

Digital Signal Processing and Applications with the **C6713** and **C6416 DSK**

Rulph Chassaing



CD-ROM INCLUDED

Topics in Digital Signal Processing

Digital Signal Processing and Applications with the C6713 and C6416 DSK

Rulph Chassaing

Worcester Polytechnic Institute



A JOHN WILEY & SONS, INC., PUBLICATION

**Digital Signal Processing
and Applications with the
C6713 and C6416 DSK**

TOPICS IN DIGITAL SIGNAL PROCESSING

C. S. BURRUS and T. W. PARKS: *DFT/FFT AND CONVOLUTION ALGORITHMS: THEORY AND IMPLEMENTATION*

JOHN R. TREICHLER, C. RICHARD JOHNSON, JR., and MICHAEL G. LARIMORE: *THEORY AND DESIGN OF ADAPTIVE FILTERS*

T. W. PARKS and C. S. BURRUS: *DIGITAL FILTER DESIGN*

RULPH CHASSAING and DARRELL W. HORNING: *DIGITAL SIGNAL PROCESSING WITH THE TMS320C25*

RULPH CHASSAING: *DIGITAL SIGNAL PROCESSING WITH C AND THE TMS320C30*

RULPH CHASSAING: *DIGITAL SIGNAL PROCESSING LABORATORY EXPERIMENTS USING C AND THE TMS320C31 DSK*

RULPH CHASSAING: *DSP APPLICATIONS USING C AND THE TMS320C6x DSK*

RULPH CHASSAING: *DIGITAL SIGNAL PROCESSING AND APPLICATIONS WITH THE C6713 AND C6416 DSK*

Digital Signal Processing and Applications with the C6713 and C6416 DSK

Rulph Chassaing

Worcester Polytechnic Institute



A JOHN WILEY & SONS, INC., PUBLICATION

Copyright © 2005 by John Wiley & Sons, Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400, fax 978-646-8600, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993 or fax 317-572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print, however, may not be available in electronic format.

Library of Congress Cataloging-in-Publication Data:

Chassaing, Rulph.

Digital signal processing and applications with the C6713 and C6416 DSK / by Rulph Chassaing.
p. cm.

Includes bibliographical references and index.

ISBN 0-471-69007-4

1. Signal processing—Digital techniques. 2. Texas Instruments TMS320 series microprocessors.

I. Title.

TK5102.9.C47422 2004
621.382'2—dc22

2004050924

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Contents

Preface	xiii
List of Examples	xvii
Programs/Files on Accompanying CD	xxi
1 DSP Development System	1
1.1 Introduction	1
1.2 DSK Support Tools	2
1.2.1 DSK Board	3
1.2.2 TMS320C6713 Digital Signal Processor	5
1.3 Code Composer Studio	5
1.3.1 CCS Installation and Support	6
1.3.2 Useful Types of Files	7
1.4 Quick Test of DSK	7
1.5 Support Files	8
1.6 Programming Examples to Test the DSK Tools	9
1.7 Support Programs/Files Considerations	27
1.7.1 Initialization/Communication File	27
1.7.2 Vector File	30
1.7.3 Linker Command File	32
1.8 Compiler/Assembler/Linker Shell	33
1.8.1 Compiler	33
1.8.2 Assembler	34
1.8.3 Linker	34
	v

1.9	Assignments	35
	References	36
2	Input and Output with the DSK	39
2.1	Introduction	39
2.2	TLV320AIC23 (AIC23) Onboard Stereo Codec for Input and Output	40
2.3	Programming Examples Using C Code	42
2.4	Assignments	71
	References	72
3	Architecture and Instruction Set of the C6x Processor	73
3.1	Introduction	73
3.2	TMS320C6x Architecture	75
3.3	Functional Units	76
3.4	Fetch and Execute Packets	79
3.5	Pipelining	79
3.6	Registers	81
3.7	Linear and Circular Addressing Modes	82
	3.7.1 Indirect Addressing	82
	3.7.2 Circular Addressing	82
3.8	TMS320C6x Instruction Set	84
	3.8.1 Assembly Code Format	84
	3.8.2 Types of Instructions	85
3.9	Assembler Directives	86
3.10	Linear Assembly	87
3.11	ASM Statement within C	88
3.12	C-Callable Assembly Function	89
3.13	Timers	89
3.14	Interrupts	89
	3.14.1 Interrupt Control Registers	90
	3.14.2 Interrupt Acknowledgment	91
3.15	Multichannel Buffered Serial Ports	92
3.16	Direct Memory Access	92
3.17	Memory Considerations	93
	3.17.1 Data Allocation	93
	3.17.2 Data Alignment	94

	Contents	vii
3.17.3	Pragma Directives	94
3.17.4	Memory Models	95
3.18	Fixed- and Floating-Point Format	95
3.18.1	Data Types	95
3.18.2	Floating-Point Format	96
3.18.3	Division	97
3.19	Code Improvement	97
3.19.1	Intrinsics	97
3.19.2	Trip Directive for Loop Count	98
3.19.3	Cross-Paths	98
3.19.4	Software Pipelining	98
3.20	Constraints	99
3.20.1	Memory Constraints	99
3.20.2	Cross-Path Constraints	99
3.20.3	Load/Store Constraints	100
3.20.4	Pipelining Effects with More Than One EP within an FP	100
3.21	Programming Examples Using C, Assembly, and Linear Assembly	101
3.22	Assignments	115
	References	117
4	Finite Impulse Response Filters	119
4.1	Introduction to the z -Transform	119
4.1.1	Mapping from s -Plane to z -Plane	122
4.1.2	Difference Equations	123
4.2	Discrete Signals	124
4.3	FIR Filters	125
4.4	FIR Lattice Structure	127
4.5	FIR Implementation Using Fourier Series	131
4.6	Window Functions	135
4.6.1	Hamming Window	136
4.6.2	Hanning Window	136
4.6.3	Blackman Window	136
4.6.4	Kaiser Window	137
4.6.5	Computer-Aided Approximation	137
4.7	Programming Examples Using C and ASM Code	137
4.8	Assignments	173
	References	174

5	Infinite Impulse Response Filters	177
5.1	Introduction	177
5.2	IIR Filter Structures	178
5.2.1	Direct Form I Structure	178
5.2.2	Direct Form II Structure	179
5.2.3	Direct Form II Transpose	181
5.2.4	Cascade Structure	182
5.2.5	Parallel Form Structure	183
5.2.6	Lattice Structure	185
5.3	Bilinear Transformation	190
5.3.1	BLT Design Procedure	191
5.4	Programming Examples Using C and ASM Code	192
5.5	Assignments	205
	References	206
6	Fast Fourier Transform	208
6.1	Introduction	208
6.2	Development of the FFT Algorithm with Radix-2	209
6.3	Decimation-in-Frequency FFT Algorithm with Radix-2	210
6.4	Decimation-in-Time FFT Algorithm with Radix-2	217
6.5	Bit Reversal for Unscrambling	221
6.6	Development of the FFT Algorithm with Radix-4	221
6.7	Inverse Fast Fourier Transform	224
6.8	Programming Examples	225
6.8.1	Fast Convolution	237
6.9	Assignments	245
	References	247
7	Adaptive Filters	249
7.1	Introduction	249
7.2	Adaptive Structures	251
7.3	Adaptive Linear Combiner	254
7.4	Performance Function	257
7.5	Searching for the Minimum	259
7.6	Programming Examples for Noise Cancellation and System Identification	262
	References	282

	Contents	ix
8 Code Optimization		284
8.1 Introduction		284
8.2 Optimization Steps		285
8.2.1 Compiler Options		285
8.2.2 Intrinsic C Functions		286
8.3 Procedure for Code Optimization		286
8.4 Programming Examples Using Code Optimization Techniques		286
8.5 Software Pipelining for Code Optimization		293
8.5.1 Procedure for Hand-Coded Software Pipelining		293
8.5.2 Dependency Graph		294
8.5.3 Scheduling Table		295
8.6 Execution Cycles for Different Optimization Schemes		302
References		303
9 DSP/BIOS and RTDX Using MATLAB, Visual C++, Visual Basic, and LabVIEW		304
9.1 Introduction to DSP/BIOS		306
9.2 RTDX Using MATLAB to Provide Interface Between PC and DSK		311
9.3 RTDX Using Visual C++ to Interface with DSK		321
9.4 RTDX Using Visual Basic to Provide Interface Between PC and DSK		332
9.5 RTDX Using LabVIEW to Provide Interface Between PC and DSK		335
Acknowledgments		342
References		342
10 DSP Applications and Student Projects		343
10.1 DTMF Detection Using Correlation, FFT, and Goertzel Algorithm		343
10.1.1 Using a Correlation Scheme and Onboard LEDs for Verifying Detection		345
10.1.2 Using RTDX with Visual C++ to Display Detected DTMF Signals on the PC		348
10.1.3 Using FFT and Onboard LEDs for Verifying Detection		350
10.1.4 Using Goertzel Algorithm		350
10.2 Beat Detection Using Onboard LEDs		352

10.3	FIR with RTDX Using Visual C++ for Transfer of Filter Coefficients	355
10.4	Radix-4 FFT with Frequency Domain Filtering	357
10.5	Radix-4 FFT with RTDX Using Visual C++ and MATLAB for Plotting	357
10.6	Spectrum Display Through EMIF Using a Bank of 32 LEDs	360
10.7	Spectrum Display Through EMIF Using LCDs	364
10.8	Time-Frequency Analysis of Signals with Spectrogram	368
10.8.1	Simulation Using MATLAB	368
10.8.2	Spectrogram with RTDX Using MATLAB	370
10.8.3	Spectrogram with RTDX Using Visual C++	372
10.9	Audio Effects (Echo and Reverb, Harmonics, and Distortion)	373
10.10	Voice Detection and Reverse Playback	375
10.11	Phase Shift Keying—BPSK Encoding and Decoding with PLL	377
10.11.1	BPSK Single-Board Transmitter/Receiver Simulation	377
10.11.2	BPSK Transmitter/Voice Encoder with Real-Time Input	381
10.11.3	Phase-Locked Loop	383
10.11.4	BPSK Transmitter and Receiver with PLL	386
10.12	Binary Phase Shift Keying	390
10.13	Modulation Schemes—PAM and PSK	393
10.13.1	Pulse Amplitude Modulation	393
10.13.2	Phase-Shift Keying	396
10.14	Selectable IIR Filter and Scrambling Scheme Using Onboard Switches	401
10.15	Convolutional Encoding and Viterbi Decoding	404
10.16	Speech Synthesis Using Linear Prediction of Speech Signals	414
10.17	Automatic Speaker Recognition	418
10.18	μ -Law for Speech Companding	422
10.19	Voice Scrambler Using DMA and User Switches	423
10.20	SB-ADPCM Encoder/Decoder: Implementation of G.722 Audio Coding	423
10.21	Encryption Using the Data Encryption Standard Algorithm	425
10.22	Phase-Locked Loop	429
10.23	Miscellaneous Projects	430
10.23.1	Multirate Filter	431
10.23.2	Acoustic Direction Tracker	436
10.23.3	Neural Network for Signal Recognition	437
10.23.4	Adaptive Temporal Attenuator	441

	Contents	xi
10.23.5	FSK Modem	442
10.23.6	Image Processing	443
10.23.7	Filter Design and Implementation Using a Modified Prony's Method	444
10.23.8	PID Controller	444
10.23.9	Four-Channel Multiplexer for Fast Data Acquisition	444
10.23.10	Video Line Rate Analysis	444
	Acknowledgments	444
	References	445
Appendix A	TMS320C6x Instruction Set	450
A.1	Instructions for Fixed- and Floating-Point Operations	450
A.2	Instructions for Floating-Point Operations	450
	References	450
Appendix B	Registers for Circular Addressing and Interrupts	452
	Reference	452
Appendix C	Fixed-Point Considerations	455
C.1	Binary and Two's-Complement Representation	455
C.2	Fractional Fixed-Point Representation	458
C.3	Multiplication	458
	Reference	461
Appendix D	MATLAB Support Tools	462
D.1	SPTool and FDATool for FIR Filter Design	462
D.2	SPTool and FDATool for IIR Filter Design	465
D.3	MATLAB for FIR Filter Design Using the Student Version	468
D.4	MATLAB for IIR Filter Design Using the Student Version	470
D.5	BLT Using MATLAB and Support Programs on CD	471
D.6	FFT and IFFT	477
	References	478
Appendix E	Additional Support Tools	479
E.1	Goldwave Shareware Utility as a Virtual Instrument	479
E.2	Filter Design Using DigiFilter	480

xii Contents

E.2.1	FIR Filter Design	480
E.2.2	IIR Filter Design	481
E.3	FIR Filter Design Using a Filter Development Package	482
E.3.1	Kaiser Window	482
E.3.2	Hamming Window	484
E.4	Visual Application Builder and LabVIEW	485
E.5	Alternative Input/Output	485
	References	485
 Appendix F Fast Hartley Transform		 486
	References	492
 Appendix G Goertzel Algorithm		 493
G.1	Design Considerations	493
	References	496
 Appendix H TMS320C6416 DSK		 497
H.1	TMS320C64x Processor	497
H.2	Programming Examples Using the C6416 DSK	498
	References	502
 Appendix I TMS320C6711 DSK		 503
	Reference	503
 Index		 505

Preface

Digital signal processors, such as the TMS320 family of processors, are used in a wide range of applications, such as in communications, controls, speech processing, and so on. They are used in cellular phones, digital cameras, high-definition television (HDTV), radio, fax transmission, modems, and other devices. These devices have also found their way into the university classroom, where they provide an economical way to introduce real-time digital signal processing (DSP) to the student.

Texas Instruments introduced the TM320C6x processor, based on the very-long-instruction-word (VLIW) architecture. This new architecture supports features that facilitate the development of efficient high-level language compilers. Throughout the book we refer to the C/C++ language simply as C. Although TMS320C6x/assembly language can produce fast code, problems with documentation and maintenance may exist. With the available C compiler, the programmer must “let the tools do the work.” After that, if the programmer is not satisfied, Chapters 3 and 8 and the last few examples in Chapter 4 can be very useful.

This book is intended primarily for senior undergraduate and first-year graduate students in electrical and computer engineering and as a tutorial for the practicing engineer. It is written with the conviction that the principles of DSP can best be learned through interaction in a laboratory setting, where students can appreciate the concepts of DSP through real-time implementation of experiments and projects. The background assumed is a course in linear systems and some knowledge of C.

Most chapters begin with a theoretical discussion, followed by representative examples that provide the necessary background to perform the concluding experiments. There are a total of 105 programming examples, most using C code, with a few in assembly and linear assembly code. A list of these examples appears on page xvii. A total of 22 students’ projects are also discussed. These projects cover a wide

range of applications in filtering, spectrum analysis, modulation techniques, speech processing, and so on.

