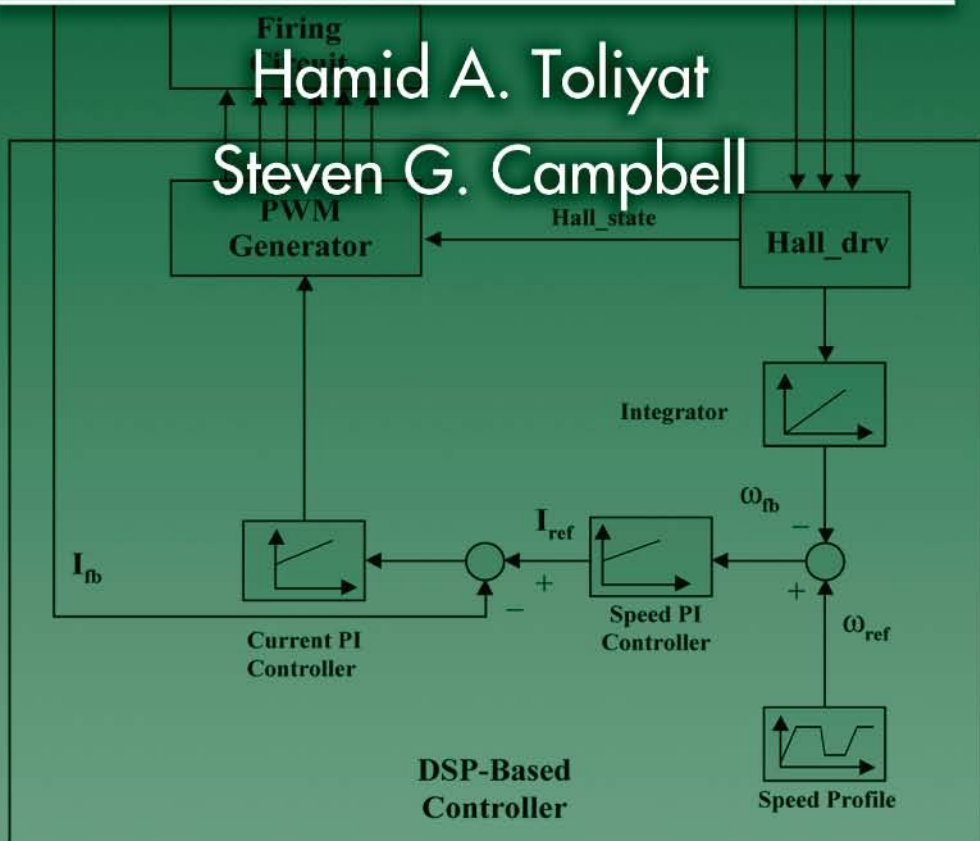


# DSP-BASED ELECTROMECHANICAL MOTION CONTROL

Hamid A. Toliyat

Steven G. Campbell



CRC PRESS

# DSP-BASED ELECTROMECHANICAL MOTION CONTROL

# POWER ELECTRONICS AND APPLICATIONS SERIES

Muhammad H. Rashid, Series Editor  
*University of West Florida*

## **PUBLISHED TITLES**

### **Complex Behavior of Switching Power Converters**

Chi Kong Tse

### **DSP-Based Electromechanical Motion Control**

Hamid A. Toliyat and Steven Campbell

### **Advanced DC/DC Converters**

Fang Lin Luo and Hong Ye

## **FORTHCOMING TITLES**

### **Renewable Energy Systems: Design and Analysis with Induction Generators**

Marcelo Godoy Simoes and Felix Alberto Farret

# DSP-BASED ELECTROMECHANICAL MOTION CONTROL

Hamid A. Toliyat  
Steven Campbell

Texas A&M University  
Department of Electrical Engineering  
College Station, Texas



**CRC PRESS**

---

Boca Raton London New York Washington, D.C.

## Library of Congress Cataloging-in-Publication Data

---

Toliyat, Hamid A.

DSP-Based electromechanical motion control / by Hamid A. Toliyat and Steven Campbell.

p. cm.-- (Power electronics and applications series)

Includes bibliographical references and index.

ISBN 0-8493-1918-8 (alk. paper)

1. Digital control systems. 2. Electromechanical devices. 3. Signal processing--Digital techniques. I. Campbell, Steven (Steven Gerard), 1979- II. Title. III. Series.

TJ223.M53.T64 2003

629.8—dc22

2003058462

This book contains information obtained from authentic and highly regarded sources. Reprinted material is quoted with permission, and sources are indicated. A wide variety of references are listed. Reasonable efforts have been made to publish reliable data and information, but the author and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

Neither this book nor any part may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, microfilming, and recording, or by any information storage or retrieval system, without prior permission in writing from the publisher.

The consent of CRC Press LLC does not extend to copying for general distribution, for promotion, for creating new works, or for resale. Specific permission must be obtained in writing from CRC Press LLC for such copying.

Direct all inquiries to CRC Press LLC, 2000 N.W. Corporate Blvd., Boca Raton, Florida 33431.

**Trademark Notice:** Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation, without intent to infringe.

**Visit the CRC Press Web site at [www.crcpress.com](http://www.crcpress.com)**

---

© 2004 by CRC Press LLC

No claim to original U.S. Government works

International Standard Book Number 0-8493-1918-8

Library of Congress Card Number 2003058462

Printed in the United States of America 1 2 3 4 5 6 7 8 9 0

Printed on acid-free paper

*To my wife Mina, and my sons Amir and Mohammad for their love  
and patience while this book was being prepared.*

*To my parents for their continuous support and encouragement.*

*- H.T.*



## PREFACE

This book was written to provide a general application guide for students and engineers of all disciplines who want to begin utilizing a Digital Signal Processor (DSP) for the task of electromechanical motion control. While the act of learning to program and use the DSP itself is not overly difficult, utilizing the DSP in applications such as motor control can sometimes seem challenging to the first-time user.

Full mastery of all the topics and concepts presented in this text would take years of study and knowledge from many areas of engineering and science. For this reason, we will attempt to survey each topic, giving readers a basic understanding of each topic without going into great depth. We will thus leave it to the reader for in-depth study of particular topics of interest.

So why would we choose to integrate a DSP into a motion control system? Well, the advantages of such a design are numerous. DSP-based control gives us a large degree of freedom in developing computationally extensive algorithms that would otherwise be very difficult or impossible without a DSP. Advanced control algorithms can sometimes drastically increase the performance and efficiency of the electromechanical system being controlled.

For example, consider a typical Heating-Ventilation-and-Air-Conditioning (HVAC) system. A standard HVAC system contains at least three electric motors: compressor motor, condenser fan motor, and the air handler fan motor. Typically, indoor temperature is controlled by simply cycling (turning on and off) the system. This control method puts unnecessary wear on system components and is inefficient. An advanced motor drive system incorporating DSP control could continuously adjust both the air-conditioner compressor speed and indoor fan to maintain the desired temperature and optimal system performance. This control scheme would be much more energy efficient and could extend the operational lifespan of the system.

We will start by visiting the LF2407 DSP processor. Device functionality, integrated components, memory, and assembly programming will be covered. Several laboratory exercises will help the reader practice the information presented in each chapter. After several chapters are presented on the DSP, more advanced topics are presented involving several real-world applications in the area of motion control and motor drives.





## ACKNOWLEDGMENTS

As most readers can imagine, creating a book is no trivial task. Besides the authors listed on the cover of each book, there are usually many others who give their time and knowledge. These contributions range from the writing of a chapter to simply proofreading the book for mistakes. This book is no exception. There are many people I would like to thank who made invaluable contributions to the creation of this book.

During the past two years that this book was in development, the many undergraduate students who took my “DSP-Based Electromechanical Motion Control Devices” course in the Department of Electrical Engineering at Texas A&M University provided invaluable feedback on the material. I am in debt to all of them.

