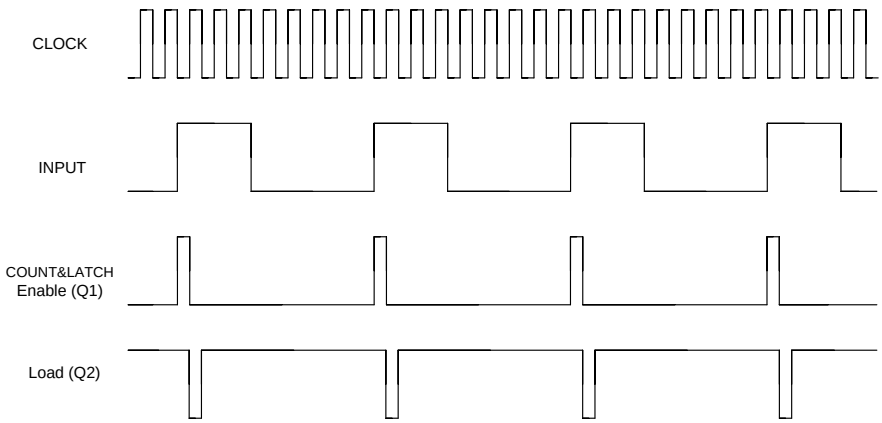


FIGURE 2.6 - Timing diagram of the proposed meter.

The whole explanation of how the FFT works can be seen in Appendix A.

2.2.4 Counting zero crossings

In mathematical terms, a "zero-crossing" is a point where the sign of a function changes (e.g. from positive to negative), represented by a crossing of the axis (zero value) in the graph of the function. In alternating current, the zero-crossing is the instantaneous point at which there is no voltage present. In a sine wave or other simple waveform, this normally occurs twice during each cycle but it can be modified depending on the noise of the input signal.

This algorithm is very sensitive to the noise superimposed to the input signals, since the zero-crossing points can be displaced by the noise itself.

Generally, it is difficult to find the probability distribution function (pdf) of the zero crossings (29), or to estimate the signal spectrum based on it (4). However, for the particular case of a single sinusoid, the interval between zero crossings gives a good estimation of its frequency

with reduced computational effort.

2.3 Aspects to keep in mind

Accuracy is intimately related to the resolution of an Analog-to-Digital Converter (ADC) you want to use and its internal DC-reference. Many NMIs possess the commercial digital voltmeter HP3458 for high accuracy sampling though, which is the state of the art in digitizers still. The primary level in ppm requires high spectral purity to preclude aliasing (and as a consequence systematic deviations on sampling).

For the highest accuracy and precision, synchronous sampling with a common quartz time-base clock is preferred. Asynchronous measurements demand a higher sampling rate and/or the use of windowing functions. Literature on sampling is extense, and intimately related to it are Digital Signal Processors (DSP). For example, using a 32-bit floating point SHARC Processor, numerical resolution poses no limitations on frequency determinations. These are restrained to FFT-Frequency resolution however, which depends on the number of samples and sampling frequency (defined by the user!).

Recently it has been developed a software that can virtually handle all modes of sampling, i.e., synchronous (tight synchronization between source and sampler), asynchronous sampling and an additional mode: synchronization by software, which uses dedicated hardware employing variable/dynamic sampling clocks settable up to nHz (or lower) resolution.

The problem with the mains are the inherent variations of the mains frequency (i.e., it is not stationary). What many people do is to synchronize measurements employing a Phase-Locked-Loop (PLL). We have such hardware which allows ppm accuracy under stationary conditions, but it suffers from frequency variations as well, besides jitter of the order of microseconds; Nevertheless it can easily be triggered synchronously (allowing ppm accuracy). However, to tackle nearly all problems in sampling (ratio measurement, frequency determination, rms, phase, ac power, energy, impedance measurements, spectral components, noise, ADC nonlinearities and so on...) the software and hardware of 3) above seem to be the best solution we could devise at the moment.

3 Chapter 3 - Using FPGAs and CPLDs

3.1 Introduction

In this section we are going to discuss the possibility of using another different microprocessor such as FPGAs (Field-Programmable Gate Array) or CPLDs (Complex Programmable Logic Device) in comparison with DSPs.

3.2 Discussion

DSP processors are widely used for implementing many DSP applications. Although DSP processors are programmable through software, the DSP processor hardware architecture is not flexible.

There are various options to implement DSP applications:

- DSP processors
- Application-specific integrated circuits (ASICs)
- Application-specific standard products (ASSPs)
- PLDs

DSP processors have a general-purpose architecture that makes them flexible for a variety of applications. However, their flexibility ultimately limits their system performance.

DSP processors are most suited for back-end signal processing at low data rates. Many DSP processors have multipliers with special instructions to speed up the math calculations, however, they lack real-time performance. DSP processors are flexible and can be used for filtering or modulating applications by changing the processor's software code.

ASSPs and ASICs, which are designed to implement a specific function, have better performance than DSP processors for a low cost. These qualities make them attractive for designers. Because ASSPs are semi-custom integrated circuits that perform specific functions, such as finite impulse response (FIR) and infinite impulse response (IIR) filters, their performance is better than other hardware solutions on a similar process technology. However, ASSPs are inflexible and must be redesigned if the DSP application changes. ASICs provide a customizable, low-cost solution. However, they have long lead times-typical design cycles are 1 to 1.5 years-and require a minimum purchase quantity. Small design changes incur additional nonrecurring engineering costs and result in a longer design cycle.

PLDs offer compelling advantages over DSP processors, ASSPs and ASICs. Designers can configure PLD logic to process complex routines in parallel or in serial like DSP processors (Figure 3.1). In parallel, they offer greater performance than DSP processors by executing the equivalent of hundreds of instructions at once. Unlike ASSPs and ASICs, PLDs provide the flexibility to make design changes without sacrificing time-to-market.

FPGAs provide a reconfigurable solution for implementing DSP applications as well as higher DSP throughput and raw data processing power than DSP processors. Since FPGAs can be reconfigured in hardware, FPGAs offer complete hardware customization while implementing various DSP applications. FPGAs now provide a cost-effective alternative for DSP implementation that can be adopted easily for a broad range of applications.

With introduction of dedicated multipliers inside the FPGAs, many of DSP products are being implemented with FPGAs. This big market success pushed FPGA vendors to made dedicated DSP processing blocks inside the FPGAs in order to allow fast multiply and accumulate structures as well as other DSP specific arithmetic functions. For example the latest Xilinx Virtex 4 family offers up to 256GMACS. Combined with FPGA speed and size increase as well as with famous FPGA reprogram ability, this has made a huge revolution in digital processing with FPGAs.

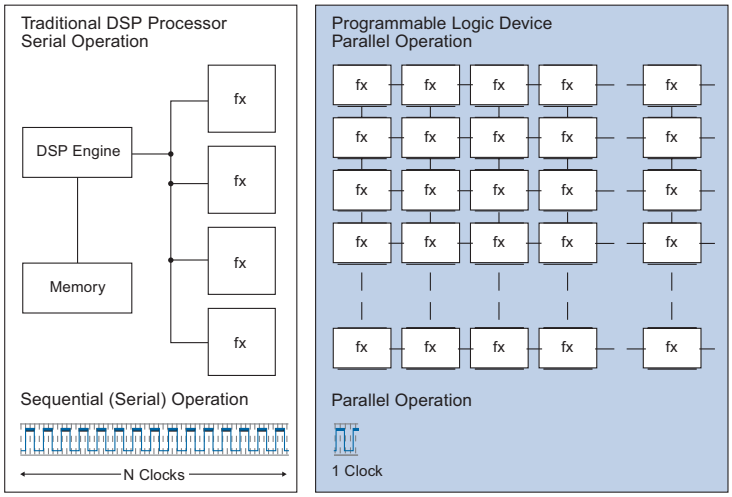


FIGURE 3.1 - Architectural Differences between PLDs and DSP Processors.

3.3 FPGAs

DSP-Enhanced FPGAs

Compared to DSP processors that only offer a limited number of multipliers, Altera FPGAs offer much more multiplier bandwidth. Since one determining factor of the overall DSP bandwidth is the multiplier bandwidth, the overall DSP bandwidth of FPGAs can be much higher than the DSP processors. For example, Stratix device DSP blocks can deliver 70 GMACS of DSP throughput while leading DSP processors available today can deliver only up to 4.8 GMACS.

Various DSP applications use external memory devices to manage large amounts of data processing. The embedded memory in FPGAs meets these requirements and also eliminates the need for external memory devices in certain cases.

Hardware Acceleration in FPGAs

FPGA devices provide a flexible platform to accelerate performance-critical functions in hardware because of the configurability of the device's logic resources. Unlike DSP processors that have predefined hardware accelerator blocks, FPGAs can implement hardware accelerators for each application, allowing the designer to achieve the best performance through hardware acceleration. The designer can implement hardware accelerator blocks by designing such blocks using parametrizable IP functions or from scratch using HDL.

Cost Benefits of FPGAs

Historically, FPGAs have been viewed as more expensive than DSP processors. However, FPGAs provide much higher throughput than DSP processors as well as hardware flexibility, such that a single FPGA can accommodate an order of magnitude more channels than DSP processors without any degradation in performance. Hence in multi-channel applications, the cost per channel of an FPGA based solution can be much less expensive than an equivalent DSP solution.

Combination of DSP algorithms and FPGAs is quite a big challenge. The DSP designers primarily use the MATLAB or C/C++ for system definition. On the other side, the FPGA designers are mainly using the hardware design languages like Verilog or VHDL for synthesis. Obviously, there is a big gap between those two technologies, Because of this; it is quite common approach to use the block diagrams for communication between these two worlds.

Today, it is possible to make downloadable FPGA bitstream with a push button after functional

verification of the HDL code. Then, after few hours of compilation, the realtime verification of DSP algorithm in FPGA can begin. Overall, it is needed weeks to months of time for development ofDSP algorithm in FPGA before real-time verification. Time to market is obviously shortened with using DSP processors. Apart from this advantage, DSP processors have two disadvantages. Firstly, due to its sequential processing for the most complex DSP algorithms they simply can not provide needed data through put. Secondly, the price of one processor unit is often too big for high volume markets. This is especially noticeable for new DSP processor families that run on high frequencies in order to provide bigger data through put.

Most of DSP algorithm developers would happily embrace a methodology that would allow them to have direct path from algorithm abstraction in MATLAB to optimized, parallel hardware without deep understanding or detailed analysis of targeted architecture. Nowadays this is possible through design abstraction with IP integration. (1) (22).

4 Chapter 4 - The chosen solution

4.1 Fast Fourier Transform

In this chapter will be shown the calculations carried out to study the feasibility of the implementation of the chosen solution. Among the available options, it has been chosen the FFT one because is more robust to face up to noise or small glitches. The Fast Fourier Transform (FFT) is going to be used to calculate the frequency in a faster way using the DSP functionality in the current dsPIC.

4.2 Calculations

The device EM4000AC is capable of measure between 45 Hz and 65 Hz with a resolution of 0.01 Hz. Depending on this requirements the number of points in the FFT will be derived.

Let's start with the calculations.

The input signal will have a frequency between 45 Hz and 65 Hz. If we use the Nyquist-Shannon sampling theorem:

> If a continuous, bandwidth limited signal contains no frequency components higher than F, then the original signal can be recovered without distortion (aliasing) if it is sampled at a rate that is greater than 2F samples per second.

The minimum sample frequency needed is 130 Hz (65 Hz x 2).

Generally, instead of sampling at twice the frequency, we will sample at five to ten times the highest frequency we are trying to capture (oversampling).

Oversampling is the process of sampling a signal with a sampling frequency significantly higher than twice the bandwidth or highest frequency of the signal being sampled. An oversampled signal is said to be oversampled by a factor of β, defined as:

$$f_s = 2\beta B$$

where:

f_s is the sampling frequency

B is the bandwidth or highest frequency of the signal

There are two main reasons for performing oversampling:

- It aids in anti-aliasing because realizable analog anti-aliasing filters are very difficult to implement with the sharp cutoff necessary to maximize use of the available bandwidth without exceeding the Nyquist limit.
- In practice, oversampling is implemented in order to achieve cheaper higher-resolution A/D and D/A conversion.