Programming examples are included throughout the text. This can be useful to the reader who is familiar with both DSP and C programming but who is not necessarily an expert in both. Many assignments are included at the end of Chapters 1–6.

This book can be used in the following ways:

1. For a DSP course with a laboratory component, using parts of Chapters 1–9. If needed, the book can be supplemented with some additional theoretical materials, since its emphasis is on the practical aspects of DSP. It is possible to cover Chapter 7 on adaptive filtering following Chapter 4 on finite impulse response (FIR) filtering (since there is only one example in Chapter 7 that uses materials from Chapter 5). It is my conviction that adaptive filtering should be incorporated into an undergraduate course in DSP.
2. For a laboratory course using many of the examples and experiments from Chapters 1–7 and Chapter 9. The beginning of the semester can be devoted to short programming examples and experiments and the remainder of the semester for a final project. The wide range of sample projects (for both undergraduate and graduate students) discussed in Chapter 10 can be very valuable.
3. For a senior undergraduate or first-year graduate design project course using selected materials from Chapters 1–10.
4. For the practicing engineer as a tutorial and reference, and for workshops and seminars, using selected materials throughout the book.

In Chapter 1 we introduce the tools through three programming examples. These tools include the powerful Code Composer Studio (CCS) provided with the TMS320C6713 DSP starter kit (DSK). It is essential to perform these examples before proceeding to subsequent chapters. They illustrate the capabilities of CCS for debugging, plotting in both the time and frequency domains, and other matters. Appendix H contains several programming examples using the TMS320C6416 DSK.

In Chapter 2 we illustrate input and output (I/O) with the AIC23 stereo codec on the DSK board through many programming examples. Chapter 3 covers the architecture and the instructions available for the TMS320C6x processor. Special instructions and assembler directives that are useful in DSP are discussed. Programming examples using both assembly and linear assembly are included in this chapter.

In Chapter 4 we introduce the z -transform and discuss FIR filters and the effect of window functions on these filters. Chapter 5 covers infinite impulse response (IIR) filters. Programming examples to implement real-time FIR and IIR filters are included. Appendix D illustrates MATLAB for the design of FIR and IIR filters.

Chapter 6 covers the development of the fast Fourier transform (FFT). Programming examples on FFT are included using both radix-2 and radix-4 FFT. In

Chapter 7 we demonstrate the usefulness of the adaptive filter for a number of applications with least mean squares (LMS). Programming examples are included to illustrate the gradual cancellation of noise or system identification. Students have been very receptive to applications in adaptive filtering. Chapter 8 illustrates techniques for code optimization.

In Chapter 9 we introduce DSP/BIOS and discuss a number of schemes (Visual C++, MATLAB, etc.) for real-time data transfer (RTDX) and communication between the PC and the DSK.

Chapter 10 discusses a total of 22 projects implemented by undergraduate and graduate students. They cover a wide range of DSP applications in filtering, spectrum analysis, modulation schemes, speech processing, and so on.

A CD is included with this book and contains all the programs discussed. See page xxi for a list of the folders that contain the support files for the examples and projects.

Over the last 10 years, faculty members from over 200 institutions have taken my workshops on “DSP and Applications.” Many of these workshops were supported by grants from the National Science Foundation (NSF) and, subsequently, by Texas Instruments. I am thankful to NSF, Texas Instruments, and the participating faculty members for their encouragement and feedback. I am grateful to Dr. Donald Reay of Heriot-Watt University, who contributed several examples during his review of my previous book based on the TMS320C6711 DSK. I appreciate the many suggestions made by Dr. Mounir Boukadoum of the University of Quebec, Dr. Subramaniam Ganesan from Oakland University, and Dr. David Kozel from Purdue University at Calumet. I also thank Dr. Darrell Horning of the University of New Haven, with whom I coauthored my first book, *Digital Signal Processing with the TMS320C25*, for introducing me to “book writing.” I thank all the students at Roger Williams University, the University of Massachusetts at Dartmouth, and Worcester Polytechnic Institute (WPI) who have taken my real-time DSP and senior design project courses, based on the TMS320 processors, over the last 20 years. The contribution of Aghogho Obi, from WPI, is very much appreciated.

The continued support of many people from Texas Instruments is also very much appreciated: Cathy Wicks and Christina Peterson, in particular, have been very supportive of this book.

Special appreciation: The laboratory assistance of Walter J. Gomes III in several workshops and during the development of many examples has been invaluable. His contribution is appreciated.

RULPH CHASSAING
Chassaing@msn.com
Chassaing@ece.wpi.edu

List of Examples

1.1	Sine Generation Using Eight Points with DIP Switch Control	9
1.2	Generation of the Sinusoid and Plotting with CCS	19
1.3	Dot Product of Two Arrays	22
2.1	Loop Program Using Interrupt	43
2.2	Loop Program Using Polling	45
2.3	Stereo Input and Stereo Output	46
2.4	Sine Generation with Two Sliders for Amplitude and Frequency Control	48
2.5	Loop Program with Input Data Stored in Memory	50
2.6	Loop with Data in a Buffer Printed to a File	52
2.7	Square-Wave Generation Using a Lookup Table	53
2.8	Ramp Generation Using a Lookup Table	54
2.9	Ramp Generation without a Lookup Table	55
2.10	Echo	56
2.11	Echo with Control for Different Effects	57
2.12	Sine Generation with Table Values Generated within the Program	59
2.13	Sine Generation with a Table Created by MATLAB	60
2.14	Amplitude Modulation	62
2.15	Sweep Sinusoid Using a Table with 8000 Points	63
2.16	Pseudorandom Noise Sequence Generation	65
2.17	Sine Generation with Dip Switch Control	66
2.18	Use of External Memory to Record Voice	67
2.19	Use of Flash Memory—Programming the Onboard Flash	69
3.1	Efficient Dot Product	102
3.2	Sum of $n + (n - 1) + (n - 2) + \dots + 1$, Using C Calling an Assembly Function	103

xviii List of Examples

3.3	Factorial of a Number Using C Calling an Assembly Function	104
3.4	32-bit Pseudorandom Noise Generation Using C Calling an Assembly Function	105
3.5	Code Detection Using C Calling an ASM Function	107
3.6	Dot Product Using Assembly Program Calling an Assembly Function	109
3.7	Dot Product Using C Function Calling a Linear Assembly Function	112
3.8	Factorial Using C Calling a Linear Assembly Function	114
4.1	FIR Filter Implementation: Bandstop and Bandpass	139
4.2	Effects on Voice Using Three FIR Lowpass Filters	144
4.3	Implementation of Four Different Filters: Lowpass, Highpass, Bandpass, and Bandstop	147
4.4	FIR Implementation with a Pseudorandom Noise Sequence as Input to a Filter	148
4.5	FIR Filter with Internally Generated Pseudorandom Noise as Input to a Filter and Output Stored in Memory	151
4.6	Two Notch Filters to Recover Corrupted Input Voice	154
4.7	FIR Implementation Using Four Different Methods	156
4.8	Voice Scrambling Using Filtering and Modulation	158
4.9	Illustration of Aliasing Effects with Down-Sampling	161
4.10	Implementation of an Inverse FIR Filter	163
4.11	FIR Implementation Using C Calling an ASM Function	164
4.12	FIR Implementation Using C Calling a Faster ASM Function	167
4.13	FIR Implementation Using C Calling an ASM Function with a Circular Buffer	168
4.14	FIR Implementation Using C Calling an ASM Function with a Circular Buffer in External Memory	172
5.1	IIR Filter Implementation Using Second-Order Stages in Cascade	192
5.2	Generation of Two Tones Using Two Second-Order Difference Equations	196
5.3	Sine Generation Using a Difference Equation	199
5.4	Generation of a Swept Sinusoid Using a Difference Equation	200
5.5	IIR Inverse Filter	202
5.6	Sine Generation Using a Difference Equation with C Calling an ASM Function	205
6.1	DFT of a Sequence of Real Numbers with Output from the CCS Window	225
6.2	FFT of a Real-Time Input Signal Using an FFT Function in C	227
6.3	FFT of a Sinusoidal Signal from a Table Using TI's C-Callable Optimized FFT Function	229

6.4	FFT of Real-Time Input Using TI's C-Callable Optimized Radix-2 FFT Function	232
6.5	Radix-4 FFT of Input from a Lookup Table Using TI's C-Callable Optimized FFT Function	234
6.6	Radix-4 FFT of Real-Time Input Using TI's C-Callable Optimized FFT Function	236
6.7	Fast Convolution With Overlap-Add for FIR Implementation Using TI's Floating-Point FFT Functions	237
6.8	Fast Convolution with Overlap-Add Simulation for FIR Implementation Using a C-Coded FFT Function	241
6.9	Graphic Equalizer	242
7.1	Adaptive Filter Using C Code Compiled with Borland C/C++	262
7.2	Adaptive Filter for Sinusoidal Noise Cancellation	265
7.3	Adaptive FIR Filter for Noise Cancellation Using External Inputs	267
7.4	Adaptive FIR Filter for System ID of a Fixed FIR as an Unknown System	270
7.5	Adaptive FIR for System ID of a Fixed FIR as an Unknown System with Weights of an Adaptive Filter Initialized as an FIR Bandpass	272
7.6	Adaptive FIR for System ID of Fixed IIR as an Unknown System	275
7.7	Adaptive Predictor for Cancellation of Narrowband Interference Added to a Desired Wideband Signal	275
7.8	Adaptive Predictor for Cancellation of Narrowband Interference Added to a Desired Wideband Signal Using External Inputs	280
8.1	Sum of Products with Word-Wide Data Access for Fixed-Point Implementation Using C Code	287
8.2	Separate Sum of Products with C Intrinsic Functions Using C Code	288
8.3	Sum of Products with Word-Wide Access for Fixed-Point Implementation Using Linear ASM Code	288
8.4	Sum of Products with Double-Word Load for Floating-Point Implementation Using Linear ASM Code	289
8.5	Dot Product with No Parallel Instructions for Fixed-Point Implementation Using ASM Code	289
8.6	Dot Product with Parallel Instructions for Fixed-Point Implementation Using ASM Code	290
8.7	Two Sums of Products with Word-Wide (32-Bit) Data for Fixed-Point Implementation Using ASM Code	290
8.8	Dot Product with No Parallel Instructions for Floating-Point Implementation Using ASM Code	291
8.9	Dot Product with Parallel Instructions for Floating-Point Implementation Using ASM Code	292

XX List of Examples

8.10	Two Sums of Products with Double-Word-Wide (64-Bit) Data for Floating-Point Implementation Using ASM Code	292
8.11	Dot Product Using Software Pipelining for a Fixed-Point Implementation	297
8.12	Dot Product Using Software Pipelining for a Floating-Point Implementation	299
9.1	Sine Generation with DIP Switch Control through DSP/BIOS	306
9.2	Blinking of LEDs at Different Rates Using DSP/BIOS	309
9.3	Sine Generation Using BIOS to Set Up Interrupt INT11	310
9.4	MATLAB–DSK Interface Using RTDX	311
9.5	MATLAB–DSK Interface Using RTDX, with MATLAB For FFT and Plotting	314
9.6	MATLAB–DSK Interface Using RTDX For FIR Filter Implementation	317
9.7	Visual C++–DSK Interface Using RTDX for Amplitude Control of the Sine Wave	321
9.8	Visual C++–DSK Interface Using RTDX, with MATLAB Functions for FFT and Plotting	327
9.9	Visual Basic–DSK Interface Using RTDX for Amplitude Control of a Sine Wave	332
9.10	Visual Basic–DSK Interface Using RTDX for Amplitude Control of Output in a Loop Program	334
9.11	LabVIEW–DSK Interface Using RTDX for FIR Filtering	336
9.12	LabVIEW–DSK Interface Using RTDX for Controlling the Gain of a Generated Sinusoid	339
9.13	LabVIEW–DSK Interface Using RTDX for Controlling the Amplitude of a Generated Sinusoid with Real-Time Output from the DSK	341
D.1	SPTool and FDATool for FIR Filter Design	462
D.2	SPTool and FDATool for IIR Filter Design	465
D.3	FIR Filter Design Using MATLAB’s Student Version	468
D.4	Multiband FIR Filter Design Using MATLAB	469
D.5	IIR Filter Design Using MATLAB’s Student Version	470
H.1	Sine Generation with DIP Switch Control Using the C6416 DSK	498
H.2	Loop Program Using the C6416 DSK	499
H.3	FIR/IIR Implementation Using the C6416 DSK	499
H.4	FFT with C-Coded FFT Function Using the C6416 DSK	500
H.5	Adaptive FIR Filter Implementation Using the C6416 DSK	501
H.6	DTMF Implementation on the C6416 DSK Using the Goertzel Algorithm and the FFT, With RTDX Using Visual C++	501
I.1	Loop Program Using the C6711 DSK	503

1

DSP Development System

- Testing the software and hardware tools with Code Composer Studio
- Use of the TMS320C6713 DSK
- Programming examples to test the tools

Chapter 1 introduces several tools available for digital signal processing (DSP). These tools include the popular Code Composer Studio (CCS), which provides an integrated development environment (IDE), and the DSP starter kit (DSK) with the TMS320C6713 floating-point processor onboard and complete support for input and output. Three examples illustrate both the software and hardware tools included with the DSK. It is strongly suggested that you review these three examples before proceeding to subsequent chapters.

1.1 INTRODUCTION

Digital signal processors such as the TMS320C6x (C6x) family of processors are like fast special-purpose microprocessors with a specialized type of architecture and an instruction set appropriate for signal processing. The C6x notation is used to designate a member of Texas Instruments' (TI) TMS320C6000 family of digital signal processors. The architecture of the C6x digital signal processor is very well suited for numerically intensive calculations. Based on a very-long-instruction-word (VLIW) architecture, the C6x is considered to be TI's most powerful processor.

Digital signal processors are used for a wide range of applications, from communications and controls to speech and image processing. The general-purpose

digital signal processor is dominated by applications in communications (cellular). Applications embedded digital signal processors are dominated by consumer products. They are found in cellular phones, fax/modems, disk drives, radio, printers, hearing aids, MP3 players, high-definition television (HDTV), digital cameras, and so on. These processors have become the products of choice for a number of consumer applications, since they have become very cost-effective. They can handle different tasks, since they can be reprogrammed readily for a different application. DSP techniques have been very successful because of the development of low-cost software and hardware support. For example, modems and speech recognition can be less expensive using DSP techniques.