I would also like to extend my gratitude to Texas Instruments for permitting me to use the materials in its manuals. I would also like to extend a special acknowledgment to Gene Frantz and Christina Peterson from Texas Instruments, whose help and support for materializing this book were fundamental.

Several individuals, including my past and present graduate students, have contributed to this book. They are as follows: Sebastien Gay, Dr. Masoud Hajiaghajni - Chapter 7; Dr. Lei Hao and Leila Parsa - Chapters 8, 9, and 12; Mehdi Abolhassani - Chapter 10; Nasser Qahtani - Chapter 11; Peyman Niazi - Chapter 13; Sang-shin Kwak - Chapter 15; and Baris Ozturk the Appendix.

Dr. Babak Fahimi of University of Missouri-Rolla wrote Chapter 14 and Dr. Syed Madani of the University of Puerto Rico-Mayaguez contributed to Chapter 7. Rebecca Morrison proofread several chapters.

I would also like to thank Nora Konopka, Helena Redshaw, Susan Fox of CRC Press for their patience and support while this book was being prepared.

Hamid A. Toliyat  
College Station, TX



## TABLE OF CONTENTS

Chapter 1	Introduction to the TMSLF2407 DSP Controller .....	1
1.1	Introduction.....	1
1.2	Brief Introduction to Peripherals.....	3
1.3	Types of Physical Memory .....	5
1.4	Software Tools.....	6
Chapter 2	C2xx DSP CPU and Instruction Set .....	19
2.1	Introduction to the C2xx DSP Core and Code Generation .....	19
2.2	The Components of the C2xx DSP Core .....	19
2.3	Mapping External Devices to the C2xx Core and the Peripheral Interface .....	21
2.4	System Configuration Registers.....	22
2.5	Memory.....	26
2.6	Memory Addressing Modes.....	31
2.7	Assembly Programming Using the C2xx DSP Instruction Set .....	36
Chapter 3	General Purpose Input/Output (GPIO) Functionality .....	49
3.1	Pin Multiplexing (MUX) and General Purpose I/O Overview .....	49
3.2	Multiplexing and General Purpose I/O Control Registers .....	50
3.3	Using the General Purpose I/O Ports .....	57
3.4	General Purpose I/O Exercise .....	58
Chapter 4	Interrupts on the TMS320LF2407 .....	61
4.1	Introduction to Interrupts .....	61
4.2	Interrupt Hierarchy .....	61
4.3	Interrupt Control Registers .....	64
4.4	Initializing and Servicing Interrupts in Software .....	70
4.5	Interrupt Usage Exercise.....	75
Chapter 5	The Analog-to-Digital Converter (ADC) .....	77
5.1	ADC Overview .....	77
5.2	Operation of the ADC .....	78
5.3	Analog to Digital Converter Usage Exercise .....	98
Chapter 6	The Event Managers (EVA, EVB).....	101
6.1	Overview of the Event Manager (EV) .....	101
6.2	Event Manager Interrupts .....	102
6.3	General Purpose (GP) Timers .....	115
6.4	Compare Units .....	134
6.5	Capture Units and Quadrature Encoded Pulse (QEP) Circuitry.....	147
6.6	General Event Manager Information .....	158
6.7	Exercise: PWM Signal Generation .....	161

Chapter 7	DSP-Based Implementation of DC-DC Buck-Boost Converters .....	163
7.1	Introduction.....	163
7.1	Converter Structure.....	163
7.2	Continuous Conduction Mode .....	164
7.3	Discontinuous Conduction Mode.....	165
7.4	Connecting the DSP to the Buck-Boost Converter .....	165
7.5	Controlling the Buck-Boost Converter .....	168
7.6	Main Assembly Section Code Description .....	171
7.7	Interrupt Service Routine.....	173
7.8	The Regulation Code Sequences.....	175
7.9	Results.....	179
Chapter 8	DSP-Based Control of Stepper Motors .....	183
8.1	Introduction.....	183
8.2	The Principle of Hybrid Stepper Motor .....	183
8.3	The Basic Operation .....	184
8.4	The Stepper Motor Drive System .....	188
8.5	The Implementation of Stepper Motor Control System Using the LF2407 DSP.....	190
8.6	The Subroutine of Speed Control Module .....	191
	Reference .....	192
Chapter 9	DSP-Based Control of Permanent Magnet Brushless DC Machines... 193	
9.1	Introduction.....	193
9.2	Principles of the BLDC Motor.....	195
9.3	Torque Generation .....	195
9.4	BLDC Motor Control System.....	196
9.5	Implementation of the BLDC Motor Control System Using the LF2407.....	200
Chapter 10	Clarke's and Park's Transformations .....	209
10.1	Introduction.....	209
10.2	Clarke's Transformation .....	209
10.3	Park's Transformation .....	210
10.4	Transformations Between Reference Frames .....	212
10.5	Field Oriented Control (FOC) Transformations.....	213
10.6	Implementing Clarke's and Park's Transformations on the LF240X.....	214
10.7	Conclusion .....	222
	References.....	222
Chapter 11	Space Vector Pulse Width Modulation .....	223
11.1	Introduction.....	223
11.2	Principle of Constant V/Hz Control for Induction Motors.....	223
11.3	Space Vector PWM Technique.....	224
11.4	DSP Implementation.....	232

References.....	240
Chapter 12 DSP-Based Control of Permanent Magnet Synchronous Machines..	241
12.1 Introduction.....	241
12.2 The Principle of the PMSM .....	241
12.3 PMSM Control System .....	244
12.4 Implementation of the PMSM System Using the LF2407 .....	248
Chapter 13 DSP-Based Vector Control of Induction Motors.....	255
13.1 Introduction.....	255
13.2 Three-Phase Induction Motor Basic Theory .....	255
13.3 Model of the Three-Phase Induction Motor in Simulink .....	257
13.4 Reference Frame Theory.....	259
13.5 Induction Motor Model in the Arbitrary q-d-0 Reference Frame .....	260
13.6 Field Oriented Control .....	261
13.7 DC Machine Torque Control .....	262
13.8 Field Oriented Control, Direct and Indirect Approaches .....	262
13.9 Simulation Results for the Induction Motor Control System.....	266
13.10 Induction Motor Speed Control System.....	266
13.11 System Components .....	268
13.12 Implementation of Field-Oriented Speed Control of Induction Motor.....	270
13.13 Experimental Results .....	287
13.14 Conclusion .....	288
References.....	288
Chapter 14 DSP-Based Control OF Switched Reluctance Motor Drives .....	289
14.1 Introduction.....	289
14.2 Fundamentals of Operation.....	290
14.3 Fundamentals of Control in SRM Drives.....	292
14.4 Open Loop Control Strategy for Torque.....	293
14.5 Closed Loop Torque Control of the SRM Drive.....	301
14.6 Closed Loop Speed Control of the SRM Drive .....	304
14.7 Summary .....	305
14.8 Algorithm for Running SRM Drive using an Optical Encoder.....	305
Chapter 15 DSP-Based Control of Matrix Converters.....	307
15.1 Introduction.....	307
15.2 Topology and Characteristics.....	308
15.3 Control Algorithms .....	309
15.4 Space Vector Modulation .....	314
15.5 Bidirectional Switch.....	319
15.6 Current Commutation .....	320
15.7 Overall Structure of Three-Phase Matrix Converter .....	321
15.8 Implementation of the Venturini Algorithm using the LF2407 ....	322
References.....	325

Appendix A	Development of Field-Oriented Control Induction Motor Using VisSim™ .....	327
A.1	Introduction.....	327
A.2	Overview of VisSim™ Placing and Wiring Blocks.....	327
A.3	Computer Simulation of Vector Control of Three-Phase Induction Motor Using VisSim™ .....	329
A.4	Summary and Improvements .....	341
	References.....	342
Index	.....	343

## Chapter 1

### INTRODUCTION TO THE TMSLF2407 DSP CONTROLLER

#### 1.1 Introduction

The Texas Instruments TMS320LF2407 DSP Controller (referred to as the LF2407 in this text) is a programmable digital controller with a C2xx DSP central processing unit (CPU) as the core processor. The LF2407 contains the DSP core processor and useful peripherals integrated onto a single piece of silicon. The LF2407 combines the powerful CPU with on-chip memory and peripherals. With the DSP core and control-oriented peripherals integrated into a single chip, users can design very compact and cost-effective digital control systems.