Sample Frequency chosen = 1000 Hz

Where the sampling period is:

$$T = \frac{1}{1000Hz} = 0.001\text{s}$$

The duration of the data window can be determined from the desired frequency resolution as:

$$T_o = \frac{1}{\text{Resolution}} = \frac{1}{0.01Hz} = 100\text{s}$$

from which it follows that

$$\mathrm{N} = \frac{T_o}{T} = \frac{100s}{0.001s} \geq 10^5 \text{points}$$

(Number of points of FFT needed)

Assuming that we are using a radix-2 FFT routine, we would choose N to be 131072 = 2^{17} which is the smallest power of 2 satisfying the constraint on N.

4.3 Problems

The main limitations in the system are a direct result of hardware limitations in the processor chosen for the system. The primary constraint is the amount of RAM; this limits the number of samples available for processing at any particular time. The amount of RAM limits the number of samples that can be operated on simultaneously; the PIC30, with 8192 bytes of RAM, can hold 4096 16-bit samples in memory.

The maximum FFT possible with the dsPIC30F6011A would be 4096 points assuming that no RAM is used for other operations. However, approximately 25% of the RAM is used solely for buffering data for communications, so the maximum FFT is 2048 points.

Figure 4.1 shows the resolution obtained as a function of the number of points used for an FFT calculation. As can be seen in Figure 4.1, to obtain a resolution of 0.01 Hz in the measured frequency 50,000 points will be necessary (with a sample frequency of 500 Hz) or 100,000 with a sampling frequency of 1000 Hz.

The first solution seems to be not possible to be implemented. Besides, the time needed to capture all the data would be excessive.

If we suppose the maximum number of points possible (2048 points) and a sample frequency of 1000 Hz, the obtained resolution would be:

$$2048\text{points} = \frac{\frac{1}{resolution}}{\frac{1}{fs}} = \frac{fs}{resolution} = \frac{1000}{resolution}$$

$$resolution = \frac{1000}{2048} = 0.488Hz$$

and it would be necessary a data window of

$$T = \frac{1}{0.488Hz} = 2.048s$$

For the implementation of this algorithm it is necessary to have all the data stored in memory at the same time so the time needed to calculate the frequency would be 2.048 s plus the time needed to make the 11 stages-FFT.

This time is huge in comparison with the current time used in the current software, 20ms.

4.4 Modification of the possible solution

An available modification is called “zoom FFT”. Basically this method consist in taking the data and tightly filter it to a passband of 45-65Hz. Then, “mix” it down; this can be done with judicious use of sampling frequency and simple undersampling. The idea is to convert 45 Hz to DC. Now the new signal runs from 0 to 20Hz. The algorithm of the FFT is applied on the new signal. Since 100 seconds of data for 0.01Hz resolution is still required, the new FFT will need 4000 samples for a 0.01Hz resolution with a sample frequency of 40 Hz. Although this is a more reasonable number of points, the sampling frequency is still too low to ensure goods results.

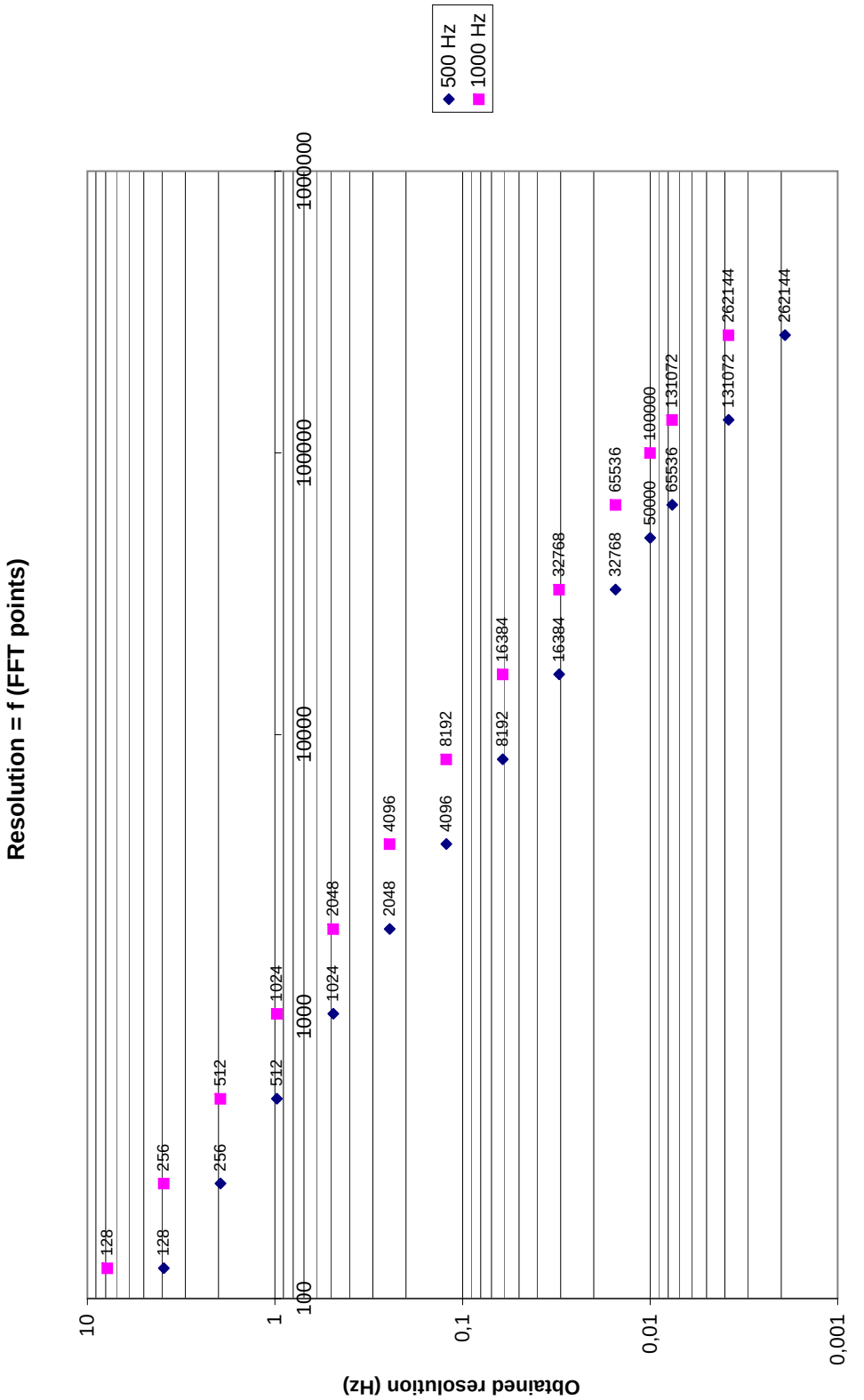


FIGURE 4.1 - Resolution obtained measuring the frequency in function of the number of points used in a FFT.

4.5 Conclusion

It can be concluded that it is not possible to implement an FFT in this device meeting the resolution requirements.

In the next chapter will be analyzed the feasibility of improving the device EM4000AC changing some part of the code.

5 Chapter 5 - Calculation of other parameters

5.1 Introduction

In this chapter a new attempt of improving the device EM4000AC will be shown. After having the frequency we need to calculate several parameters such as apparent power, current, voltage We will try to show if it is worthwhile to change some part of this code to speed up all the calculations.

We will begin with some parameters such as apparent power, line voltage...

First of all, we have to calculate the real frequency of the calculations, MIPS (Million Instructions Per Second).

With an external clock of 12MHz and the clock configuration in mode HS/3, PLL16.

$$F_{osc} = \frac{F_{ext}}{3} = \frac{12MHz}{3}$$

$$PLLx = 16$$

$$F_{real} = \frac{F_{osc} \cdot PLLx}{4} = \frac{12MHz}{3} \cdot \frac{16}{4} = 16MIPS$$

With $16MIPS$ we can calculate one cycle every 62.5 ns.

5.2 Calculation of the parameters

5.2.1 Calculation of apparent power

Original code in C.

Here we have the original code for the calculation of the apparent power.

```
void calculateApparentPower(uint16 uPhase)
{
uValuesUint32Ram[eRamApparentPower1+uPhase] =
(uint32)((float)(((float)uValuesUint32Ram[eRamPhaseVoltage1+uPhase]/
(float)VOLTAGE_RESOLUTION)*((float)((float)uValuesUint32Ram[eRamCurrent1+
uPhase]/(float)CURRENT_RESOLUTION))*(float)POWER_RESOLUTION);
}
```

Original code in assembler.

```
_calculateApparentPower:
.def .bf
.val .
.scl 101
.line 247
.endef
.def _uPhase
.val 0
.scl 9
.type 016
.endef
.def .bb
.val .
.scl 100
.line 247
.endef
.set ___PA___,1
lnk #24
mov w0,[w14]
.def ___FP
.val 14
.scl 4
.type 4
.endef
.ln 248
mov [w14],w0
add w0,#17,w0
mov w0,[w14+22]
mov [w14],w1
add w1,#4,w0
sl w0,#2,w1
mov #_uValuesUint32Ram,w0
add w1,w0,w0
mov.d [w0],w2
mov w2,[w14+18]
mov w3,[w14+20]
```

```
mov [w14+18],w4
mov [w14+20],w5
sub w4,#0,[w15]
subb w5,#0,[w15]
.set ___BP___,0
bra lt,.L107
mov [w14+18],w0
mov [w14+20],w1
rcall ___floatsisf
mov w0,[w14+14]
mov w1,[w14+16]
bra .L106
.L107:
mov [w14+18],w0
and w0,#1,w2
mul.uu w0,#0,w0
mov #0,w3
ior w0,w2,w0
ior w1,w3,w1
mov [w14+18],w4
mov [w14+20],w5

lsr w5,w3
rrc w4,w2
ior w0,w2,w0
ior w1,w3,w1
rcall ___floatsisf
mov w0,[w14+14]
mov w1,[w14+16]
mov [w14+14],w2
mov [w14+16],w3
mov [w14+14],w0
mov [w14+16],w1
rcall ___addsf3
mov w0,[w14+14]
mov w1,[w14+16]
.L106:
mov #9216,w2
mov #18804,w3
mov [w14+14],w0
mov [w14+16],w1
rcall ___divsf3
mov w0,[w14+10]
mov w1,[w14+12]
mov [w14],w0
sl w0,#2,w1
mov #_uValuesUint32Ram,w0
add w1,w0,w0
mov.d [w0],w2
mov w2,[w14+6]
mov w3,[w14+8]
mov [w14+6],w4
mov [w14+8],w5
sub w4,#0,[w15]
subb w5,#0,[w15]
.set ___BP___,0
bra lt,.L109
mov [w14+6],w0
mov [w14+8],w1
rcall ___floatsisf
mov w0,[w14+2]
mov w1,[w14+4]
bra .L108
.L109:
mov [w14+6],w0
and w0,#1,w2
mul.uu w0,#0,w0
mov #0,w3
ior w0,w2,w0
ior w1,w3,w1
mov [w14+6],w4
mov [w14+8],w5

lsr w5,w3
rrc w4,w2
ior w0,w2,w0
ior w1,w3,w1
rcall ___floatsisf
mov w0,[w14+2]
mov w1,[w14+4]
mov [w14+2],w2
mov [w14+4],w3
mov [w14+2],w0
mov [w14+4],w1
rcall ___addsf3
mov w0,[w14+2]
mov w1,[w14+4]
.L108:
mov #9216,w2
mov #18804,w3
mov [w14+2],w0
mov [w14+4],w1
rcall ___divsf3
mov.d w0,w2
mov [w14+10],w0
mov [w14+12],w1
rcall ___mulsf3
mov #0,w2
mov #17530,w3
rcall ___mulsf3
rcall ___fixunssfsi
mov.d w0,w2
mov [w14+22],w5
sl w5,#2,w1
mov #_uValuesUint32Ram,w0
add w1,w0,w0
mov.d w2,[w0]
.ln 249
ulnk
return
.set ___PA___,0
.def .eb
.val .
.scl 100
.line 249
.endef
.def .ef
.val .
.scl 101
.line 249
.endef
.def _calculateApparentPower
.val .
.scl -1
.endef
.align 2
```

The code shown in Section 5.2.1 has 186 cycles. The time needed to calculate this algorithm is:

$$Time = 186 cycles \cdot 62.5 ns/cycle = 11.63 \mu s$$

If we compare 11.63 μs with 20 ms used to calculate the frequency, it can be concluded that it is not worthwhile improve this part of the code.

5.2.2 Calculation of Line Voltage

In this case (one of the biggest calculations), the original code in C is bigger than the previous one, and the original code in assembler has aproximately 1000 cycles what means 61.88 μs. See Table 5.1.

We still have to spend at least 20ms in the calculation of the frequency.