DSP processors are concerned primarily with real-time signal processing. Real-time processing requires the processing to keep pace with some external event, whereas non-real-time processing has no such timing constraint. The external event to keep pace with is usually the analog input. Whereas analog-based systems with discrete electronic components such as resistors can be more sensitive to temperature changes, DSP-based systems are less affected by environmental conditions. DSP processors enjoy the advantages of microprocessors. They are easy to use, flexible, and economical.

A number of books and articles address the importance of digital signal processors for a number of applications [1–22]. Various technologies have been used for real-time processing, from fiber optics for very high frequency to DSPs very suitable for the audio-frequency range. Common applications using these processors have been for frequencies from 0 to 96 kHz. Speech can be sampled at 8 kHz (the rate at which samples are acquired), which implies that each value sampled is acquired at a rate of $1/(8\text{ kHz})$ or 0.125 ms. A commonly used sample rate of a compact disk is 44.1 kHz. Analog/digital (A/D)-based boards in the megahertz sampling rate range are currently available.

The basic system consists of an analog-to-digital converter (ADC) to capture an input signal. The resulting digital representation of the captured signal is then processed by a digital signal processor such as the C6x and then output through a digital-to-analog converter (DAC). Also included within the basic system are a special input filter for anti-aliasing to eliminate erroneous signals and an output filter to smooth or reconstruct the processed output signal.

1.2 DSK SUPPORT TOOLS

Most of the work presented in this book involves the design of a program to implement a DSP application. To perform the experiments, the following tools are used:

1. *TI's DSP starter kit (DSK)*. The DSK package includes:
 - (a) *Code Composer Studio (CCS)*, which provides the necessary software support tools. CCS provides an integrated development environment (IDE), bringing together the C compiler, assembler, linker, debugger, and so on.

- (b) A board, shown in Figure 1.1, that contains the TMS320C6713 (C6713) floating-point digital signal processor as well as a 32-bit stereo codec for input and output (I/O) support.
 - (c) A universal synchronous bus (USB) cable that connects the DSK board to a PC.
 - (d) A 5 V power supply for the DSK board.
2. *An IBM-compatible PC.* The DSK board connects to the USB port of the PC through the USB cable included with the DSK package.
 3. *An oscilloscope, signal generator, and speakers.* A signal/spectrum analyzer is optional. Shareware utilities are available that utilize the PC and a sound card to create a virtual instrument such as an oscilloscope, a function generator, or a spectrum analyzer.

All the files/programs listed and discussed in this book (except some student project files in Chapter 10) are included on the accompanying CD. Most of the examples (with some minor modifications) can also run on the fixed-point C6416-based DSK. See Appendix H for the appropriate support files along with five illustrative examples. Reference 1 contains examples implemented on the C6711-based DSK (which has been discontinued). A list of all the examples is given on pages xv–xviii.

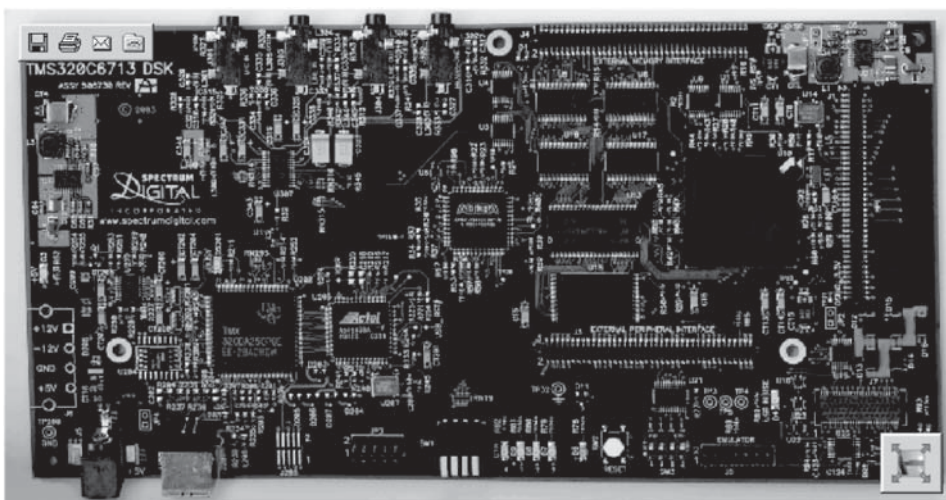
1.2.1 DSK Board

The DSK package is powerful, yet relatively inexpensive (\$395), with the necessary hardware and software support tools for real-time signal processing [23–43]. It is a complete DSP system. The DSK board, with an approximate size of 5×8 in., includes the C6713 floating-point digital signal processor and a 32-bit stereo codec TLV320AIC23 (AIC23) for input and output.

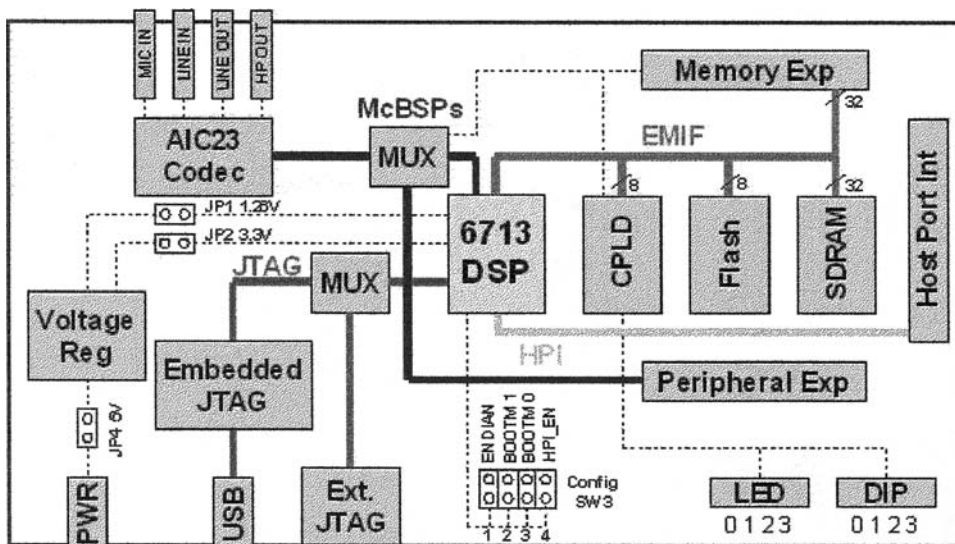
The onboard codec AIC23 [37] uses a sigma–delta technology that provides ADC and DAC. It connects to a 12-MHz system clock. Variable sampling rates from 8 to 96 kHz can be set readily.

A daughter card expansion is also provided on the DSK board. Two 80-pin connectors provide for external peripheral and external memory interfaces. Two project examples in Chapter 10 illustrate the use of the external memory interface (EMIF) with light-emitting diodes (LEDs) and liquid-crystal displays (LCDs) for spectrum display.

The DSK board includes 16 MB (megabytes) of synchronous dynamic random access memory (SDRAM) and 256 kB (kilobytes) of flash memory. Four connectors on the board provide input and output: MIC IN for microphone input, LINE IN for line input, LINE OUT for line output, and HEADPHONE for a headphone output (multiplexed with line output). The status of the four user dip switches on the DSK board can be read from a program and provides the user with a feedback control interface. The DSK operates at 225 MHz. Also onboard the DSK are voltage



(a)



(b)

FIGURE 1.1. TMS320C6713-based DSK board: (a) board; (b) diagram. (Courtesy of Texas Instruments)

regulators that provide 1.26 V for the C6713 core and 3.3 V for its memory and peripherals.

Appendix H illustrates a DSK based on the fixed-point processor C6416.

1.2.2 TMS320C6713 Digital Signal Processor

The TMS320C6713 (C6713) is based on the VLIW architecture, which is very well suited for numerically intensive algorithms. The internal program memory is structured so that a total of eight instructions can be fetched every cycle. For example, with a clock rate of 225 MHz, the C6713 is capable of fetching eight 32-bit instructions every $1/(225\text{ MHz})$ or 4.44 ns.

Features of the C6713 include 264 kB of internal memory (8 kB as L1P and L1D Cache and 256 kB as L2 memory shared between program and data space), eight functional or execution units composed of six arithmetic-logic units (ALUs) and two multiplier units, a 32-bit address bus to address 4 GB (gigabytes), and two sets of 32-bit general-purpose registers.

The C67xx (such as the C6701, C6711, and C6713) belong to the family of the C6x floating-point processors, whereas the C62xx and C64xx belong to the family of the C6x fixed-point processors. The C6713 is capable of both fixed- and floating-point processing. The architecture and instruction set of the C6713 are discussed in Chapter 3.

1.3 CODE COMPOSER STUDIO

CCS provides an IDE to incorporate the software tools. CCS includes tools for code generation, such as a C compiler, an assembler, and a linker. It has graphical capabilities and supports real-time debugging. It provides an easy-to-use software tool to build and debug programs.

The C compiler compiles a C source program with extension *.c* to produce an assembly source file with extension *.asm*. The assembler assembles an *.asm* source file to produce a machine language object file with extension *.obj*. The linker combines object files and object libraries as input to produce an executable file with extension *.out*. This executable file represents a linked common object file format (COFF), popular in Unix-based systems and adopted by several makers of digital signal processors [25]. This executable file can be loaded and run directly on the C6713 processor. Chapter 3 introduces the linear assembly source file with extension *.sa*, which is a cross between C and assembly code. A linear optimizer optimizes this source file to create an assembly file with extension *.asm* (similar to the task of the C compiler).

To create an application project, one can “add” the appropriate files to the project. Compiler/linker options can readily be specified. A number of debugging features are available, including setting breakpoints and watching variables; viewing memory, registers, and mixed C and assembly code; graphing results; and monitor-

ing execution time. One can step through a program in different ways (step into, over, or out).

Real-time analysis can be performed using real-time data exchange (RTDX) (Chapter 9). RTDX allows for data exchange between the host PC and the target DSK, as well as analysis in real time without stopping the target. Key statistics and performance can be monitored in real time. Through the joint team action group (JTAG), communication with on-chip emulation support occurs to control and monitor program execution. The C6713 DSK board includes a JTAG interface through the USB port.

1.3.1 CCS Installation and Support

Use the USB cable to connect the DSK board to the USB port on the PC. Use the 5-V power supply included with the DSK package to connect to the +5-V power connector on the DSK to turn it on. Install CCS with the CD-ROM included with the DSK, preferably using the `c:\C6713` structure (in lieu of `c:\ti` as the default).

The CCS icon should be on the desktop as “C6713DSK CCS” and is used to launch CCS. The code generation tools (C compiler, assembler, linker) are used with CCS version 2.x.

CCS provides useful documentations included with the DSK package on the following (see the Help icon):

1. Code generation tools (compiler, assembler, linker, etc.)
2. Tutorials on CCS, compiler, RTDX
3. DSP instructions and registers
4. Tools on RTDX, DSP/basic input/output system (DSP/BIOS), and so on.

An extensive amount of support material (*pdf* files) is included with CCS. There are also examples included with CCS within the folder `c:\C6713\examples`. They illustrate the board and chip support library files, DSP/BIOS, and so on. CCS Version 2.x was used to build and test the examples included in this book. A number of files included in the following subfolders/directories within `c:\C6713` (suggested structure during CCS installation) can be very useful:

1. *myprojects*: a folder supplied only for your projects. All the folders in the accompanying book CD should be placed within this subdirectory.
2. *bin*: contains many utilities.
3. *docs*: contains documentation and manuals.
4. *c6000\cgtools*: contains code generation tools.
5. *c6000\RTDX*: contains support files for real-time data transfer.
6. *c6000\bios*: contains support files for DSP/BIOS.
7. *examples*: contains examples included with CCS.
8. *tutorial*: contains additional examples supplied with CCS.

Note that all the folders containing the programs and support files in the accompanying book CD should be transferred to the subdirectory `myprojects`. Change the properties of all the files included so that they are not read-only (all the folders can be highlighted to change the properties of their contents at once).

1.3.2 Useful Types of Files

You will be working with a number of files with different extensions. They include:

1. `file.pjt`: to create and build a project named file
2. `file.c`: C source program
3. `file.asm`: assembly source program created by the user, by the C compiler, or by the linear optimizer
4. `file.sa`: linear assembly source program. The linear optimizer uses `file.sa` as input to produce an assembly program `file.asm`
5. `file.h`: header support file
6. `file.lib`: library file, such as the run-time support library file `rts6700.lib`
7. `file.cmd`: linker command file that maps sections to memory
8. `file.obj`: object file created by the assembler
9. `file.out`: executable file created by the linker to be loaded and run on the C6713 processor
10. `file.cdb`: configuration file when using DSP/BIOS

1.4 QUICK TEST OF DSK

1. On power, a program `post.c` (Power On Self Test), stored in onboard flash memory, uses the board support library (BSL) to test the DSK. It tests the internal, external, and flash memories, the two multichannel buffered serial ports (McBSP), direct memory access (DMA), the onboard codec, and the LEDs. If all tests are successful, all four LEDs blink three times and stop (with all LEDs on). During the testing of the codec, a 1-kHz tone is generated for 1 sec.
2. Launch CCS from the icon on the desktop. A USB enumeration process takes place. Then CCS will be opened and the LEDs will turn off. Press GEL → Check DSK → Quick Test. The Quick Test can be used for confirmation of correct operation and installation. The following message is then displayed:

Switches: 15

Board Revision: 1

CPLD Revision: 2

This assumes that the four dip switches (0, 1, 2, 3) are all in the up position. Change the switches to $(1110)_2$ so that the first three switches (0, 1, 2) are up and press the

fourth switch (3) down. Repeat the procedure to select GEL → Check DSK → Quick Test and verify that the value of the switches is now 7 (with the display “Switches: 7”). You can set the value of the four user switches from 0 to 15. Within your program you can then direct the execution of your code based on these 16 values.

Alternative Quick Test of DSK

1. Open/launch CCS from the icon on the desktop if this has not been done already. Select File → Load Program. Click on the folder *sine8_LED\Debug* within myprojects to load the file *sine8_LED.out*. This loads the executable file *sine8_LED.out* into the C6713 processor. This assumes that you have already copied all the folders on the accompanying CD into your folder: *c:\c6713\myprojects*.
2. Select Debug → Run. Press the dip switch #0, which should light LED #0 on and generate a 1-kHz tone. Connect the LINE OUT (or the HEADPHONE) on the DSK board to a speaker or to an oscilloscope and verify the generation of the 1-kHz tone. The four connectors on the DSK board for I/O (MIC, LINE IN, LINE OUT, and HEADPHONE) use a 3.5-mm jack audio cable.