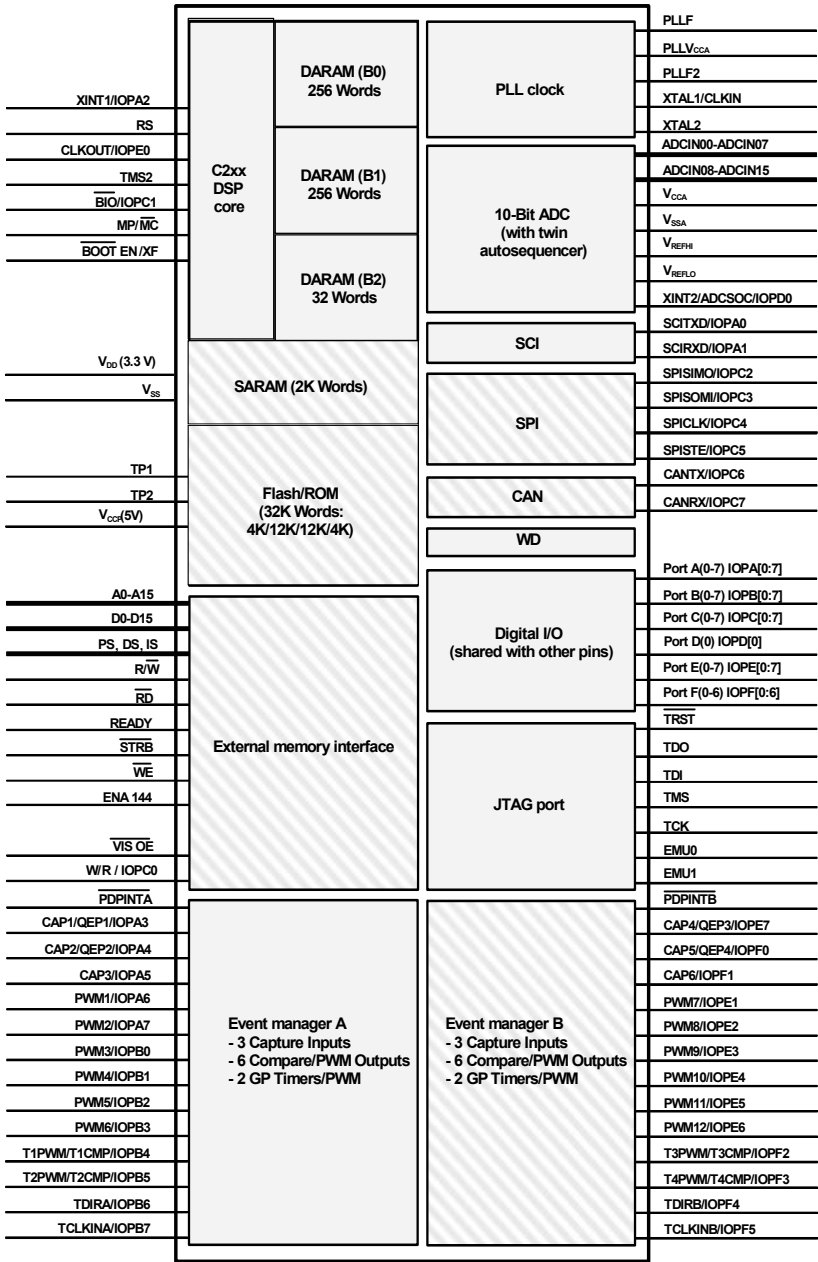
The LF2407 DSP controller offers 40 million instructions per second (MIPS) performance. This high processing speed of the C2xx CPU allows users to compute parameters in real time rather than look up approximations from tables stored in memory. This fast performance is well suited for processing control parameters in applications such as notch filters or sensorless motor control algorithms where a large amount of calculations must be computed quickly.

While the “brain” of the LF2407 DSP is the C2xx core, the LF2407 contains several control-orientated peripherals onboard (see Fig. 1.1). The peripherals on the LF2407 make virtually any digital control requirement possible. Their applications range from analog to digital conversion to pulse width modulation (PWM) generation. Communication peripherals make possible the communication with external peripherals, personal computers, or other DSP processors. Below is a brief listing of the different peripherals onboard the LF2407 followed by a graphical layout depicted in Fig. 1.1.

The LF2407 peripheral set includes:

- Two Event Managers (A and B)
- General Purpose (GP) timers
- PWM generators for digital motor control
- Analog-to-digital converter
- Controller Area Network (CAN) interface
- Serial Peripheral Interface (SPI) – synchronous serial port
- Serial Communications Interface (SCI) – asynchronous serial port
- General-Purpose bi-directional digital I/O (GPIO) pins
- Watchdog Timer (“time-out” DSP reset device for system integrity)






 Indicates optional modules in the 240x family. The memory size and peripheral selection of these modules change for different 240xA devices

Figure 1.1                      Graphical overview of DSP core and peripherals on the LF2407.  
(Courtesy of Texas Instruments)

## 1.2 Brief Introduction to Peripherals

The following peripherals are those that are integrated onto the LF2407 chip. Refer to Fig. 1.1 to view the pin-out associated with each peripheral.

### Event Managers (EVA, EVB)

There are two Event Managers on the LF2407, the EVA and EVB. The Event Manager is the most important peripheral in digital motor control. It contains the necessary functions needed to control electromechanical devices. Each EV is composed of functional “blocks” including timers, comparators, capture units for triggering on an event, PWM logic circuits, quadrature-encoder-pulse (QEP) circuits, and interrupt logic.

### The Analog-to-Digital Converter (ADC)

The ADC on the LF2407 is used whenever an external analog signal needs to be sampled and converted to a digital number. Examples of ADC applications range from sampling a control signal for use in a digital notch filtering algorithm or using the ADC in a control feedback loop to monitor motor performance. Additionally, the ADC is useful in motor control applications because it allows for current sensing using a shunt resistor instead of an expensive current sensor.

### The Control Area Network (CAN) Module

While the CAN module will not be covered in this text, it is a useful peripheral for specific applications of the LF2407. The CAN module is used for multi-master serial communication between external hardware. The CAN bus has a high level of data integrity and is ideal for operation in noisy environments such as in an automobile, or industrial environments that require reliable communication and data integrity.

### Serial Peripheral Interface (SPI) and Serial Communications Interface (SCI)

The SPI is a high-speed synchronous communication port that is mainly used for communicating between the DSP and external peripherals or another DSP device. Typical uses of the SPI include communication with external shift registers, display drivers, or ADCs.

The SCI is an asynchronous communication port that supports asynchronous serial (UART) digital communication between the CPU and other asynchronous peripherals that use the standard NRZ (non-return-to-zero) format. It is useful in communication between external devices and the DSP. Since these communication peripherals are not directly related to motion control applications, they will not be discussed further in this text.

## Watchdog Timer (WD)

The Watchdog timer (WD) peripheral monitors software and hardware operations and asserts a system reset when its internal counter overflows. The WD timer (when enabled) will count for a specific amount of time. It is necessary for the user's software to reset the WD timer periodically so that an unwanted reset does not occur. If for some reason there is a CPU disruption, the watchdog will generate a system reset. For example, if the software enters an endless loop or if the CPU becomes temporarily disrupted, the WD timer will overflow and a DSP reset will occur, which will cause the DSP program to branch to its initial starting point. Most error conditions that temporarily disrupt chip operation and inhibit proper CPU function can be cleared by the WD function. In this way, the WD increases the reliability of the CPU, thus ensuring system integrity.