5.2.3 Calculation of the rest of parameters

Following the same steps as in Section 5.2.1 we can determine the time needed to calculate all the parameters. These times can be seen in Table 5.1.

Parameter	Number of Instructions	Time (μs)
Current	239	14.94
CurrentAvg	138	8.63
PhaseVoltage	238	14.88
PhaseVoltageAvg	152	9.50
LineVoltage	990	61.88
LineVoltageAvg	243	15.19
ApparentPower	186	11.63
ApparentPowerSum	246	15.38
Temperature	122	7.63
ActivePowerRaw	152	9.50
CosPhiRaw	264	16.50
LeadLag	213	13.31
Phi	668	41.75
CosPhi	79	4.94
CosPhiAvg	111	6.94
PhiAvg	106	6.63
SinPhi	80	5.00
SinPhiAvg	111	6.94
ActivePower	124	7.75
ActivePowerSum	112	7.00
ReactivePower	127	7.94
ReactivePowerSum	117	7.31
WhImport	300	18.75
WhImportSum	75	4.69
VArImport	303	18.94
VArImportSum	75	4.69
WhExport	300	18.75
WhExportSum	75	4.69
VArExport	303	18.94
VArExportSum	70	4.38
ValidateMeasurement	223	13.94
DetermineLinearisationFactor	265	16.56
DeterminePhiCorrectionFactor	160	10.00
Total Time	$\Rightarrow$	**0.44 ms**

TABLE 5.1 - Time used to calculte different parameters.

5.3 Conclusion.

As can be seen in Section 5.2.1, Section 5.2.2 and Section 5.2.3, the maximum time of calculation would be 61.88 μs, which is nothing in comparison with 20 ms used to calculate the

frequency.

As can be seen in Table 5.1 the whole code of calculations takes approximately 0.435 ms. Even if we are able to increase the performance of all the parameters, the calculation of the frequency limits it.

It is not useful to improve this part of the software.

6 Chapter 6 - Conclusions and recommendations

6.1 Conclusions

The functionality of a commercial power measurement device, EM4000AC, has been analysed. Focus was put on an accurate but faster determination of the frequency (50 Hz) using this device. Under the requirement of no modification of physical characteristics of EM4000AC, the implementation of new software using the DSP functionality has been studied. Having compared the various options, it appeared not well possible to significantly reduce the calculation time for the frequency, when based on FFT. Therefore, another microcontroller or microprocessor has been considered to increase the performance of the device. Several options have been shown to increase the overall performance using other hardware.

The implementation of a new method for measuring the frequency of the input signal in a faster way is not possible using the current hardware with the studied algorithm, the FFT. To increase the measurement speed, other hardware should be used.

6.2 Recommendations

The first recommendation will be to continue using the current DSP because, though it is not a powerful device in terms of calculation, is very versatile and already used. Seen that using the FFT we can not obtain better results, it will be necessary the implementation of some other algorithm shown in chapter 2.

If the results are not satisfactory it would be necessary a new study for changing the microprocessor with a bigger power of calculation than at present.

A Appendix - How FFT works

A.1 What a Fast Fourier Transform is

In the field of digital signal processing, the Discrete Fourier Transform (DFT) plays an important role in the analysis, design, and implementation of discrete-time signal-processing algorithms and systems. For instance, the DFT can be used to calculate a signal's frequency response, find a system's frequency response from the system's impulse response, and serve as an intermediate step in more elaborate signal processing techniques. The Fast Fourier Transform (FFT) is an efficient class of computational algorithms of the DFT. FFT algorithms are based on the fundamental principle of decomposing the computation of the DFT of a sequence of length N into successively smaller DFTs, all with comparable improvements in computational speed.

The study of FFT algorithms not only has a long history and large bibliography, and it is still an exciting research field where new results are used in practical applications. Efficient FFT algorithms were first discovered by Gauss, and later by Runge and Konig. The importance of FFT algorithms was not fully recognized until its rediscovery by Cooley and Tukey in 1960s. Since then, the research in FFT has been proliferated, to name a few, higher radix algorithms, mixed-radix, prime-factor, Winograd (WFTA), the split-radix Fourier transform algorithms, recursive FFT algorithm, and the combination of decimation-in-time and the decimation-in-frequency FFT algorithms.

A DFT decomposes a sequence of values into components of different frequencies. This operation is useful in many fields but computing it directly from the definition is often too slow to be practical. An FFT is a way to compute the same result more quickly: computing a DFT of N points in the obvious way, using the definition, takes O(N^2) arithmetical operations, while an FFT can compute the same result in only O($N \log_2 N$) operations. The difference in speed can be substantial, especially for long data sets where N may be in the thousands or millions-in practice, the computation time can be reduced by several orders of magnitude in such cases, and the improvement is roughly proportional to $N / \log_2 N$.

An FFT computes the DFT and produces exactly the same result as evaluating the DFT definition directly; the only difference is that an FFT is much faster.

Let $x_0, ..., x_{N-1}$ be complex numbers. The DFT is defined by:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi j}{N} nk}$$

$$k = 0, ..., n-1$$

FFTs are algorithms for quick calculation of Discrete Fourier Transform of a data vector. The FFT is a DFT algorithm which reduces the number of computations needed for N points from O(N^2) to O($N \log_2 N$).

The 'Radix-2' algorithms are useful if N is a regular power of 2 ($N = 2^p$). If we assume that algorithmic complexity provides a direct measure of execution time and that the relevant logarithm base is 2 then as shown in Table A.1, ratio of execution times for the (DFT) vs. (Radix-2 FFT) (denoted as 'Speed Improvement Factor') increases tremendously with increase in N.

The term 'FFT' is actually slightly ambiguous, because there are several commonly used 'FFT' algorithms. There are two different Radix-2 algorithms, the so-called 'Decimation in Time' (DIT) and 'Decimation in Frequency' (DIF) algorithms. Both of these rely on the recursive decomposition of an N point transform into 2 (N/2) point transforms. This decomposition process can be applied to any composite (non prime) N. The method is particularly simple if N

Number of Points, N	Complex Multipications in Direct Computation, N^2	Complex Multiplications in FFT Algorithm, $(N/2)\log_2 N$	Speed Improvement Factor
4	16	4	4,0
8	64	12	5,3
16	256	32	8,0
32	1024	80	12,8
64	4096	192	21,3
128	16384	448	36,6
256	65536	1024	64,0
512	262144	2304	113,8
1024	1048576	5120	204,8

TABLE A.1 - Comparison of execution times, DFT and Radix-2 FFT.

is divisible by 2 and if N is a regular power of 2, the decomposition can be applied repeatedly until the trivial '1 point' transform is reached.

The decimation in frequency (DIF) algorithm is obtained essentially by dividing the output sequence X(k), rather than input sequence x(n), into smaller subsequences. To derive this algorithm, we group the first N/2 points and the last N/2 points of the sequence x(n) together and write

$$\begin{aligned} X(k) &= \sum_{n=0}^{(N/2)-1} x(n)W_N^{nk} + \sum_{n=N/2}^{N-1} x(n)W_N^{nk} = \\ &= \sum_{n=0}^{N/2-1} x(n)W_N^{nk} + W_N^{N/2k} \sum_{n=0}^{N/2-1} x\left(n+\tfrac{N}{2}\right) W_N^{nk} \end{aligned} \tag{A.1}$$

We can combine the two terms in Equation A.1 by nothing that $W_N^{Nk/2} = (-1)^k$, to get

$$X(k) = \sum_{n=0}^{(N/2)-1} \left[x(n) + (-1)^k x\left(n+\frac{N}{2}\right) \right] W_N^{nk} \tag{A.2}$$

Let

$$g(n) = x(n) + x\left(n+\frac{N}{2}\right)$$

and

$$h(n) = \left[x(n) - x\left(n+\frac{N}{2}\right) \right] W_N^n$$

where $0 \leq n \leq (N/2)-1$

For k even, we can set $k = 2r$ and write the Equation A.2 as

$$X(2r) = \sum_{n=0}^{(N/2)-1} g(n)W_N^{2rn} = \sum_{n=0}^{(N/2)-1} g(n)W_{N/2}^{rn} \tag{A.3}$$

Similarly, setting $k = 2r+1$ fives the expression for odd values of k:

$$X(2r+1) = \sum_{n=0}^{(N/2)-1} h(n)W_N^{2rn} = \sum_{n=0}^{(N/2)-1} h(n)W_{N/2}^{rn} \tag{A.4}$$

Equation A.3 and Equation A.4 represent the N/2-point DFTs of the sequences G(k) and H(k), respectively. Thus, the computation of X(k) involves first forming the sequences g(n) and h(n) and then computing their DFTs to obtain the even and odd values of X(k). This is illustrated in Figure A.1 for the case N=8.

We can proceed to determine the two (N/2)-point DFTs G(k) and H(k) by computing the even and odd values separately using a similar procedure. That is, we form the sequences

$$g_1(n) = g(n) + g\left(n + \frac{N}{4}\right)$$

$$g_2(n) = \left[g(n) - g\left(n + \frac{N}{4}\right)\right] W_{N/2}^n$$

and

$$h_1(n) = h(n) + h\left(n + \frac{N}{4}\right)$$

$$h_2(n) = \left[h(n) - h\left(n + \frac{N}{4}\right)\right] W_{N/2}^n$$

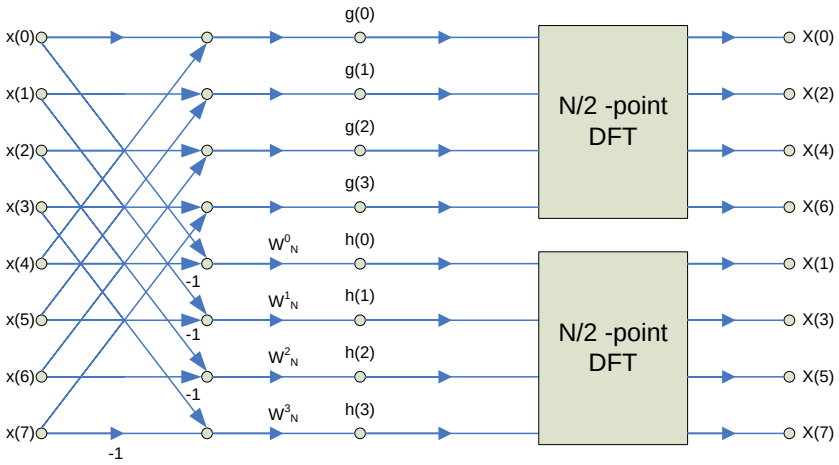


FIGURE A.1 - First stage of DIF graph.

We can continue this procedure until we have a set of two-point sequences, which are implemented by adding and subtracting the input values. Figure A.2 shows the complete flow graph for the computation of an eight-point DFT. As can be seen from the figure, the input in this case is in its natural order, and the output is in bit-reversed order.

By far the most common FFT is the Cooley-Tukey algorithm. This is a 'divide and conquer' algorithm that recursively breaks down a DFT of any composite size N = N1N2 into many smaller DFTs of sizes N1 and N2, along with O(N) multiplications by complex roots of unity, traditionally called twiddle factors.

A twiddle factor, in fast Fourier transform (FFT) algorithms, is any of the trigonometric constant coefficients that are multiplied by the data in the course of the algorithm. More specifically, "twiddle factors" originally referred to the root-of-unity complex multiplicative constants in the

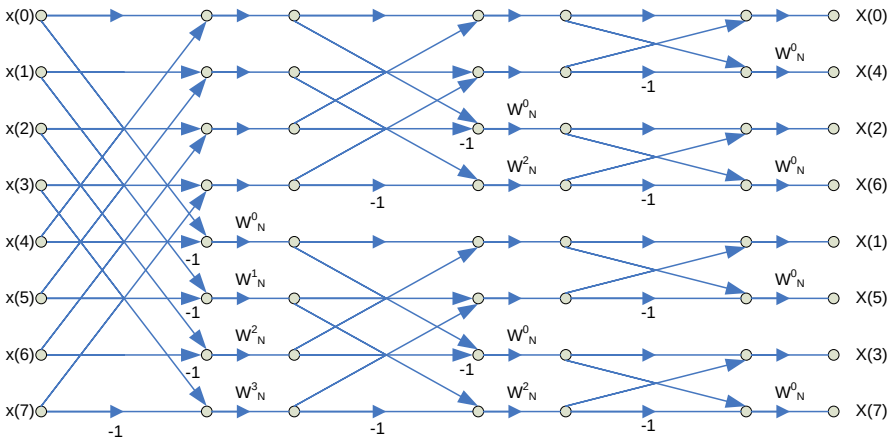


FIGURE A.2 - Complete flow graph for DIF algorithm, N=8.

butterfly operations of the Cooley-Tukey FFT algorithm, used to recursively combine smaller discrete Fourier transforms. This remains the term's most common meaning, but it may also be used for any data-independent multiplicative constant in an FFT.