1.5 SUPPORT FILES

The following support files located in the folder *support* (except the library files) are used for most of the examples and projects discussed in this book:

1. *C6713dskinit.c*: contains functions to initialize the DSK, the codec, the serial ports, and for I/O. It is not included with CCS.
2. *C6713dskinit.h*: header file with function prototypes. Features such as those used to select the mic input in lieu of line input (by default), input gain, and so on are obtained from this header file (modified from a similar file included with CCS).
3. *C6713dsk.cmd*: sample linker command file. This generic file can be changed when using external memory in lieu of internal memory.
4. *Vectors_intr.asm*: a modified version of a vector file included with CCS to handle interrupts. Twelve interrupts, INT4 through INT15, are available, and INT11 is selected within this vector file. They are used for interrupt-driven programs.
5. *Vectors_poll.asm*: vector file for programs using polling.
6. *rts6700.lib*, *dsk6713bsl.lib*, *cs16713.lib*: run-time, board, and chip support library files, respectively. These files are included with CCS and are located in *C6000\cgtools\lib*, *C6000\dsk6713\lib*, and *c6000\bios\lib*, respectively.

1.6 PROGRAMMING EXAMPLES TO TEST THE DSK TOOLS

Three programming examples are introduced to illustrate some of the features of CCS and the DSK board. The primary focus is to become familiar with both the software and hardware tools. It is strongly suggested that you complete these three examples before proceeding to subsequent chapters.

Example 1.1: Sine Generation Using Eight Points with DIP Switch Control (*sine8_LED*)

This example generates a sinusoid using a table lookup method. More important, it illustrates some features of CCS for editing, building a project, accessing the code generation tools, and running a program on the C6713 processor. The C source program *sine8_LED.c* shown in Figure 1.2 implements the sine generation and is included in the folder *sine8_LED*.

Program Consideration

Although the purpose is to illustrate some of the tools, it is useful to understand the program *sine8_LED.c*. A table or buffer *sine_table* is created and filled with eight points representing $\sin(t)$, where $t = 0, 45, 90, 135, 180, 225, 270$, and 315 degrees

```
//Sine8_LED.c Sine generation with DIP switch control

#include "dsk6713_aic23.h"           //support file for codec,DSK
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
short loop = 0;                     //table index
short gain = 10;                    //gain factor
short sine_table[8]={0,707,1000,707,0,-707,-1000,-707}; //sine values

void main()
{
    comm_poll();                     //init DSK, codec, McBSP
    DSK6713_LED_init();               //init LED from BSL
    DSK6713_DIP_init();               //init DIP from BSL
    while(1)                          //infinite loop
    {
        if(DSK6713_DIP_get(0)==0)    //==0 if switch #0 pressed
        {
            DSK6713_LED_on(0);        //turn LED #0 ON
            output_sample(sine_table[loop]*gain); //output every Ts (SW0 on)
            if (++loop > 7) loop = 0;  //check for end of table
        }
        else DSK6713_LED_off(0);      //LED #0 off
    }                                  //end of while (1)
}                                     //end of main
```

FIGURE 1.2. Sine generation program using eight points with dip switch control (*sine8_LED.c*).

(scaled by 1000). Within the function *main*, another function, *comm_poll*, is called that is located in the communication and initialization support file *c6713dskinit.c*. It initializes the DSK, the AIC23 codec onboard the DSK, and the two McBSPs on the C6713 processor. Within *c6713dskinit.c*, the function *DSK6713_init* initializes the BSL file, which must be called before the two subsequent BSL functions, *DSK6713_LED_init* and *DSK6713_DIP_init*, are invoked that initialize the four LEDs and the four dip switches.

The statement `while (1)` within the function *main* creates an infinite loop. When dip switch #0 is pressed, LED #0 turns on and the sinusoid is generated. Otherwise, *DSK6713_DIP_get(0)* will be false (true if the switch is pressed) and LED #0 will be off.

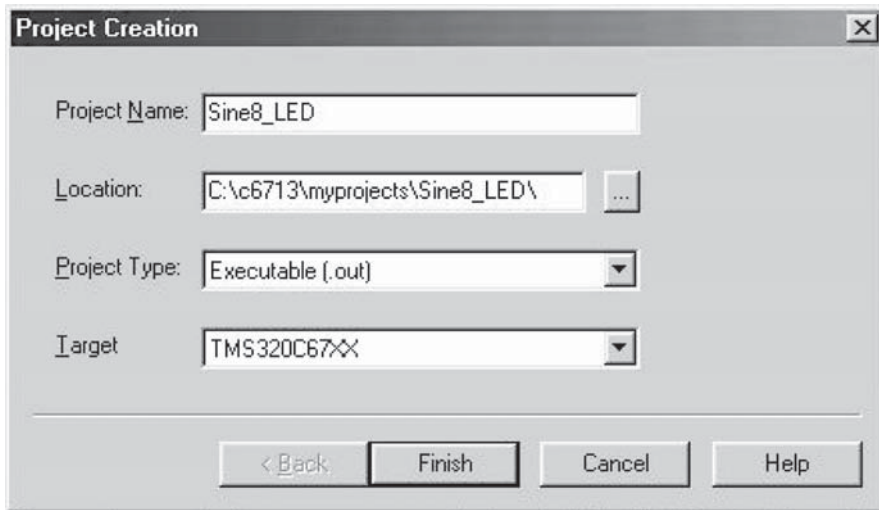
The function *output_sample*, located in the communication support file *C6713dskinit.c*, is called to output the first data value in the buffer or table *sine_table[0] = 0*. The loop index is incremented until the end of the table is reached, after which it is reinitialized to zero.

Every sample period $T = 1/F_s = 1/8000 = 0.125$ ms, the value of dip switch #0 is tested, and a subsequent data value in *sine_table* (scaled by *gain* = 10) is sent for output. Within one period, eight data values (0.125 ms apart) are output to generate a sinusoidal signal. The period of the output signal is $T = 8(0.125 \text{ ms}) = 1$ ms, corresponding to a frequency of $f = 1/T = 1$ kHz.

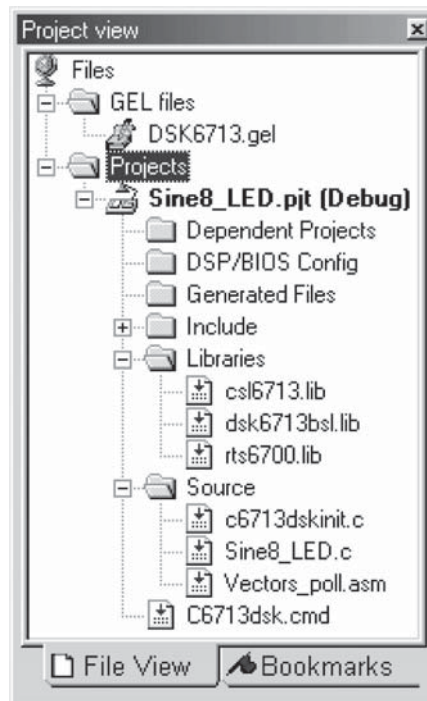
Create Project

In this section we illustrate how to create a project, adding the necessary files for building the project **sine8_LED**. Back up the folder *sine8_LED* (change its name) or delete its content (which can be retrieved from the book CD if needed), keeping only the C source file *sine8_LED.c* and the file *gain.gel* in order to recreate the content of that folder. Access CCS (from the desktop).

1. To create the project file *sine8_LED.pjt*. Select Project → New. Type *sine8_LED* for the project name, as shown in Figure 1.3. This project file is saved in the folder *sine8_LED* (within `c:\c6713\myprojects`). The .pjt file stores project information on build options, source filenames, and dependencies.
2. To add files to the project. Select Project → Add Files to Project. Look in the folder *support*, Files of type C Source Files. Double-click on the C source file *C6713dskinit.c* to add it to the project. Click on the “+” symbol to the left of the Project Files window within CCS to expand and verify that this C source file has been added to the project.
3. Repeat step 2, use the pull-down menu for Files of type, and select ASM Source Files. Double-click on the assembly source vector file *vectors_poll.asm* to add it to the project. Repeat again and select Files of type: Linker Command File, and add *C6713dsk.cmd* to the project.



(a)



(b)

FIGURE 1.3. CCS Project windows for sine8_LED: (a) project creation; (b) project view files.

4. To add the library support files to the project. Repeat the previous step, but select files of type: Object and Library Files. Look in `c:\c6713\c6000\cgtools\lib` and select the run-time support library file `rts6700.lib` (which supports the C67x architecture) to add to the project. Continue this process to add the BSL file `dsk6713bs1.lib` located in `c:\c6713\c6000\dsk6713\lib`, and the chip support library (CSL) file `cs16713.lib` located in `c:\c6713\c6000\bios\lib`.
5. Verify from the Files window that the project (`.pjx`) file, the linker command (`.cmd`) file, the three library (`.lib`) files, the two C source (`.c`) files, and the assembly (`.asm`) file have been added to the project. The GEL file `dsk6713.gel` is added automatically when you create the project. It initializes the C6713 DSK invoking the BSL to use the phase-locked loop (PLL) to set the central processing unit (CPU) clock to 225 MHz (otherwise, the C6713 runs at 50 MHz by default).
6. Note that there are no “include” files yet. Select Project → Scan All File Dependencies. This adds/includes the header files `c6713dskinit.h`, along with several board and chip support header files included with CCS.

The Files window in CCS should look as in Figure 1.3b. Any of the files (except the library files) from CCS’s Files window can be displayed by clicking on it. You should not add header or include files to the project. They are added to the project automatically when you select: Scan All File Dependencies. (They are also added when you build the project.)

It is also possible to add files to a project simply by “dragging” the file (from a different window) and dropping it into the CCS Project window.

Code Generation and Options

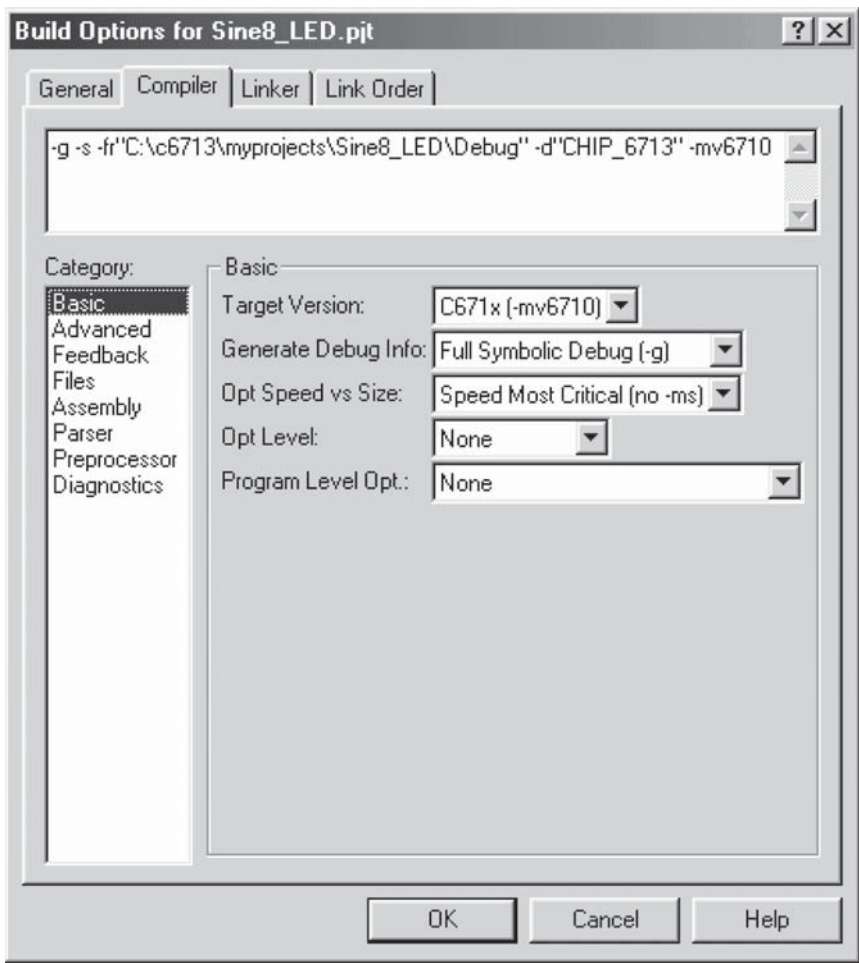
Various options are associated with the code generation tools: C compiler and linker to build a project.

Compiler Option

Select Project → Build Options. Figure 1.4a shows the CCS window Build Options for the compiler. Select the following for the compiler option with Basic (for Category): (1) `c671x{-mv6710}` (for Target Version), (2) Full Symbolic Debug (for Generate Debug Info), (3) Speed most critical (for Opt Speed vs. Size), and (4) None (for Opt Level and Program Level Opt). Select the Preprocessor Category and type for Define Symbols[d]: `CHIP_6713`, and from the Feedback Category, select for Interlisting: `OPT/C` and `ASM{-s}`. The resulting compiler option is

```
-g -s
```

The `-g` option is used to enable symbolic debugging information, useful during the debugging process, and is used in conjunction with the option `-s` to interlist the C



(a)

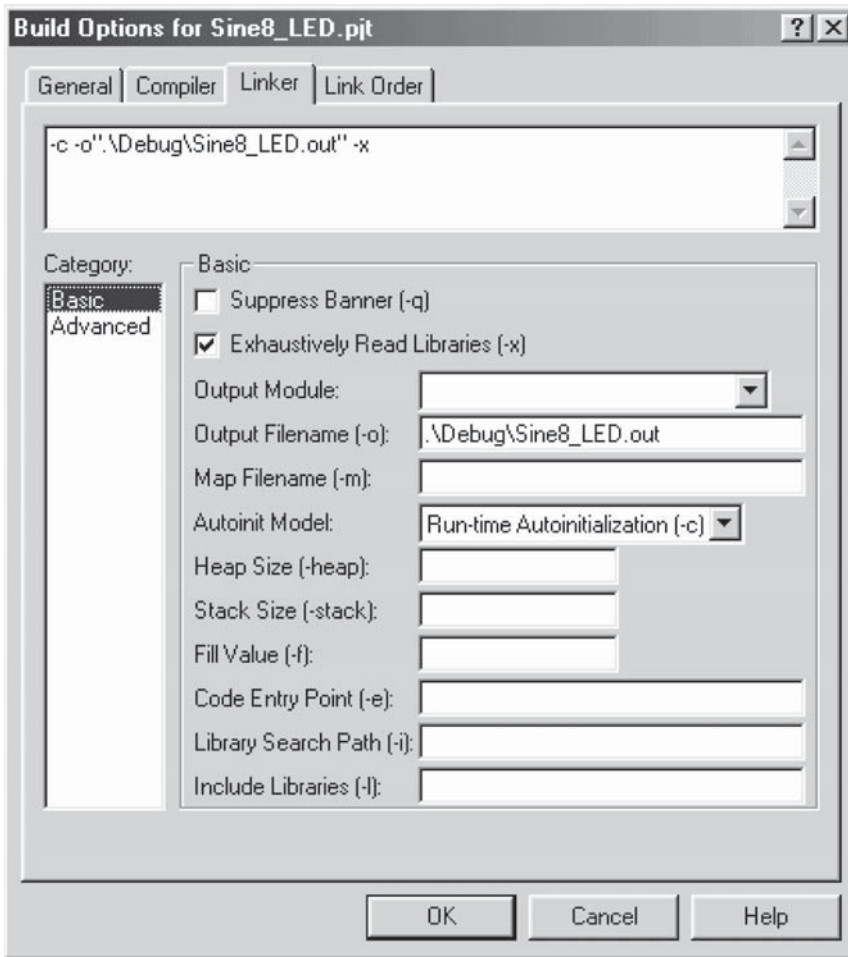
FIGURE 1.4. CCS Build options: (a) compiler; (b) linker.

source file with the assembly source file *sine8_LED.asm* generated (an additional option, `-k`, can be used to retain the assembly source file). The `-g` option disables many code optimizations to facilitate the debugging process. Press OK.