## General Purpose Bi-Directional Digital I/O (GPIO) Pins

Since there are only a finite number of pins available on the LF2407 device, many of the pins are multiplexed to either their primary function or the secondary GPIO function. In most cases, a pin's second function will be as a general-purpose input/output pin. The GPIO capability of the LF2407 is very useful as a means of controlling the functionality of pins and also provides another method to input or output data to and from the device. Nine 16-bit control registers control all I/O and shared pins. There are two types of these registers:

- I/O MUX Control Registers (MCRx) – Used to control the multiplexer selection that chooses between the primary function of a pin or the general-purpose I/O function.
- Data and Direction Control Registers (PxDATDIR) – Used to control the data and data direction of bi-directional I/O pins.

## Joint Test Action Group (JTAG) Port

The JTAG port provides a standard method of interfacing a personal computer with the DSP controller for emulation and development. The XDS510PP or equivalent emulator pod provides the connection between the JTAG module on the LF2407 and the personal computer. The JTAG module allows the PC to take full control over the DSP processor while Code Composer Studio™ is running. Figure 1.2 shows the connection scheme from computer to the DSP board.

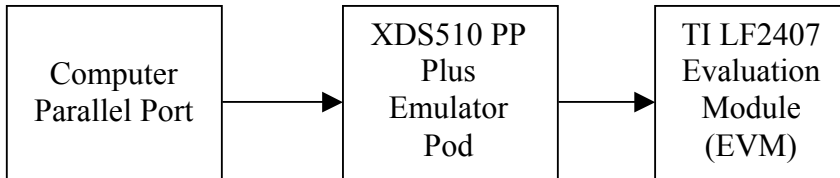


Figure 1.2 PC to DSP connection scheme.

## Phase Locked Loop (PLL) Clock Module

The phase locked loop (PLL) module is basically an input clock multiplier that allows the user to control the input clocking frequency to the DSP core. External to the LF2407, a clock reference (can oscillator/crystal) is generated. This signal is fed into the LF2407 and is multiplied or divided by the PLL. This new (higher or lower frequency) clock signal is then used to clock the DSP core. The LF2407's PLL allows the user to select a multiplication factor ranging from 0.5X to 4X that of the external clock signal. The default value of the PLL is 4X.

## Memory Allocation Spaces

The LF2407 DSP Controller has three different allocations of memory it can use: Data, Program, and I/O memory space. Data space is used for program calculations, look-up tables, and any other memory used by an algorithm. Data memory can be in the form of the on-chip random access memory (RAM) or external RAM. Program memory is the location of user's program code. Program memory on the LF2407 is either mapped to the off-chip RAM (MP/MC- pin =1) or to the on-chip flash memory (MP/MC- = 0), depending on the logic value of the MP/MC-pin.

I/O space is not really memory but a virtual memory address used to output data to peripherals external to the LF2407. For example, the digital-to-analog converter (DAC) on the Spectrum Digital™ evaluation module is accessed with I/O memory. If one desires to output data to the DAC, the data is simply sent to the configured address of I/O space with the "OUT" command. This process is similar to writing to data memory except that the OUT command is used and the data is copied to and outputted on the DAC instead of being stored in memory.

### 1.3 Types of Physical Memory

#### Random Access Memory (RAM)

The LF2407 has 544 words of 16 bits each in the on-chip DARAM. These 544 words are partitioned into three blocks: B0, B1, and B2. Blocks B1 and B2 are allocated for use only as data memory. Memory block B0 is different than B1 and B2. This memory block is normally configured as Data Memory, and hence primarily used to hold data, but in the case of the B0 block, it can also be configured as Program Memory. B0 memory can be configured as program or data memory depending on the value of the core level "CNF" bit.

- (CNF=0) maps B0 to data memory.
- (CNF=1) maps B0 to program memory.

The LF2407 also has 2K of single-access RAM (SARAM). The addresses associated with the SARAM can be used for both data memory and program memory, and are software configurable to the internal SARAM or external memory.

## Non-Volatile Flash Memory

The LF2407 contains 32K of on-chip flash memory that can be mapped to program space if the MP/MC-pin is made logic 0 (tied to ground). The flash memory provides a permanent location to store code that is unaffected by cutting power to the device. The flash memory can be electronically programmed and erased many times to allow for code development. Usually, the external RAM on the LF2407 Evaluation Module (EVM) board is used instead of the flash for code development due to the fact that a separate “flash programming” routine must be performed to flash code into the flash memory. The on-chip flash is normally used in situations where the DSP program needs to be tested where a JTAG connection is not practical or where the DSP needs to be tested as a “stand-alone” device. For example, if a LF2407 was used to develop a DSP control solution to an automobile braking system, it would be somewhat impractical to have a DSP/JTAG/PC interface in a car that is undergoing performance testing.

## 1.4 Software Tools

Texas Instrument’s Code Composer Studio™ (CCS) is a user-friendly Windows-based debugger for developing and debugging software for the LF2407. CCS allows users to write and debug code in C or in TI assembly language. CCS has many features that can aid in developing code. CCS features include:

- User-friendly Windows environment
- Ability to use code written in C and assembly
- Memory displays and on-the-fly editing capability
- Disassembly window for debugging
- Source level debugging, which allows stepping through and setting breakpoints in original source code
- CPU register visibility and modification
- Real-time debugging with watch windows and continuous refresh
- Various single step/step over/ step-into command icons
- Ability to display data in graph formats
- General Extension Language (GEL) capability, allows the user to create functions that extend the usefulness of CCS™

### 1.4.1 *Becoming Aquatinted with Code Composer Studio (CCS)*

This exercise will help you become familiar with the software and emulation tools of the LF2407 DSP Controller. CCS™, the current emulation and debugging software, is user-friendly and a powerful development tool.

The hardware required for this exercise and all others is the Spectrum Digital TMS320LF2407 EVM package, which includes LF2407 EVM board and the XDS510PP Plus JTAG emulator pole. You will also need a Windows-based

personal computer with a parallel printer port. In this lab exercise you will learn how to:

- Open a program, build it, and load the program onto the DSP.
- View the disassembly
- View and edit memory locations
- View and edit CPU registers
- Open a Watch Window
- Reset the DSP
- Run the program in Real-time Mode
- Set breakpoints
- Single step through code
- Save and load a workspace

Since some readers may not have connected their EVM to their PC, we will start with the necessary PC to EVM connection and setup. Follow this procedure if you are first connecting the LF2407 EVM to your PC.

- First, if you have not done so, configure the parallel port of your PC and connect the emulator and target board according to the documentation that came with the LF2407 EVM.
- Before you can start using CCS<sup>TM</sup>, CCS needs to be configured for the particular DSP emulator you are going to be using.

Run *CC\_setup.exe*, which should be an icon under *Start/Programs/Code Composer* or at *C:\tic2xx\cc\bin\cc\_setup.exe*. You should see a window appear similar to that shown in Fig. 1.3.

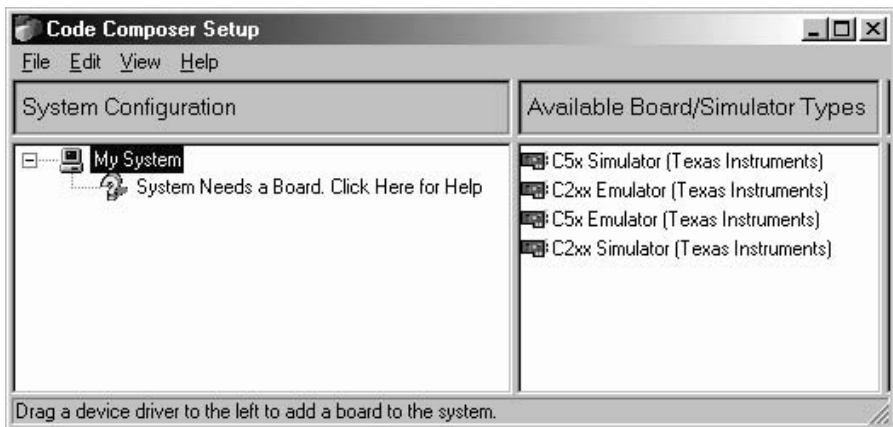


Figure 1.3 Code Composer setup window (from running Setup.exe).