The other point of consideration in the design of the FFT system is the handling of the twiddle factors. For most small DSP systems, the twiddle factors can be stored in on-chip memory. This works fine for smaller systems but would require huge amounts of memory for large systems like the one under consideration. For the 100.000-point FFT under consideration, a 100.000 twiddle factors would have to be stored in memory. The other alternative is to generate the twiddle factors needed for the FFT on the chip itself. This option has been ruled out due to the time used to calculate the twiddle factors would be excessive.

The most well-known use of the Cooley-Tukey algorithm is to divide the transform into two pieces of size N/2 at each step, and is therefore limited to power-of-two sizes, but any factorization can be used in general (as was known to both Gauss and Cooley/Tukey). These are called the radix-2 and mixed-radix cases, respectively. Although the basic idea is recursive, most traditional implementations rearrange the algorithm to avoid explicit recursion. Also, because the Cooley-Tukey algorithm breaks the DFT into smaller DFTs, it can be combined arbitrarily with any other algorithm for the DFT. (34)

A.2 How the FFT algorithm works

In complex notation, the time and frequency domains each contain one signal made up of N complex points. Each of these complex points is composed of two numbers, the real part and the imaginary part. Each complex variable holds two numbers. When two complex variables are multiplied, the four individual components must be combined to form the two components of the product.

The FFT operates by decomposing an N point time domain signal into N time domain signals each composed of a single point. The second step is to calculate the N frequency spectra corresponding to these N time domain signals. Lastly, the N spectra are synthesized into a single frequency spectrum.

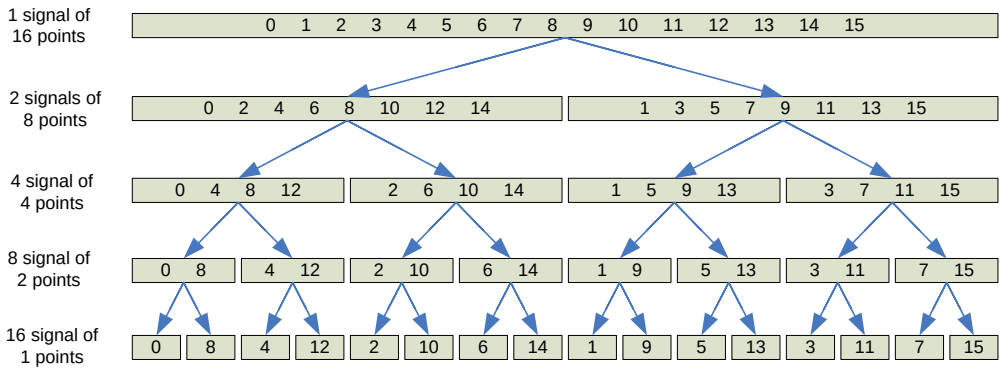


FIGURE A.3 - The FFT decomposition. An N point signal is decomposed into N signals each containing a single point. Each stage uses an interlace decomposition, separating the even and odd numbered samples.

Sample numbers in normal order			Sample numbers after bit reversal	
Decimal	Binary		Decimal	Binary
0	0000		0	0000
1	0001		8	1000
2	0010		4	0100
3	0011	⇒	12	1100
4	0100		2	0010
5	0101		10	1010
6	0110		6	0110
7	0111	⇒	14	1110
8	1000		1	0001
9	1001		9	1001
10	1010		5	0101
11	1011	⇒	13	1101
12	1100		3	0011
12	1101		11	1011
14	1110		7	0111
15	1111	⇒	15	1111

TABLE A.2 - The FFT bit reversal sorting. The FFT time domain decomposition can be implemented by sorting the samples according to bit reversed order.

Figure A.3 shows an example of the time domain decomposition used in the FFT. In this example, a 16 point signal is decomposed through four separate stages. The first stage breaks the 16 point signal into two signals each consisting of 8 points. The second stage decomposes the data into four signals of 4 points. This pattern continues until there are N signals composed of a single point. An interlaced decomposition is used each time a signal is broken in two, that is, the signal is separated into its even and odd numbered samples. There are $\log_2 N$ stages required in this decomposition.

The decomposition is nothing more than a reordering of the samples in the signal. Table A.2 shows the rearrangement pattern required. On the left, the sample numbers of the original signal are listed along with their binary equivalents. On the right, the rearranged sample numbers are listed, also along with their binary equivalents. The important idea is that the binary numbers are the reversals of each other. For example, sample 3 (0011) is exchanged with sample number 12 (1100). Likewise, sample number 14 (1110) is swapped with sample number 7 (0111), and so forth. The FFT time domain decomposition is usually carried out by a bit reversal sorting algorithm. This involves rearranging the order of the N time domain samples by counting in binary with the bits flipped left-for-right (such as in the far right column in Table A.2).

The next step in the FFT algorithm is to find the frequency spectra of the 1 point time domain signals. The frequency spectrum of a 1 point signal is equal to itself. This means that nothing is required to do in this step. Each of the 1 point signals is now a frequency spectrum, and not a time domain signal.

The last step in the FFT is to combine the N frequency spectra in the exact reverse order that the time domain decomposition took place. Unfortunately, the bit reversal shortcut is not applicable, and we must go back one stage at a time. In the first stage, 16 frequency spectra (1 point each) are synthesized into 8 frequency spectra (2 points each). In the second stage, the 8 frequency spectra (2 points each) are synthesized into 4 frequency spectra (4 points each), and so on. The last stage results in the output of the FFT, a 16 point frequency spectrum.

Figure A.4 shows how two frequency spectra, each composed of 4 points, are combined into a single frequency spectrum of 8 points. This synthesis must undo the interlaced decomposition done in the time domain. In other words, the frequency domain operation must correspond to the time domain procedure of combining two 4 point signals by interlacing. Consider two time domain signals, abcd and efgh. An 8 point time domain signal can be formed by two steps: dilute each 4 point signal with zeros to make it an 8 point signal, and then add the signals together. That is, abcd becomes a0b0c0d0, and efgh becomes 0e0f0g0h. Adding these two 8 point signals produces aebfcgdh. As shown in Figure A.4, diluting the time domain with zeros corresponds to a duplication of the frequency spectrum. Therefore, the frequency spectra are combined in the FFT by duplicating them, and then adding the duplicated spectra together.

In order to match up when added, the two time domain signals are diluted with zeros in a slightly different way. In one signal, the odd points are zero, while in the other signal, the even points are zero. In other words, one of the time domain signals (0e0f0g0h in Figure A.4) is shifted to the right by one sample. This time domain shift corresponds to multiplying the spectrum by a sinusoid. To see this, recall that a shift in the time domain is equivalent to convolving the signal with a shifted delta function. This multiplies the signal's spectrum with the spectrum of the shifted delta function. The spectrum of a shifted delta function is a sinusoid (see Figure A.3).

Figure A.5 shows a flow diagram for combining two 4 point spectra into a single 8 point spectrum. To reduce the situation even more, notice that Figure A.5 is formed from the basic pattern in Figure A.6 repeated over and over.

This simple flow diagram is called a butterfly due to its winged appearance. The butterfly is the basic computational element of the FFT, transforming two complex points into two other

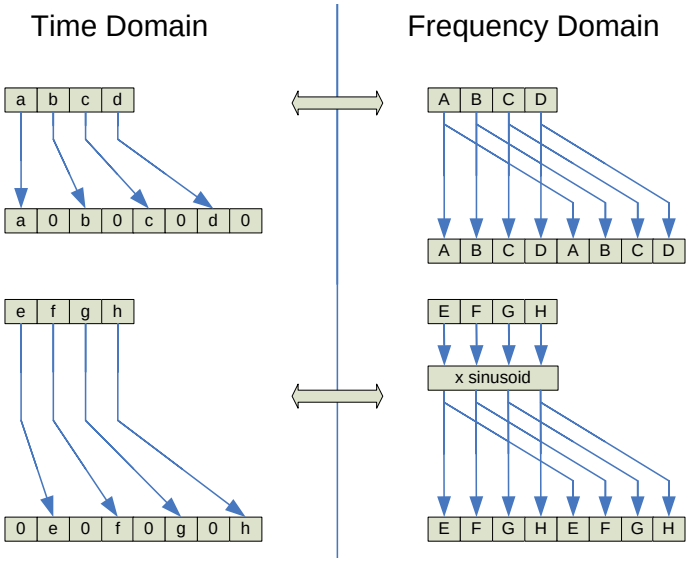


FIGURE A.4 - The FFT synthesis. When a time domain signal is diluted with zeros, the frequency domain is duplicated. If the time domain signal shifted by one sample during the dilution, the spectrum will additionally be multiplied by a sinusoid.

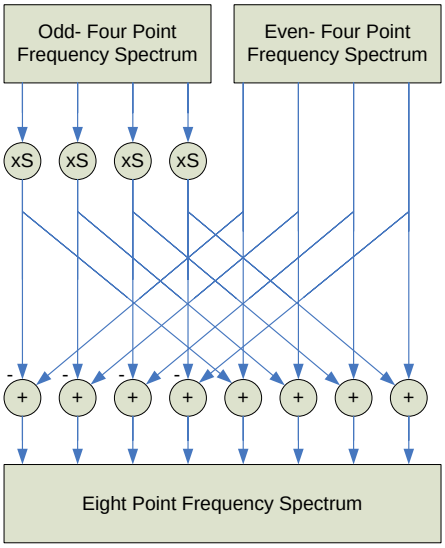


FIGURE A.5 - The FFT synthesis flow diagram. This shows the method of combining two 4 point frequency spectra into a single 8 point frequency spectrum. The xS operation means that the signal is multiplied by a sinusoid with an appropriately selected frequency.

complex points.

Figure A.7 shows the structure of the entire FFT. The time domain decomposition is accomplished with a bit reversal sorting algorithm. Transforming the decomposed data into the frequency domain involves nothing and therefore does not appear in the figure.

The frequency domain synthesis requires three loops. The outer loop runs through the $\log_2 N$ stages. The middle loop moves through each of the individual frequency spectra in the stage being worked on. The innermost loop uses the butterfly to calculate the points in each frequency

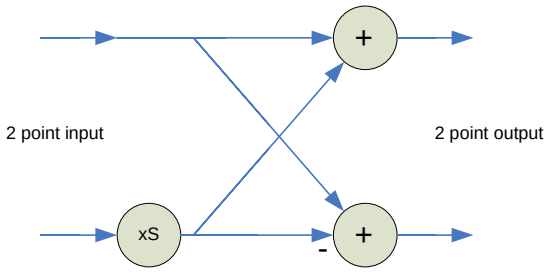


FIGURE A.6 - The FFT butterfly. This is the basic calculation element in the FFT, taking two complex points and converting them into two other complex points.

spectra. The overhead boxes in Figure A.7 determine the beginning and ending indexes for the loops, as well as calculating the sinusoids needed in the butterflies. (32)

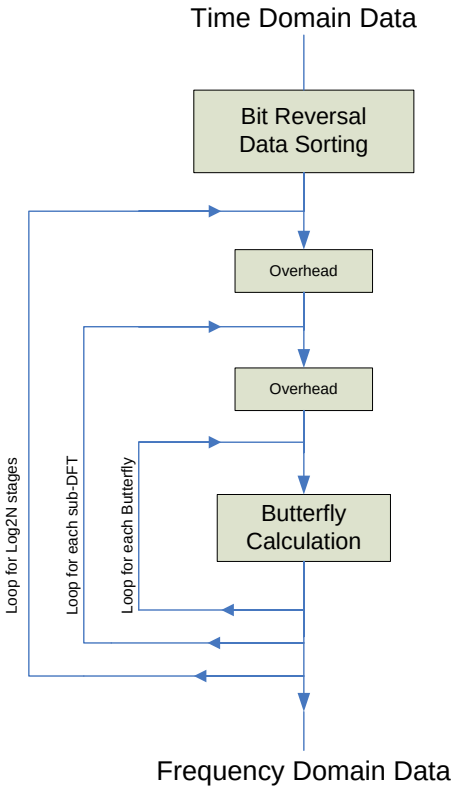


FIGURE A.7 - Flow diagram of the FFT. This is based on three steps: (1) decompose an N point time domain signal into N signals each containing a single point, (2) find the spectrum of each of the N point signals (nothing required), and (3) synthesize the N frequency spectra into a single frequency spectrum.