Selecting C621x or C64xx for Target Version invokes a fixed-point implementation. The C6713-based DSK can use either fixed- or floating-point processing. Most examples implemented in this book can run using fixed-point processing. Selecting C671x as Target Version invokes a floating-point implementation.

If No Debug is selected (for Generate Debug Info) and `-o3:File` is selected (for Opt Level), the Compiler option is automatically changed to

`-s -o3`



(b)

FIGURE 1.4. (Continued)

The `-o3` option invokes the highest level of optimization for performance or execution speed. For now, speed is not critical (neither is debugging). Use the compiler options `-gs` (which you can also type directly in the compiler command window). Initially, one would not optimize for speed but to facilitate debugging. A number of compiler options are described in Ref. 28.

Linker Option

Click on Linker (from CCS Build Options). The output filename `sine8_LED.out` defaults to the name of the `.pjt` filename, and Run-time Autoinitialization defaults for Autoinit Model. The linker option should be displayed as in Figure 1.4b. The map file can provide useful information for debugging (memory locations of func-

tions, etc.). The `-c` option is used to initialize variables at run time, and the `-o` option is used to name the linked executable output file `sine8_LED.out`. Press OK.

Note that you can/should choose to store the executable file in the subfolder “Debug,” within the folder `sine8_LED`, especially during the debugging stage of a project.

Again, these various compiler and linker options can be typed directly within the appropriate command windows.

In lieu of adding the three library files to the project by retrieving them from their specific locations, it is more convenient to add them within the linker option window Include Libraries{-l}, typing them directly, separated by a comma. However, they will not be shown in the Files window.

Building and Running the Project

The project `sine8_LED` can now be built and run.

1. Build this project as **sine8_LED**. Select Project → Rebuild All or press the toolbar with the three down arrows. This compiles and assembles all the C files using `cl6x` and assembles the assembly file `vectors_poll.asm` using `asm6x`. The resulting object files are then linked with the library files using `lnk6x`. This creates an executable file `sine8_LED.out` that can be loaded into the C6713 processor and run. Note that the commands for compiling, assembling, and linking are performed with the Build option. A log file `cc_build_Debug.log` is created that shows the files that are compiled and assembled, along with the compiler options selected. It also lists the support functions that are used. Figure 1.5 shows several windows within CCS for the project `sine8_LED`. The building process causes all the dependent files to be included (in case one forgets to scan for all the file dependencies).
2. Select File → Load Program in order to load `sine_LED.out` by clicking on it (CCS includes an option to load the program automatically after a build). It should be in the folder `sine8_LED\Debug`. Select Debug → Run or use the toolbar with the “running man.” Connect a speaker to the LINE OUT connector on the DSK. Press the dip switch #0. You should hear a tone. You can also use the headphone output at the same time.

The sampling rate F_s of the codec is set at 8 kHz. The frequency generated is $f = F_s/(\text{number of points}) = 8\text{ kHz}/8 = 1\text{ kHz}$. Connect the output of the DSK to an oscilloscope to verify a 1-kHz sinusoidal signal with an approximate amplitude of 0.8 V p-p (peak to peak).

Correcting Program Errors

1. Delete the semicolon in the statement

```
short gain = 10;
```

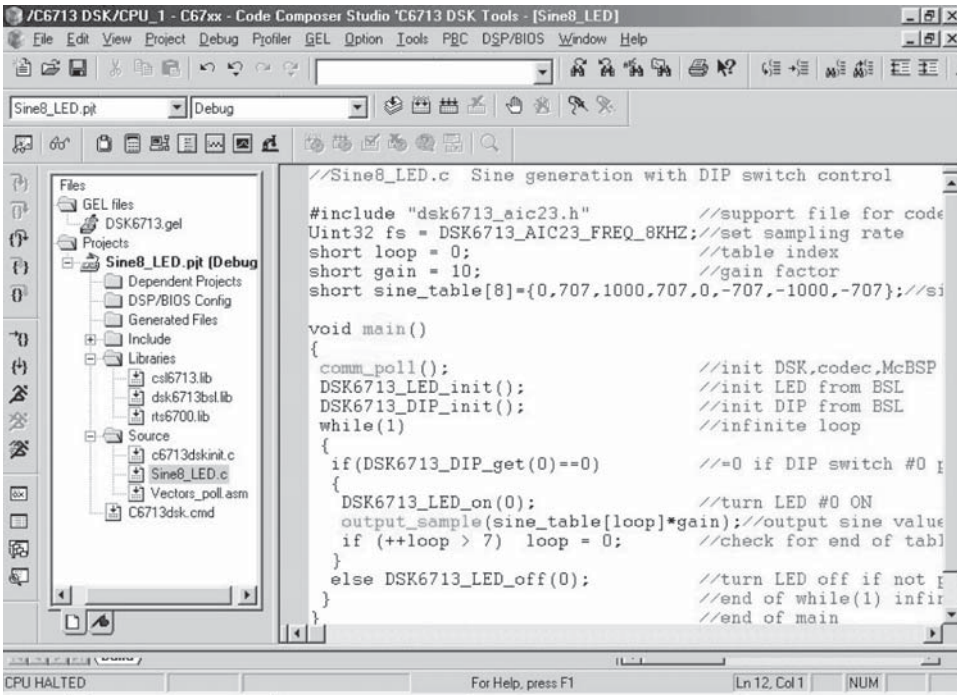



FIGURE 1.5. CCS windows for project sine8_LED.

in the C source file `sine8_LED.c`. If it is not displayed, double-click on it (from the Files window).

2. Select Project → Build to perform an incremental build or use the toolbar with the two (not three) arrows. The incremental build is chosen so that only the C source file `sine8_LED.c` is compiled. With the Rebuild option (toolbar with three arrows), files compiled and/or assembled previously would again go through this unnecessary process.
3. An error message, highlighted in red, stating that a “;” is expected, should appear in the Build window of CCS (lower left). You may need to scroll up the Build window for a better display of this error message. Double-click on the highlighted error message line. This should bring the cursor to the section of code where the error occurs. Make the appropriate correction, Build again, load, and run the program to verify your previous results.

Monitoring the Watch Window

Verify that the processor is still running (and dip switch #0 is pressed). Note the indicator “DSP RUNNING” at the bottom left of CCS. The Watch window allows you to change the value of a parameter or to monitor a variable:

1. Select View → Quick Watch window, which should be displayed on the lower section of CCS. Type *gain*, then click on “Add to Watch.” The gain value of 10 set in the program in Figure 1.2 should appear in the Watch window.
2. Change *gain* from 10 to 30 in the Watch window. Press Enter. Verify that the volume of the generated tone has increased (with the processor still running and dip switch #0 is pressed). The amplitude of the sine wave has increased from approximately 0.8V p-p to approximately 2.5V p-p.
3. Change *gain* to 33 (as in step 2). Verify that a higher-pitched tone exists, which implies that the frequency of the sine wave has changed just by changing its amplitude. This is not so. You have exceeded the range of the codec AIC23. Since the values in the table are scaled by 33, the range of these values is now between $\pm 33,000$. The range of output values is limited from -2^{15} to $(2^{15} - 1)$, or from $-32,768$ to $+32,767$.

Since the AIC23 is a stereo codec, we can send data to both 16-bit channels within each sampling period. This is introduced in Chapter 2. This can be useful to experiment with the stereo effects of output signals. In Chapter 7, we use both channels for adaptive filtering where it is necessary to input one type of signal (such as noise) on one 16-bit channel and another signal (such as a desired signal) on the other 16-bit channel. In this book, we will mostly use the codec as a mono device without the need to use an adapter that is required when using both channels.

Applying the Slider Gel File

The General Extension Language (GEL) is an interpretive language similar to (a subset of) C. It allows you to change a variable such as *gain*, sliding through different values while the processor is running. All variables must first be defined in your source program.

1. Select File → Load GEL and open the file *gain.gel*, which you retained from the original folder, *sine8_LED* (that you backed up). Double-click on the file *gain.gel* to view it within CCS. It should be displayed in the Files window. This file is shown in Figure 1.6. By creating the slider function *gain* shown in Figure 1.6, you can start with an initial value of 10 (first value) for the variable *gain* that is set in the C program, up to a value of 35 (second value), incremented by 5 (third value).
2. Select GEL → Sine Gain → Gain. This should bring out the Slider window shown in Figure 1.7, with the minimum value of 10 set for the gain.
3. Press the up-arrow key to increase the gain value from 10 to 15, as displayed in the Slider window. Verify that the volume of the sine wave generated has increased. Press the up-arrow key again to continue increasing the slider, incrementing by 5 up to 30. The amplitude of the sine wave should be about 2.5V p-p with a *gain* value set at 30. Now use the mouse to click directly on the Slider window and slowly increase the slider position to 31, then 32, and

```

/*gain.gel Create slider and vary amplitude (gain) of sinewave*/

menuitem "Sine Gain"

slider Gain(10,35,5,1,gain_parameter) /*incr by 5, up to 35*/
{
    gain = gain_parameter;          /*vary gain of sine*/
}

```

FIGURE 1.6. GEL file to slide through different gain values in the sine generation program (gain.gel).

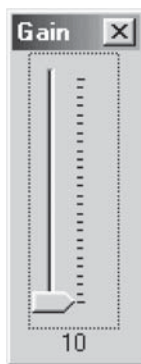


FIGURE 1.7. Slider window for varying the gain of generated sine wave.

verify that the frequency generated is still 1 kHz. Increase the slider to 33 and verify that you are no longer generating a 1-kHz sine wave. The table values, scaled by the gain value, are now between $\pm 33,000$ (beyond the acceptable range by the codec).

Changing the Frequency of the Generated Sinusoid

1. Change the sampling frequency from 8 to 16 kHz by setting f_s in the C source program to `DSK6713_AIC23_FREQ_16KHZ`. Rebuild (use incremental build) the project, load and run the new executable file, and verify that the frequency of the generated sinusoid is 2 kHz. The sampling frequencies supported by the AIC23 codec are 8, 16, 24, 32, 44.1, 48, and 96 kHz.
2. Change the number of points in the lookup table to four points in lieu of eight points—for example, `{0, 1000, 0, -1000}`. The size of the array `sine_table` and the loop index also need to be changed. Verify that the generated frequency is $f = F_s / (\text{number of points})$.

Note that the sinusoid is no longer generated if the dip switch #0 is not pressed. If a different dip switch such as switch #3 is desired (in lieu of switch #0), the BSL functions `DSK6713_DIP_get(3)`, `DSK6713_LED_on(3)`, and `DSK6713_LED_off(3)` can be substituted in the C source program.

Two sliders can readily be used, one to change the gain and the other to change the frequency. A different signal frequency can be generated by changing the loop index within the C program (e.g., stepping through every two points in the table). When you exit CCS after you build a project, all changes made to the project can be saved. You can later return to the project with the status as you left it before. For example, when returning to the project after launching CCS, select Project → Open to open an existing project such as `sine8_LED.pjt` (with all the necessary files for the project already added).

Example 1.2: Generation of the Sinusoid and Plotting with CCS (*sine8_buf*)

This example generates a sinusoid with eight points, as in Example 1.1. More important, it illustrates CCS capabilities for plotting in both time and frequency domains. The program `sine8_buf.c`, shown in Figure 1.8, implements this project. This program creates a buffer to store the output data in memory.

Create this project as `sine8_buf.pjt`, and add the necessary files to the project, as in Example 1.1 (use the C source program `sine8_buf.c` in lieu of `sine8_LED.c`). Note that the necessary header support files are added to the project by selecting Project → Scan All File Dependencies. The necessary

```
//sine8_buf Sine generation. Output buffer plotted within CCS

#include "dsk6713_aic23.h"                //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;      //set sampling rate
int loop = 0;                            //table index
short gain = 10;                          //gain factor
short sine_table[8]={0,707,1000,707,0,-707,-1000,-707}; //sine values
short out_buffer[256];                   //output buffer
const short BUFFERLENGTH = 256;         //size of output buffer
int i = 0;                                //for buffer count

interrupt void c_int11()                  //interrupt service routine
{
    output_sample(sine_table[loop]*gain); //output sine values
    out_buffer[i] = sine_table[loop]*gain; //output to buffer
    i++;                                  //increment buffer count
    if(i==BUFFERLENGTH) i=0;             //if @ bottom reinit count
    if (++loop > 7) loop = 0;             //check for end of table
    return;                               //return from interrupt
}

void main()
{
    comm_intr();                          //init DSK, codec, McBSP
    while(1);                             //infinite loop
}
```

FIGURE 1.8. Sine generation with output stored in memory as well (`sine8_buf.c`).

support files for this project, `c6713dskinit.c`, `vectors_intr.asm` and `C6713dsk.cmd`, are in the folder `support`, and the three library support files can be added using Project → Build Options and selecting the linker option (Include Libraries). Type them, separating each by a comma. Note that since this program is interrupt-driven (in lieu of polling), the vector file `vectors_intr.asm` (in lieu of `vectors_poll.asm`) is added to the project.

Within the function `main`, `comm_intr` (in lieu of `comm_poll` in Example 1.1) is called. This function resides in `c6713dskinit.c` to support interrupt-driven programs. The statement `while(1)` within the function `main` creates an infinite loop to wait for an interrupt to occur. On interrupt, execution proceeds to the interrupt service routine (ISR) `c_int11`. This ISR address is specified in the file `vectors_intr.asm` with a branch instruction to this address, using interrupt INT11. Interrupts are discussed in more detail in Chapter 3.

Within the ISR, the function `output_sample`, located in the communication and initialization file `c6713dskinit.c`, is called to output the first data value in `sine_table`. The loop index is incremented until the end of the table is reached; after that, it is reinitialized to zero. An output buffer is created to capture a total of 256 (specified by `BUFFERLENGTH`) sine data values. Execution returns from ISR to the `while(1)` infinite loop to wait for each subsequent interrupt.

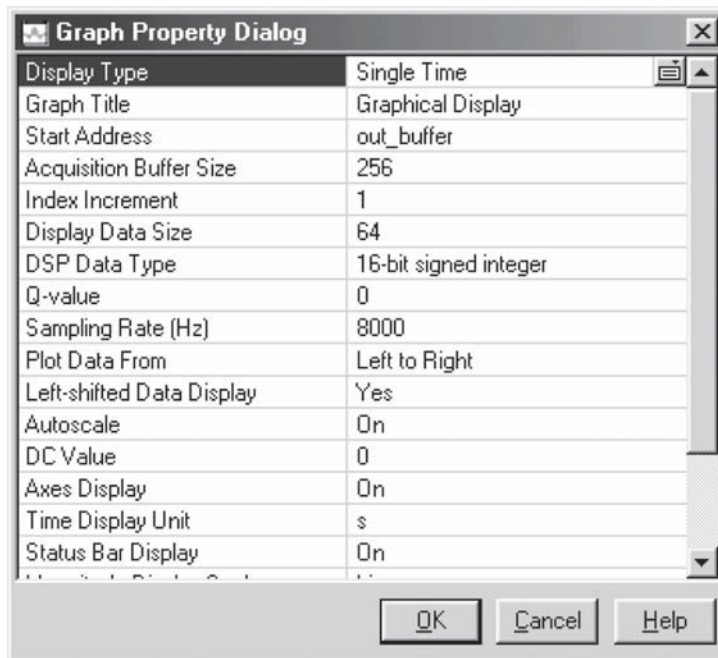
Build this project as `sine8_buf`. Load and run the executable file `sine8_buf.out` and verify that a 1-kHz sinusoid is generated with the output connected to a speaker or a scope (as in Example 1.1).

Plotting with CCS

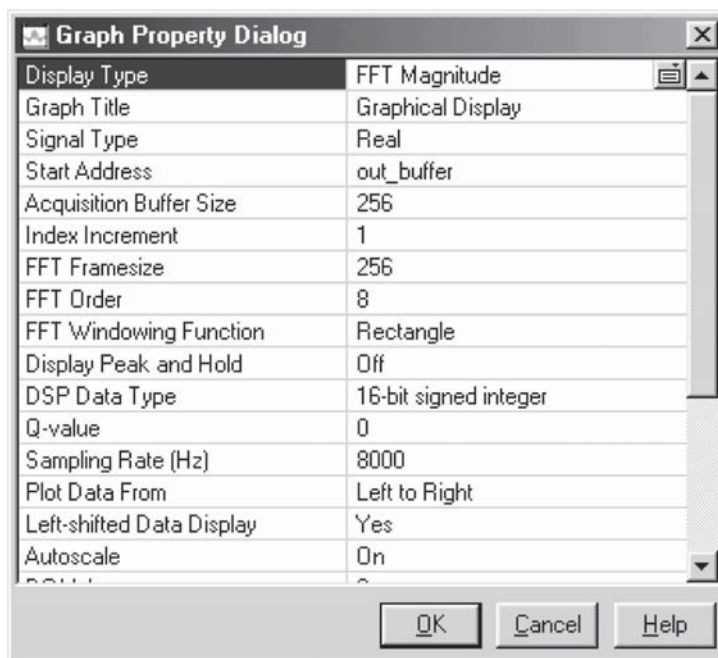
The output buffer is being updated continuously every 256 points (you can readily change the buffer size). Use CCS to plot the current output data stored in the buffer `out_buffer`.

1. Select View → Graph → Time/Frequency. Change the Graph Property Dialog so that the options in Figure 1.9a are selected for a time-domain plot (use the pull-down menu when appropriate). The starting address of the output buffer is `out_buffer`. The other options can be left as default. Figure 1.10 shows a time-domain plot of the sinusoidal signal within CCS.
2. Figure 1.9b shows CCS's Graph Property Display for a frequency-domain plot. Choose a fast Fourier transform (FFT) order so that the frame size is 2^{order} . Press OK and verify that the FFT magnitude plot is as shown in Figure 1.10. The spike at 1000 Hz represents the frequency of the sinusoid generated.

You can obtain many different windows within CCS. From the Build window, right-click and select Float In Main Window. To change the screen size, right-click on the Build window and deselect Allow Docking. For example, you can get the time-domain plot (separated). Right-click on the time-domain plot, select Float In Main Window, and again right-click on the same time-domain plot window and deselect Allow Docking. You can then move it.



(a)



(b)

FIGURE 1.9. CCS Graph Property Dialog for `sine8_buf`: (a) for time-domain plot; (b) for frequency-domain plot.

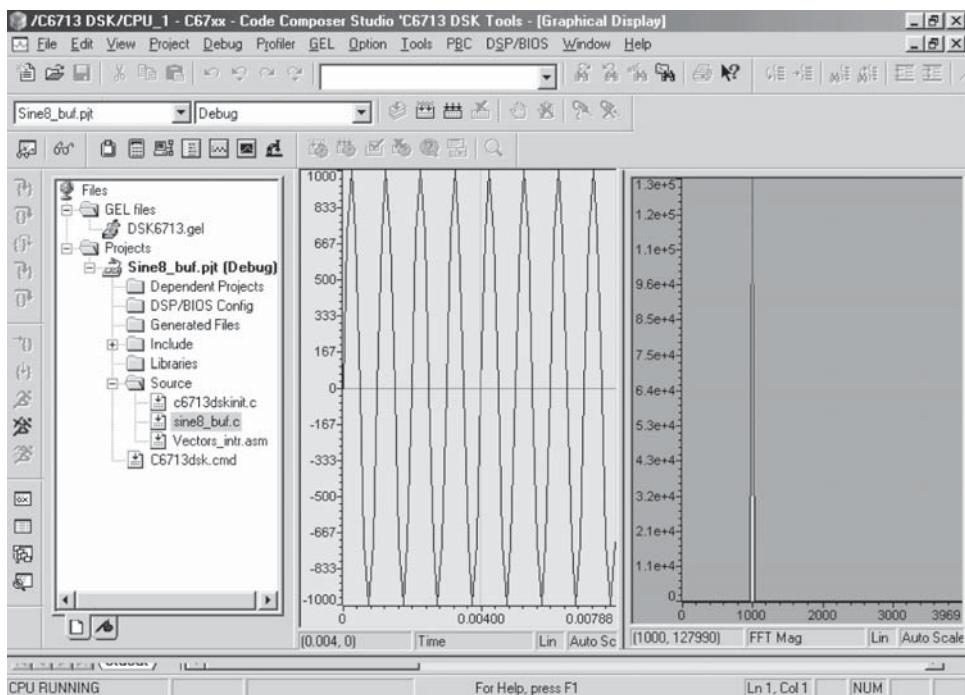


FIGURE 1.10. CCS windows for sine8_buf showing both time- and frequency-domain plots of a generated 1-kHz sine wave.

Viewing and Saving Data from Memory in a File

To view the content of that buffer, select View → Memory and specify out_buffer for the address, and select the 16-bit signed integer (or hex, etc.) for the format.

To save the content of the output buffer in a file, select File → Data → Save. Save the file as sine8_buf.dat (as type hex, for example) in the folder sine8_buf. From the Storing Memory window, use out_buffer as the buffer's address with length 256. You can then plot this data [with MATLAB for example] and verify the 1-kHz sinusoidal waveform (with 8 kHz as the sampling rate).

Example 1.3: Dot Product of Two Arrays (dotp4)

Operations such as addition/subtraction and multiplication are the key operations in a DSP. A very important operation is multiply/accumulate, which is useful in a number of applications requiring digital filtering, correlation, and spectrum analysis. Since the multiplication operation is executed commonly and is essential for most DSP algorithms, it is important that it executes in a single cycle. With the C6713 we can actually perform two multiply/accumulate operations within a single cycle.

This example illustrates additional features of CCS, such as single-stepping, setting breakpoints, and profiling for the benchmark. Again, the purpose here is to


```
//Dotp4.c Multiplies two arrays, each array with 4 numbers

int dotp(short *a,short *b,int ncount);//function prototype
#include <stdio.h>                      //for printf
#include "dotp4.h"                      //header file with data
#define count 4                        // # data in each array
short x[count] = {x_array};           //declaration of 1st array
short y[count] = {y_array};           //declaration of 2nd array

main()
{
    int result = 0;                    //result sum of products

    result = dotp(x, y, count);        //call dotp function
    printf("result = %d (decimal) \n", result); //print result
}

int dotp(short *a,short *b,int ncount) //dot product function
{
    int sum = 0;                      //init sum
    int i;

    for (i = 0; i < ncount; i++)
        sum += a[i] * b[i];           //sum of products
    return(sum);                      //return sum as result
}
```

FIGURE 1.11. Sum-of-products program using C code (dotp4.c).

```
//dotp4.h Header file with two arrays of numbers

#define x_array 1,2,3,4

#define y_array 0,2,4,6
```

FIGURE 1.12. Header file with two arrays each with four numbers (dotp4.h).

become more familiar with the tools. We invoke C compiler optimization to see how performance or execution speed can be drastically increased.

The C source file `dotp4.c` in Figure 1.11 takes the sum of products of two arrays, each with four numbers, contained in the header file `dotp4.h` in Figure 1.12. The first array contains the four numbers 1, 2, 3, and 4, and the second array contains the four numbers 0, 2, 4, and 6. The sum of products is $(1 \times 0) + (2 \times 2) + (3 \times 4) + (4 \times 6) = 40$.

The program can be readily modified to handle a larger set of data. No real-time implementation is used in this example, and no real-time I/O support files are needed. The support functions for interrupts are not needed here.

Create this project as **dotp4** and add the following files to the project (see Example 1.1):

1. `dotp4.c`: C source file
2. `vectors_poll.asm`: vector file defining the entry address `c_int00`
3. `C6713dsk.cmd`: generic linker command file
4. `rts6700.lib`: library file

Do not add any “include” files using “Add Files to Project” since they are added by selecting Project → Scan All File Dependencies. The header file `stdio.h` is needed due to the `printf` statement in the program `dotp4.c` to print the result.

Implementing a Variable Watch

1. Select Project → Options with `-gs` as the compiler option and the default linker option with no optimization.
2. Rebuild All by selecting the toolbar with the three arrows (or select Project → Rebuild All). Load the executable file `dotp4.out` within the folder `dotp4\Debug`.
3. Select View → Quick Watch. Type `sum` to watch the variable `sum` and click on “Add to Watch.” The message “identifier not found” associated with `sum` is displayed (as Value) because this local variable does not exist yet.
4. Set a breakpoint at the line of code

```
sum += a[i] * b[i];
```

by placing the mouse cursor (clicking) on that line, then right-click and select the Toggle breakpoint. Or, preferably, with the cursor on that line of code (at the extreme left), double-click. A red circle to the left of that line of code should appear. (Note: placing the cursor on a line of code with a set breakpoint and double clicking will remove the breakpoint.)

5. Select Debug → Run (or use the “running man” toolbar). The program executes up to (excluding) the line of code with the set breakpoint. A yellow arrow will also point to that line of code.
6. Single-step using F8. Repeat or continue to single-step and observe/watch the variable `sum` in the Watch window change in value to 0, 4, 16, 40. Select Debug → Run and verify that the resulting value of `sum` is printed as

```
sum = 40 (decimal)
```

7. Note the `printf` statement in the C program `dotp4.c` for printing the result. This statement (while excellent for debugging) should be avoided after the debugging stage, since it takes over 6000 cycles to execute.

Animating

1. Select File → Reload Program to reload the executable file `dotp4.out`. Or, preferably, select Debug → Restart. Note that after the executable file is

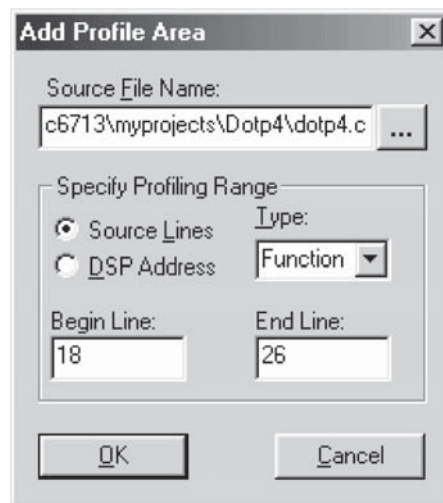
loaded, the entry address for execution is `c_int00`, as can be verified by the disassembled file.

2. The same breakpoint should be set already at the same line of code as before. Select **Debug** → **Animate** or use the equivalent toolbar in the left window (below the Halt running man). Observe the variable `sum` change in values through the Watch window. The speed of animation can be controlled by selecting **Option** → **Customize** → **Animate Speed** (the maximum speed is set to default at 0 second).

Benchmarking (Profiling) without Optimization

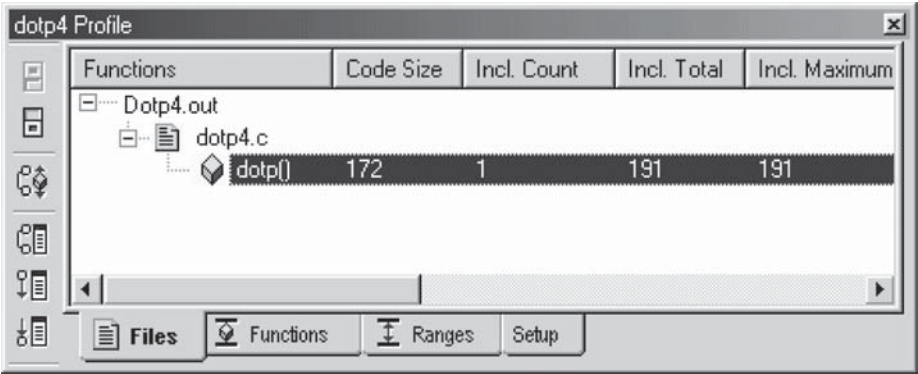
In this section we illustrate how to benchmark a section of code: in this case, the `dotp` function. Verify that the options for the compiler (`-g`) and linker (`-c -o dotp4.out`) are still set. To profile code, you must use the compiler option `-g` for symbolic debugging information. Remove any breakpoint by double-clicking on the line of code with the set breakpoint (or right-click and select the Toggle breakpoint).

1. Select **Debug** → **Restart**.
2. Select **Profiler** → **Start New Session** and enter `dotp4` as the Profile Session Name. Then press **OK**.
3. Click on the icon to “Create Profile Area” (see Figure 1.13a). This icon is the third icon from the bottom left in Figure 1.13b. Figure 1.13b shows the added profile area for the function `dotp` within the C source file `dotp4.c`.

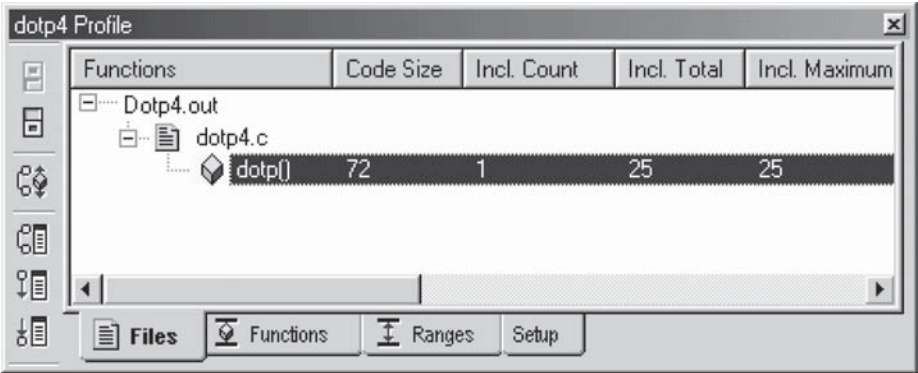


(a)

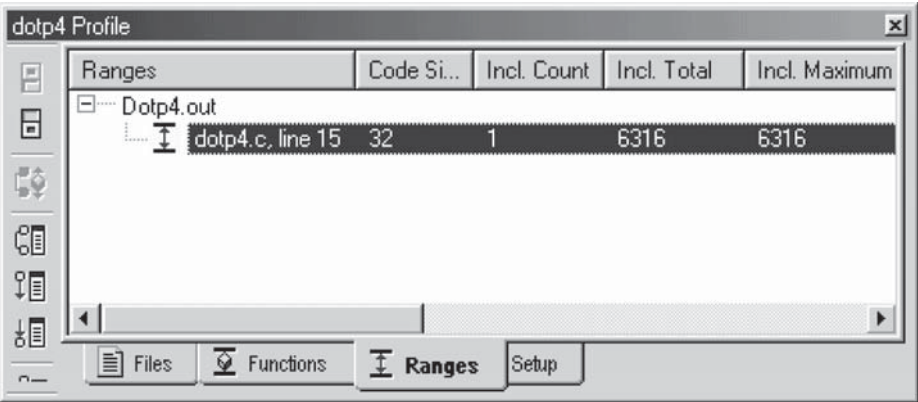
FIGURE 1.13. CCS display of project `dotp4` for profiling: (a) profile area for function `dotp`; (b) profiling function `dotp` with no optimization; (c) profiling function `dotp` with level 3 optimization; (d) profiling printf.



(b)



(c)



(d)

FIGURE 1.13. (Continued)

4. Run the program. Verify the results shown in Figure 1.13*b*. This indicates that it takes 191 cycles to execute the function `dotp` (with no optimization).

Benchmarking (Profiling) with Optimization

In this section we illustrate how to optimize the program using one of the optimization options, `-o3`. The program's execution speed can be increased using the optimizing C compiler. Change the compiler option (select Project → Build Options) to

```
-g -o3
```

and use the same linker options as before (you can type this option directly). The option `-o3` invokes the highest level of compiler optimization. Various compiler options are described in Ref. 28. Rebuild All (toolbar with three arrows) and load the executable file `dotp4.out` (or select File → Reload Program). Re-create the Profile Area as in Figure 1.13*a*.

Select Debug → Run. Verify that it takes now 25 cycles (from 191) to execute the `dotp` function, as shown in Figure 1.13*c*. This is a considerable improvement using the C compiler optimizer. The code size is reduced from 172 to 72. The dot product example can be also optimized using an intrinsic function or the code optimization techniques discussed in Chapter 8.

Profiling Printf

Again restart the program (Debug → Restart). Click on the icon *Ranges* at the bottom of the profile area. Highlight *printf* from the C source program, drag it to the profiling area window, and drop it by releasing the cursor. Verify that the code size of *printf* is 32 and that it takes 6316 cycles to execute, as shown in Figure 1.13*d*.

Note that in lieu of using Figure 1.13*a* to profile the function `dotp`, you can highlight it, drag it, and drop it with your mouse in the profiling area.

1.7 SUPPORT PROGRAMS/FILES CONSIDERATIONS

The following support files are used for practically all the examples in this book:

- (1) *c6713dskinit.c*, (2) *vectors_intr.asm* or *vectors_poll.asm*, and
- (3) *c6713dsk.cmd*. For now, the emphasis associated with these files should be on using them.

1.7.1 Initialization/Communication File (*c6713dskinit.c*)

Several BSL and CSL support functions are included in the initialization and communication (init/comm) file *c6713dskinit.c*. A partial listing is shown in Figure 1.14. It includes functions to initialize the DSK and provide for input and output.

```

//C6713dskinit.c Partial list of init/comm file.Includes CSL/BSL funct
...
void c6713_dsk_init()                //dsp-peripheral init
{
    DSK6713_init();                  //BSL to init DSK-EMIF,PLL
    hAIC23_handle=DSK6713_AIC23_openCodec(0, &config);//handle to codec
    DSK6713_AIC23_setFreq(hAIC23_handle, fs); //set sample rate
    MCBSP_config(DSK6713_AIC23_DATAHANDLE,&AIC23CfgData);//32bits interface
    MCBSP_start(DSK6713_AIC23_DATAHANDLE,MCBSP_XMIT_START | MCBSP_RCV_START
        | MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC,220); //start data channel
}

void comm_poll()                     //comm/init using polling
{
    poll = 1;                        //1 if using polling
    c6713_dsk_init();                //init DSP and codec
}

void comm_intr()                     //for comm/init using interrupt
{
    poll = 0;                        //0 since not polling
    IRQ_globalDisable();              //disable interrupts
    c6713_dsk_init();                //init DSP and codec
    CODECEventId=MCBSP_getXmtEventId(DSK6713_AIC23_codecdatahandle); //Xmit
    ...
    IRQ_setVecs(vectors);              //point to the IRQ vector
    IRQ_map(CODECEventId, 11);          //map McBSP1 Xmit to INT11
    IRQ_reset(CODECEventId);            //reset codec INT 11
    IRQ_globalEnable();                 //globally enable interrupts
    IRQ_nmiEnable();                   //enable NMI interrupt
    IRQ_enable(CODECEventId);           //enable CODEC eventXmit INT11
    output_sample(0);                  //start McBSP interrup out a sample
}

void output_sample(int out_data)      //out to Left and Right channels
{
    short CHANNEL_data;
    AIC_data.uint=0;                  //clear data structure
    AIC_data.uint=out_data;            //32-bit data -->data structure
    ...
    if(poll) while(!MCBSP_xrdy(DSK6713_AIC23_DATAHANDLE)); //ready to Xmit?
        MCBSP_write(DSK6713_AIC23_DATAHANDLE,AIC_data.uint); //write data
}

void output_left_sample(short out_data) //for output->left channel
{
    AIC_data.uint=0;                  //clear data structure
    AIC_data.channel[LEFT]=out_data; //data->Left channel->data structure
    if(poll) while(!MCBSP_xrdy(DSK6713_AIC23_DATAHANDLE)); //ready to Xmit?
        MCBSP_write(DSK6713_AIC23_DATAHANDLE,AIC_data.uint); //out->leftchannel
}

void output_right_sample(short out_data) //for output->right channel
...
Uint32 input_sample()                //for 32-bit input
{
    short CHANNEL_data;

```

FIGURE 1.14. Partial listing of communication/initialization support program (C6713dskinit.c).

```

if (poll) while(!MCBSP_rrdy(DSK6713_AIC23_DATAHANDLE)); //receiverready?
    AIC_data.uint=MCBSP_read(DSK6713_AIC23_DATAHANDLE); //read data
    ...
return(AIC_data.uint);
}

short input_left_sample()                                //input to left channel
{
    if(poll) while(!MCBSP_rrdy(DSK6713_AIC23_DATAHANDLE)); //receiverready?
        AIC_data.uint=MCBSP_read(DSK6713_AIC23_DATAHANDLE); //read->left chan
    return(AIC_data.channel[LEFT]);                      //return left channel data
}

short input_right_sample()                               //input to right channel
...

```

FIGURE 1.14. (*Continued*)

The function *comm_intr()* in an interrupt-driven program or *comm_poll()* in a polling-based program calls the appropriate functions to initialize the DSK. These two functions are located in the *init/comm.* file. When using an interrupt-driven program, interrupt #11 (INT11) is configured and enabled (selected). The nonmaskable interrupt bit must be enabled as well as the global interrupt enable (GIE) bit. A different interrupt, such as INT12, can be selected readily by modifying slightly the *init/comm.* file and the vector file that contains the branching address to the corresponding ISR in the main C source program. INT11 is generated via the serial port (McBSP).

The function *input_sample()* is used to input data and the function *output_sample()* to output data. Most of the examples throughout the book utilize the AIC23 codec in a mono format, defaulting to the left channel to read or write a 16-bit data. The example *loop_stereo.c* in Chapter 2 illustrates the stereo capability of the codec to input 16-bit data into each (left and right) channel and output a 16-bit data from each channel. Some adaptive filtering examples in Chapter 7 use both input channels to acquire two different 16-bit input data signals.

The code *input = input_sample();*, casting *input* as a short, acquires 16-bit data through the left (default) channel. Similarly, *output_sample((short) ...);* outputs 16-bit data from the left (default) channel.

A polling-based program (non-interrupt-driven) continuously polls or tests whether or not data are ready to be received or transmitted. This scheme is in general less efficient than the interrupt scheme. For input, the content of the serial port control register (SPCR) bit 1 [the second least significant bit (LSB)], as shown in Figure B.8 (Appendix B), is continuously tested to determine when data are available to be received or read. For output, the content of SPCR bit 17 is tested (Figure B.8) to determine when data are available to be transmitted. An input data value is accessed through the data receive register of the McBSP. An output data value is sent through the data transmit register of McBSP.

The MCBSP1 transmit interrupt is used and INT11 is selected in the examples throughout the book. If the program is polling-based, the McBSP is continuously tested before reading (for input) or writing (for output).

Within the function `output_sample()` used for output, in the code segment

```
If (poll) while(!MCBSP_xrdy(...)); MCBSP_write(...);
```

the first line of code continuously tests (if polling-based) the transmit ready *xrdy* register bit. If it is a 1, then the subsequent line of code is executed to write (output). If the transmit ready bit is a 0 (not ready), then the *while()* statement becomes *while (true)* and execution remains in an infinite loop until the transmit ready bit becomes a 1 (ready). If the program is not polling-based, then the transmit ready bit is not tested and writing (output) occurs every sample period.

Similarly, within the function `input_sample()` used for input, in the code segment

```
If (poll) while(!MCBSP_rrdy(...)); MCBSP_read(...);
```

the first line of code continuously tests (if polling-based) the receive ready *rrdy* register bit. If it is a 1 (ready), the subsequent line of code reads the data. If it is a 0 (not ready), the *while ()* statement causes execution to remain in an infinite loop until the receive ready bit register becomes a 1. If the program is not polling-based, the receive ready bit is not tested and reading occurs every sample period.

The examples throughout the book use both interrupt-driven and polling-based programs. A polling-based program can be readily changed to interrupt-driven and vice versa. Interrupts are discussed further in Chapter 3.

Header File (*c6713dskinit.h*)

The corresponding header support file *c6713dskinit.h* contains the function prototypes as well as various register settings associated with the AIC23 codec. For example (see *c6713dskinit.h*):

1. The mic input can be set in lieu of the line input by changing the value of register 4 from the (default) value of 0x0011 to 0x0015.
2. In Chapter 2, a loop program yields an output that is the delayed input, with the same frequency but attenuated (by default). To increase the gain of the (default) left line input channel, change the value of register 0 from 0x0017 to 0x001c. This value will produce an output of the same amplitude as the input. Note that either the line input or the mic input can be made active.

1.7.2 Vector File (*vectors_intr.asm/vectors_poll.asm*)

To select interrupt INT11, a branch instruction to the ISR `c_int11` located in the C program (see `sine8_buf.c`) is placed at the address INT11 in `vectors_intr.asm`. A listing of the file `vectors_intr.asm` is shown in Figure 1.15. Note

```

*Vectors_intr.asm Vector file for interrupt INT11
.global _vectors                ;global symbols
.global _c_int00
.global _vector1
.global _vector2
.global _vector3
.global _vector4
.global _vector5
.global _vector6
.global _vector7
.global _vector8
.global _vector9
.global _vector10
.global _c_int11                ;for INT11
.global _vector12
.global _vector13
.global _vector14
.global _vector15

.ref _c_int00                    ;entry address

VEC_ENTRY .macro addr            ;macro for ISR
    STW    B0, *--B15
    MVKL   addr, B0
    MVKH   addr, B0
    B      B0
    LDW    *B15++, B0
    NOP    2
    NOP
    NOP
.endm

_vec_dummy:
    B      B3
    NOP    5

.sect ".vecs"                    ;aligned IST section
.align 1024
_vectors:
_vector0:  VEC_ENTRY _c_int00      ;RESET
_vector1:  VEC_ENTRY _vec_dummy    ;NMI
_vector2:  VEC_ENTRY _vec_dummy    ;RSVD
_vector3:  VEC_ENTRY _vec_dummy
_vector4:  VEC_ENTRY _vec_dummy
_vector5:  VEC_ENTRY _vec_dummy
_vector6:  VEC_ENTRY _vec_dummy
_vector7:  VEC_ENTRY _vec_dummy
_vector8:  VEC_ENTRY _vec_dummy
_vector9:  VEC_ENTRY _vec_dummy
_vector10: VEC_ENTRY _vec_dummy
_vector11: VEC_ENTRY _c_int11      ;ISR address
_vector12: VEC_ENTRY _vec_dummy
_vector13: VEC_ENTRY _vec_dummy
_vector14: VEC_ENTRY _vec_dummy
_vector15: VEC_ENTRY _vec_dummy

```

FIGURE 1.15. Vector file for an interrupt-driven program (vectors_intr.asm).

the underscore preceding the name of the routine or function being called. The ISR is also referenced in `vectors_intr.asm` using `.ref_c_int11`.

For a non-interrupt-driven or polling-based program, a separate file `vectors_poll.asm` is used, in lieu of `vectors_intr.asm`, by

1. Deleting the reference to the interrupt service routine (ISR) `.ref_c_int11`
2. Replacing the branch instruction to the ISR for interrupt INT11 by (NOP), which is a no operation instruction.

1.7.3 Linker Command File (`c6713dsk.cmd`)

The linker command file `C6713dsk.cmd` is listed in Figure 1.16. It shows that sections such as `.text` reside in internal RAM (IRAM), which is mapped to the internal memory of the C6713 digital signal processor. It can be used as a generic sample linker command file even though some portion of it is not necessary. Chapter 2 contains an example illustrating the use of the *pragma* directive to specify a section such as EXT_RAM in synchronous DRAM (SDRAM). SDRAM is a section in external memory that starts at the address `0x80000000`. Chapter 2 contains an example illustrating the use of the onboard flash memory (burning the flash) that starts at address `0x90000000`. In Chapter 4, we illustrate the implementation of a digital filter in assembly code using external memory SDRAM. Chapter 10 contains

```
/*C6713dsk.cmd Linker command file*/

MEMORY
{
    IVECS:      org=0h,          len=0x220
    IRAM:       org=0x00000220, len=0x0002FDE0 /*internal memory*/
    SDRAM:      org=0x80000000, len=0x00100000 /*external memory*/
    FLASH:      org=0x90000000, len=0x00020000 /*flash memory*/
}
SECTIONS
{
    .EXT_RAM :> SDRAM
    .vectors :> IVECS      /*in vector file*/
    .text    :> IRAM
    .bss     :> IRAM
    .cinit   :> IRAM
    .stack   :> IRAM
    .sysmem  :> IRAM
    .const   :> IRAM
    .switch  :> IRAM
    .far     :> IRAM
    .cio     :> IRAM
    .csldata :> IRAM
}
```

FIGURE 1.16. Generic linker command file (`C6713dsk.cmd`).

two projects that utilize the EMIF 80-pin connector on the DSK, which starts at address 0xA0000000, to interface to external LEDs and LCDs.

Linker options include `-heap size` to specify the heap size in bytes for dynamic memory allocation (default is 1 kB) and the option `-stack size` to specify the C system stack size in bytes. Other linker options can be found in Ref. 26.

The linker allocates the program in memory using a default location algorithm. It places the various sections into appropriate memory locations, where code and data reside. By using a linker command file with extension `.cmd`, one can customize the allocation process, specifying `MEMORY` and `SECTIONS` directives within the linker command file. The linker directive `MEMORY` (uppercase) defines a memory model and designates the origin and length of various available memory spaces. The directive `SECTIONS` (uppercase) allocate the output sections into defined memory and designate the various code sections to available memory spaces.

Most of the examples in the book invoke internal memory. The generic sample linker command file, shown in Figure 1.16, can be used for almost all of the examples in the book, even if neither external nor flash memory is utilized.

1.8 COMPILER/ASSEMBLER/LINKER SHELL

In previous examples the code generation tools for compiling, assembling, and linking were invoked within CCS while building a project. The tools may also be invoked directly outside CCS using a DOS shell.

1.8.1 Compiler

The compiler shell can be invoked using

```
c16x [options] [files]
```

to compile and assemble files that can be C files with extension `.c`, assembly files with extension `.asm`, and linear assembly (introduced in Chapter 3) with extension `.sa`. A linear assembly program file is a cross between C and assembly that can provide a compromise between the more versatile C program and the most efficient assembly program. For example, the command

```
C16x -gks -o3 file1.c, file2, file3.asm, file4.sa
```

invokes the C compiler to compile `file1` and `file2` (defaults to extension `.c`) and generates the assembly files `file1.asm` and `file2.asm`. This also invokes the assembler optimizer to optimize `file4.sa` and create `file4.asm`. Then the assembler (invoked with the shell command `c16x`) assembles the four assembly source files and creates the four object files `file1.obj`, . . . , `file4.obj`. The option `-gs` adds debugger-specific information for debugging purposes and interlists C

statements into assembly files, respectively. The `-k` option is used to keep the assembly source files generated.

Four levels of compiler optimizations are available, with `-o3` to invoke the highest level of optimization. Level 0 allocates variables to registers. Level 1 performs all level 0 optimizations, eliminates local common expressions, and removes unused assignments. Level 2 performs all the level 1 optimizations plus loop optimizations and rolling. Level 3 performs all level 2 optimizations and removes functions that are not called. There are also compiler optimizations to minimize code size (with possible degradation in execution speed).

Note that full optimization may change memory locations that can affect the functionality of a program. In such cases, these memory locations must be declared as volatile. The compiler does not optimize volatile variables. A volatile variable is allocated to an uninitialized section in lieu of a register. Volatiles can be used when memory access is to be exactly as specified in the C code.

Initially, the functionality of a program is of primary importance. One should *not* invoke any (or too-high-level) optimization option initially while debugging, since additional debugger-specific information is provided to enhance the debugging process. Such additional information suppresses the level of performance. It is also difficult to debug a program after optimization, since the lines of code are usually no longer arranged in a serial fashion. Compiler options can also be set using the environment variable with `C_OPTION`.

1.8.2 Assembler

An assembly-coded source file `file3.asm` can also be assembled using

```
asm6x file3.asm
```

to create `file3.obj`. The `.asm` extension is optional. The resulting object file is then linked with a run-time support library to create an executable COFF file with extension `.out` that can be loaded directly and run on the DSP. Examples using assembly-coded source files are introduced in Chapter 3.

1.8.3 Linker

The linker can be invoked using

```
lnk6x -c prog1.obj -o prog1.out -l rts6700.lib
```

The `-c` option tells the linker to use special conventions defined by the C environment for automatic variable initialization at run time (another linker option, `-cr`, initializes the variables at load time). The `-l` option invokes a library file such as the run-time support library file `rts6700.lib`. These options [`-c` (or `-cr`) and

-l] must be used when linking. The object file `prog1.obj` is linked with the library file(s) and creates the executable file `prog1.out`. Without the `-o` option, the executable file `a.out` (by default) is created.

The linker can also be invoked with the compiler shell command with the `-z` option

```
C16x -gks -o3 prog1.c prog2.asm -z -o prog.out -m prog.map
-l rts6700.lib
```

to create the executable file `prog.out`. The `-m` option creates a map file that provides a list of all the addresses of sections, symbols, and labels that can be useful for debugging.

The linker also links automatically a *boot* program when using C programs to initialize the run-time environment, setting the entry point to `c_int00`. The symbol `_c_int00` is defined automatically when the linker option `-c` (or `-cr`) is invoked. The function `_c_int00`, included in the run-time support library, is the entry point in the *boot* program that sets up the stack and calls *main*. The run-time library support program `boot.c` is used to auto-initialize variables. The linker option `-c` invokes the initialization process with `boot.c`. Note that it is defined in the vector files `vectors_intr.asm` and `vectors_poll.asm`.

The book CD contains all the main source files used in this book, located in separate folders, and some support files necessary for many examples and projects are located in the folder *support*. Other needed support files are included with CCS within `c:\C6713`.

1.9 ASSIGNMENTS

1. Write a program to generate a cosine with a frequency of 666.66 Hz. Verify your output result using LINE OUT, as well as plotting the generated cosine in both time and frequency domains.
2. Write a polling-based program so that once dip switch #3 is pressed, LED #3 turns on and a 666.66 Hz cosine is generated for approximately 5 seconds. [*Hint*: also use (incorporate) the delay associated with turning a LED on.]
3. Write a program to multiply two arrays, each containing the five numbers 1, 2, 3, 4, and 5 (i.e., $1^2 + 2^2 + 3^2 + 4^2 + 5^2$). Verify your result using a watch window and printing it within CCS in the Build window.
4. Write an interrupt-driven program to capture an input sinusoidal signal of amplitude 3 V p-p and a frequency of 1 kHz, and output that sampled signal every 0.0625 ms. Use the function `input_sample` in a similar fashion as the function `output_sample` used in Examples 1.1 and 1.2—for example,

```
input = input_sample();
```

casting input as short (16-bit). Verify that the output signal has the same frequency as the input signal but is reduced in amplitude. Increase the input signal frequency until the output is reduced drastically. What is the approximate frequency at which this occurs? This represents the bandwidth of the onboard AIC23 codec (as illustrated in Chapter 2).

REFERENCES

Note: References 23 to 43 are included with the DSK package.

1. R. Chassaing, *DSP Applications Using C and the TMS320C6x DSK*, Wiley, New York, 2002.
2. R. Chassaing, *Digital Signal Processing Laboratory Experiments Using C and the TMS320C31 DSK*, Wiley, New York, 1999.
3. R. Chassaing, *Digital Signal Processing with C and the TMS320C30*, Wiley, New York, 1992.
4. R. Chassaing and D. W. Horning, *Digital Signal Processing with the TMS320C25*, Wiley, New York, 1990.
5. N. Kehtarnavaz and M. Keramat, *DSP System Design Using the TMS320C6000*, Prentice Hall, Upper Saddle River, NJ, 2001.
6. N. Kehtarnavaz and B. Simsek, *C6x-Based Digital Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 2000.
7. N. Dahnoun, *DSP Implementation Using the TMS320C6x Processors*, Prentice Hall, Upper Saddle River, NJ, 2000.
8. Steven A. Tretter, *Communication System Design Using DSP Algorithms with Laboratory Experiments for the TMS320C6701 and TMS320C6711*, Kluwer Academic, New York, 2003.
9. J. H. McClellan, R. W. Schafer, and M. A. Yoder, *DSP First: A Multimedia Approach*, Prentice Hall, Upper Saddle River, NJ, 1998.
10. C. Marven and G. Ewers, *A Simple Approach to Digital Signal Processing*, Wiley, New York, 1996.
11. J. Chen and H. V. Sorensen, *A Digital Signal Processing Laboratory Using the TMS320C30*, Prentice Hall, Upper Saddle River, NJ, 1997.
12. S. A. Tretter, *Communication System Design Using DSP Algorithms*, Plenum Press, New York, 1995.
13. A. Bateman and W. Yates, *Digital Signal Processing Design*, Computer Science Press, New York, 1991.
14. Y. Dote, *Servo Motor and Motion Control Using Digital Signal Processors*, Prentice Hall, Upper Saddle River, NJ, 1990.
15. J. Eyre, The newest breed trade off speed, energy consumption, and cost to vie for an ever bigger piece of the action, *IEEE Spectrum*, June 2001.

16. J. M. Rabaey, ed., VLSI design and implementation fuels the signal-processing revolution, *IEEE Signal Processing*, Jan. 1998.
17. P. Lapsley, J. Bier, A. Shoham, and E. Lee, *DSP Processor Fundamentals: Architectures and Features*, Berkeley Design Technology, Berkeley, CA, 1996.
18. R. M. Piedra and A. Fritsh, Digital signal processing comes of age, *IEEE Spectrum*, May 1996.
19. R. Chassaing, The need for a laboratory component in DSP education: a personal glimpse, *Digital Signal Processing*, Jan. 1993.
20. R. Chassaing, W. Anakwa, and A. Richardson, Real-time digital signal processing in education, *Proceedings of the 1993 International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Apr. 1993.
21. S. H. Leibson, DSP development software, *EDN Magazine*, Nov. 8, 1990.
22. D. W. Horning, An undergraduate digital signal processing laboratory, *Proceedings of the 1987 ASEE Annual Conference*, June 1987.
23. *TMS320C6000 Programmer's Guide*, SPRU198G, Texas Instruments, Dallas, TX, 2002.
24. *TMS320C6211 Fixed-Point Digital Signal Processor–TMS320C6711 Floating-Point Digital Signal Processor*, SPRS073C, Texas Instruments, Dallas, TX, 2000.
25. *TMS320C6000 CPU and Instruction Set Reference Guide*, SPRU189F, Texas Instruments, Dallas, TX, 2000.
26. *TMS320C6000 Assembly Language Tools User's Guide*, SPRU186K, Texas Instruments, Dallas, TX, 2002.
27. *TMS320C6000 Peripherals Reference Guide*, SPRU190D, Texas Instruments, Dallas, TX, 2001.
28. *TMS320C6000 Optimizing C Compiler User's Guide*, SPRU187K, Texas Instruments, Dallas, TX, 2002.
29. *TMS320C6000 Technical Brief*, SPRU197D, Texas Instruments, Dallas, TX, 1999.
30. *TMS320C64x Technical Overview*, SPRU395, Texas Instruments, Dallas, TX, 2000.
31. *TMS320C6x Peripheral Support Library Programmer's Reference*, SPRU273B, Texas Instruments, Dallas, TX, 1998.
32. *Code Composer Studio User's Guide*, SPRU328B, Texas Instruments, Dallas, TX, 2000.
33. *Code Composer Studio Getting Started Guide*, SPRU509, Texas Instruments, Dallas, TX, 2001.
34. *TMS320C6000 Code Composer Studio Tutorial*, SPRU301C, Texas Instruments, Dallas, TX, 2000.
35. *TLC320AD535C/I Data Manual Dual Channel Voice/Data Codec*, SLAS202A, Texas Instruments, Dallas, TX, 1999.
36. *TMS320C6713 Floating-Point Digital Signal Processor*, SPRS186, Texas Instruments, Dallas, TX.
37. *TLV320AIC23 Stereo Audio Codec, 8- to 96-kHz, with Integrated Headphone Amplifier Data Manual*, SLWS106G, Texas Instruments, Dallas, TX, 2003.

38. *TMS320C6000 DSP Phase-Locked Loop (PLL) Controller Peripheral Reference Guide*, SPRU233, Texas Instruments, Dallas, TX.
39. *Migrating from TMS320C6211/C6711 to TMS320C6713*, SPRA851, Texas Instruments, Dallas, TX, 2003.
40. *How to begin Development Today with the TMS320C6713 Floating-Point DSP*, SPRA809, Texas Instruments, Dallas, TX, 2003.
41. *TMS320C6000 DSP/BIOS User's Guide*, SPRU423, Texas Instruments, Dallas, TX, 2002.
42. *TMS320C6000 Optimizing C Compiler Tutorial*, SPRU425A, Texas Instruments, Dallas, TX, 2002.
43. *TMS320C6000 Chip Support Library API User's Guide*, SPRU401F, Texas Instruments, Dallas, TX, 2003.
44. B. W. Kernigan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Upper Saddle River, NJ, 1988.
45. G. R. Gircys, *Understanding and Using COFF*, O'Reilly & Associates, Newton, MA, 1988.