Once you have entered Code Composer Setup window, the proper board/simulator needs to be added to the “System Configuration”.

- a. Drag the appropriate icon from the “Available Board/Simulator Types” list to the “System Configuration” list. To use the LF2407 DSP select, use the sdgo2xx icon as shown in Fig. 1.4.

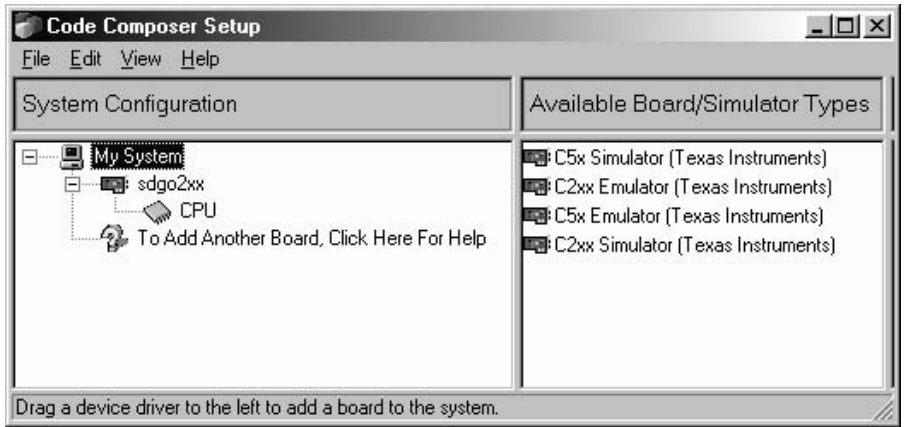


Figure 1.4 Simulator types.

- b. Once you drag the sdgo2xx icon into the “System Configuration” section, a “Board Properties” box (shown in Fig. 1.5) should appear. Click on the “Board Properties” tab and set the I/O port for 378.

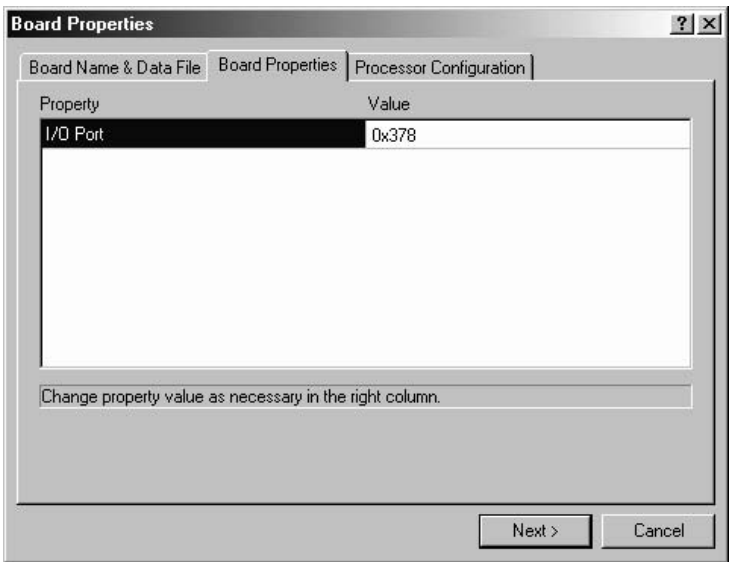


Figure 1.5 Port setting for Printer/Parallel Port.

- c. Click on the “Processor Configuration” tab and select the TMS320C2400 processor. Click on the “Add Single” button. You should see “CPU\_1” under the “processors on the board” list.
- d. Click on the “Finish” button located at the lower right corner of the “Board Properties” box. The setup is now complete. Go to *File/Save* to save the configuration. Close the Code Composer Setup window.

Now that everything is connected properly, we shall begin with the CCS exercise:

1. Turn on the EVM. The green LED on the top right of the board will confirm that there is power to the board.
2. Open Code Composer Studio by running *cc\_app.exe* either from the desktop icon, *Start/Programs/Code Composer*, or *C:\tic2xx\cc\bin\cc\_app.exe*.
3. Go to the “Project” menu, select “Open” as seen in Fig. 1.6. Open *realtime.mak*, which is found under *C:\tic2xx\c2000\tutorial\realtime*. The project file is the master file that “holds” the other files together to build a working program.



Figure 1.6 Project open window.

4. Once you have the project opened, look at the frame on the left side of the screen where “Files”, “GEL files”, and “Project” are listed. Expand everything in the “Project” folder. When you are done, you should see the “Include” files, “Libraries”, and “Source” files as shown in Fig. 1.7. The project file (\*.mak) is the master file that links the other necessary files together as a common filename. When you want to create a program with Code Composer, you will want to first create a new project, add a new source file(s) (\*.asm or \*.c) to the project, add the linker command file (\*.cmd), and add “include” (\*.i) or “header” (\*.h) files.



As in other programming languages, “include” (\*.i) and “header” (\*.h) files are user-defined files that are common to most programs. Functionally \*.h and \*.i files are the same. Both types of files can define constants, macros (user defined callable functions), or variables. In this case, we want to run our program in real-time mode. Therefore, we need a real-time monitor program (*C200mnt.r* in this program). The file *X24x.h* contains variable names for data memory mapped control registers. The code that is in the header (\*.h) or include (\*.i) file could be written in the actual source code, but it is easier to just make general register definitions as a header file that can be used with many projects.

The linker command file (\*.cmd) is vital to the proper building of your code. It specifies where in the program memory to place sections of the program code, defines memory blocks, contains linker options, and names input files for the linker, names the (.out) etc. The linker command file also specifies memory allocations. Without a proper linker command file, CCS will not build the program properly. In this case, the linker command file is named *realtime.cmd*.

Source (\*.c or \*.asm) files contain the actual program that is to be run on the DSP. You must have at least one source file, but may have source files that call other source files. Be sure *all* relevant source files are added to the project.

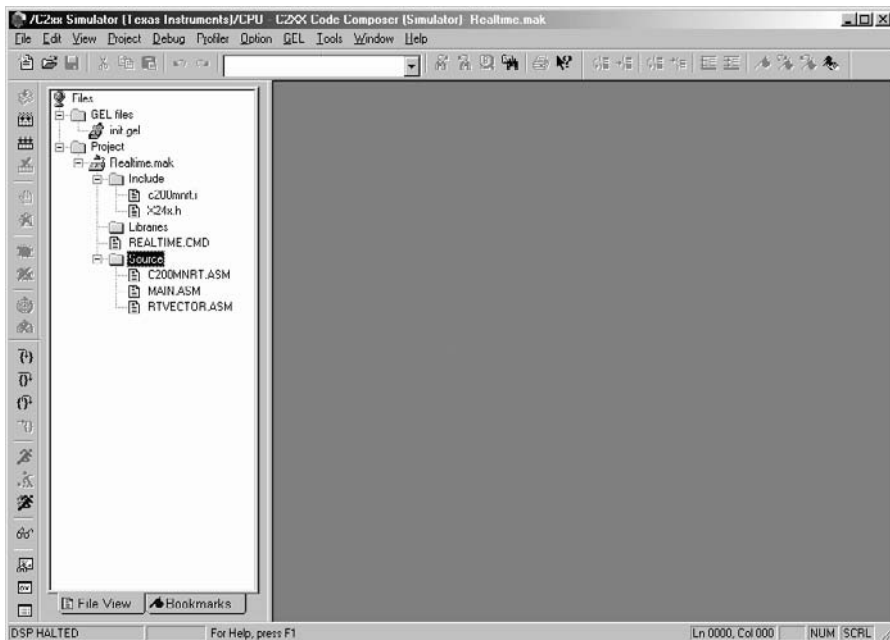


Figure 1.7 CCS window with opened project.