A.3 Microchip FFT code

Introduction

This example has been used to measure different signal in the input with different configurations in the FFT code, using 256 and 512 points.

The signals have been generated by dsPICworks and then exported as an assembler file from dsPICworks to be able to use in this example code.

Description

Microchip's 16-bit dsPIC Digital Signal Controllers feature a DSP Engine in the CPU that is capable of executing a Fast Fourier Transform (FFT) with great efficiency (high speed and low RAM usage). The on-chip features enabling the FFT implementation include, bit-reversed addressing, Multiply-accumulate (MAC) type instructions and the ability to store and retrieve constants stored in Program memory.

The code example is reconfigurable to perform an FFT of any size, including common sizes of 64, 128, 256 and 512 points. The code example also allows the user to place the FFT coefficients (known as Twiddle Factors) in RAM or in Program Flash Memory. The project may be easily reconfigured modyfing the file *FFT.h* in Appendix A.3.2.

The FFT operation is performed on the input signal, in-place. This means that the output of the FFT resides in the same RAM locations where the input signal used to reside. The FFT is performed in the following steps as can be seen in the file *mainFFTExample.c* in Appendix A.3.1:

1 Initialization: Generate Twiddle Factor Coefficients and store them in X-RAM or alternately use twiddle factor coefficients stored in Program Flash (*twiddleFactors.c* in Appendix A.3.3).
2 Scale the input signal to lie within the range [-0.5, +0.5]. For fixed point fractional input data, this translates to input samples in the range [0xC000,0x3FFF]. The scaling is achieved by simply right-shifting the input samples by 1 bit, assuming the input samples lie in the fixed point range [0x8000,0x7FFF] or [-1,+1).
3 Convert the real input signal vector to a complex vector by placing zeros in every other location to signify a complex input whose imaginary part is 0x0000.
4 Butterfly computation: This is achieved by performing a call to the *FFTComplexIP()* function.
5 Bit-Reversed Re-ordering: The output array is re-ordered to be in bit-reversed order of the addresses. This is achieved by a function call to *BitReverseComplex()*.
6 SquareMagnitude computation: We then need to compute the magnitude of each complex element in the output vector, so that we can estimate the energy in each spectral component. This is achieved by a call to a special C-callable routine, *SquareMagnitudeCplx()*, written in assembler language.
7 Peak-picking: We then find the frequency component with the largest energy by using the *VectorMax()* routine in the DSP library.
8 Frequency Calculation: The value of the spectral component with the highest energy, in Hz, is calculated by multiplying the array index of the largest element in the output array with the spectral (bin) resolution (= sampling rate/FFT size).

A.3.1 mainFFTExample.c

```
/************************************************************************
* 2005 Microchip Technology Inc.
*
* FileName:         mainFFTExample.c
* Dependencies:     Header (.h) files if applicable, see below
* Processor:        dsPIC30Fxxxx
* Compiler:         MPLAB® C30 v3.00 or higher
* IDE:              MPLAB® IDE v7.52 or later
* Dev. Board Used: dsPICDEM 1.1 Development Board
* Hardware Dependencies: None
*
* SOFTWARE LICENSE AGREEMENT:
* Microchip Technology Incorporated ("Microchip") retains all ownership and
* intellectual property rights in the code accompanying this message and in all
* derivatives hereto.  You may use this code, and any derivatives created by
* any person or entity by or on your behalf, exclusively with Microchip's
* proprietary products.  Your acceptance and/or use of this code constitutes
* agreement to the terms and conditions of this notice.
*
* CODE ACCOMPANYING THIS MESSAGE IS SUPPLIED BY MICROCHIP "AS IS".  NO
* WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED
* TO, IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A
* PARTICULAR PURPOSE APPLY TO THIS CODE, ITS INTERACTION WITH MICROCHIP'S
* PRODUCTS, COMBINATION WITH ANY OTHER PRODUCTS, OR USE IN ANY APPLICATION.
*
* YOU ACKNOWLEDGE AND AGREE THAT, IN NO EVENT, SHALL MICROCHIP BE LIABLE, WHETHER
* IN CONTRACT, WARRANTY, TORT (INCLUDING NEGLIGENCE OR BREACH OF STATUTORY DUTY),
* STRICT LIABILITY, INDEMNITY, CONTRIBUTION, OR OTHERWISE, FOR ANY INDIRECT, SPECIAL,
* PUNITIVE, EXEMPLARY, INCIDENTAL OR CONSEQUENTIAL LOSS, DAMAGE, FOR COST OR EXPENSE OF
* ANY KIND WHATSOEVER RELATED TO THE CODE, HOWSOEVER CAUSED, EVEN IF MICROCHIP HAS BEEN
```

```
* ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE.  TO THE FULLEST EXTENT
* ALLOWABLE BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO
* THIS CODE, SHALL NOT EXCEED THE PRICE YOU PAID DIRECTLY TO MICROCHIP SPECIFICALLY TO
* HAVE THIS CODE DEVELOPED.
*
* You agree that you are solely responsible for testing the code and
* determining its suitability.  Microchip has no obligation to modify, test,
* certify, or support the code.
*
* REVISION HISTORY:
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
* Author            Date      Comments on this revision
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
* HV                09/30/05  First release of source file
*
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
*
* ADDITIONAL NOTES:
*
*
**************************************************************************/
#include <p30Fxxxx.h>
#include <dsp.h>
#include "fft.h"

/* Device configuration register macros for building the hex file */
_FOSC(CSW_FSCM_OFF & XT_PLL8);          /* XT with 8xPLL oscillator, Failsafe clock off */
_FWDT(WDT_OFF);                         /* Watchdog timer disabled */
_FBORPOR(PBOR_OFF & MCLR_EN);           /* Brown-out reset disabled, MCLR reset enabled */
_FGS(CODE_PROT_OFF);                    /* Code protect disabled */
```

```
/* Extern definitions */
extern fractcomplex sigCmpx[FFT_BLOCK_LENGTH]          /* Typically, the input signal to an FFT  */
__attribute__ ((section (".ydata, data, ymemory"),     /* routine is a complex array containing samples */
aligned (FFT_BLOCK_LENGTH * 2 *2)));                   /* of an input signal. For this example, */
                                                       /* we will provide the input signal in an */
                                                       /* array declared in Y-data space. */
/* Global Definitions */
#ifndef FFTTWIDCOEFFS_IN_PROGMEM
fractcomplex twiddleFactors[FFT_BLOCK_LENGTH/2]        /* Declare Twiddle Factor array in X-space*/
__attribute__ ((section (".xbss, bss, xmemory"), aligned (FFT_BLOCK_LENGTH*2)));
#else
extern const fractcomplex twiddleFactors[FFT_BLOCK_LENGTH/2]    /* Twiddle Factor array in Program memory */
__attribute__ ((space(auto_psv), aligned (FFT_BLOCK_LENGTH*2)));
#endif

int      peakFrequencyBin = 0;                         /* Declare post-FFT variables to compute the */
unsigned long peakFrequency = 0;                       /* frequency of the largest spectral component */

int main(void)
{
        int i = 0;
        fractional *p_real = &sigCmpx[0].real ;
        fractcomplex *p_cmpx = &sigCmpx[0] ;


#ifndef FFTTWIDCOEFFS_IN_PROGMEM                                    /* Generate TwiddleFactor Coefficients */
        TwidFactorInit (LOG2_BLOCK_LENGTH, &twiddleFactors[0], 0);  /* We need to do this only once at start-up */
#endif

        for ( i = 0; i < FFT_BLOCK_LENGTH; i++ )/* The FFT function requires input data */
```

```
        {                                       /* to be in the fractional fixed-point range [-0.5, +0.5] */
                *p_real = *p_real >>1 ;         /* So, we shift all data samples by 1 bit to the right. */
                *p_real++;                      /* Should you desire to optimize this process, perform */
        }                                       /* data scaling when first obtaining the time samples */
                                                /* Or within the BitReverseComplex function source code */

        p_real = &sigCmpx[(FFT_BLOCK_LENGTH/2)-1].real ;        /* Set up pointers to convert real array */
        p_cmpx = &sigCmpx[FFT_BLOCK_LENGTH-1] ; /* to a complex array. The input array initially has all */
                                                /* the real input samples followed by a series of zeros */


        for ( i = FFT_BLOCK_LENGTH; i > 0; i-- ) /* Convert the Real input sample array */
        {                                       /* to a Complex input sample array  */
                (*p_cmpx).real = (*p_real--);   /* We will simpy zero out the imaginary  */
                (*p_cmpx--).imag = 0x0000;      /* part of each data sample */
        }

        /* Perform FFT operation */
#ifndef FFTTWIDCOEFFS_IN_PROGMEM
        FFTComplexIP (LOG2_BLOCK_LENGTH, &sigCmpx[0], &twiddleFactors[0], COEFFS_IN_DATA);
#else
        FFTComplexIP (LOG2_BLOCK_LENGTH, &sigCmpx[0], (fractcomplex *) __builtin_psvoffset(&twiddleFactors[0]),
        (int) __builtin_psvpage(&twiddleFactors[0]));
#endif

        /* Store output samples in bit-reversed order of their addresses */
        BitReverseComplex (LOG2_BLOCK_LENGTH, &sigCmpx[0]);

        /* Compute the square magnitude of the complex FFT output array so we have a Real output vetor */
        SquareMagnitudeCplx(FFT_BLOCK_LENGTH, &sigCmpx[0], &sigCmpx[0].real);
```

```
/* Find the frequency Bin ( = index into the SigCmpx[] array) that has the largest energy*/
/* i.e., the largest spectral component */
VectorMax(FFT_BLOCK_LENGTH/2, &sigCmpx[0].real, &peakFrequencyBin);

/* Compute the frequency (in Hz) of the largest spectral component */
peakFrequency = peakFrequencyBin*(SAMPLING_RATE/FFT_BLOCK_LENGTH);

while (1);      /* Place a breakpoint here and observe the watch window variables */
}
```

A.3.2 FFT.h

```
/* Constant Definitions */
#define FFT_BLOCK_LENGTH        256     /* = Number of frequency points in the FFT */
#define LOG2_BLOCK_LENGTH       8       /* = Number of "Butterfly" Stages in FFT processing */
#define SAMPLING_RATE           10000   /* = Rate at which input signal was sampled */
                                        /* SAMPLING_RATE is used to calculate the frequency*/
                                        /* of the largest element in the FFT output vector*/

#define FFTTWIDCOEFFS_IN_PROGMEM        /*<---Comment out this line of the code if twiddle factors (coefficients) */
                                        /*reside in data memory (RAM) as opposed to Program Memory */
                                        /*Then remove the call to "TwidFactorInit()" and add the twiddle factor*/
                                        /*coefficient file into your Project. An example file for a 256-pt FFT*/
                                        /*is provided in this Code example */
```

A.3.3 twiddleFactors.c

```
/*********************************************************************
* 2005 Microchip Technology Inc.
*
* FileName:         twiddleFactors.c
* Dependencies:     Header (.h) files if applicable, see below
* Processor:        dsPIC30Fxxxx
* Compiler:         MPLAB® C30 v3.00 or higher
* IDE:              MPLAB® IDE v7.52 or later
* Dev. Board Used: dsPICDEM 1.1 Development Board
* Hardware Dependencies: None
*
* SOFTWARE LICENSE AGREEMENT:
* Microchip Technology Incorporated ("Microchip") retains all ownership and
* intellectual property rights in the code accompanying this message and in all
* derivatives hereto.  You may use this code, and any derivatives created by
* any person or entity by or on your behalf, exclusively with Microchip's
* proprietary products.  Your acceptance and/or use of this code constitutes
* agreement to the terms and conditions of this notice.
*
* CODE ACCOMPANYING THIS MESSAGE IS SUPPLIED BY MICROCHIP "AS IS".  NO
* WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED
* TO, IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A
* PARTICULAR PURPOSE APPLY TO THIS CODE, ITS INTERACTION WITH MICROCHIP'S
* PRODUCTS, COMBINATION WITH ANY OTHER PRODUCTS, OR USE IN ANY APPLICATION.
*
* YOU ACKNOWLEDGE AND AGREE THAT, IN NO EVENT, SHALL MICROCHIP BE LIABLE, WHETHER
* IN CONTRACT, WARRANTY, TORT (INCLUDING NEGLIGENCE OR BREACH OF STATUTORY DUTY),
* STRICT LIABILITY, INDEMNITY, CONTRIBUTION, OR OTHERWISE, FOR ANY INDIRECT, SPECIAL,
* PUNITIVE, EXEMPLARY, INCIDENTAL OR CONSEQUENTIAL LOSS, DAMAGE, FOR COST OR EXPENSE OF
* ANY KIND WHATSOEVER RELATED TO THE CODE, HOWSOEVER CAUSED, EVEN IF MICROCHIP HAS BEEN
```

```
* ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE.  TO THE FULLEST EXTENT
* ALLOWABLE BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO
* THIS CODE, SHALL NOT EXCEED THE PRICE YOU PAID DIRECTLY TO MICROCHIP SPECIFICALLY TO
* HAVE THIS CODE DEVELOPED.
*
* You agree that you are solely responsible for testing the code and
* determining its suitability.  Microchip has no obligation to modify, test,
* certify, or support the code.
*
* REVISION HISTORY:
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
* Author            Date      Comments on this revision
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
* HV                09/30/05  First release of source file
*
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
*
* ADDITIONAL NOTES:
*
*
**************************************************************************/

#include <dsp.h>
#include "fft.h"

#ifdef FFTTWIDCOEFFS_IN_PROGMEM

#if (FFT_BLOCK_LENGTH == 64)
        const fractcomplex twiddleFactors[] __attribute__ ((space(auto_psv), aligned (FFT_BLOCK_LENGTH*2)))=
        {
        0x7FFF, 0x0000, 0x7F62, 0xF374, 0x7D8A, 0xE707, 0x7A7D, 0xDAD8,
```

```
    0x7642, 0xCF04, 0x70E3, 0xC3A9, 0x6A6E, 0xB8E3, 0x62F2, 0xAECC,
    0x5A82, 0xA57E, 0x5134, 0x9D0E, 0x471D, 0x9592, 0x3C57, 0x8F1D,
    0x30FC, 0x89BE, 0x2528, 0x8583, 0x18F9, 0x8276, 0x0C8C, 0x809E,
    0x0000, 0x8000, 0xF374, 0x809E, 0xE707, 0x8276, 0xDAD8, 0x8583,
    0xCF04, 0x89BE, 0xC3A9, 0x8F1D, 0xB8E3, 0x9592, 0xAECC, 0x9D0E,
    0xA57D, 0xA57D, 0x9D0E, 0xAECC, 0x9592, 0xB8E3, 0x8F1D, 0xC3A9,
    0x89BE, 0xCF04, 0x8583, 0xDAD8, 0x8276, 0xE707, 0x809E, 0xF374
    } ;
#endif
#if (FFT_BLOCK_LENGTH == 128)
    const fractcomplex twiddleFactors[] __attribute__ ((space(auto_psv), aligned (FFT_BLOCK_LENGTH*2)))=
    {
    0x7FFF, 0x0000, 0x7FD9, 0xF9B8, 0x7F62, 0xF374, 0x7E9D, 0xED38,
    0x7D8A, 0xE707, 0x7C2A, 0xE0E6, 0x7A7D, 0xDAD8, 0x7885, 0xD4E1,
    0x7642, 0xCF04, 0x73B6, 0xC946, 0x70E3, 0xC3A9, 0x6DCA, 0xBE32,
    0x6A6E, 0xB8E3, 0x66D0, 0xB3C0, 0x62F2, 0xAECC, 0x5ED7, 0xAA0A,
    0x5A82, 0xA57E, 0x55F6, 0xA129, 0x5134, 0x9D0E, 0x4C40, 0x9930,
    0x471D, 0x9592, 0x41CE, 0x9236, 0x3C57, 0x8F1D, 0x36BA, 0x8C4A,
    0x30FC, 0x89BE, 0x2B1F, 0x877B, 0x2528, 0x8583, 0x1F1A, 0x83D6,
    0x18F9, 0x8276, 0x12C8, 0x8163, 0x0C8C, 0x809E, 0x0648, 0x8027,
    0x0000, 0x8000, 0xF9B8, 0x8027, 0xF374, 0x809E, 0xED38, 0x8163,
    0xE707, 0x8276, 0xE0E6, 0x83D6, 0xDAD8, 0x8583, 0xD4E1, 0x877C,
    0xCF04, 0x89BE, 0xC946, 0x8C4A, 0xC3A9, 0x8F1D, 0xBE32, 0x9236,
    0xB8E3, 0x9592, 0xB3C0, 0x9931, 0xAECC, 0x9D0E, 0xAA0A, 0xA129,
    0xA57E, 0xA57E, 0xA129, 0xAA0A, 0x9D0E, 0xAECC, 0x9931, 0xB3C0,
    0x9592, 0xB8E3, 0x9236, 0xBE32, 0x8F1D, 0xC3A9, 0x8C4A, 0xC946,
    0x89BE, 0xCF04, 0x877C, 0xD4E1, 0x8583, 0xDAD8, 0x83D6, 0xE0E6,
    0x8276, 0xE707, 0x8163, 0xED38, 0x809E, 0xF374, 0x8027, 0xF9B8
    } ;
#endif
#if (FFT_BLOCK_LENGTH == 256)
```

```
const fractcomplex twiddleFactors[] __attribute__ ((space(auto_psv), aligned (FFT_BLOCK_LENGTH*2))) =
{
0x7FFF, 0x0000, 0x7FF6, 0xFCDC, 0x7FD9, 0xF9B8, 0x7FA7, 0xF695,
0x7F62, 0xF374, 0x7F0A, 0xF055, 0x7E9D, 0xED38, 0x7E1E, 0xEA1E,
0x7D8A, 0xE707, 0x7CE4, 0xE3F4, 0x7C2A, 0xE0E6, 0x7B5D, 0xDDDC,
0x7A7D, 0xDAD8, 0x798A, 0xD7D9, 0x7884, 0xD4E1, 0x776C, 0xD1EF,
0x7642, 0xCF04, 0x7505, 0xCC21, 0x73B6, 0xC946, 0x7255, 0xC673,
0x70E3, 0xC3A9, 0x6F5F, 0xC0E9, 0x6DCA, 0xBE32, 0x6C24, 0xBB85,
0x6A6E, 0xB8E3, 0x68A7, 0xB64C, 0x66CF, 0xB3C0, 0x64E8, 0xB140,
0x62F2, 0xAECC, 0x60EC, 0xAC65, 0x5ED7, 0xAA0A, 0x5CB4, 0xA7BD,
0x5A82, 0xA57E, 0x5843, 0xA34C, 0x55F6, 0xA129, 0x539B, 0x9F14,
0x5134, 0x9D0E, 0x4EC0, 0x9B18, 0x4C40, 0x9931, 0x49B4, 0x9759,
0x471D, 0x9592, 0x447B, 0x93DC, 0x41CE, 0x9236, 0x3F17, 0x90A1,
0x3C57, 0x8F1D, 0x398D, 0x8DAB, 0x36BA, 0x8C4A, 0x33DF, 0x8AFB,
0x30FC, 0x89BE, 0x2E11, 0x8894, 0x2B1F, 0x877C, 0x2827, 0x8676,
0x2528, 0x8583, 0x2224, 0x84A3, 0x1F1A, 0x83D6, 0x1C0B, 0x831C,
0x18F9, 0x8276, 0x15E2, 0x81E3, 0x12C8, 0x8163, 0x0FAB, 0x80F7,
0x0C8C, 0x809E, 0x096B, 0x8059, 0x0648, 0x8028, 0x0324, 0x800A,
0x0000, 0x8000, 0xFCDC, 0x800A, 0xF9B8, 0x8028, 0xF695, 0x8059,
0xF374, 0x809E, 0xF055, 0x80F7, 0xED38, 0x8163, 0xEA1E, 0x81E3,
0xE707, 0x8276, 0xE3F5, 0x831C, 0xE0E6, 0x83D6, 0xDDDC, 0x84A3,
0xDAD8, 0x8583, 0xD7D9, 0x8676, 0xD4E1, 0x877C, 0xD1EF, 0x8894,
0xCF04, 0x89BE, 0xCC21, 0x8AFB, 0xC946, 0x8C4A, 0xC673, 0x8DAB,
0xC3A9, 0x8F1D, 0xC0E9, 0x90A1, 0xBE32, 0x9236, 0xBB85, 0x93DC,
0xB8E3, 0x9593, 0xB64C, 0x975A, 0xB3C0, 0x9931, 0xB140, 0x9B18,
0xAECC, 0x9D0E, 0xAC65, 0x9F14, 0xAA0A, 0xA129, 0xA7BD, 0xA34C,
0xA57E, 0xA57E, 0xA34C, 0xA7BD, 0xA129, 0xAA0A, 0x9F14, 0xAC65,
0x9D0E, 0xAECC, 0x9B18, 0xB140, 0x9931, 0xB3C0, 0x975A, 0xB64C,
0x9593, 0xB8E3, 0x93DC, 0xBB85, 0x9236, 0xBE32, 0x90A1, 0xC0E9,
0x8F1D, 0xC3A9, 0x8DAB, 0xC673, 0x8C4A, 0xC946, 0x8AFB, 0xCC21,
0x89BF, 0xCF04, 0x8894, 0xD1EF, 0x877C, 0xD4E1, 0x8676, 0xD7D9,
```

```
    0x8583, 0xDAD8, 0x84A3, 0xDDDC, 0x83D6, 0xE0E6, 0x831C, 0xE3F5,
    0x8276, 0xE707, 0x81E3, 0xEA1E, 0x8163, 0xED38, 0x80F7, 0xF055,
    0x809E, 0xF374, 0x8059, 0xF695, 0x8028, 0xF9B8, 0x800A, 0xFCDC
    } ;
#endif
#if (FFT_BLOCK_LENGTH == 512 )
    const fractcomplex twiddleFactors[] __attribute__ ((space(auto_psv), aligned (FFT_BLOCK_LENGTH*2*2))) =
    {
    0x7FFF, 0x0000, 0x7FFE, 0xFE6E, 0x7FF6, 0xFCDC, 0x7FEA, 0xFB4A,
    0x7FD9, 0xF9B8, 0x7FC2, 0xF827, 0x7FA7, 0xF695, 0x7F87, 0xF505,
    0x7F62, 0xF374, 0x7F38, 0xF1E4, 0x7F0A, 0xF055, 0x7ED6, 0xEEC6,
    0x7E9D, 0xED38, 0x7E60, 0xEBAB, 0x7E1E, 0xEA1E, 0x7DD6, 0xE892,
    0x7D8A, 0xE707, 0x7D3A, 0xE57D, 0x7CE4, 0xE3F4, 0x7C89, 0xE26D,
    0x7C2A, 0xE0E6, 0x7BC6, 0xDF61, 0x7B5D, 0xDDDC, 0x7AEF, 0xDC59,
    0x7A7D, 0xDAD8, 0x7A06, 0xD958, 0x798A, 0xD7D9, 0x790A, 0xD65C,
    0x7885, 0xD4E1, 0x77FB, 0xD367, 0x776C, 0xD1EF, 0x76D9, 0xD079,
    0x7642, 0xCF04, 0x75A6, 0xCD92, 0x7505, 0xCC21, 0x7460, 0xCAB2,
    0x73B6, 0xC946, 0x7308, 0xC7DB, 0x7255, 0xC673, 0x719E, 0xC50D,
    0x70E3, 0xC3A9, 0x7023, 0xC248, 0x6F5F, 0xC0E9, 0x6E97, 0xBF8C,
    0x6DCA, 0xBE32, 0x6CF9, 0xBCDA, 0x6C24, 0xBB85, 0x6B4B, 0xBA33,
    0x6A6E, 0xB8E3, 0x698C, 0xB796, 0x68A7, 0xB64C, 0x67BD, 0xB505,
    0x66D0, 0xB3C0, 0x65DE, 0xB27F, 0x64E9, 0xB140, 0x63EF, 0xB005,
    0x62F2, 0xAECC, 0x61F1, 0xAD97, 0x60EC, 0xAC65, 0x5FE4, 0xAB36,
    0x5ED8, 0xAA0A, 0x5DC8, 0xA8E2, 0x5CB4, 0xA7BD, 0x5B9D, 0xA69C,
    0x5A83, 0xA57E, 0x5964, 0xA463, 0x5843, 0xA34C, 0x571E, 0xA238,
    0x55F6, 0xA128, 0x54CA, 0xA01C, 0x539B, 0x9F14, 0x5269, 0x9E0F,
    0x5134, 0x9D0E, 0x4FFB, 0x9C11, 0x4EC0, 0x9B17, 0x4D81, 0x9A22,
    0x4C40, 0x9930, 0x4AFB, 0x9843, 0x49B4, 0x9759, 0x486A, 0x9674,
    0x471D, 0x9592, 0x45CD, 0x94B5, 0x447B, 0x93DC, 0x4326, 0x9307,
    0x41CE, 0x9236, 0x4074, 0x9169, 0x3F17, 0x90A1, 0x3DB8, 0x8FDD,
    0x3C57, 0x8F1D, 0x3AF3, 0x8E62, 0x398D, 0x8DAB, 0x3825, 0x8CF8,
```

```
0x36BA, 0x8C4A, 0x354E, 0x8BA0, 0x33DF, 0x8AFB, 0x326E, 0x8A5A,
0x30FC, 0x89BE, 0x2F87, 0x8927, 0x2E11, 0x8894, 0x2C99, 0x8805,
0x2B1F, 0x877B, 0x29A4, 0x86F6, 0x2827, 0x8676, 0x26A8, 0x85FA,
0x2528, 0x8583, 0x23A7, 0x8511, 0x2224, 0x84A3, 0x209F, 0x843A,
0x1F1A, 0x83D6, 0x1D93, 0x8377, 0x1C0C, 0x831C, 0x1A83, 0x82C6,
0x18F9, 0x8276, 0x176E, 0x822A, 0x15E2, 0x81E2, 0x1455, 0x81A0,
0x12C8, 0x8163, 0x113A, 0x812A, 0x0FAB, 0x80F6, 0x0E1C, 0x80C8,
0x0C8C, 0x809E, 0x0AFB, 0x8079, 0x096B, 0x8059, 0x07D9, 0x803E,
0x0648, 0x8027, 0x04B6, 0x8016, 0x0324, 0x800A, 0x0192, 0x8002,
0x0000, 0x8000, 0xFE6E, 0x8002, 0xFCDC, 0x800A, 0xFB4A, 0x8016,
0xF9B8, 0x8027, 0xF827, 0x803E, 0xF695, 0x8059, 0xF505, 0x8079,
0xF374, 0x809E, 0xF1E4, 0x80C8, 0xF055, 0x80F6, 0xEEC6, 0x812A,
0xED38, 0x8163, 0xEBAB, 0x81A0, 0xEA1E, 0x81E2, 0xE892, 0x822A,
0xE707, 0x8276, 0xE57D, 0x82C6, 0xE3F4, 0x831C, 0xE26D, 0x8377,
0xE0E6, 0x83D6, 0xDF61, 0x843A, 0xDDDC, 0x84A3, 0xDC59, 0x8511,
0xDAD8, 0x8583, 0xD958, 0x85FA, 0xD7D9, 0x8676, 0xD65C, 0x86F6,
0xD4E1, 0x877B, 0xD367, 0x8805, 0xD1EF, 0x8894, 0xD079, 0x8927,
0xCF04, 0x89BE, 0xCD92, 0x8A5A, 0xCC21, 0x8AFB, 0xCAB2, 0x8BA0,
0xC946, 0x8C4A, 0xC7DB, 0x8CF8, 0xC673, 0x8DAB, 0xC50D, 0x8E62,
0xC3A9, 0x8F1D, 0xC248, 0x8FDD, 0xC0E9, 0x90A1, 0xBF8C, 0x9169,
0xBE32, 0x9236, 0xBCDA, 0x9307, 0xBB85, 0x93DC, 0xBA33, 0x94B5,
0xB8E3, 0x9592, 0xB796, 0x9674, 0xB64C, 0x9759, 0xB505, 0x9843,
0xB3C0, 0x9930, 0xB27F, 0x9A22, 0xB140, 0x9B17, 0xB005, 0x9C11,
0xAECC, 0x9D0E, 0xAD97, 0x9E0F, 0xAC65, 0x9F14, 0xAB36, 0xA01C,
0xA0A, 0xA128, 0xA8E2, 0xA238, 0xA7BD, 0xA34C, 0xA69C, 0xA463,
0xA57D, 0xA57D, 0xA463, 0xA69C, 0xA34C, 0xA7BD, 0xA238, 0xA8E2,
0xA128, 0xAA0A, 0xA01C, 0xAB36, 0x9F14, 0xAC65, 0x9E0F, 0xAD97,
0x9D0E, 0xAECC, 0x9C11, 0xB005, 0x9B17, 0xB140, 0x9A22, 0xB27F,
0x9930, 0xB3C0, 0x9843, 0xB504, 0x9759, 0xB64C, 0x9674, 0xB796,
0x9592, 0xB8E3, 0x94B5, 0xBA33, 0x93DC, 0xBB85, 0x9307, 0xBCDA,
0x9236, 0xBE32, 0x9169, 0xBF8C, 0x90A1, 0xC0E9, 0x8FDD, 0xC248,
```

```
        0x8F1D, 0xC3A9, 0x8E62, 0xC50D, 0x8DAB, 0xC673, 0x8CF8, 0xC7DB,
        0x8C4A, 0xC946, 0x8BA0, 0xCAB2, 0x8AFB, 0xCC21, 0x8A5A, 0xCD92,
        0x89BE, 0xCF04, 0x8927, 0xD079, 0x8894, 0xD1EF, 0x8805, 0xD367,
        0x877B, 0xD4E1, 0x86F6, 0xD65C, 0x8676, 0xD7D9, 0x85FA, 0xD958,
        0x8583, 0xDAD8, 0x8510, 0xDC59, 0x84A3, 0xDDDC, 0x843A, 0xDF61,
        0x83D6, 0xE0E6, 0x8377, 0xE26D, 0x831C, 0xE3F4, 0x82C6, 0xE57D,
        0x8275, 0xE707, 0x8229, 0xE892, 0x81E2, 0xEA1E, 0x81A0, 0xEBAB,
        0x8163, 0xED38, 0x812A, 0xEEC6, 0x80F6, 0xF055, 0x80C8, 0xF1E4,
        0x809E, 0xF374, 0x8079, 0xF505, 0x8059, 0xF695, 0x803E, 0xF827,
        0x8027, 0xF9B8, 0x8016, 0xFB4A, 0x800A, 0xFCDC, 0x8002, 0xFE6E
        } ;
#endif

#endif
```