- Now that you have the project opened, go to *Option/Program Load*, and check the “load program after build” box (Fig. 1.8). This will automatically load the DSP compatible version of the program (\*.out) file into the DSP after the build is complete. Building the project causes Code Composer to assemble and link your code. Basically, this creates a file that the DSP can be loaded with and run. Loading the program can also be done manually under the “File” menu.

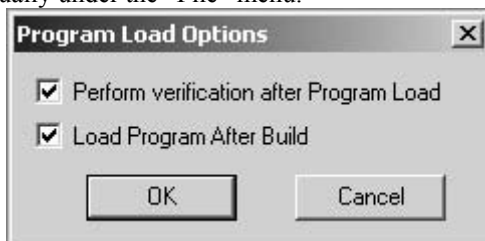


Figure 1.8 Program load options box.

- Now go to *Project/ Rebuild All*. This will build and load the program into the DSP. If the program is being loaded onto the DSP, the disassembly window will open up automatically.

**Note:** It is good practice to ALWAYS RESET THE DSP each time you build or rebuild the project. Do this by going to “Debug” menu, then “Reset DSP”.

To view the disassembly window as in Fig. 1.9 if it is not already open, go to *View/Dis-Assembly*. The disassembly window shows the assembly code that is stored in program memory. It also has a highlighted line that serves as the position marker when running the program.

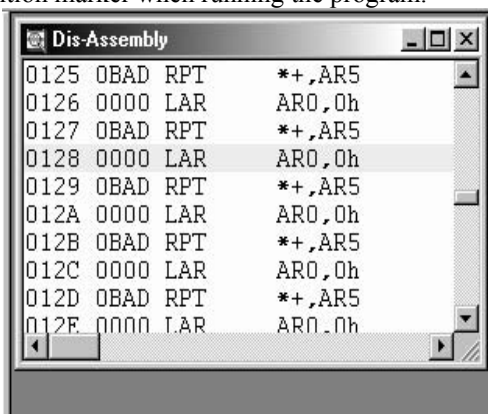
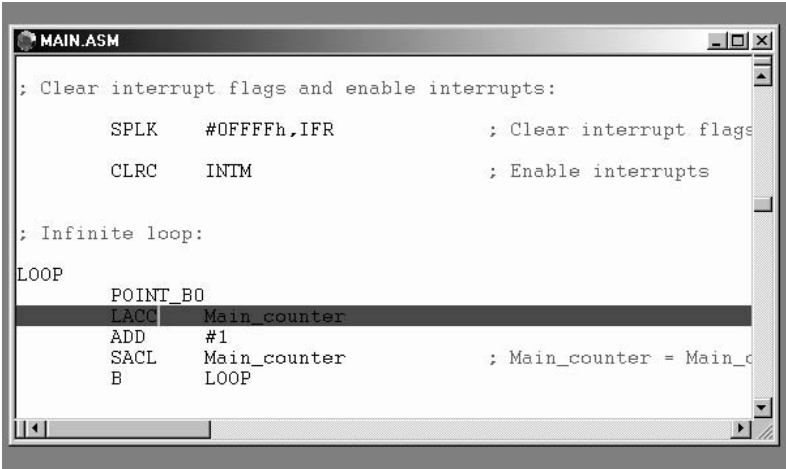


Figure 1.9 Disassembly window.

**Note:** When *Source Level Debugging* is selected (we’ll get to this in a minute), a position marker also appears in the appropriate source code window (if a program is loaded into the DSP) (Fig. 1.10).



```
MAIN.ASM
; Clear interrupt flags and enable interrupts:
        SPLK    #0FFFh,IFR                ; Clear interrupt flags
        CLRC    INTM                      ; Enable interrupts

; Infinite loop:
LOOP
        POINT_B0
        LACC    Main_counter
        ADD     #1
        SACL    Main_counter              ; Main_counter = Main_c
        B       LOOP
```

Figure 1.10 Source level debugging.

- 7. The CPU registers and CPU status registers are very helpful in debugging code. To view these registers, go to *View/CPU Registers* (both registers are under this menu). Open both CPU registers. You should see the registers appear in new frames on the screen.
- 8. The ability to view memory locations is also vital to debugging. To view memory, go to *View/Memory*. You should see a box pop up which will configure the memory window that is about to open (see Fig. 1.11). Enter 0x0300 for the start address.

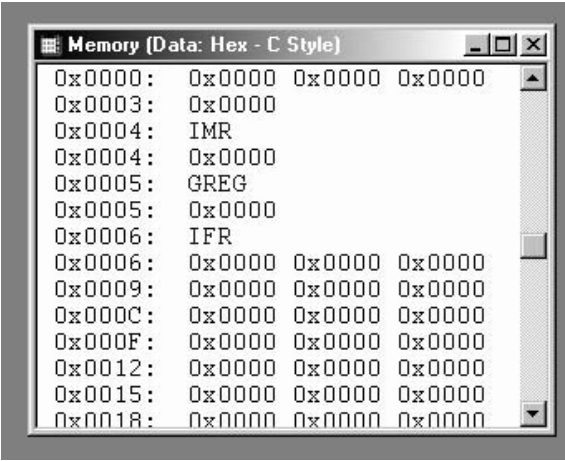


Figure 1.11 Memory viewing window.

You can also change the values for the CPU registers and memory locations by double clicking on the register or memory location. A box will pop up that will allow you to enter in a new value.

Double click on the 0x300 location in the memory window and change the value to 0x0555. The new value will appear in red signifying that the memory location has been changed.

Using the same technique, change a few registers in the CPU status and CPU register frames. Observe how the values in the registers change to the new value entered.

9. In MAIN.ASM scroll down until you see the line `“.bss Main_counter,1”`. Highlight `“Main_counter”` and add that variable to the watch window.

A watch window allows us to view variables that we use in our code. Open a watch window by going to *View/Watch Window*. You can add variables to this window by highlighting the variable name in the source code and then right clicking the mouse button and selecting `“add to watch window”`. Now, let us edit the display format of this variable in the watch window. Double click on the variable name in the watch window. When the `“edit variable”` box appears, add the command `“(int*)”` in front of the variable name (see Fig. 1.12). This configures the variable in the watch window to be displayed as an integer, thus ensuring that a decimal value is displayed. Otherwise, a hex value will be displayed.

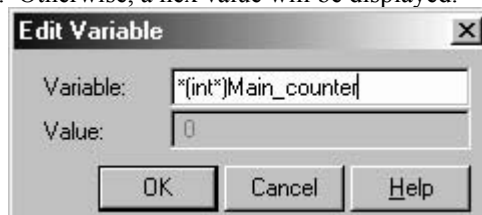


Figure 1.12 Editing a variable while in the watch window.

10. Rebuild the project (which should load the program as well) and reset the DSP by going to *Debug Menu/Reset DSP*.

**Note:** If a source code window opens up as well as the disassembly window when the project is built, Source Level Debugging is enabled. If not, enable Source Level Debugging by going to *Project/Options/Assembler Tab* and check the `“enable source level debugging”` (Fig. 1.13). Source level debugging lets you see where in the source code the program is running instead of having to decipher the disassembly window information. If you have just enabled Source Level Debugging, you need to rebuild the project before it takes effect.

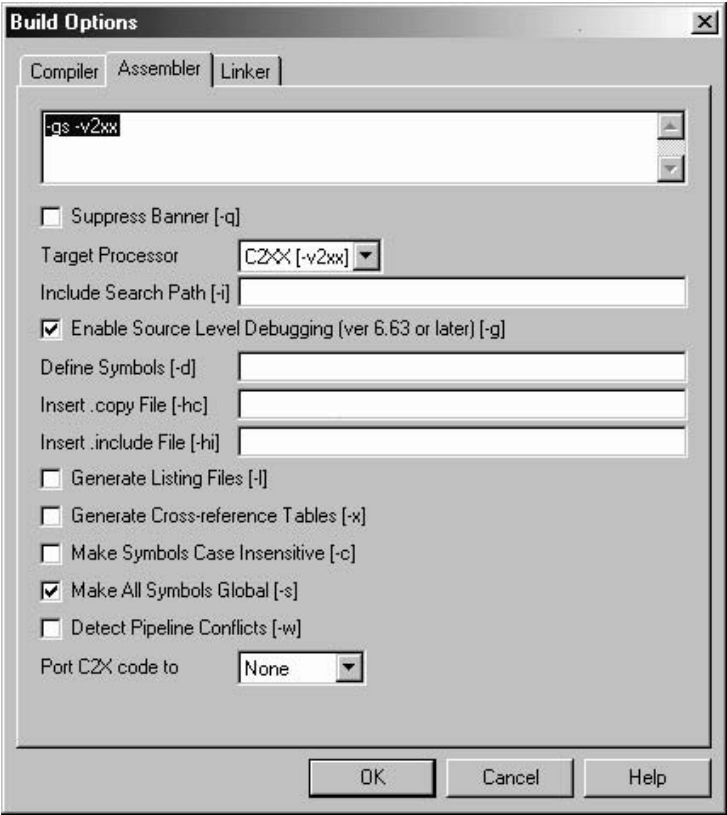


Figure 1.13 Build options menu box.

11. Enable Real Time mode by performing the following steps:
- a. The DSP *must* have the program already loaded in order to enable real-time mode. (While in real-time mode, programs cannot be loaded to the DSP.)
  - b. Reset the DSP by going to *Debug Menu / Reset DSP*.
  - c. Open the Command Window by going to *Tools Menu / Command Window*.
  - d. Type in the Command Window “go MON\_GO”.
  - e. Put CCS in Real-time mode by going to *Debug Menu / Real-time Mode*. When in real-time mode, you will see the word “REALTIME” in the bottom of the code composer screen.
  - f. Reset the DSP again and the program is ready to RUN.

**Note:** Real-time mode is a useful feature of CCS that allows you to see changes as they happen but is not necessary for program debugging. When CCS is not in real time mode, the values in all windows will update as soon as the program is halted or a break point occurs.

Right click on the watch window and choose “Continuous Refresh”. This will allow the values in the Watch window to change.

12. We are now ready to run the demonstration program. First make sure that no breakpoints have been set or the DSP will stop when it reaches the breakpoint.

Run the program by going to *Debug/Run*. Running and halting the DSP can also be performed by hitting F5 to run and Shift-F5 to halt. Observe as the value of “Main\_counter” in the watch window changes.

13. Halt the DSP by going to *Debug/Halt*. In the disassembly or source window you should see that the program is halted somewhere in the area of code entitled “Loop” (hex address 0159-015D in the disassembly window (program memory)). Left click on a line in the “Loop” section and toggle a breakpoint by right clicking the mouse and selecting “toggle breakpoint”. You should see a purple line appear where the breakpoint is set (Fig. 1.14). Notice how the breakpoint appears in both the disassembly window *and* the window containing the assembly source code.

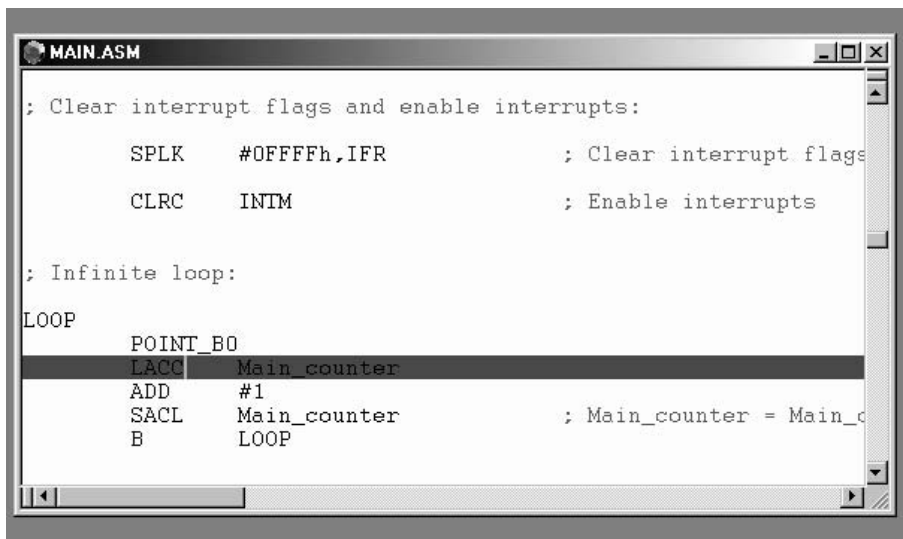


Figure 1.14 Breakpoint is located at the highlighted line (source level debug).

- 14. Run the program and watch as the DSP stops at the breakpoint each time it passes through the “Loop” section. (You will need to “run” the DSP each time after it hits a breakpoint because the breakpoint essentially “pauses” the DSP.) Observe as the value of Main\_counter increments by 1 in the watch window each time the code is restarted after the breakpoint. Remove the breakpoint by toggling it off.

***Note:** If you wish to single step through the code regardless of whether or not a breakpoint is set, you can do this by choosing Debug/Step Into or pressing F8.*

- 15. If you wish to save the screen configuration (position of windows, what appears on the screen, etc.) go to *File Menu/Workspace/save workspace* shown in Fig. 1.15.

Now, when you re-open CCS in the future, you will only have to load the workspace, saving you the trouble of opening the memory, CPU, and source code windows shown in Fig. 1.16. Saving a workspace not only saves window configuration, but project configuration as well. If a previously saved workspace is opened, the project that was open at the time of the workspace save will also open. While saving a workspace saves screen configuration, it *does not* save the contents of any files or the project!

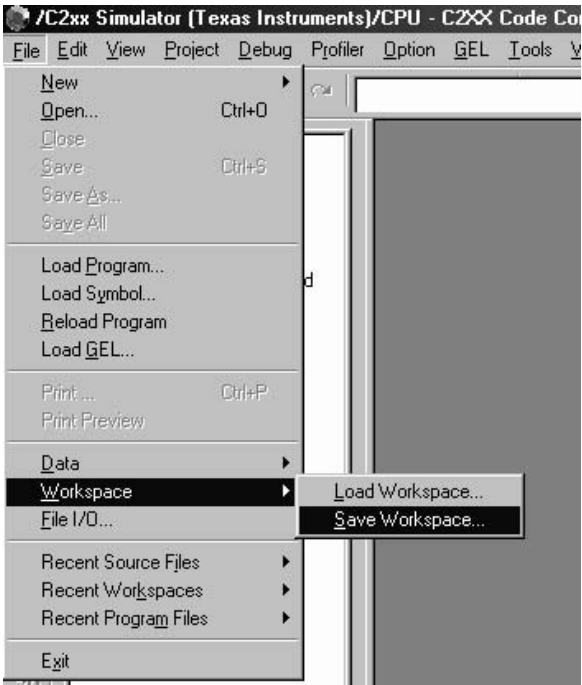


Figure 1.15 Saving a workspace.