A.3.4 inputsignalsquare1khz.c

```
/********************************************************************
* © 2005 Microchip Technology Inc.
*
* FileName:         inputsignalsquare1khz.c
* Dependencies:     Header (.h) files if applicable, see below
* Processor:        dsPIC30Fxxxx
* Compiler:         MPLAB® C30 v3.00 or higher
* IDE:              MPLAB® IDE v7.52 or later
* Dev. Board Used: dsPICDEM 1.1 Development Board
* Hardware Dependencies: None
*
* SOFTWARE LICENSE AGREEMENT:
* Microchip Technology Incorporated ("Microchip") retains all ownership and
* intellectual property rights in the code accompanying this message and in all
* derivatives hereto.  You may use this code, and any derivatives created by
```

```
* any person or entity by or on your behalf, exclusively with Microchip's
* proprietary products.  Your acceptance and/or use of this code constitutes
* agreement to the terms and conditions of this notice.
*
* CODE ACCOMPANYING THIS MESSAGE IS SUPPLIED BY MICROCHIP "AS IS".  NO
* WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED
* TO, IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A
* PARTICULAR PURPOSE APPLY TO THIS CODE, ITS INTERACTION WITH MICROCHIP'S
* PRODUCTS, COMBINATION WITH ANY OTHER PRODUCTS, OR USE IN ANY APPLICATION.
*
* YOU ACKNOWLEDGE AND AGREE THAT, IN NO EVENT, SHALL MICROCHIP BE LIABLE, WHETHER
* IN CONTRACT, WARRANTY, TORT (INCLUDING NEGLIGENCE OR BREACH OF STATUTORY DUTY),
* STRICT LIABILITY, INDEMNITY, CONTRIBUTION, OR OTHERWISE, FOR ANY INDIRECT, SPECIAL,
* PUNITIVE, EXEMPLARY, INCIDENTAL OR CONSEQUENTIAL LOSS, DAMAGE, FOR COST OR EXPENSE OF
* ANY KIND WHATSOEVER RELATED TO THE CODE, HOWSOEVER CAUSED, EVEN IF MICROCHIP HAS BEEN
* ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE.  TO THE FULLEST EXTENT
* ALLOWABLE BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO
* THIS CODE, SHALL NOT EXCEED THE PRICE YOU PAID DIRECTLY TO MICROCHIP SPECIFICALLY TO
* HAVE THIS CODE DEVELOPED.
*
* You agree that you are solely responsible for testing the code and
* determining its suitability.  Microchip has no obligation to modify, test,
* certify, or support the code.
*
* REVISION HISTORY:
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
* Author            Date        Comments on this revision
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
* HV                09/30/05    First release of source file
*
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
*
* ADDITIONAL NOTES:
*
*
*****************************************************************/

#include <dsp.h>
#include "fft.h"

fractcomplex sigCmpx[FFT_BLOCK_LENGTH] __attribute__ ((section (".ydata, data, ymemory"),
aligned (FFT_BLOCK_LENGTH * 2 *2))) =
{
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001, 0x8001, 0x8001, 0x8001, 0x8001,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001, 0x8001, 0x8001, 0x8001, 0x8001,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001, 0x8001, 0x8001, 0x8001, 0x8001,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001, 0x8001, 0x8001, 0x8001, 0x8001,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001, 0x8001, 0x8001, 0x8001, 0x8001,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001, 0x8001, 0x8001, 0x8001, 0x8001,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001, 0x8001, 0x8001, 0x8001, 0x8001,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001, 0x8001, 0x8001, 0x8001, 0x8001,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001, 0x8001, 0x8001, 0x8001, 0x8001,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
```

```
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x8001 , 0x8001 , 0x8001 , 0x8001 , 0x8001 ,
```

```
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x8001,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
```

```
0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 ,
0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 ,
0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 ,
0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 ,
0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 ,
0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 ,
0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 ,
0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 ,
0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 ,
0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 ,
0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 ,
0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 , 0x0000 ,
0x0000 , 0x0000 , 0x0000
};
```