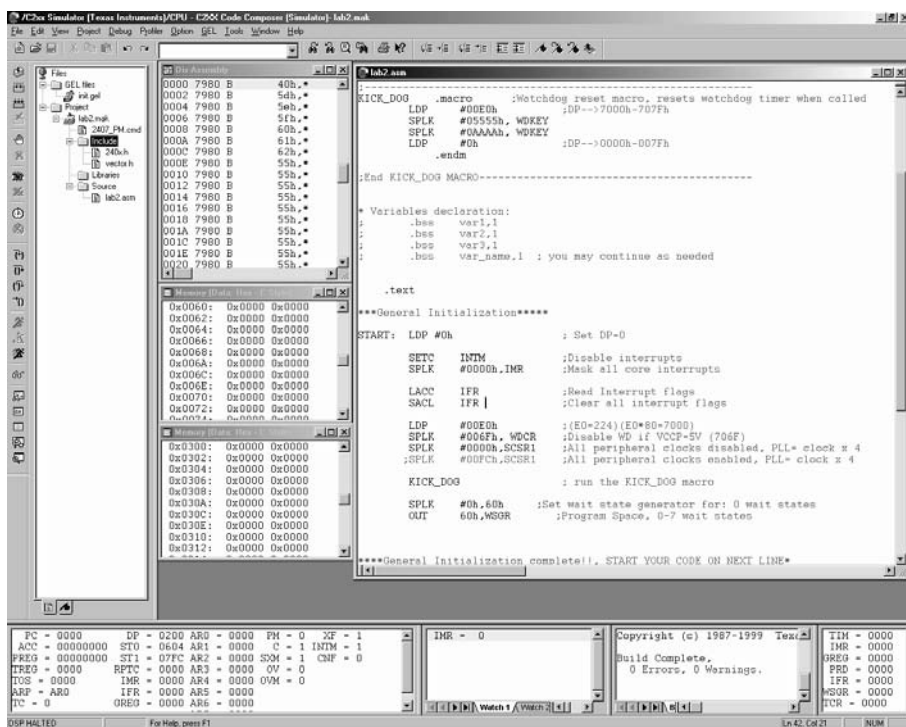


Figure 1.16 Screenshot of typical CCS™ workspace.

The screenshot shown in Fig. 1.16 displays what a typical workspace might contain. The workspace includes: several memory windows, watch window, CPU register windows, source code, and project window.

This concludes the introduction of the most common features of Code Composer Studio. There are many features not covered by this introduction that may be useful to advanced users. Consult the program Help or the Code Composer Users Guide for more information on Code Composer functions.





## Chapter 2

### C2xx DSP CPU AND INSTRUCTION SET

#### 2.1 Introduction to the C2xx DSP Core and Code Generation

The heart of the LF2407 DSP Controller is the C2xx DSP core. This core is a 16-bit fixed point processor, meaning that it works with 16-bit binary numbers. One can think of the C2xx as the central processor in a personal computer. The LF2407 DSP consists of the C2xx DSP core plus many peripherals such as Event Managers, ADC, etc., all integrated onto one single chip. This chapter will discuss the C2xx DSP core, subcomponents, and instruction set.

The C2xx core has its own native instruction set of assembly mnemonics or commands. Through the use of CCS and the associated compiler, one has the freedom of writing code in both C language and the native assembly language. However, to write compact, fast executing programs, it is best to compose code in assembly language. Due to this reason, programming in assembly will be the focus of this book. However, we will also include an example of a software tool called VisSim™, by Visual Solutions. VisSim allows users to simulate algorithms and develop code in “block” form. More on VisSim will be presented in the Appendix.

#### 2.2 The Components of the C2xx DSP Core

The DSP core (like all microprocessors) consists of several subcomponents necessary to perform arithmetic operations on 16-bit binary numbers. The following is a list of the multiple subcomponents found in the C2xx core which we will discuss further:

- A 32-bit central arithmetic logic unit (CALU)
- A 32-bit accumulator (used frequently in programs)
- Input and output data-scaling shifters for the CALU
- A (16-bit by 16-bit) multiplier
- A product-scaling shifter
- Eight auxiliary registers (AR0 – AR7) and an auxiliary register arithmetic unit (ARAU)

Each of the above components is either accessed directly by the user code or is indirectly used during the execution of an assembly command.

#### Central Arithmetic Logic Unit (CALU)

The C2xx performs 2s-complement arithmetic using the 32-bit CALU. The CALU uses 16-bit words taken from data memory, derived from an immediate instruction, or from the 32-bit multiplier result. In addition to arithmetic operations, the CALU can perform Boolean operations. The CALU is somewhat transparent to

the user. For example, if an arithmetic command is used, the user only needs to write the command and later read the output from the appropriate register. In this sense, the CALU is “transparent” in that it is not accessed directly by the user.

### Accumulator

The accumulator stores the output from the CALU and also serves as another input to the CALU (many arithmetic commands perform operations on numbers that are currently stored in the accumulator; versus other memory locations). The accumulator is 32 bits wide and is divided into two sections, each consisting of 16 bits. The high-order bits consist of bits 31 through 16, and the low-order bits are made up of bits 15 through 0. Assembly language instructions are provided for storing the high- and low-order accumulator words to data memory. In most cases, the accumulator is written to and read from directly by the user code via assembly commands. In some instances, the accumulator is also transparent to the user (similar to the CALU operation in that it is accessed “behind the scenes”).

### Scaling Shifters

The C2xx has three 32-bit shifters that allow for scaling, bit extraction, extended arithmetic, and overflow-prevention operations. The scaling shifters make possible commands that shift data left or right. Like the CALU, the operation of the scaling shifters is “transparent” to the user. For example, the user needs only to use a shift command, and observe the result. Any one of the three shifters could be used by the C2xx depending on the specific instruction entered. The following is a description of the three shifters:

- **Input data-scaling shifter (input shifter):** This shifter left-shifts 16-bit input data by 0 to 16 bits to align the data to the 32-bit input of the CALU. For example, when the user uses a command such as “ADD 300h, 5”, the input shifter is responsible for first shifting the data in memory address “300h” to the left by five places before it is added to the contents of the accumulator.
- **Output data-scaling shifter (output shifter):** This shifter left-shifts data from the accumulator by 0 to 7 bits before the output is stored to data memory. The content of the accumulator remains unchanged. For example, when the user uses a command such as “SACL 300h, 4”, the output shifter is responsible for first shifting the contents of the accumulator to the left by four places before it is stored to the memory address “300h”.

- **Product-scaling shifter (product shifter):** The product register (PREG) receives the output of the multiplier. The product shifter shifts the output of the PREG before that output is sent to the input of the CALU. The product shifter has four product shift modes (no shift, left shift by one bit, left shift by four bits, and right shift by six bits), which are useful for performing multiply/accumulate operations, fractional arithmetic, or justifying fractional products.

## Multiplier

The multiplier performs 16-bit, 2s-complement multiplication and creates a 32-bit result. In conjunction with the multiplier, the C2xx uses the 16-bit temporary register (TREG) and the 32-bit product register (PREG).

The operation of the multiplier is not as “transparent” as the CALU or shifters. The TREG **always** needs to be loaded with one of the numbers that are to be multiplied. Other than this prerequisite, the multiplication commands do not require any more actions from the user code. The output of the multiply is stored in the PREG, which can later be read by the user code.

## Auxiliary Register Arithmetic Unit (ARAU) and Auxiliary Registers

The ARAU generates data memory addresses when an instruction uses indirect addressing to access data memory (more on indirect addressing will be covered later along with assembly programming). Eight auxiliary registers (AR0 through AR7) support the ARAU, each of which can be loaded with a 16-bit value from data memory or directly from an instruction. Each auxiliary register value can also be stored in data memory. The auxiliary registers are mainly used as “pointers” to data memory locations to more easily facilitate looping or repeating algorithms. They are directly written to by the user code and are automatically incremented or decremented by particular assembly instructions during a looping or repeating operation. The auxiliary register pointer (ARP) embedded in status register ST0 references the auxiliary register. The status registers (ST0, ST1) are core level registers where values such as the Data Page (DP) and ARP located. More on the operation and use of auxiliary registers will be covered in subsequent chapters.

## 2.3 Mapping External Devices to the C2xx Core and the Peripheral

### Interface

Since the LF2407 contains many peripherals that need to be accessed by the C2xx core, the C2xx needs a way to read and write to the different peripherals. To make this possible, peripherals are mapped to data memory (memory will be covered shortly). Each peripheral is mapped to a corresponding block of data memory addresses. Where applicable, each corresponding block contains configuration registers, input registers, output registers, and status registers. Each peripheral is accessed by simply writing to the appropriate registers in data memory, provided the peripheral clock is enabled (see System Configuration registers).