B Appendix - dsPIC30F6011A

B.1 dsPIC30F6011A Block diagram

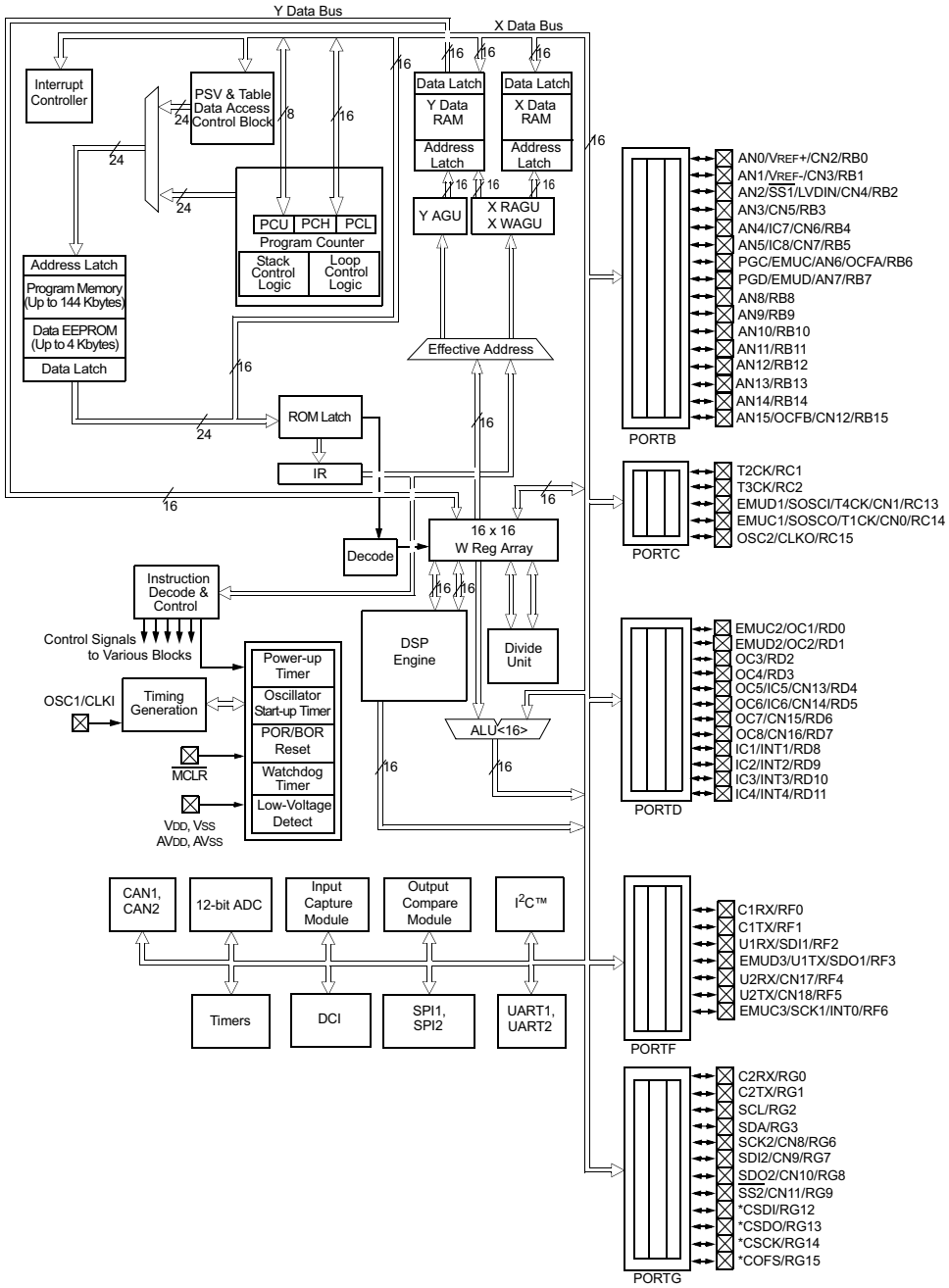


FIGURE B.1 - Block diagram of the dsPIC30F6011A

B.2 DSP engine block diagram

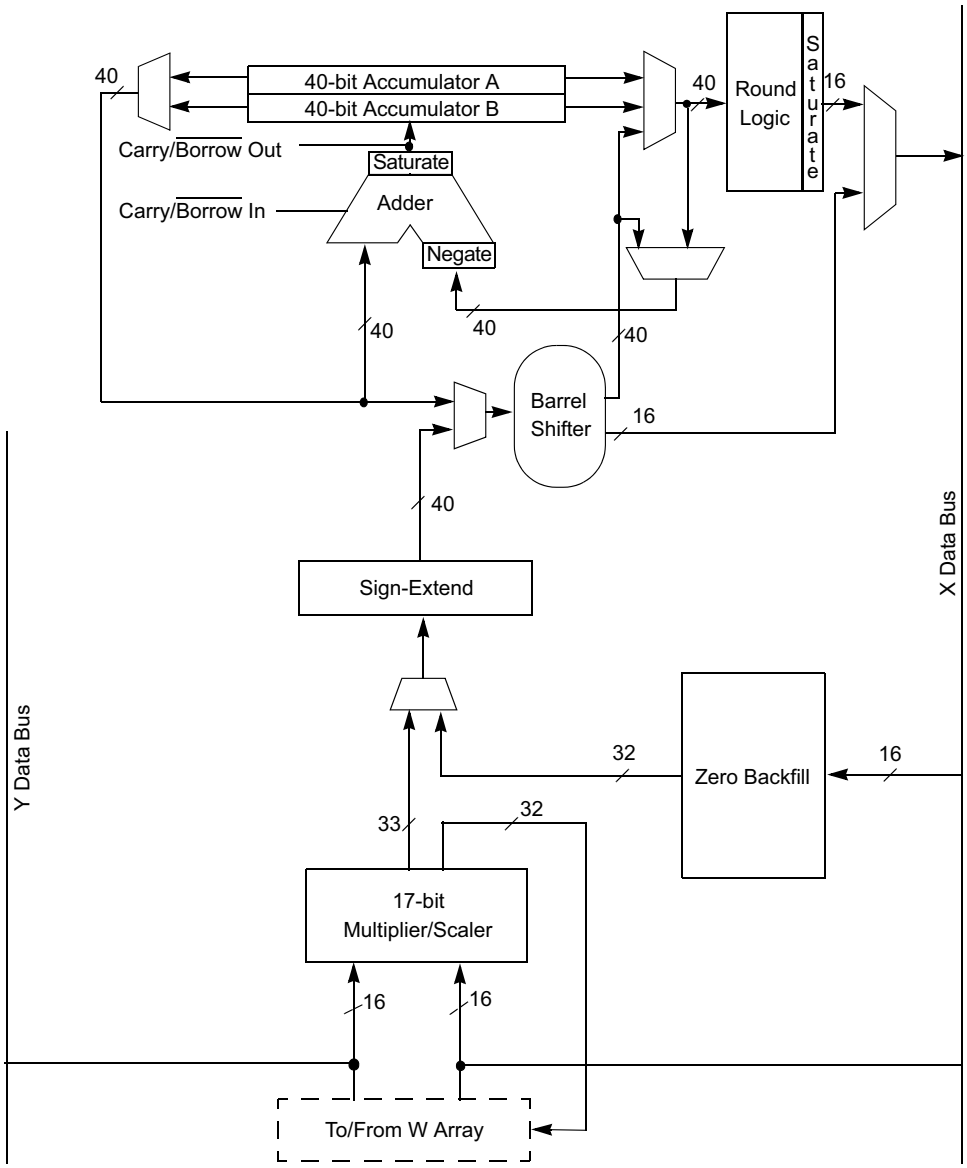


FIGURE B.2 - DSP engine block diagram.

Bibliography

[1] ALTERA. Using plds for high-performance dsp applications. 2005.

[2] Gabriele D Antona, Alessandro Ferrero, and Roberto Ottoboni. A real-time instantaneous frequency estimator for rotatin ii u magnetic islands in a tokamak thermonuclear plasma. *Instrumentation and Measurement, IEEE Transactions on*, 44, 1995.

[3] J.P. Bentley. *Principles of Measurement Systems*. Prentice Hall, 2005.

[4] N. Bom and BW Conoly. Zero-crossing shift as a detection method. *The Journal of the Acoustical Society of America*, 47:1408, 1970.

[5] V. Eckhardt, P. Hippe, and G. Hosemann. Dynamic measuring of frequency and frequency oscillations inmultiphase power systems. *Power Delivery, IEEE Transactions on*, 4(1):95–102, 1989.

[6] ELEQ. `http://www.eleq.com/ENG/Home/index.php`.

[7] Vladimir Friedman. A zero crossing algorithm for the estimation of the frequency of a single sinusoid in white noise. *Instrumentation and Measurement, IEEE Transactions on*, 42, 1994.

[8] MM Giray and MS Sachdev. Off-nominal frequency measurements in electric power systems. *Power Delivery, IEEE Transactions on*, 4(3):1573–1578, 1989.

[9] AA Girgis and TLD Hwang. Optimal estimation of voltage phasors and frequency deviation using linear and non-linear kalman filtering: Theory and limitations. *IEEE Transactions on Power Apparatus and Systems*, pages 2943–2951, 1984.

[10] AA Girgis and WL Peterson. Adaptive estimation of power system frequency deviation and itsrate of change for calculating sudden power system overloads. *Power Delivery, IEEE Transactions on*, 5(2):585–594, 1990.

[11] VK Govindan, M.V. Vaidyan, and C. Velayudhan. A novel method for digital monitoring of frequency deviation in power systems. *International Journal of Electronics*, 56(1):127–134, 1984.

[12] V. HAMILAKIS and NC VOULGARIS. An accurate method for the measurement of line frequency and its deviation using a microprocessor. *IEEE transactions on instrumentation and measurement*, 36(1):104–109, 1987.

[13] M.T. Inc. dspic30f6011a. Technical report, Data Sheet. `www.microchip.com`.

[14] MI Irshid, WA Shahab, and BR El-Asir. A simple programmable frequency meter for low frequencies withknown nominal values. *Instrumentation and Measurement, IEEE Transactions on*, 40(3):640–642, 1991.

[15] I. Kamwa and R. Grondin. Fast adaptive schemes for tracking voltage phasor and localfrequency in power transmission and distribution systems. In *Transmission and Distribution Conference, 1991., Proceedings of the 1991 IEEE Power Engineering Society*, pages 930–936, 1991.

[16] B. Kedem. Spectral analysis and discrimination by zero-crossings. *Proceedings of the IEEE*, 74(11):1477–1493, 1986.

[17] M. Kezunovic, P. Spasojevic, and B. Perunicic. New digital signal processing algorithms for frequency deviation measurement. *Power Delivery, IEEE Transactions on*, 7(3):1563–1573, 1992.

[18] R. Kumaresan and D. Tufts. Estimating the parameters of exponentially damped sinusoids and pole-zero modeling in noise. *Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing], IEEE Transactions on*, 30(6):833–840, 1982.

[19] Miodrag D. Kusljevic. A simple recursive algorithm for frequency estimation. *Instrumentation and Measurement, IEEE Transactions on*, 53, 2004.

[20] MP Mathur. A digital frequency meter for measuring low frequencies. *IEEE Trans. Instrum. Meas*, 27:295–296, 1978.

[21] EP McCarthy. A digital instantaneous frequency meter. *IEEE Transactions on Instrumentation and Measurement*, 28:224–226, 1979.

[22] Razak Mohammedali. Fpgas provide reconfigurable dsp solutions. 2005.

[23] PJ Moore, RD Carranza, and AT Johns. A new numeric technique for high-speed evaluation of power system frequency. In *Generation, Transmission and Distribution, IEE Proceedings-*, volume 141, pages 529–536, 1994.

[24] AG Phadke, JS Thorp, and MG Adamiak. A new measurement technique for tracking voltage phasors, local system frequency, and rate of change of frequency. *IEEE Transactions on Power Apparatus and Systems*, pages 1025–1038, 1983.

[25] M. Prokin. Dma transfer method for wide-range speed and frequency measurement. *Instrumentation and Measurement, IEEE Transactions on*, 42(4):842–846, 1993.

[26] D. Rife and R. Boorstyn. Single tone parameter estimation from discrete-time observations. *Information Theory, IEEE Transactions on*, 20(5):591–598, 1974.

[27] MS Sachdev and MM Giray. A least error squares technique for determining power system frequency. *IEEE Transactions on Power Apparatus and Systems*, pages 437–444, 1985.

[28] MS Sachdev and M. Nagpal. A recursive least error squares algorithm for power system relaying and measurement applications. *Power Delivery, IEEE Transactions on*, 6(3):1008–1015, 1991.

[29] R. Scarr. Zero crossings as a means of obtaining spectral information in speech analysis. *Audio and Electroacoustics, IEEE Transactions on*, 16(2):247–255, 1968.

[30] TS Sidhu. Accurate measurement of power system frequency using a digitalsignal processing technique. *Instrumentation and Measurement, IEEE Transactions on*, 48(1):75–81, 1999.

[31] Priyabrata Sinha. dspic30f addressing modes. 2002.

[32] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing San Diego, CA, USA, 1997.

[33] SA SOLIMAN, GS CHRISTENSEN, DH KELLY, and N. LIU. An algorithm for frequency relaying based on least absolute value approximations. *Electric power systems research*, 19(2):73–84, 1990.

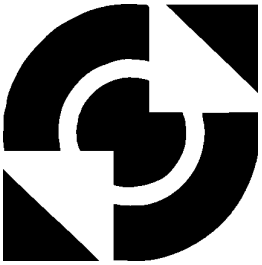
[34] Samir S. Soliman and Mandyam D. Srinath. *Continuous and Discrete Signals and Systems*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1990.

[35] H. TAO and IF MORRISON. The measurement of power system frequency using a microprocessor. *Electric power systems research*, 11(3):103–108, 1986.

[36] VV Terzija, MB Djuric, and BD Kovacevic. Voltage phasor and local system frequency estimation using newtontype algorithm. *Power Delivery, IEEE Transactions on*, 9(3):1368–1374, 1994.

[37] L. Van Den Steen. A low frequency meter with instantaneous response. *Biomedical Engineering, IEEE Transactions on*, pages 137–140, 1979.

University of Twente



EEMCS / Electrical Engineering
Control Engineering

Universidad de Navarra
Nafarroako Unibertsitatea
Escuela Superior de Ingenieros
Ingeniarien Goi Mailako Eskola

Feasibility study of implementing a new algorithm to measure the frequency in a universal power measuring device.

Victor Gutiérrez Perez

MSc report

Supervisors
Prof.dr.ir. P.P. Regtien
Ing. .M. Klomp
Dr. L. Fontán

January 2009

Report nr. 002CE2009

Control Engineering
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands