

Digital Signal Processing and Applications with the OMAP-L138 eXperimenter

Digital Signal Processing and Applications with the OMAP-L138 eXperimenter

Donald Reay
Heriot-Watt University



A JOHN WILEY & SONS, INC., PUBLICATION

Copyright © 2012 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) 750-4470, 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, or online at <http://www.wiley.com/go/permission>.

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 or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States 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 may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data:

Reay, Donald (Donald S.)

Digital signal processing and applications with the OMAP-L138 eXperimenter /
Donald Reay.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-470-93686-3 (hardback)

1. Signal processing—Digital techniques—Experiments. 2.

Microprocessors—Experiments. I. Title.

TK5102.9.R4325 2012

621.382'2078—dc23

2011038412

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To Reiko

Contents

Preface	xi
List of Examples	xiii
1. OMAP-L138 Development System	1
1.1 Introduction	1
1.1.1 Digital Signal Processors	3
1.2 Hardware and Software Tools	4
1.2.1 Zoom OMAP-L138 eXperimenter Board	6
1.2.2 C6748 Processor	6
1.2.3 Code Composer Studio IDE	6
1.2.4 Installation of Code Composer Studio Software Version 4 and Support Files	7
1.3 Initial Test of the Experimenter Using a Program Supplied with this Book	8
1.4 Programming Examples to Test the Experimenter	14
1.5 Support Files	31
1.5.1 Initialization and Configuration File (L138_aic3106_init.c)	31
1.5.2 Header File (L138_aic3106_init.h)	32
1.5.3 Vector Files (vectors_intr.asm and vectors_poll.asm)	32
1.5.4 Linker Command File (linker_dsp.cmd)	34
Exercises	36
References	37
2. Analog Input and Output with the OMAP-L138 eXperimenter	38
2.1 Introduction	38
2.1.1 Sampling, Reconstruction, and Aliasing	39
2.2 TLV320AIC3106 (AIC3106) On-Board Stereo Codec for Analog Input and Output	39
2.3 Programming Examples Using C Code	41
2.3.1 Real-Time Input and Output Using Polling, Interrupts, and Direct Memory Access	41

2.3.2 Real-Time Sine Wave Generation	64
References	102
3. Finite Impulse Response Filters	103
3.1 Introduction to Digital Filters	103
3.1.1 FIR Filter	103
3.1.2 Introduction to the z -Transform	105
3.1.3 Properties of the z -Transform	107
3.1.4 z -Transfer Functions	109
3.1.5 Mapping from the s -Plane to the z -Plane	109
3.1.6 Difference Equations	111
3.1.7 Frequency Response and the z -Transform	112
3.1.8 Ideal Filter Response Classifications: LP, HP, BP, and BS	112
3.1.9 Window Method of Filter Design	113
3.1.10 Window Functions	114
3.1.11 Design of Band-Pass and High-Pass Filters Using Frequency Shifting	120
3.2 Programming Examples Using C And ASM Code	123
References	158
4. Infinite Impulse Response Filters	159
4.1 Introduction	159
4.2 IIR Filter Structures	160
4.2.1 Direct Form I Structure	160
4.2.2 Direct Form II Structure	161
4.2.3 Direct Form II Transpose	162
4.2.4 Cascade Structure	164
4.2.5 Parallel Form Structure	165
4.3 Impulse Invariance	166
4.4 Bilinear Transformation	167
4.4.1 Bilinear Transform Design Procedure	169
4.5 Programming Examples Using C and ASM Code	169
4.5.1 Design of a Simple IIR Low-Pass Filter	169
Reference	211
5. Fast Fourier Transform	212
5.1 Introduction	212
5.2 Development of the FFT Algorithm with Radix-2	213
5.3 Decimation-In-Frequency FFT Algorithm with Radix-2	214
5.4 Decimation-In-Time FFT Algorithm with RADIX-2	218
5.4.1 Reordered Sequences in the Radix-2 FFT and Bit-Reversed Addressing	220

5.5 Decimation-In-Frequency FFT Algorithm with Radix-4	221
5.6 Inverse Fast Fourier Transform	223
5.7 Programming Examples Using C Code	223
5.7.1 Frame- or Block-Based Processing	233
5.7.2 Fast Convolution	258
References	278
6. Adaptive Filters	279
6.1 Introduction	279
6.2 Adaptive Filter Configurations	280
6.2.1 Adaptive Prediction	280
6.2.2 System Identification or Direct Modeling	281
6.2.3 Noise Cancellation	281
6.2.4 Equalization	283
6.3 Performance Function	283
6.3.1 Visualizing the Performance Function	285
6.4 Searching for the Minimum	285
6.5 Least Mean Squares Algorithm	287
6.5.1 LMS Variants	288
6.6 Programming Examples	288
7. DSP/BIOS and Platform Support Package	307
7.1 Introduction to DSP/BIOS	307
7.1.1 DSP/BIOS Threads	307
7.1.2 DSP/BIOS Configuration Tool	308
7.1.3 DSP/BIOS Start-Up Sequence	309
7.1.4 Hardware Interrupts	310
7.1.5 Software Interrupts	320
7.1.6 Tasks and Idle Functions	322
7.1.7 Periodic Functions	327
7.1.8 Real-Time Analysis with DSP/BIOS	329
7.2 DSP/BIOS Platform Support Package	329
References	335
Index	337

Preface

This book continues the series started in 1990 by Rulph Chassaing and Darrell Horning's *Digital Signal Processing with the TMS320C25* and which has reflected the development of successive generations of digital signal processors by Texas Instruments. More specifically, each book in this series has complemented a different one of the inexpensive DSP development tools promoted by the Texas Instruments University Programme for teaching purposes. A consistent theme in the books has been the provision of a large number of simple example programs illustrating DSP concepts in real-time in a laboratory setting.

It was Rulph Chassaing's belief, and also mine, that hands-on teaching of DSP, using hardware development kits and laboratory test equipment to process analog audio frequency signals, is a valuable and effective way of reinforcing the theory taught in lectures.

The contents of the books, insofar as they concern fundamental concepts of digital signal processing such as analog-to-digital and digital-to-analog conversion, FIR and IIR filtering, the Fourier transform, and adaptive filtering, have changed little. Every year, in the context of university teaching, brings another set of students wanting to study this material. However, each successive book has concerned a different hardware development kit. The latest hardware development kit to be promoted by the Texas Instruments University Programme is the Logic PD OMAP-L138 eXperimenter.

This book is suitable for senior undergraduate and postgraduate electrical engineering students who have a basic knowledge of C programming and of linear systems theory.

The architecture of Texas Instruments' DSP devices has reached a level of complexity that places assembly language programming out of reach of such students. Certainly, I have found that it is beyond the scope and time available in a digital signal processing class. Even some of the optimized DSP functions supplied by Texas Instruments in support libraries are written in C rather than assembly language.

For this reason, this book does not contain chapters concerning processor architecture or assembly language programming.

The OMAP-L138 is a dual-core processor, the capabilities of which are far beyond what can be covered in a single text. This book uses only a fraction of its features in order to provide teaching materials specifically for DSP.

It is intended and hoped that this book will prove a useful resource for anyone involved in teaching or learning DSP and as a starting point for teaching or learning more.

I am grateful to Robert Owen and Cathy Wicks at the TI University Programme for their support and encouragement, to George Telecki at Wiley for his patience, to Keith Brown at Heriot-Watt University for his help in testing some of the example programs and for his suggestions, and to He Xiangning at Zhejiang University for giving me the opportunity to try out the example programs in a teaching environment with some very able students. But above all, I thank Rulph Chassaing for inspiring me to get involved in teaching hands-on DSP.

The ftp site (ftp://ftp.wiley.com/public/sci_tech_med/signal_processing) contains the code for the example programs described in the book.

DONALD REAY

List of Examples

- Example 1.1: Sine generation using forty-eight points with output recorded in a buffer for plotting using Code Composer Studio software and MATLAB (L138_sine48_buf_intr)
- Example 1.2: Dot product of two arrays (L138_dotp4)
- Example 2.1: Basic input and output using polling (L138_loop_poll)
- Example 2.2: Basic input and output using interrupts (L138_loop_intr)
- Example 2.3: Basic input and output using DMA (L138_loop_edma)
- Example 2.4: Modifying Program L138_loop_intr.c to create a delay (L138_delay_intr)
- Example 2.5: Modifying program L138_loop_intr.c to create an echo (L138_echo_intr)
- Example 2.6: Modifying program L138_loop_intr.c to create a flanging effect (L138_flanger_intr)
- Example 2.7: Loop program with input data stored in a buffer (L138_loop_buf_intr)
- Example 2.8: Sine wave generation using a look up table (L138_sine48_intr)
- Example 2.9: Sine wave generation using sin() function call (L138_sine_intr)
- Example 2.10: Sine generation with DIP switches for amplitude and frequency control (L138_sine_DIP_intr)
- Example 2.11: Sweep Sinusoid Using Table with 8000 Points (L138_sweep_poll)
- Example 2.12: Generation of DTMF tones using a look up table (L138_sineDTMF_intr)
- Example 2.13: Signal reconstruction, aliasing and the properties of the AIC23 codec (L138_sine_intr.c)
- Example 2.14: Square wave generation using a look up table (L138_squarewave_intr)

- Example 2.15: Impulse response of the AIC3106 DAC reconstruction filter (L138_dimpulse_intr)
- Example 2.16: Frequency response of the DAC reconstruction filter using a pseudo random binary sequence (L138_prbs_intr)
- Example 2.17: Frequency response of the DAC reconstruction filter using pseudo random noise (L138_prandom_intr)
- Example 2.18: Step response of the AIC3106 codec antialiasing filter (L138_loop_buf_intr)
- Example 2.19: Demonstration of the AIC3106 codec antialiasing filter (L138_sine48_loop_intr)
- Example 2.20: Demonstration of aliasing (L138_aliasing_intr)
- Example 2.21: Identification of AIC3106 codec bandwidth using an adaptive filter (L138_sysid_intr)
- Example 2.22: Identification of AIC3106 Codec Bandwidth using two experimenters
- Example 2.23: Ramp Generation (L138_ramp_intr)
- Example 2.24: Amplitude Modulation (L138_am_poll)
- Example 2.25: Use of External Memory to Record Music (L138_record_poll)
- Example 3.1: Design of an ideal low pass FIR filter using the window method
- Example 3.2: Moving average filter (L138_average_intr)
- Example 3.3: Moving average filter with internally generated pseudo random noise as input (L138_average_prn_intr)
- Example 3.4: Identification of moving average filter frequency response using a second eXperimenter board (L138_sysid_intr)
- Example 3.5: Identification of moving average filter frequency response using a single eXperimenter board (L138_sysid_average_intr)
- Example 3.6: FIR filter with moving average, lowpass, bandstop, and bandpass characteristics defined in separate coefficient files (L138_fir_intr)
- Example 3.7: FIR Implementation with a pseudo random noise sequence as input (L138_firprn_intr)
- Example 3.8: FIR filter with internally generated pseudo random noise as input and output stored in memory (L138_firprnbuf_intr)
- Example 3.9: Effects on voice or music using three FIR low pass filters (L138_fir3lp_intr)

- Example 3.10: Implementation of four different filters: low pass, high pass, band pass, and band stop (L138_fir4types_intr)
- Example 3.11: Two notch filters to recover a corrupted speech recording (L138_notch2_intr)
- Example 3.12: Voice scrambling using filtering and modulation (L138_scrambler_intr)
- Example 3.13: FIR filter implemented using DMA-based I/O (L138_fir_edma)
- Example 3.14: FIR filter implemented using a DSPLIB function (L138_fir_dsplib_edma)
- Example 3.15: FIR implementation using C calling an ASM function (L138_FIRcasm_intr.c)
- Example 3.16: FIR implementation using C calling a faster ASM function (FIRcasmfast)
- Example 4.1: Implementation of an IIR filter using cascaded second order direct form II sections (L138_iirssos_intr)
- Example 4.2: Implementation of IIR Filter using cascaded second order transposed direct form II sections (L138_iirssostr_intr)
- Example 4.3: Estimating the frequency response of an IIR filter using pseudo random noise as input (L138_iirssosprn_intr)
- Example 4.4: Estimating the Frequency response of an IIR filter using a sequence of impulses as input (L138_iirssosdelta_intr)
- Example 4.5: Fourth Order Elliptic Low Pass IIR Filter designed using fdatoool
- Example 4.6: Band pass filter design using fdatoool
- Example 4.7: Implementation of IIR Filter using DSPLIB function DSPF_sp_biquad() (L138_iirssos_DSPLIB_edma)
- Example 4.8: Fixed point implementation of an IIR filter (L138_iir_intr)
- Example 4.9: Implementation of a fourth order IIR filter using the AIC3106 digital effects filter (L138_sysid_biquad_intr)
- Example 4.10: Generation of a Sine Wave using a Difference Equation (L138_sinegenDE_intr)
- Example 4.11: Generation of DTMF signal using difference equations (L138_sinegenDTMF_intr)
- Example 4.12: Generation of a swept sinusoid using a difference equation (L138_sweepDE_intr)

- Example 4.13: Sine Generation Using a Difference Equation with C Calling an ASM Function (L138_sinegencasm_intr)
- Example 5.1: DFT of a Sequence of Real Numbers with Output in the CCS Graphical Display Window and in MATLAB (L138_dft)
- Example 5.2: Estimating Execution Times for DFT and FFT Functions (L138_dft, L138_dftw, L138_fft, L138_fft_dsplibr2)
- Example 5.3: EDMA3 memory move (L138_mem_edma)
- Example 5.4: DFT of a Signal in Real-Time Using a DFT function with pre-calculated Twiddle Factors (L138_dft128_edma)
- Example 5.5: FFT of a Real-Time Input Signal Using an FFT Function in C (L138_fft128_edma.c)
- Example 5.6: FFT of Real-Time Input Using TI's C-Callable Optimized Radix-2 FFT Function (L138_fft128_dsplibr2_edma)
- Example 5.7: FFT of a Sinusoidal Signal from a Table Using TI's C Callable Optimized DSPLIB FFT Function (L138_FFTsinetable_edma)
- Example 5.8: Demonstration of fast convolution (L138_fastconv_demo)
- Example 5.9: Real-time fast convolution (L138_fastconv_edma)
- Example 5.10: Graphic equalizer (L138_graphicEQ_DSPLIB_edma)
- Example 6.1: Adaptive Filter Using C Code (L138_adaptc)
- Example 6.2: Adaptive Filter for Sinusoidal Noise Cancellation (L138_adaptnoise_intr)
- Example 6.3: Adaptive FIR Filter for Noise Cancellation Using External Inputs (L138_adaptnoise_2IN_iir_intr)
- Example 6.4: Adaptive FIR Filter for System Identification of a Fixed FIR Filter as an Unknown System (L138_adaptIDFIR_intr)
- Example 6.5: Adaptive FIR for System ID of a Fixed FIR as an Unknown System with Weights of an Adaptive Filter Initialized as a FIR Bandpass (L138_adaptIDFIR_init_intr)
- Example 6.6: Adaptive FIR for System ID of Fixed IIR as an Unknown System (L138_iirsoadapt_intr)
- Example 6.7: Adaptive FIR Filter for System Identification of System External to the eXperimenter (L138_sysid_intr)

- Example 6.8: Adaptive FIR Filter for System Identification of System External to the eXperimenter using DSPLIB function DSPF_sp_fir_gen() (L138_sysid_DSPLIB_edma)
- Example 7.1: Sine Generation Using DSP/BIOS hardware interrupts HWIs (L138_bios_sine48_intr_HWI)
- Example 7.2: FIR filter using DSP/BIOS hardware interrupts HWIs and Software Interrupts SWIs (L138_bios_firprn_intr_SWI)
- Example 7.3: System Identification using DMA-based i/o and semaphore to signal transfer complete (L138_bios_sysid_edma_TSK.c)
- Example 7.4: System Identification using DMA-based i/o and semaphore to signal transfer complete (L138_bios_sysid_edma_IDL.c)
- Example 7.5: Blinking of LEDs at Different Rates Using DSP/BIOS PRDs (L138_bios_LED_PRD)
- Example 7.6: Basic Input and Output (L138_psp_loop.c)
- Example 7.7: Generation of pseudo random noise (L138_psp_prbs.c)
- Example 7.8: FIR filter using pseudo random noise as input (L138_psp_firprn)

Chapter 1

OMAP-L138 Development System

- OMAP-L138 processor
- Code Composer Studio™ IDE version 4
- Use of the OMAP-L138 eXperimenter
- Programming examples

This chapter gives an overview of the OMAP-L138 processor and Logic PD's Zoom OMAP-L138 eXperimenter development system. It describes how to install and start using version 4 of Texas Instruments (TI) Code Composer Studio integrated development environment (IDE). Two example programs that demonstrate hardware and software features of the eXperimenter board and of the Code Composer Studio IDE are presented. It is recommended strongly that you review these examples before proceeding to subsequent chapters.

1.1 INTRODUCTION

The Logic PD Zoom OMAP-L138 eXperimenter kit is a low-cost development platform for the Texas Instruments OMAP-L138 processor. This device is a dual-core system on a chip comprising an ARM926EJ-S general-purpose processor (GPP) and a TMS320C6748 digital signal processor. In addition, a number of peripherals and interfaces are built into the OMAP-L138 as shown in Figure 1.1.

The eXperimenter makes a significant number of the OMAP-L138 interfaces available to the user, as shown in Figure 1.2. This book is concerned with the development of real-time digital signal processing (DSP) applications and therefore makes use of the DSP (C6748) side of the device and of the TLC320AIC3106 (AIC3106) analog interface circuit (codec) connected to the OMAP-L138's

2 Chapter 1 OMAP-L138 Development System

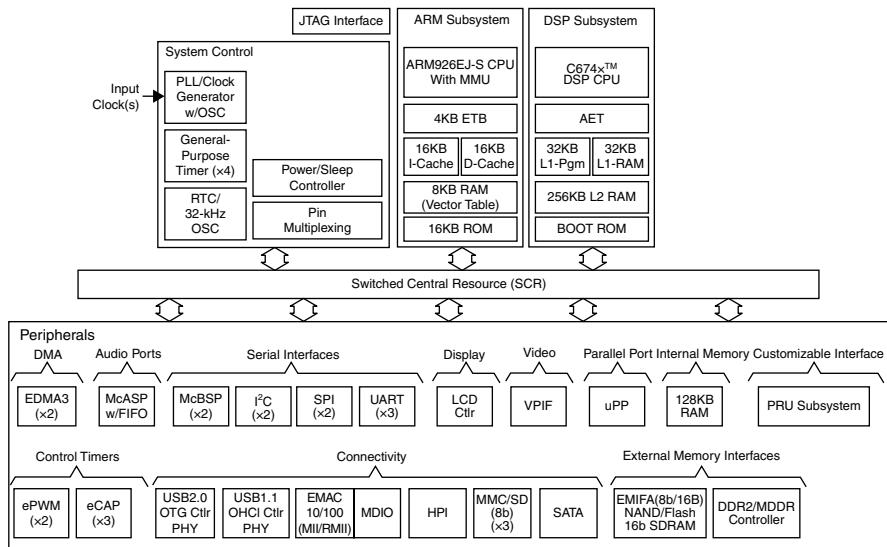


Figure 1.1 Functional block diagram of OMAP-L138 processor. (courtesy of Texas Instruments)

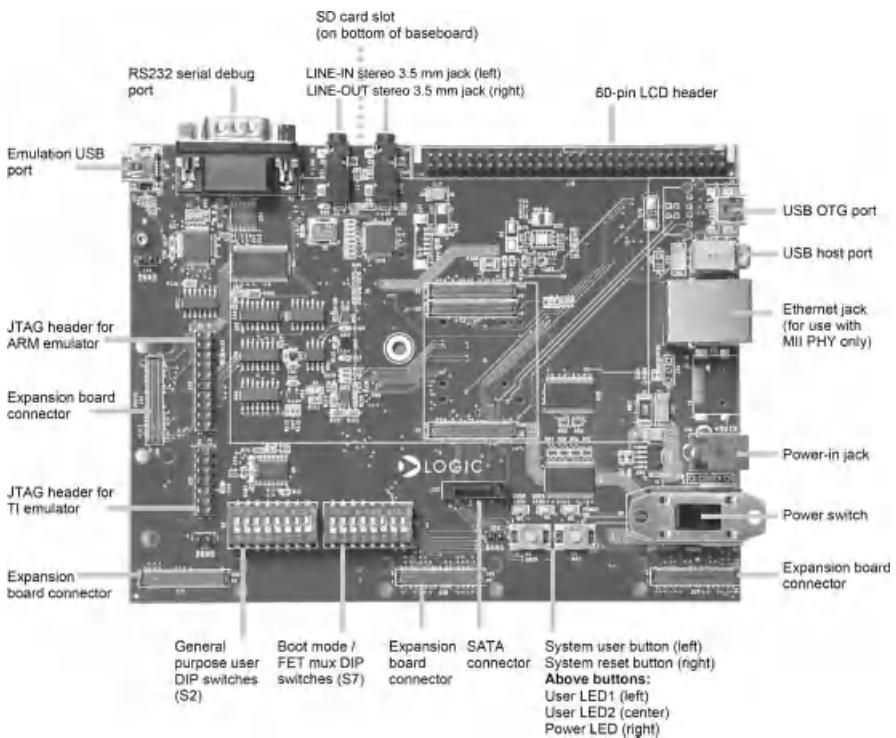


Figure 1.2 Logic PD Zoom OMAP-L138 eXperimenter baseboard (courtesy of Logic PD).

multichannel audio serial port (McASP). The ARM side of the device is not used by the examples in this book. Connection to a host PC running the Code Composer Studio IDE is via XDS100v1 JTAG emulation built in to the eXperimenter. The Code Composer Studio IDE enables software written in C or assembly language to be compiled and/or assembled, linked, and downloaded to run on the C6748. Details of the OMAP-L138, TMS320C6748, TLC320AIC3106, eXperimenter, and Code Composer Studio IDE can be found in their associated datasheets [1–5] and in the TI wiki [6]. The purpose of this chapter is to introduce the installation and use of the eXperimenter for hands-on DSP experiments.

1.1.1 Digital Signal Processors

A digital signal processor is a specialized form of microprocessor. Its architecture and instruction set are optimized for real-time digital signal processing. Typical optimizations include hardware multiply accumulate (MAC) provision, hardware circular and bit-reversed addressing capabilities (for efficient implementation of data buffers and fast Fourier transform (FFT) computation), and Harvard architecture (independent program and data memory systems). In many respects, digital signal processors resemble microcontrollers. Typically, they provide single-chip computer solutions integrating on-board volatile and nonvolatile memory and a range of peripheral interfaces, and have a small footprint, making them ideal for embedded applications. In addition, digital signal processors tend to have low power consumption requirements. This attribute has been extremely important in establishing the use of digital signal processors in cellular handsets. However, the distinctions between digital signal processors and other more general-purpose microprocessors are blurred. No strict definition of a digital signal processor exists and semiconductor manufacturers apply the term to products exhibiting some, but not necessarily all, of the above characteristics as they see fit.

Digital signal processors are used for a wide range of applications, from communications and control to speech and image processing. They are found in cellular phones, disk drives, radios, printers, MP3 players, HDTV, digital cameras, and so on. Specialized (particularly in terms of their on-board peripherals) DSPs are used in electric motor drives and in a range of associated automotive and industrial applications. Overall, digital signal processors are concerned primarily with real-time signal processing. Real-time processing means that the processing must 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. While analog-based systems with discrete electronic components including resistors and capacitors are sensitive to temperature changes, DSP-based systems are less affected by environmental conditions such as temperature. Digital signal processors embody the major advantages of microprocessors. They are easy to use, flexible, and economical.

Texas Instruments OMAP-L138 device combines a C6748 DSP with an ARM926EJ-S general-purpose processor to produce a dual-core solution for

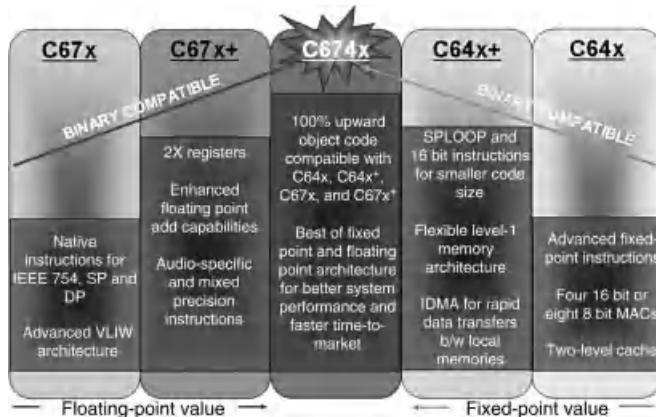


Figure 1.3 The C674x combines C64x fixed-point and C67x floating-point DSP architectures. (courtesy of Texas Instruments).

handheld and other embedded applications. ARM926EJ-S provides the benefits of a 32-bit RISC processor, well suited to implementing user interfaces and running operating systems.

C6748 is a member of the Texas Instruments TMS320C6000™ DSP family of digital signal processors. Its architecture is very well suited to numerically intensive calculations and it is one of TI's most powerful digital signal processors. More specifically, as a member of the C674x family, it combines C64x™ DSP fixed-point and C67x™ DSP floating-point architectures in one core, as illustrated in Figure 1.3.

1.2 HARDWARE AND SOFTWARE TOOLS

Most of the examples presented in this book involve the development and testing of short programs intended to demonstrate fundamental DSP concepts in a laboratory setting. To perform the experiments described in the book, a number of hardware and software tools are required.

- (1) **A Logic PD Zoom OMAP-L138 eXperimenter kit** This package includes the following:
 - (a) Three separate circuit boards, a baseboard, a 4.3" LCD, and an OMAP-L138 SOM-M1. OMAP-L138 SOM-M1 must be connected to the baseboard, as described in the instructions included in the eXperimenter kit. The LCD is not used in the experiments in this book and therefore need not be connected to the baseboard.
 - (b) A universal serial bus (USB) cable that connects the eXperimenter board to a host PC.
 - (c) A 5 V universal power supply for the eXperimenter board.
- (2) **A host PC** This is used to run the Code Composer Studio IDE. The eXperimenter board is connected to a USB port on the host PC.

- (3) **Code Composer Studio software, version 4** The eXperimenter kit includes a DVD containing Code Composer Studio software version 4. Alternatively, the Code Composer Studio IDE may be downloaded from the Texas Instruments wiki at http://processors.wiki.ti.com/index.php/Download_CCS. Code Composer Studio software provides an IDE, bringing together C compiler, assembler, linker, and debugger.
- (4) **Board support library** Board-level support routines specific to eXperimenter are not supplied with the kit, but may be downloaded from the Logic PD website at <http://www.logicpd.com/product-support>. Access to the board support library BSL file 1017292A_OMAP-L138_GEL_BSL_Files_v2.3.zip requires a login and product registration.
- (5) **TMS320C674x DSP library** A number of example programs make use of optimized DSP routines from this library. It can be downloaded from the Texas Instruments wiki at http://processors.wiki.ti.com/index.php/C674x_DSPLIB or from the Texas Instruments website at <http://focus.ti.com/docs/toolsw/folders/print/sprc900.html>.
- (6) **DSP/BIOS™ software Kernel Foundation Platform Support Package (PSP)** This is used for examples in Chapter 7. Details of how to download file BIOSPSP_01_30_00_06_Setup.exe, which is part of the OMAP-L138 and C6748 software development kits (SDKs) may be found at http://processors.wiki.ti.com/index.php/GSG_C6748:_Installing_the_SDK_Software.
- (7) **An oscilloscope, spectrum analyzer, signal generator, headphones, microphone, and loudspeakers** The experiments presented in subsequent chapters of this book are intended to demonstrate digital signal processing concepts in real-time, using audio frequency analog input and output signals. In order to appreciate these concepts and to get the greatest benefit from the experiments, some forms of signal source and sink are required. As a bare minimum, an audio source with line level output and either headphones or loudspeakers are required. Greater benefit will be accrued if a signal generator is used to generate sinusoidal, and other, test signals and if an oscilloscope and spectrum analyzer are used to display, measure, and analyze input and output signals. Many modern digital oscilloscopes incorporate FFT functions, allowing the frequency content of signals to be displayed. Alternatively, a number of software packages that use a PC equipped with a sound card to implement virtual instruments are available. In this book, *Goldwave* is used, which may be downloaded from www.goldwave.com.
- (8) The files and example programs listed and discussed in this book are included on the partner website ftp://ftp.wiley.com/public/sci_tech_med/signal_processing.

1.2.1 Zoom OMAP-L138 eXperimenter Board

The eXperimenter board is a powerful, yet inexpensive, development system with the necessary hardware and software support tools for real-time signal processing [4]. From the point of view of the example programs in this book, it is a complete DSP system. The board, which measures approximately 5×7 inches, includes a 375 MHz OMAP-L138 processor and a 16-bit stereo codec TLV320AIC3106 (AIC3106) for analog input and output. Numerous other interfaces are provided by the eXperimenter but are not used by the example programs in this book.

The onboard codec AIC3106 [3] uses sigma-delta technology that provides analog-to-digital conversion (ADC) and digital-to-analog conversion (DAC) functions. It uses a 24.576 MHz system clock and its sampling rate can be selected from a range of alternative settings from 8 to 48 kHz.

The eXperimenter boards each include 128 MB of synchronous dynamic RAM (mDDR SDRAM) and 8 MB of NOR flash memory. Two 3.5 mm jack socket connectors on the boards provide analog input and output: LINE IN for line level input and LINE OUT for line level output. The status of eight user DIP switches on the board can be read from within a program running on the processor and provide a simple means of user interaction. The states of two LEDs on the board can be controlled from within a program running on the processor.

1.2.2 C6748 Processor

The DSP core in the OMAP-L138 device (L138) is a C6748 DSP based on Texas Instruments very long instruction word (VLIW) architecture and is very well suited to numerically intensive algorithms. The internal program memory is structured so that a total of eight instructions can be fetched every cycle. With a clock rate of 375 MHz, the C6748 is capable of fetching eight 32-bit instructions every $1/(375 \text{ MHz})$ or 2.67 ns. As part of the C674x family, it incorporates both floating-point and fixed-point architectures in one core.

Features of the C6748 include 326 kB of internal memory (32 kB of L1P program RAM/cache, 32 kB of L1D data RAM/cache, and 256 kB of L2 RAM/cache), eight functional or execution units composed of six ALUs and two multiplier units, an external memory interface addressing 256 MB of 16-bit mDDR SDRAM, and 64 32-bit general-purpose registers. In addition, the OMAP-L138 features 128 kB of on-chip RAM shared by its C6748 and ARM9 processor cores [1].

1.2.3 Code Composer Studio IDE

Code Composer Studio software provides an IDE for real-time digital signal processing applications based on the C programming language. It incorporates a C compiler, an assembler, and a linker. It has graphical capabilities and supports real-time debugging. Version 4 of the Code Composer Studio IDE is based on the

open-source Eclipse framework [7], widely used in embedded systems development.

Code Composer Studio software is project based. A Code Composer Studio software project comprises all the files (or links to all the files) required in order to generate an executable file. In addition, a Code Composer Studio software project contains information about exactly how files are to be used in order to generate an executable file. Compiler/linker options can be specified.

A number of debugging features are available in Code Composer Studio software, including setting breakpoints and watching variables, viewing memory, registers, and mixed C and assembly code, graphing results, and monitoring execution time.

Communication between the Code Composer Studio IDE and the eXperimenter is via a USB connection and the XDS100v1 JTAG emulation [8] built into the board. Compared to previous versions, Code Composer Studio software version 4 separates code development and debugging activities through the use of perspectives. Perspectives are sets of windows, views, and menus and as a default Code Composer Studio software provides a default *C/C++* perspective, including, among others, *Project View*, *Editor*, and *Outline* windows, and a default *Debug* perspective, including *Debug*, *Disassembly*, and *Console* views. Users may customize these perspectives or create new ones. The *View* menu gives an idea of the many other windows available.

1.2.4 Installation of Code Composer Studio Software Version 4 and Support Files

The example programs described in this book are intended to be used with Code Composer Studio software version 4 and were tested using version 4.2.1. Code Composer Studio software is supplied on a DVD as part of the eXperimenter kit or alternatively may be downloaded from the Texas Instruments wiki at http://processors.wiki.ti.com/index.php/Download_CCS. Installation instructions for Code Composer Studio software are included on the DVD. A typical (default) location for the files is `c:\Program Files\Texas Instruments\ccsv4`, but this is not mandatory. The default location for the Code Composer Studio IDE was used during preparation of the example programs in this book. Once installed, an icon with the label *Code Composer Studio v4* should appear on the desktop, as shown in Figure 1.4.

The example programs make use of the Logic PD BSL which may be downloaded from <http://www.logicpd.com/product-support>. During preparation of this book and testing of the example programs, the BSL was installed at `c:\omap1138`.

Some example programs make use of the optimized c674x DSPLIB library of digital signal processing routines [9] that may be downloaded from http://processors.wiki.ti.com/index.php/C674x_DSPLIB. During preparation of this book and testing of the example programs, it was installed at `c:\C6748_dsp_1_00_00_11`.



Figure 1.4 Code Composer Studio software desktop icon.

Example programs in Chapter 7 make use of the DSP/BIOS software kernel foundation PSP. During testing of the example programs, the SDK that includes the PSP was installed according to the instructions at http://processors.wiki.ti.com/index.php/GSG_C6748:_Installing_the_SDK_Software, resulting in installation of PSP at c:\C6748_dsp_1_00_00_11\pspdrivers_01_30_01.

The files accompanying this book should be copied to c:\eXperimenter so that, for example, files relating to this chapter may be found in folder c:\eXperimenter\L138_chapter1.

Alternative locations for the various software tools described are possible, but corresponding changes to some of the *Build Properties* within Code Composer Studio software that are shown in this book would have to be made. The path names used and detailed instructions given in this book assume that the software tools have been installed as just described.

1.3 INITIAL TEST OF THE EXPERIMENTER USING A PROGRAM SUPPLIED WITH THIS BOOK

Follow these instructions in order to quickly test the correct installation of the software tools and the eXperimenter board:

- (1) Connect the eXperimenter board to the host PC using the USB cable provided. There are two mini-USB sockets on the eXperimenter. The socket used for connection to the host PC is *Emulation USB port* (J21), located adjacent to the 9-way D-type RS232 *serial debug port* (Figure 1.2).
- (2) Make sure that all the *Boot mode* DIP switches (S7) are OFF, except for DIP switches #5 and #8 that should be ON (Figure 1.2). This sets the appropriate *Boot Mode* (EMU Debug) for the use made of the eXperimenter in this book.
- (3) Connect a line level audio source, for example, the output from a PC sound card, to the LINE IN socket on the eXperimenter. Make sure that the output level from the source is sufficiently low that it will not damage the input circuits of the AIC3106 codec.
- (4) Connect either headphones or loudspeakers to the LINE OUT socket on the eXperimenter.
- (5) Connect the power supply provided to the eXperimenter board.



Figure 1.5 Code Composer Studio software version 4 splash screen.

- (6) Switch on the eXperimenter using the Power Switch. The *Power On* LED (D5) should light.
- (7) Launch Code Composer Studio software by double-clicking on the *Code Composer Studio v4* icon on the desktop. You should see a splash screen as shown in Figure 1.5 and then a pop-up window similar to that shown in Figure 1.6.
- (8) Enter `c:\eXperimenter\L138_chapter1` as a workspace, as shown in Figure 1.6 and click *OK*. Next, you should see a welcome screen as shown in Figure 1.7. Click on the *Start using CCS* icon in the top right-hand corner and Code Composer Studio should start in the *C/C++* perspective, but show no projects in the *Project View* window.

The Code Composer Studio software version 4 debugger makes use of a *Target Configuration* file containing details of the hardware system being debugged. For the example programs in this book, target configuration file `L138_experimenter.ccxml` is provided. It is located in folder `c:\eXperimenter\L138_support`.

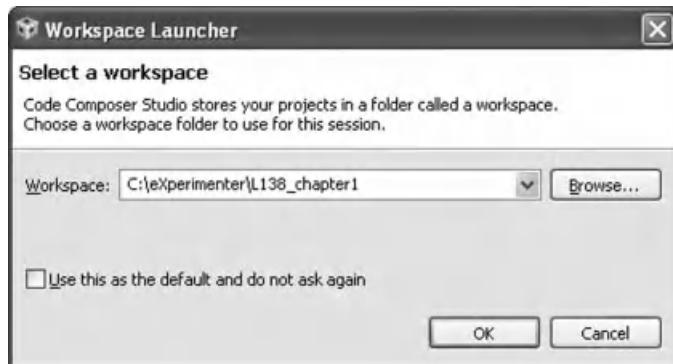


Figure 1.6 Code Composer Studio software version 4 pop-up window.



Figure 1.7 Code Composer Studio software version 4 Welcome screen.

- (9) Copy the target configuration file `L138_eXperimenter.ccxml` from `c:\eXperimenter\L138_support` to the default location used by the Code Composer Studio IDE for target configuration files, that is, `c:\Documents and Settings\YOUR_ID\user\CCSTargetConfigurations` if you are using Window XP, or `c:\User\Your_ID\user\CCSTargetConfigurations` if you are using Windows 7.
- (10) Copy file `C6748.gel` from folder `c:\eXperimenter\L138_support` to `c:\Documents and Settings\YOUR_ID\user\CCSTargetConfigurations` if you are using Window XP, or `c:\User\Your_ID\user\CCSTargetConfigurations` if you are using Windows 7. This general extension language (GEL) script is run every time you *Connect to Target* in the debugger and carries out a number of important initialization procedures on the eXperimenter.
- (11) In the *C/C++* perspective, select *View > Target Configurations*, right-click on `L138_eXperimenter.ccxml`, under *User Defined* and select *Set as Default*. The *Project View* window should then appear as shown in Figure 1.8.
- (12) Launch the debugger by selecting *Target > Launch TI Debugger*. This should cause Code Composer Studio to switch from the *C/C++* perspective to the *Debug* perspective, including a *Debug* window as shown in Figure 1.9. If Code Composer Studio software does not automatically switch to the *Debug* perspective, you can do so using the *Debug* button in the top right-hand corner of the Code Composer Studio IDE window, as shown in Figure 1.10.

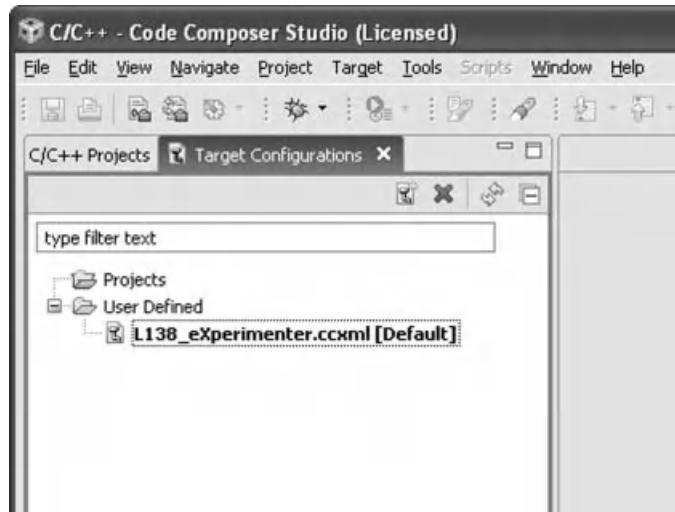


Figure 1.8 Target Configurations view following selection of default target configuration file L138_eXperimenter.ccxml.

(13) Select *Target > Connect Target*. At this point, the GEL script C6748.gel specified by the target configuration file L138_eXperimenter.ccxml will be run and some diagnostic messages will appear in the console. Following this, the *Connect* icon in the toolbar should change from grayed out to highlighted, as shown in Figure 1.11. During this step, the error message window shown in Figure 1.12 may appear. If this happens, simply ignore it. It should disappear after a few seconds.

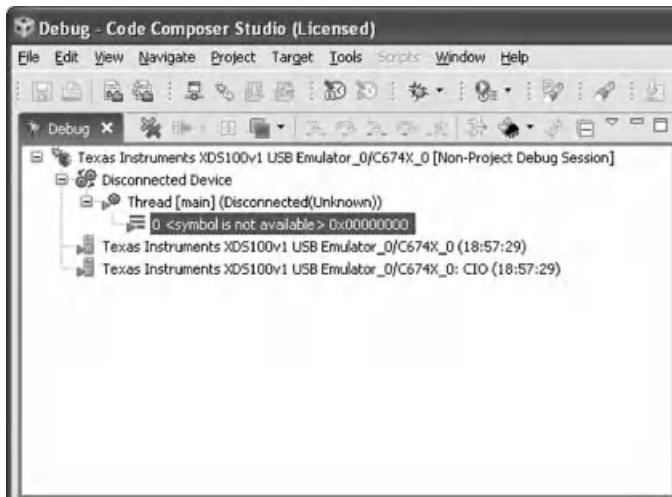


Figure 1.9 Debug view as it should appear in the Debug perspective after launching the debugger (Connect icon grayed out).

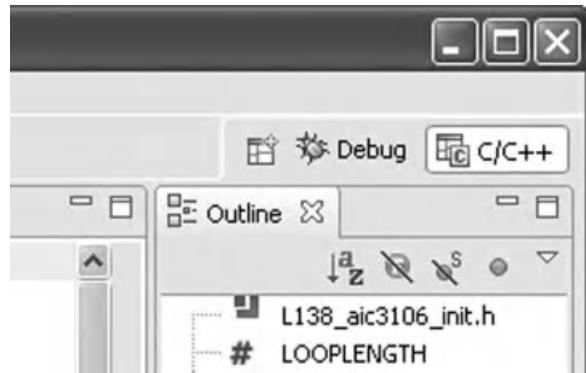


Figure 1.10 Switch between *Debug* and *C/C++* perspectives by clicking these buttons.

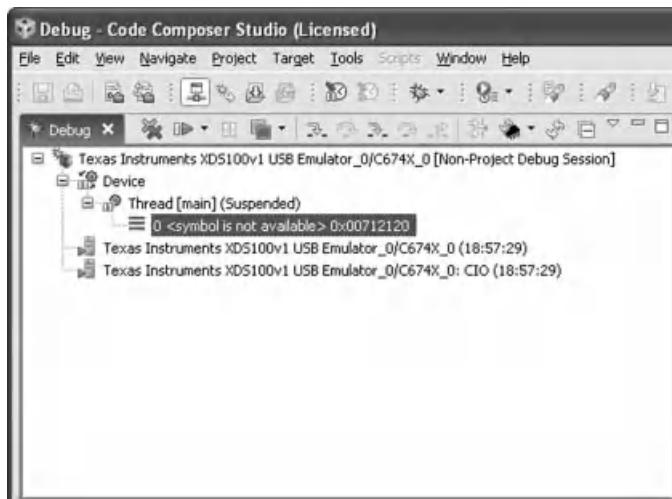


Figure 1.11 *Debug* view as it should appear in the *Debug* perspective after connecting to target (*Connect* icon highlighted).

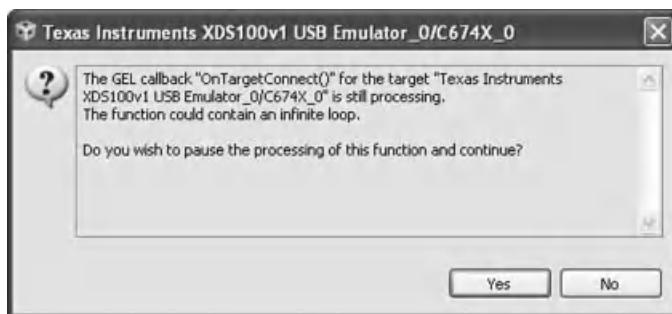


Figure 1.12 Error message window.



Figure 1.13 *Browse* to select program L138_loop_poll.out.

- (14) Select *Target > Load program* and *Browse* to find the executable file L138_loop_poll.out in folder c:\eXperimenter\L138_chapter2\L138_loop_poll\Debug, as shown in Figure 1.13.
- (15) Click *OK* and the *Debug* view should change to that shown in Figure 1.14, indicating that the program has been loaded.
- (16) Execute the program by selecting *Target > Run* or by clicking on the green *Run* toolbar button in the *Debug* window.

Program L138_loop_poll simply reads input samples from the ADC and writes them to the DAC so that an audio signal input to LINE IN, for example, from a PC sound card should be output at LINE OUT.

The program may be halted by selecting *Target > Halt* or by clicking the yellow *Halt* toolbar button in the *Debug* view. The program may be run again after clicking the *Restart* toolbar button. If this is unsuccessful, select *Target > Reset > System Reset* and then *Target > Reload Program* before clicking *Run* again.

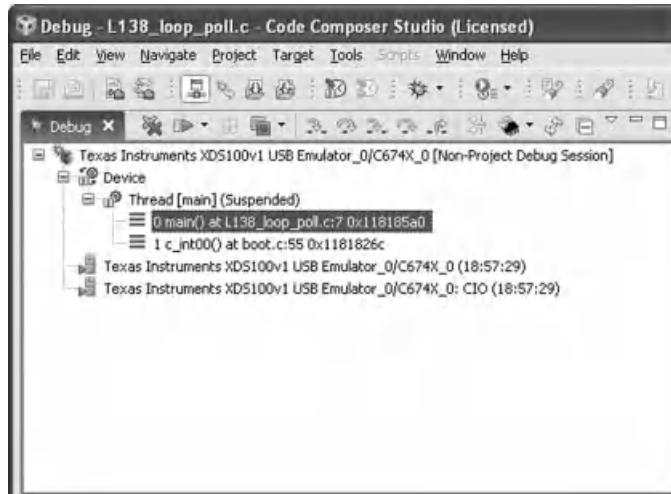


Figure 1.14 *Debug* view as it should appear in the *Debug* perspective after loading executable file L138_loop_poll.out.

1.4 PROGRAMMING EXAMPLES TO TEST THE EXPERIMENTER

Two programming examples are presented in this chapter to illustrate some of the features of the Code Composer Studio software and the eXperimenter board. The aim of these examples is to enable the reader to become familiar with both the software and hardware tools that will be used throughout this book. It is strongly recommended that you complete these two examples before proceeding to subsequent chapters.

EXAMPLE 1.1: Sine Generation Using 48 Points with Output Recorded in a Buffer for Plotting Using Code Composer Studio Software and MATLAB
(L138_sine48_buf_intr)

This example concerns generation of a sinusoidal analog output waveform using a table lookup method. More importantly, it illustrates some of the features of Code Composer Studio software for editing source files, building a project, accessing the code generation tools, and running a program on the OMAP-L138 processor.

Program description

The operation of program L138_sine48_buf_intr.c is as follows. An array, sine_table, of 48 16-bit signed integers is declared and initialized to contain samples of exactly one cycle of a sinusoid. The value of sine_table[i] is equal to

$$10\,000\sin(2\pi i/48), \quad \text{for } i = 0, 1, 2, 3, \dots, 47.$$

Within function main(), a call to function L138_initialise_intr() initializes the eXperimenter board. The AIC3106 codec is configured to operate at a sampling rate of 48 kHz with 0 dB ADC gain and 0 dB DAC attenuation and the McASP0 interface, used to communicate between the OMAP-L138 processor and the AIC3106 codec, is configured so that an interrupt will occur at each sampling instant. Function L138_initialise_intr() is defined in the file L138_aic3106_init.c, which can be found in the folder c:\eXperimenter\L138_support. Its actions are described in more detail in Chapter 2. The program statement while(1); within the function main() creates an infinite loop. Following initialization, the program does nothing except wait for interrupts.

Each time an interrupt occurs, that is, at each sampling instant, function interrupt4() is called. Within that function, the next sample value read from the array sine_table is output via the AIC3106 codec using function output_left_sample(), and the variable sine_ptr, used as an index into the array, is incremented. When the value of sine_ptr exceeds the allowable range for the array sine_table, that is, it exceeds the value LOOPLLENGTH-1, it is reset to zero. The BUFSIZE sample values most recently output via the codec are stored in the array buffer using variable buf_ptr as an index.

A sinusoidal analog output waveform is generated only on the left channel of the AIC3106 codec and via the LINE OUT socket. One cycle of the sinusoidal analog output waveform

```

// L138_sine48_buf_intr.c
//

#include "L138_aic3106_init.h"
#define LOOPLENGTH 48
#define BUFSIZE 256

int16_t sine_table[LOOPLENGTH] =
{0, 1305, 2588, 3827, 5000, 6088, 7071, 7934,
 8660, 9239, 9659, 9914, 10000, 9914, 9659, 9239,
 8660, 7934, 7071, 6088, 5000, 3827, 2588, 1305,
 0, -1305, -2588, -3827, -5000, -6088, -7071, -7934,
 -8660, -9239, -9659, -9914, -10000, -9914, -9659, -9239,
 -8660, -7934, -7071, -6088, -5000, -3827, -2588, -1305};

int16_t sine_ptr = 0; // pointer into lookup table

int32_t buffer[BUFSIZE];
int16_t buf_ptr = 0;

interrupt void interrupt4(void) // interrupt service routine
{
    int16_t sample;

    sample = sine_table[sine_ptr];           // read sample from table
    output_left_sample(sample);             // output sample
    sine_ptr = (sine_ptr+1)%LOOPLENGTH;     // increment table index
    buffer[buf_ptr] = (int32_t)(sample);    // store sample in buffer
    buf_ptr = (buf_ptr+1)%BUFSIZE;          // increment buffer index
    return;
}

int main(void)
{
    L138_initialise_intr(FS_48000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
    while(1);
}

```

Figure 1.15 Listing of program L138_sine48_buf_intr.c.

corresponds to 48 output samples and hence the frequency of the sinusoidal analog output waveform is equal to the codec sampling rate (48 kHz) divided by 48, that is 1 kHz.

The C source file L138_sine48_buf_intr.c listed in Figure 1.15 is provided in the folder c:\eXperimenter\L138_chapter1. Before it can be compiled, assembled, linked, downloaded, and executed on the OMAP-L138, a project must be created around it. The following steps assume that the eXperimenter has been connected to the host PC and powered up and that the Code Composer Studio IDE has been launched as described in the previous section.

Create a project

In the *C/C++* perspective (switch to the *C/C++* perspective by clicking the *C/C++* button in the top right corner of Code Composer Studio software), create a new project in the

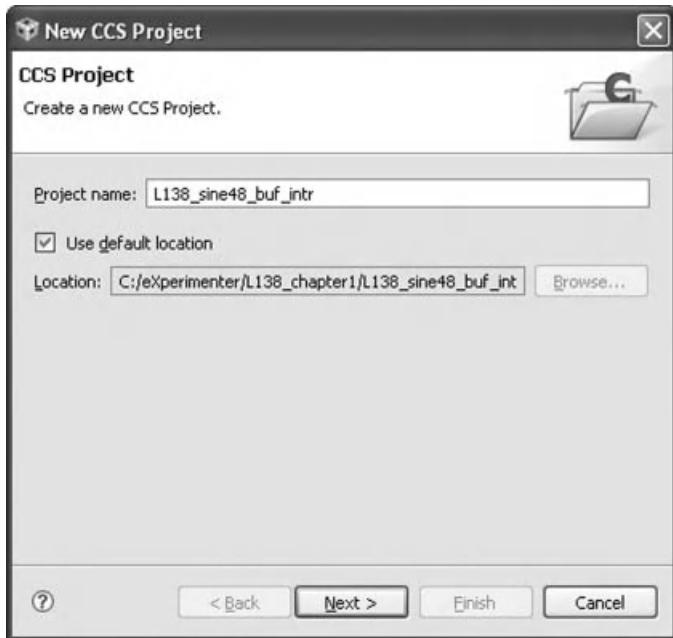


Figure 1.16 New CCS Project window.

workspace L138_chapter1 by selecting *File > New > CCS Project*. You should see a pop-up window similar to that shown in Figure 1.16.

Enter L138_sine48_buf_intr as a project name. This will create a project folder in the c:\eXperimenter\L138_chapter1 workspace. Click *Next* and the window shown in Figure 1.17 should appear.

Select *C6000* as *Project Type* and leave *Debug* and *Release* configuration options checked. Click *Next*.

Do not reference any other projects. In the *Additional Project Settings* window (Figure 1.18), click *Next*. Figure 1.19 shows the Project settings you should use: *Output type Executable*, *Device Variant Generic C674x Device*, *Device Endianness Little*, *Code Generation Tools TI v7.0.3*, and *Runtime Support Library rts6740.lib*.

Click *Finish*. At this point, the *C/C++ Project View* in Code Composer Studio software should appear, as shown in Figure 1.20. A new project has been created but it does not contain any source files. Add source file L138_sine48_buf_intr.c to the new project, using Windows Explorer to copy the source file L138_sine48_buf_intr.c from folder c:\eXperimenter\L138_chapter1 to the newly created project folder c:\eXperimenter\L138_chapter1\L138_sine48_buf_intr. After a few seconds, the file should appear in the *C/C++ Project View* window, as shown in Figure 1.21.

Link support files to project

A number of support files are required in order to build an executable version of the program and these are provided in folder c:\eXperimenter\L138_support. Links to

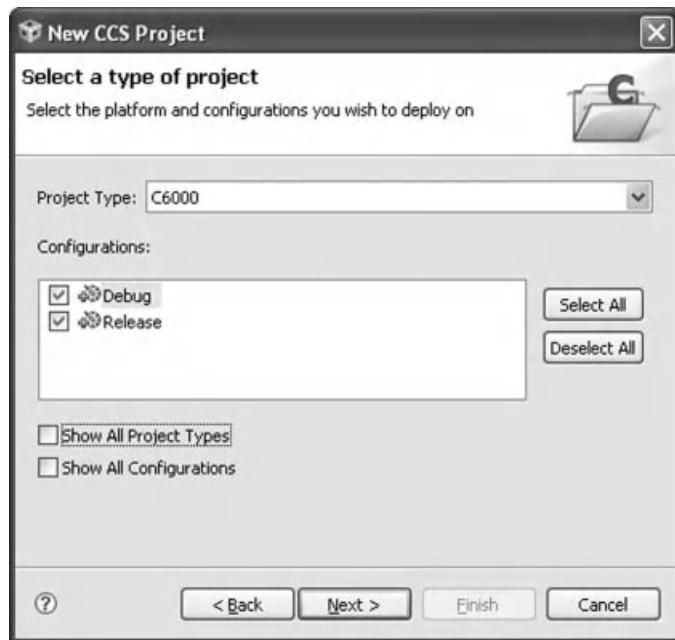


Figure 1.17 Select Project window.

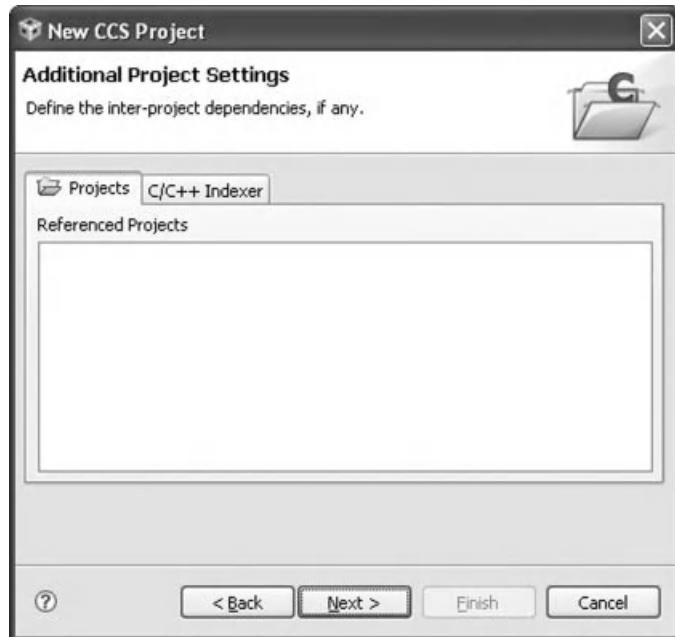


Figure 1.18 Additional Project Settings window.

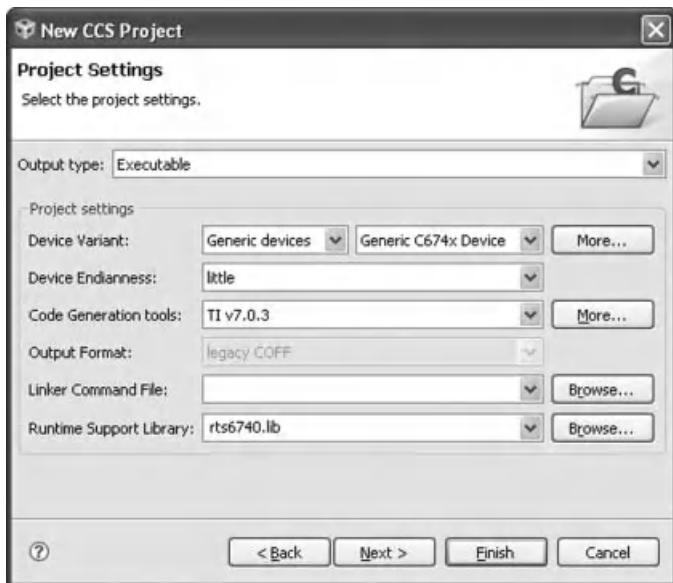


Figure 1.19 Project Settings window.

these files must be made from the L138_sine48_buf_intr project. Select *Project > Link Files to Active Project* and *Open* the following files.

- (1) L138_aic3016_init.c
- (2) L138_aic3016_init.h
- (3) linker_DSP.cmd
- (4) vectors_intr.asm

These should appear in the *C/C++ Project View* window as shown in Figure 1.22.



Figure 1.20 Project View window after creating new project.



Figure 1.21 Project View window after copying source file L138_sine48_buf_intr.c to project folder L138_sine48_buf_intr.

Set build options

Right-click on the project name L138_sine48_buf_intr in the C/C++ Project View window and select *Build Properties*. A new *Properties for L138_sine48_buf_intr (Filtered)* window should appear. In the *Tool Settings* tab under *Configuration Settings*, click *C6000 Compiler > Include Options*. The compiler needs to know the locations of the BSL and of the other support files linked to the project. Click the *Add* button next to *Add dir to #include search path (--include_path, -I)* and add the paths to the required folders. Assuming that the BSL has been installed at c:\omap138 and the files accompanying this book copied to



Figure 1.22 Project View window after linking support files to project L138_sine48_buf_intr.

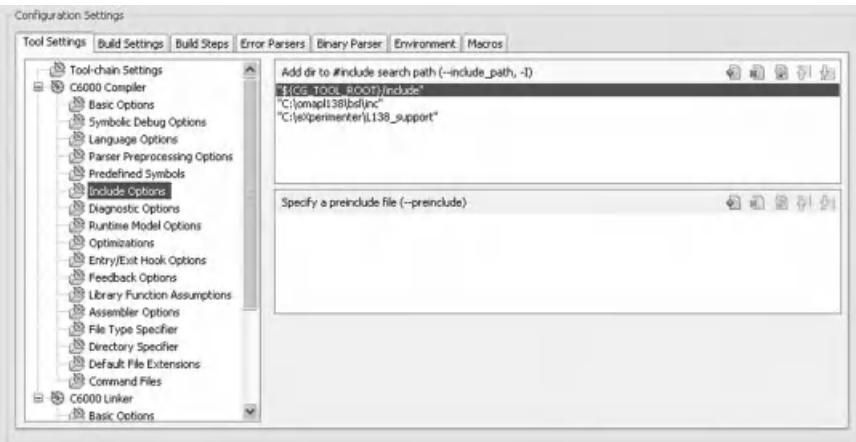


Figure 1.23 Configuration Settings in the *Build Properties* window after making additions to the *C6000 Compiler > Include Options*.

c:\eXperimenter, the appropriate paths will be c:\omap1138\bsl\inc and c:\eXperimenter\L138_support, respectively. Enter these directly or browse for them by clicking the *File System* button in the *Add directory path* pop-up window. After adding these paths, the *Configuration Settings* in the *Build Properties* window should appear as shown in Figure 1.23.

Next, click *C6000 Linker > File Search Path* and Add the library c:\omap1138\bsl\lib\evmomap1138_bsl.lib. The *Build Properties* window should then appear as shown in Figure 1.24.

Click *Apply* and then *OK* to close the *Build Properties* window.

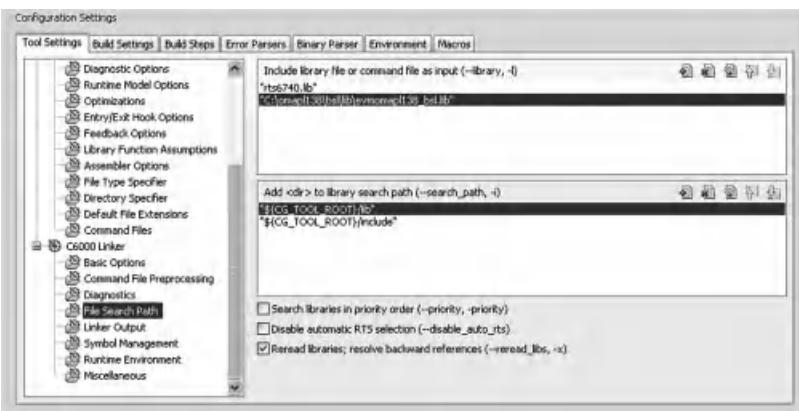


Figure 1.24 Configuration Settings in the *Build Properties* window after making additions to the *C6000 Linker > File Search Path*.

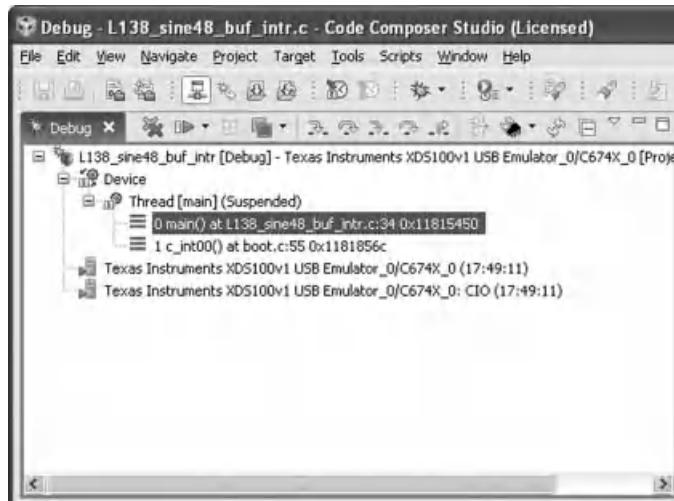


Figure 1.25 Debug window following project build and download.

Building, loading and running the program

If you have not specified a default target configuration file as part of the previous example, copy files C6748.gel and L138_eXperimenter.ccxml from folder `c:\L138_support` to folder `c:\Documents and Settings\YOUR_ID\user\CCSTargetConfigurations`. Select *View > Target Configurations*, right-click on `L138_eXperimenter.ccxml`, and select *Set as Default*.

Select *Target > Debug Active Project* or click the *Debug Active Project* toolbar button. This will build the active project, launch the debugger, and load the program. Once the project has been built and the program downloaded onto the eXperimenter, the Code Composer Studio software should have switched to the *Debug* perspective and a *Debug* window similar to that shown in Figure 1.25 should have appeared. If you still see the *C/C++* perspective, then click the *Debug* button in the top right corner of the Code Composer Studio IDE window.

To run the program, select *Target > Run* or click the *Run* toolbar button. The program should generate a 1 kHz tone via the left channel of LINE OUT. Verify this using headphones or loudspeakers connected to LINE OUT.

The program may be halted by selecting *Target > Halt* or by clicking the *Halt* toolbar button. The program may be run again after clicking the *Restart* toolbar button. If this is unsuccessful, select *Target > Reset > System Reset* and *Target > Reload Program* before clicking *Run* again.

Editing source files within Code Composer Studio software

Edit the source file `L138_sine48_buf_intr.c` in an *Editor* window. If you have carried out the steps described above, then there should already be a tabbed *Editor* window open in the *Debug* perspective containing file `L138_sine48_buf_intr.c` there. If not, then select *File > Open File* and open file `c:\eXperimenter\L138_chapter1\L138_sine48_buf_intr\L138_sine48_buf_intr.c`.

Changing the frequency of the generated sinusoid

There are several different means by which the frequency of the sinusoid generated by program L138_sine48_buf_intr.c can be altered.

- (1) Change the AIC3106 codec sampling frequency from 48 to 24 kHz by changing the line that reads

```
initialise_intr(FS_48000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
```

to read

```
initialise_intr(FS_24000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
```

Select *Target > Reset > CPU Reset* and then *Project > Build Active Project*. Run the program and verify that the frequency of the sinusoid generated is 500 Hz. The different sampling frequencies supported by the AIC3106 codec are 8, 9.6, 11.025, 12, 16, 19.2, 22.05, 24, 32, 44.1, and 48 kHz.

- (2) Change the number of samples stored in the lookup table to 24. By changing the statements

```
#define LOOPLENGTH 48
int16_t sinetable[LOOPLENGTH] =
{0, 1305, 2588, 3827, 5000, 6088, 7071, 7934,
8660, 9239, 9659, 9914, 10000, 9914, 9659, 9239,
8660, 7934, 7071, 6088, 5000, 3827, 2588, 1305,
0, -1305, -2588, -3827, -5000, -6088, -7071, -7934,
-8660, -9239, -9659, -9914, -10000, -9914, -9659, -9239,
-8660, -7934, -7071, -6088, -5000, -3827, -2588, -1305};
```

to

```
#define LOOPLENGTH 24
int16_t sinetable[LOOPLENGTH] =
{0, 2588, 5000, 7071,
8660, 9659, 10000, 9659,
8660, 7071, 5000, 2588,
0, -2588, -5000, -7071,
-8660, -9659, -10000, -9659,
-8660, -7071, -5000, -2588};
```

Select *Target > Reset > System Reset* and then *Project > Build Active Project*. Run the program and verify that the frequency of the sinusoid generated is 2 kHz (assuming a 48 kHz sampling frequency).

Plotting memory contents using the Code Composer Studio IDE

In addition to generating an analog voltage waveform, program L138_sine48_buf_intr.c uses array buffer to store the BUFSIZE most recent output sample

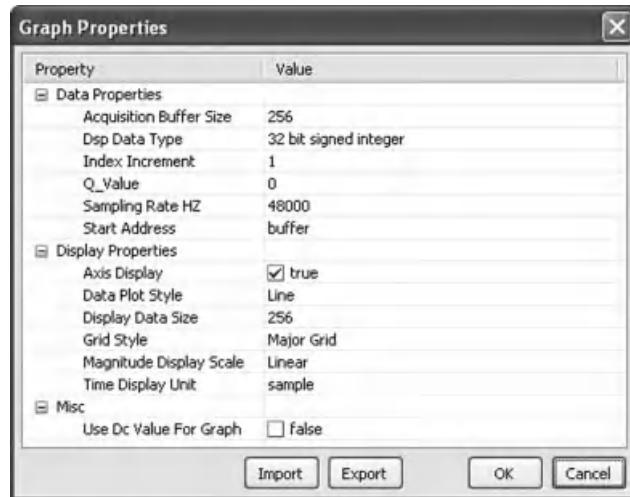


Figure 1.26 *Graph Properties* for plotting contents of array `buffer`.

values. Once the program has been halted, Code Composer Studio software may be used to plot data in both time and frequency domains.

Halt program execution by clicking the yellow *Halt* toolbar button in the *Debug* window. In order to plot a graph of the contents of the array `buffer`, select *Tools > Graph > Single Time* and set the *Graph Properties*, as shown in Figure 1.26. You should then see a graph of 256 buffered output samples, as shown in Figure 1.27. This assumes that the original version of the program has been run.

A frequency domain representation of the signal generated can be produced by selecting *Tools > Graph > FFT Magnitude* and setting the *Graph Properties*, as shown in Figure 1.28. Because the 256 buffered output samples do not represent an integer number of cycles of the 1 kHz sinusoid, spectral leakage is evident in Figure 1.29. Nonetheless, the 1 kHz tone is represented clearly in the resulting graph.

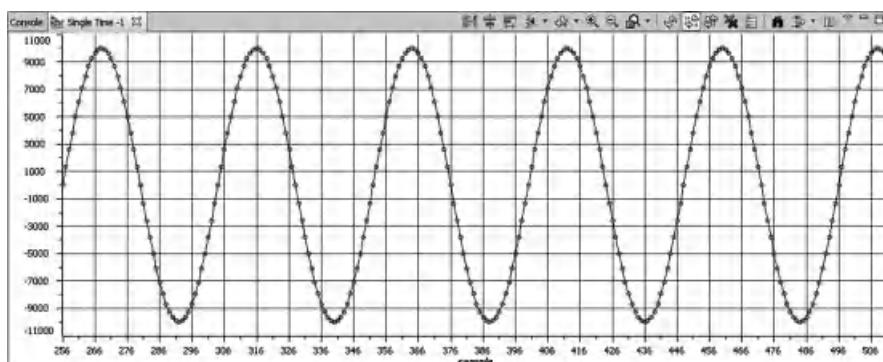


Figure 1.27 Contents of array `buffer` following execution of program `L138_sine48_buf_intr.c`.

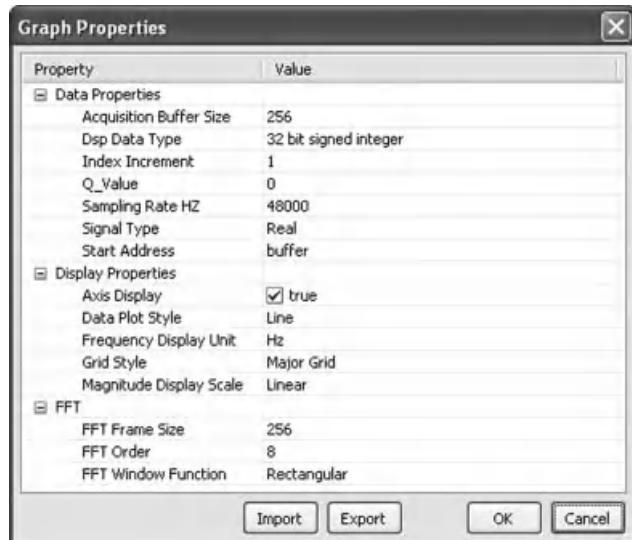


Figure 1.28 Graph properties used to plot magnitude of FFT of contents of array buffer.

Plotting memory contents using MATLAB

An alternative and more flexible method of plotting the buffered output samples is to save the buffer contents in a file and use MATLAB as a graph plotting tool.

In order to save the buffered samples to a file, select *View > Memory* and enter *buffer* as the *Address Text* and *32 Bit Signed Integer* as the *Data Type*, as shown in Figure 1.30.

Click the *Save* button in the *Memory* view, choose a filename, and fill in the details, as shown in Figure 1.31.

Although the values stored in array *sine_table* and passed to function *output_left_sample()* are 16-bit signed integers, array *buffer* is declared as type

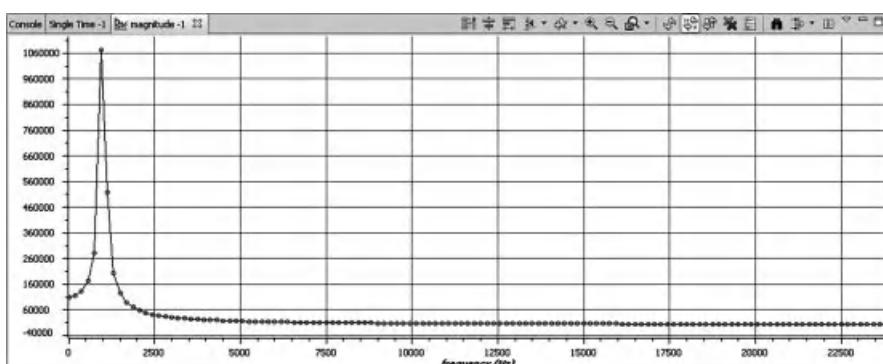


Figure 1.29 Magnitude of FFT of contents of array buffer following execution of program L138_sine48_buf_intr.c.

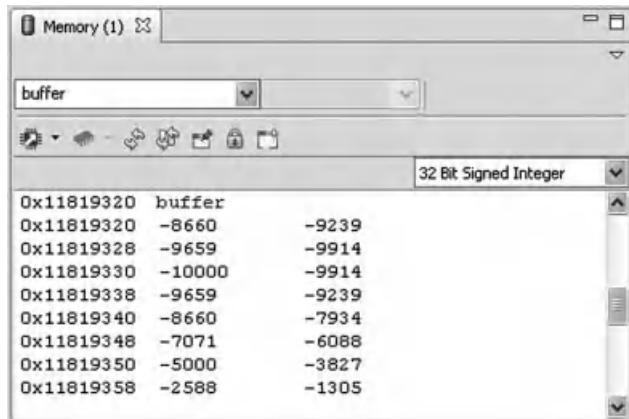


Figure 1.30 *Memory View* window showing contents of array *buffer*.

`int32_t` (32-bit signed integer) in program `L138_sine48_buf_intr.c` to allow for the fact that there is no 16-bit Signed Integer data type option in the *Save Memory* window, as shown in Figure 1.31.

The *TI Data Format (.dat)* file saved can be opened and its contents plotted using the MATLAB function `L138_logfft()`.

Type `L138_logfft` at the MATLAB command line and enter the filename, sampling frequency, frequency scale type, and the number of sample values to be plotted when prompted. Figures 1.32 and 1.33 show the resultant plots. A listing of `L138_logfft.m` is given in Chapter 2.

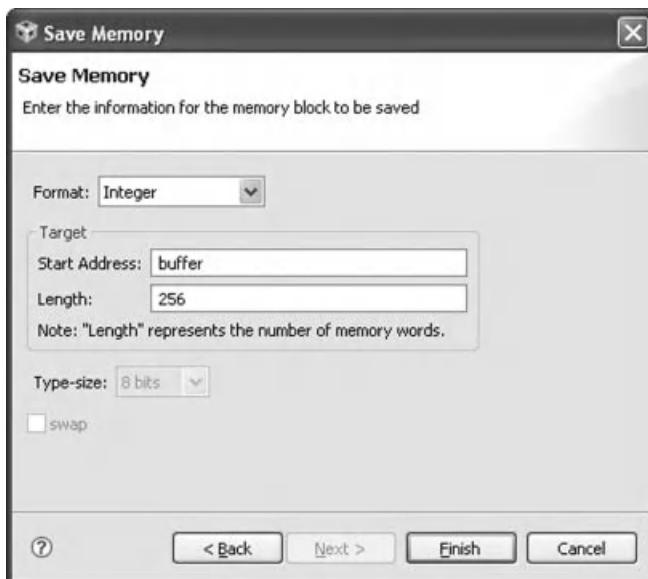


Figure 1.31 *Save Memory* window showing parameters for saving contents of array *buffer*.

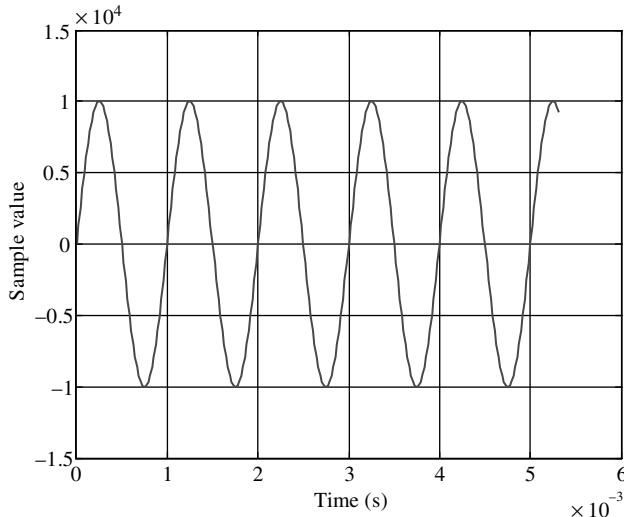


Figure 1.32 Contents of array `buffer` plotted using MATLAB function `L138_logfft()`.

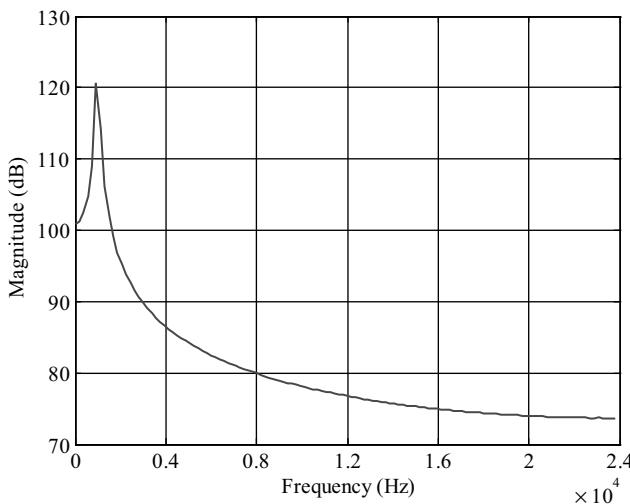


Figure 1.33 Magnitude of FFT of contents of array `buffer` plotted using MATLAB function `L138_logfft()`.

EXAMPLE 1.2: Dot Product of Two Arrays (L138_dotp4)

This example illustrates the use of breakpoints and *Watch* windows within the Code Composer Studio IDE. In addition, it illustrates how to use the Code Composer Studio software *Profile Clock* in order to estimate the time taken to execute a section of code.

```
// L138_dotp4.c
//

#include <stdio.h >
#define count 4

int dotp(short *a, short *b, int ncount);

short x[count] = {1,2,3,4};
short y[count] = {0,2,4,6};

main()
{
    int result = 0;

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

int dotp(short *a, short *b, int ncount)
{
    int i;
    int sum = 0;

    for (i=0 ; i< count ; i++)
        sum += a[i] *b[i];

    return(sum);
}
```

Figure 1.34 Listing of program L138_dotp4.c.

The C source file L138_dotp4.c, listed in Figure 1.34, calculates the dot product of two arrays of integer values. The first array is initialized using the four values 1, 2, 3, and 4, and the second array using the four values 0, 2, 4, and 6. The dot product is equal to $(1 \times 0) + (2 \times 2) + (3 \times 4) + (4 \times 6) = 40$.

The program can readily be modified to handle larger arrays. No real-time input or output is used in this example, and so the real-time support files L138_aic3106_init.c, L138_aic3106_init.h, and vectors_intr.asm are not needed.

In the C/C++ perspective, click *Project > Import Existing CCS/CCE Eclipse Project* and *Browse for search-directory c:\eXperimenter\138_chapter1\138_dotp4*. The *Import CCS Eclipse Projects* window should appear, as shown in Figure 1.35. Make sure that *Discovered Project L138_dotp4* is checked.

Click *Finish* and the L138_dotp4 project should appear in the *C/C++ Project View* window, along with the L138_sine48_buf_intr project. The L138_dotp4 project should be *Active*. If it is not, right-click on the project name L138_dotp4 in the *C/C++ Project View* window and select *Set as Active Project* (Figure 1.36).

Select *Target > Debug Active Project* or click the *Debug Active Project* toolbar button. The *Debug* window in the *Debug* perspective should then appear, as shown in Figure 1.37.

Click on the *Run* toolbar button in the *Debug* window and you should see the text

```
result = 40 (decimal)
```

appear in the console. The program can be run again, without reloading, after clicking the *Restart* toolbar button in the *Debug* window.



Figure 1.35 Import CCS Eclipse Projects window.

Use of breakpoints and watch window

Open a Watch window in the Debug perspective by selecting *View > Watch*. Then, enter the variable names *sum* and *i* in the Watch window, as shown in Figure 1.38.

At this point, the Watch window will report that it cannot find the identifiers *sum* and *i*. This is because they are declared locally in function *dotp()*. When the program is halted within that function, their values will be displayed in the Watch window. Set a breakpoint at line



Figure 1.36 Project View window as it should appear after importing project L138_dotp4.

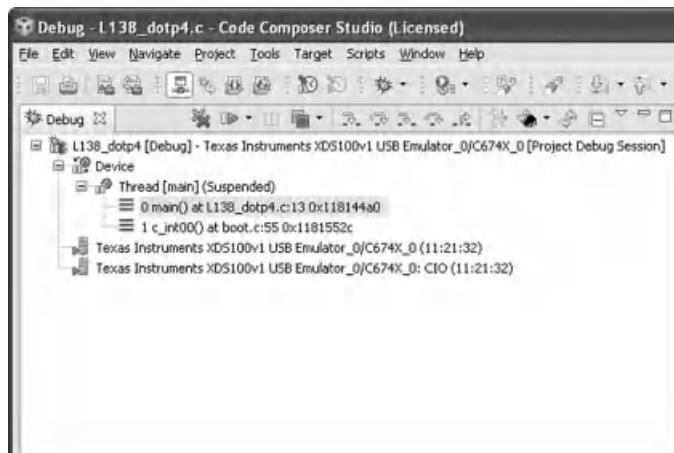


Figure 1.37 Debug window as it should appear after selecting *Target > Debug Active Project*.

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

by double-clicking in the left margin beside that line in the *Editor* window in which a listing of the file `L138_dotp4.c` appears. A symbol should appear in the left margin, as shown in Figure 1.39.

Restart and *Run* the program. Execution should halt at the breakpoint and when this happens, the *Watch* window should display the values of variables `sum` and `i` (Figure 1.40). Click on the *Run* toolbar button several more times and you should see the values change as execution progresses (and function `dotp()` is called repeatedly). Once execution of the program is complete and the message

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

is displayed in the console, the *Watch* window will once again report that identifiers `sum` and `i` cannot be found.

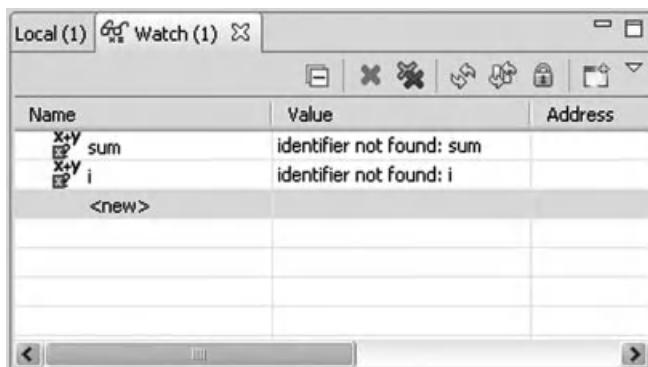
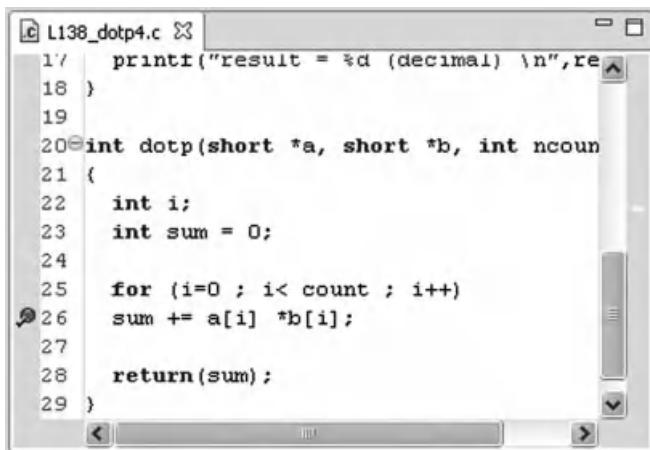


Figure 1.38 Watch window after entering variable names `sum` and `i`.



```

17     printf("result = %d (decimal) \n",re
18 }
19
20 int dotp(short *a, short *b, int ncount)
21 {
22     int i;
23     int sum = 0;
24
25     for (i=0 ; i< count ; i++)
26         sum += a[i] *b[i];
27
28     return(sum);
29 }

```

Figure 1.39 Breakpoint indicated in left margin of program listing.

If the program is edited so that the variables `sum` and `i` are declared as global variables, then their values can be displayed in the *Watch* window at all times. Cut the lines that read

```
int i;
int sum = 0;
```

from the definition of function `dotp()` and paste them immediately following the function declaration

```
int dotp(short *a, short *b, int ncount);
```

Rebuild and reload the program (by clicking *Build Active Project* or by selecting *Project > Build Active Project*) and repeat the previously described steps involving the breakpoint and *Watch* window in order to verify that the values of `sum` and `i` can be displayed at all times.

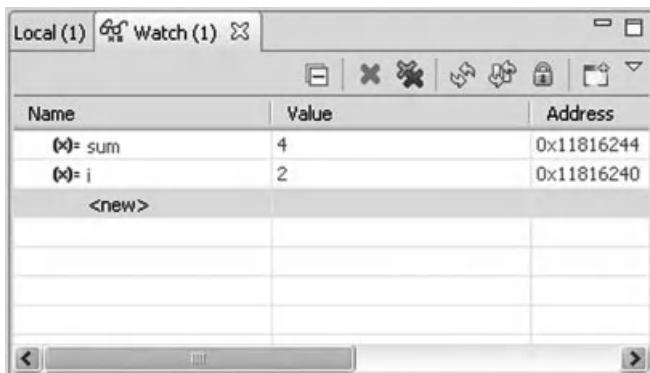


Figure 1.40 *Watch* window showing the values of variables `sum` and `i` during execution of program `L138_dotp4.c`.

Estimating execution time for function `dotp()` using the profile clock

The time taken to execute function `dotp()` can be estimated using the Code Composer software *Profile Clock*. In the *Debug* perspective,

- (1) Select *Target > Clock > Enable*.
- (2) Click the *Restart* toolbar button in the *Debug* window.
- (3) Clear all existing breakpoints (by double-clicking on the breakpoint symbols in the left-hand margin of the Editor window) and then set breakpoints at the lines
- (4) `result = dotp(x, y, count);` and `printf("result = %d (decimal)\n", result);`
- (5) Run the program. It should halt at the first breakpoint.
- (6) Reset the *Profile Clock* by double-clicking on its icon in the bottom right-hand corner of the Code Composer Studio IDE window.
- (7) Run the program. It should halt at the second breakpoint.

The number of instruction cycles counted by the profile clock between the two breakpoints, that is, during the execution of function `dotp()`, should be displayed next to the *Profile Clock* icon. On a 375 MHz C6748 processor, each instruction cycle takes 2.67 ns. With the default *Build Properties* setting of no compiler optimization, the function should take around 112 instruction cycles to execute.

To change the compiler optimization level, right-click on the project name L138_dotp4 in the *C/C++* perspective *Project View* window and select *Build Properties*. In the *Tool Settings* tab under *Configuration Settings*, click on *C6000 Compiler > Basic Options* and set the *Optimization Level*.

Repeat the experiment having set the compiler *Optimization Level* to 2 and you should see a significant reduction in the number of instruction cycles used by function `dotp()`. Using breakpoints and the profile clock can give an indication of the execution times of sections of program, but it does not always work with higher levels of compiler optimization. More detailed profiling of program execution can be achieved using a simulator.

1.5 SUPPORT FILES

The support files L138_aic3106_init.c, L138_aic3106_init.h, vectors_intr.asm, vectors_poll.asm, L138_eXperimenter.ccxml, C6748.gel, and linker_dsp.cmd are used by nearly all the examples in this book. These files are stored in folder c:\eXperimenter\L138_support.

1.5.1 Initialization and Configuration File (L138_aic3106_init.c)

Source file L138_aic3106_init.c contains definitions for a number of functions that may be called from C source files and which are used in many of the example programs in the following chapters.

Functions `L138_initialise_poll()`, `L138_initialise_intr()`, and `L138_initialise_edma()` initialize communications between the OMAP-L138 processor and the AIC3106 codec for polling-, interrupt-, or DMA-based input and output. In each case, McASP0 is configured slightly differently, using functions `L138_mcasp_init_poll()`, `L138_mcasp_init_intr()`, and `L138_mcasp_init_edma()`, respectively. In the case of DMA-based input and output, the parameters of the EDMA3 controller are configured using function `EDMA3_PaRAM_setup()`. More details of the three different I/O methods are given in Chapter 2.

Functions `input_sample()`, `input_left_sample()`, `input_right_sample()`, `output_sample()`, `output_left_sample()`, and `input_right_sample()` are used to read and write data to and from the codec. These functions make calls to lower level functions provided by the BSL `evmomap138_bsl.lib`.

Functions `prand()` and `prbs()` are concerned with generating pseudorandom noise sequences. `prand()` uses a Parks–Miller method to generate a pseudorandom sequence of integer values. `prbs()` uses a shift register to generate a PRBS sequence.

1.5.2 Header File (`L138_aic3106_init.h`)

The corresponding header support file `L138_aic3106_init.h` contains function prototypes as well as definitions of several constants.

1.5.3 Vector Files (`vectors_intr.asm` and `vectors_poll.asm`)

To make use of CPU interrupt INT4, a branch instruction (jump) to the interrupt service routine (ISR) `interrupt4()` defined in a C program, for example, `L138_sine48_buf_intr.c`, must be placed at the appropriate point, as part of a fetch packet, in the interrupt service table (IST). Assembly language file `vectors_intr.asm`, which sets up the IST, is listed in Figure 1.41. Note the underscore preceding the name of the routine or function being called. By convention, this indicates a C function.

For programs that use polling-based I/O, file `vectors_poll.asm` is used in place of `vectors_intr.asm`. The main difference between these files is that there is no branch to function `interrupt4()` in the IST set up by `vectors_poll.asm`. Common to both files is a branch to `c_int00()`, the start of a C program, associated with the reset interrupt (Figures 1.41 and 1.42).

```

; vectors_intr.asm
;
; interrupt service table for interrupt- and DMA-based i/o
; example programs

.global _vectors
.global _c_int00
.global _vector1
.global _vector2
.global _vector3
.global _interrupt4      ; interrupt service routine
.global _vector5
.global _vector6
.global _vector7
.global _vector8
.global _vector9
.global _vector10
.global _vector11
.global _vector12
.global _vector13
.global _vector14
.global _vector15

.ref _c_int00           ; C program entry address

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

; dummy interrupt service fetch packet
_vec_dummy:
    B      B3
    NOP    5

; This is the actual interrupt service table (IST).
.sect ".vecs"
.align 1024

_vectors:
_vector0:  VEC_ENTRY _c_int00      ;RESET

```

Figure 1.41 Listing of interrupt vector file vectors_intr.asm.

```

_vector1:  VEC_ENTRY _vec_dummy      ;NMI
_vector2:  VEC_ENTRY _vec_dummy      ;RSVD
_vector3:  VEC_ENTRY _vec_dummy      ;RSVD
_vector4:  VEC_ENTRY _interrupt4    ;Interrupt4 ISR
_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 _vec_dummy
_vector12: VEC_ENTRY _vec_dummy
_vector13: VEC_ENTRY _vec_dummy
_vector14: VEC_ENTRY _vec_dummy
_vector15: VEC_ENTRY _vec_dummy

```

Figure 1.41 (Continued)

1.5.4 Linker Command File (`linker_dsp.cmd`)

Linker command file `linker_dsp.cmd` is listed in Figure 1.43. It specifies the memory configuration of the internal and external memory available on the eXperimenter board and the placing of sections of code and data to absolute addresses in that memory. For example, the `.text` section, produced by the C compiler, is placed into level 2 RAM/cache, that is, the internal memory of the C6748 digital signal processor, starting at address 0x11800000. The section `.vecs` created by `vectors_intr.asm` or by `vectors_poll.asm` is mapped into IVECS, that is, internal memory starting at address 0x00000000 (the interrupt service table). Chapter 2 contains an example illustrating the use of the *pragma* directive to create a section named `EXT_RAM` to be mapped into external memory starting at address 0xC0000000 (SDRAM).

```

; vectors_poll.asm
;
; interrupt service table for polling-based i/o
; example programs

.global _vectors
.global _c_int00
.global _vector1
.global _vector2
.global _vector3
.global _vector4
.global _vector5
.global _vector6
.global _vector7
.global _vector8

```

Figure 1.42 Listing of interrupt vector file `vectors_poll.asm`.

```

.global _vector9
.global _vector10
.global _vector11
.global _vector12
.global _vector13
.global _vector14
.global _vector15

.ref _c_int00           ; C program entry address

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

; dummy interrupt service fetch packet
_vec_dummy:
    B      B3
    NOP    5

; This is the actual interrupt service table (IST).
.sect ".vecs"
.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    ;RSVD
_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 _vec_dummy
_vector12:  VEC_ENTRY _vec_dummy
_vector13:  VEC_ENTRY _vec_dummy
_vector14:  VEC_ENTRY _vec_dummy
_vector15:  VEC_ENTRY _vec_dummy

```

Figure 1.42 (Continued)

```
/* linker_dsp.cmd */

-stack          0x00000800
-heap           0x00010000

MEMORY
{
    dsp_12_ram:      ORIGIN = 0x11800000 LENGTH = 0x00040000
    shared_ram:      ORIGIN = 0x80000000 LENGTH = 0x00020000
    external_ram:    ORIGIN = 0xC0000000 LENGTH = 0x08000000
}

SECTIONS
{
    .text      > dsp_12_ram
    .const     > dsp_12_ram
    .bss       > dsp_12_ram
    .far       > dsp_12_ram
    .switch    > dsp_12_ram
    .stack     > dsp_12_ram
    .data      > dsp_12_ram
    .cinit     > dsp_12_ram
    .sysmem   > dsp_12_ram
    .cio       > dsp_12_ram
    .vecs      > dsp_12_ram
    .EXT_RAM   > external_ram
}
```

Figure 1.43 Listing of linker command file `linker_dsp.cmd`.

EXERCISES

1. Modify program `L138_sine48_buf_intr.c` to generate a sine wave with a frequency of 3000 Hz. Verify your result using an oscilloscope connected to the LINE OUT socket on the eXperimenter as well as by using Code Composer to plot the 256 most recently output samples in both time and frequency domains.
2. Write an interrupt-driven program that maintains a buffer containing the 128 most recent input samples read at a sampling frequency of 16 kHz from the AIC3106 codec. Halt the program and plot the buffer contents using Code Composer.
3. Write a program that reads input samples from the left channel of the AIC3106 codec ADC at a sampling frequency of 32 kHz using function `input_left_sample()` and writes each sample value to the right channel of the AIC3106 codec DAC using function `output_right_sample()`. Verify the effective connection of the left-hand channel of the LINE IN socket to the right-hand channel of the LINE OUT socket using a signal generator and an oscilloscope. Gradually increase the frequency of the input signal until the amplitude of the output signal is reduced drastically. This frequency corresponds to the bandwidth of the DSP system (discussed in more detail in Chapter 2).

REFERENCES

1. *OMAP-L138 C6-Integra™ DSP +ARM® Processor*, SPRS586C, Texas Instruments, Dallas, TX, 2011.
2. *TMS320C6748 Fixed/Floating-Point DSP*, SPRS590C, Texas Instruments, Dallas, TX, 2011.
3. *Low-Power Stereo Audio Codec for Portable Audio/Telephony*, SLAS509E, Texas Instruments, Dallas, TX, 2008.
4. http://www.logicpd.com/sites/default/files/1013568D_OMAP-L138_experimenter_Brief.pdf.
5. <http://processors.wiki.ti.com/index.php/CCSV4>.
6. http://processors.wiki.ti.com/index.php/Main_Page.
7. <http://eclipse.org>.
8. <http://processors.wiki.ti.com/index.php/XDS100>.
9. http://processors.wiki.ti.com/index.php/C674x_DSPLIB.

Chapter 2

Analog Input and Output with the OMAP-L138 eXperimenter

- Analog input and output using the on-board AIC3106 stereo codec
- Programming examples using C code

2.1 INTRODUCTION

A basic DSP system, suitable for processing audio frequency signals, comprises a digital signal processor and analog interfaces, as shown in Figure 2.1. The eXperimenter provides just such a system, using the C6748 floating-point processor core contained within an OMAP-L138 device and a TLV320AIC3106 (AIC3106) codec [1]. The term codec refers to the coding of analog waveforms as digital signals and the decoding of digital signals as analog waveforms. The AIC3106 codec performs both the analog-to-digital conversion (ADC) and digital-to-analog conversion (DAC) functions shown in Figure 2.1.

The AIC3106 communicates with the C6748 via multichannel audio serial port (McASP) and I₂C interfaces and these are also made available on the eXperimenter board via an audio expansion connector, allowing the addition of further codecs. The programming examples in this book use only the eXperimenter's on-board AIC3106 codec.

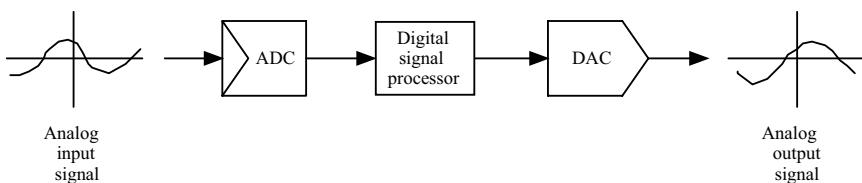


Figure 2.1 Basic digital signal processing system.

2.2.1 Sampling, Reconstruction, and Aliasing

Within digital signal processors, signals are represented as sequences of discrete samples and whenever signals are sampled, the possibility of aliasing arises. Later in this chapter, the phenomenon of aliasing is explored in more detail. Suffice to say at this stage that aliasing is undesirable and that it may be avoided by the use of an antialiasing filter placed at the input to the system shown in Figure 2.1, and by suitable design of the DAC. In a low-pass system, an effective antialiasing filter is one that allows frequency components below half the sampling frequency to pass but which attenuates greatly, or stops, frequency components equal to or greater than half the sampling frequency. A suitable DAC for a low-pass system is itself a low-pass filter having characteristics similar to the aforementioned antialiasing filter. The term DAC commonly refers to an electronic device that converts sample values represented in digital hardware into an analog electrical signal. Viewed purely from a signal processing perspective, a DAC is a reconstruction filter. The AIC3106 codec contains both digital and analog antialiasing and reconstruction filters and, therefore, does not require additional, external antialiasing filters.

2.2 TLV320AIC3106 (AIC3106) ON-BOARD STEREO CODEC FOR ANALOG INPUT AND OUTPUT

The eXperimenter makes use of a TLV320AIC3106 (AIC3106) codec for analog input and output. The AIC3106 is a low-power stereo audio codec, based on sigma-delta technology and designed for use in portable, battery-powered applications. It features a number of microphone and line level inputs, configurable for single-ended or differential connection. On its output side, a number of differential and high-power outputs are provided. The high-power outputs are capable of driving headphones. A number of different sampling rates ranging from 8 to 96 kHz are supported by the device.

The analog-to-digital converter, or coder, part of the codec converts an analog input signal into a sequence of (16-, 24-, or 32-bit signed integer) sample values to be processed by the digital signal processor. The digital-to-analog converter, or decoder, part of the codec reconstructs an analog output signal from a sequence of (16-, 24-, or 32-bit signed integer) sample values that have been processed by the digital signal processor and written to the DAC.

Also contained in the device are a number of programmable digital filters and gain blocks (Figures 2.2 and 2.3). The codec is configured using a number of control registers, offering so many options that it is beyond the scope of this text to describe them fully. However, the choice of sampling frequency, input gain, and output attenuation is made available in the example programs through the parameters passed to functions `L138_initialise_poll()`, `L138_initialise_intr()`, and `L138_initialise_edma()`. Later in this chapter, high-pass filtering of the input signal and de-emphasis of the output signal, and how these may be achieved by writing to the control registers of the AIC3106, are described. In Chapter 4, the programmable digital filters within the AIC3106 are examined in greater detail.

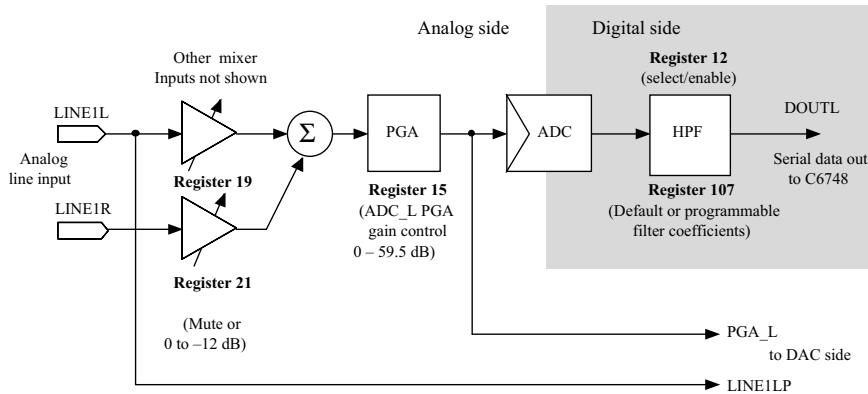


Figure 2.2 Simplified block diagram of analog-to-digital side of TLV320AIC3106 codec showing selected blocks and signal paths used by the example programs in this book (left channel only).

Data is passed to and from the AIC3106 via its McASP serial interface and control is via its I2C interface.

Most of the programming examples in this book configure the codec for a sampling rate of either 8, 16, or 48 kHz, 32-bit stereo data transfer (independent 16-bit left and right channel samples) and 0 dB gain in the LINE IN and LINE OUT signal paths (input gain and output attenuation).

On the eXperimenter board, only the LINE1/MIC1 analog input and the LINE OUT analog output connections on the codec are made available to the user. They are connected to 3.5 mm stereo jack sockets via AC-coupling networks, as shown in Figure 2.4. The maximum allowable input signal level at the analog inputs to the codec is 0.707 V rms and it is important that this signal level is not exceeded otherwise damage to the codec input circuits may occur.

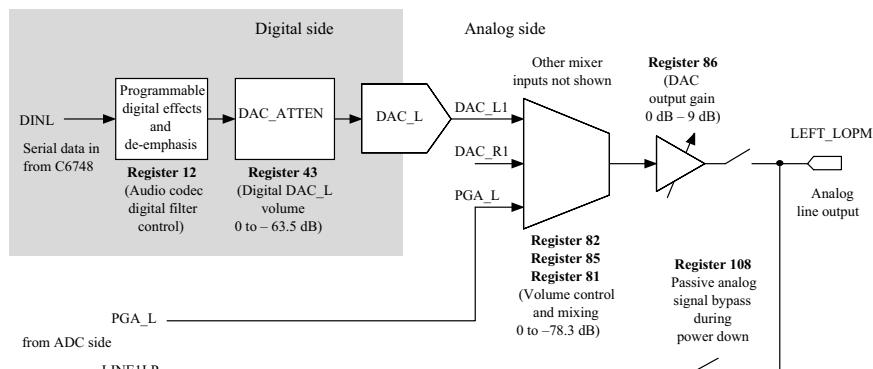


Figure 2.3 Simplified block diagram of digital-to-analog side of TLV320AIC3106 codec showing selected blocks and signal paths used by the example programs in this book (left channel only).

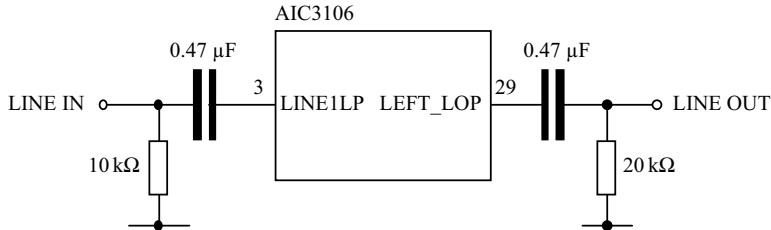


Figure 2.4 AC-coupling of AIC3106 codec analog input and output connections on the eXperimenter board (left channel only).

2.3 PROGRAMMING EXAMPLES USING C CODE

The following examples illustrate analog input and output using the eXperimenter. They are included in order to introduce both the eXperimenter hardware and the Code Composer Studio development environment. The example programs demonstrate some important concepts associated with analog-to-digital and digital-to-analog conversion, including sampling, reconstruction, and aliasing. In addition, they illustrate the use of polling-, interrupt-, and direct memory access (DMA)-based I/O in order to implement real-time applications using the eXperimenter. The concepts and techniques described in this chapter are used again in subsequent chapters.

2.3.1 Real-Time Input and Output Using Polling, Interrupts, and Direct Memory Access

Three basic forms of real-time I/O are demonstrated in the following examples. Polling- and interrupt-based methods work on a sample-by-sample basis. DMA-based I/O deals with blocks, or frames, of input and output samples and is inherently more efficient in terms of computer processing requirements. Block- or frame-based processing is closely linked to, but not restricted to use with, frequency domain processing (using the FFT), as described in Chapter 5.

The following examples illustrate implementation of the three different I/O mechanisms on the eXperimenter. Through the rest of this book use is made of all three. Compared to polling- and interrupt-based methods, there is a greater time delay introduced by the DMA-based method, although it is possible (as illustrated in Example 2.3) to make use of the DMA mechanism with a frame size of just one sample per channel.

The programs described make minimum use of support from the OMAP-L138 board support library (BSL) supplied by the board manufacturer. In Chapter 7, program examples using interrupt- and DMA-based I/O methods with the real-time operating system DSP/BIOSTM [2] are described. Example programs using the Texas Instruments DSP/BIOS™ software kernel foundation platform support package (PSP) that use DMA-based I/O exclusively are also described in Chapter 7.

EXAMPLE 2.1: Basic Input and Output Using Polling (L138_loop_poll)

The C language source file for program L138_loop_poll.c, which simply copies input samples read from the AIC3106 codec ADC back to the AIC3106 codec DAC as output samples is listed in Figure 2.5. Effectively, the LINE IN input socket is connected straight through to the LINE OUT socket on the eXperimenter via the AIC3106 codec and the C6748 digital signal processor.

Input and output functions defined in support file 1138_aic3106_init.c

The functions L138_initialise_poll(), input_sample(), and output_sample() called by program L138_loop_poll.c are defined in the support file L138_aic3106_init.c. In this way, the C source file L138_loop_poll.c is kept as short as possible and potentially distracting low-level detail is hidden. The implementation details of these, and other, functions defined in L138_aic3106_init.c need not be studied in depth in order to carry out the examples presented in this book and hence are described only briefly.

L138_initialise_poll() carries out a number of initialization tasks, including setting the value of the global variable poll equal to 1 and configuring the AIC3106 codec and the McASP serial connection between the codec and the C6748. In order to achieve this, further calls are made by L138_initialise_poll() to lower level functions contained in the board support library evmomap1138_bsl.lib.

The AIC3106 can be configured for many different modes of operation and has many programmable parameters, several of which are demonstrated later in this chapter. For the sake of brevity, only three aspects of its configuration are made accessible via the parameters passed to function L138_initialise_poll(). These are the sampling frequency fs, the programmable analog gain applied immediately preceding the ADC adc_gain, and the programmable digital attenuation applied immediately preceding the DAC dac_atten.

Functions input_sample() and output_sample() are used in polling- and interrupt-based I/O to pass pairs of samples (left and right channels) between C6748 and AIC3106.

In polling mode (poll = 1), function input_sample() polls, or tests, the receive ready bit (RRDY) of the McASP serializer control register 12 (SRCTRL12) until

```
// L138_loop_poll.c
//

#include "L138_aic3106_init.h"

int main(void)
{
    uint32_t sample;

    L138_initialise_poll(FS_48000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
    while(1)
    {
        sample = input_sample();
        output_sample(sample);
    }
}
```

Figure 2.5 Listing of loop program using polling (L138_loop_poll.c).

this indicates that newly converted data is available to be read from XRBUF12. Function `output_sample()` polls, or tests, the transmit ready bit (XRDY) of the McASP serializer control register 11 (SRCTL11) until this indicates that the codec is ready to receive a new output sample. A new output sample is sent to the codec by writing to XRBUF11.

The McASP interface [3] contains 16 serializers. On the eXperimenter board serializers 11 and 12 (corresponding to hardware pins AXR11 and AXR12, respectively) are connected to codec data lines DIN and DOUT.

In interrupt mode (`poll = 0`), functions `input_sample()` and `output_sample()`, do not test the RRDY or XRDY bits of the serializer control registers before reading from XRBUF12 or writing to XRBUF11 because it is assumed that they have been called at the appropriate time. DMA-based I/O does not use functions `input_sample()` and `output_sample()`.

Format of data transferred to and from the AIC3106 codec

In all the example programs in this book, the AIC3106 ADC is configured to convert left- and right-hand channel analog input signals into 16-bit signed integer samples and the DAC is configured to convert 16-bit signed integer samples into left- and right-hand channel analog output signals. Left and right channel samples are combined to form 32-bit values that are communicated via the multichannel audio serial port (McASP) to and from the C6748. Access to the ADC and DAC from a C program is via the functions

```
int32_t input_sample(), int16_t input_left_sample(),
int16_t input_right_sample(), void output_sample(int32_t out_data),
void output_left_sample(int16_t out_data), and
void output_right_sample(int16_t out_data).
```

Functions `input_sample()` and `output_sample()` read and write both left and right channels and are suitable, for example, for playing (stereo) music through the eXperimenter. Functions `input_left_sample()`, `input_right_sample()`, `output_left_sample()`, and `output_right_sample()` are used in many of the examples in this book that use a laboratory signal generator as a single-channel input device and process only one channel.

The 32-bit integers (`int32_t`) returned by `input_sample()` and passed to `output_sample()` contain both left and right channel samples. The statement in file `L138_aic3106_init.h`

```
typedef union
{
    uint32_t uint;
    uint16_t channel[2];
} AIC31_data_type;
```

defines a type `AIC31_data_type` that may be used for 32-bit values read from or written to the AIC3106 codec.

The statement

```
AIC31_data_type codec_data;
```

declares a variable `codec_data` that may be handled either as one 32-bit unsigned integer (`codec_data.uint`) containing both left and right channel sample values or as two 16-bit unsigned integers (`codec_data.channel[LEFT]` and `codec_data.channel[RIGHT]`).

Many of the program examples in this book use only one channel for input and output and for brevity most use the functions `input_left_sample()` and `output_left_sample()`. Within functions `input_left_sample()` and `output_left_sample()`, the unpacking and packing of the signed 16-bit integer left-hand channel sample values out of and into the 32-bit words received and transmitted from and to the codec are carried out using data type `AIC31_data_type`. Data type `AIC31_data_type` may also be used elsewhere. For example, a number of programs use only one input channel, but write output to both left and right channels. Reading an input sample from the left channel and writing it to both output channels might be achieved using the following statements (assuming prior declarations of a variable `left_sample` of type `uint16_t` and a variable `codec_data` of type `AIC31_data_type`)

```
left_sample = input_left_sample();
codec_data.channel[LEFT] = left_sample;
codec_data.channel[RIGHT] = left_sample;
output_sample(codec_data.uint);
```

or alternatively

```
codec_data.uint = input_sample();
codec_data.channel[RIGHT] = codec_data.channel[LEFT];
output_sample(codec_data.uint);
```

Programmable ADC gain and DAC attenuation

Two of the AIC3106's many programmable parameters are an analog gain immediately preceding the ADC and a digital attenuation just before the DAC (Figures 2.2 and 2.3). The default value for each is 0 dB, but the gain preceding the ADC can be set anywhere between 0 and 59.5 dB in 0.5 dB steps, using page 0 control registers 15 and 16, and the attenuation preceding the DAC can be set anywhere between 0 and -63.5 dB in -0.5 dB steps, using page 0 control registers 43 and 44. Left and right channel gains and attenuations are independently programmable, but in function `L138_initialise_poll()` they are set equal. Most of the example programs in this book use 0 dB gain settings which are suitable for line level input signals from sound cards, mp3 players, signal generators, and so on. In order to use a dynamic microphone as an input device, the gain preceding the ADC should be increased from 0 to 24 dB.

Although polling is a simple method of I/O, it is inefficient because the processor can spend nearly all of its time repeatedly testing whether the codec is ready to transmit or receive data.

Running the program

The files making up the `L138_loop_poll` project are contained in the `c:\eXperimenter\L138_chapter2` workspace folder. Launch the Code Composer Studio IDL, selecting that workspace. A number of projects should appear in the *C/C++ Project View* window in the *C/C++* perspective, as shown in Figure 2.6.

Right-click on project name `L138_loop_poll` and *Set as Active Project*. Click on the *Debug Active Project* toolbar button or select *Target > Debug Active Project* (Figure 2.7). Once

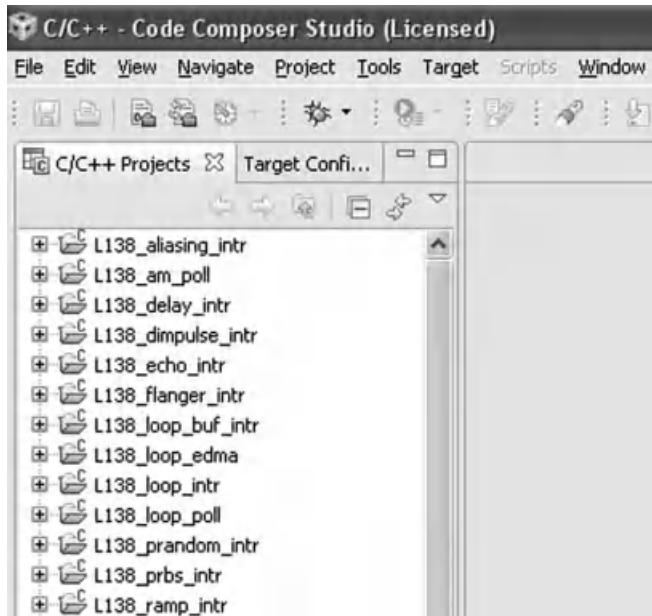


Figure 2.6 Project View window as it should appear, having selected workspace L138_chapter2 when launching the Code Composer Studio IDL.

the program has been loaded into C6748 memory and execution halted at `main()`, click on the *Run* button in the *Debug* window (in the *Debug* perspective). Connect a signal source, for example, an mp3 player, signal generator, or sound card output to the LINE IN socket on the eXperimenter board and headphones or loudspeakers to the LINE OUT socket to verify that the program operates as intended. Take care that the signal level applied to LINE IN does not exceed 0.707 V rms or else damage to the AIC3106 input circuits may result.

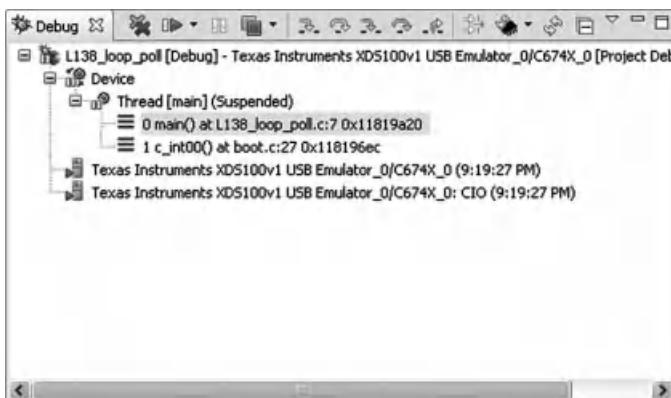


Figure 2.7 Debug perspective view of program L138_loop_poll.c prior to execution.

Changing the sampling frequency of the AIC3106 codec

Change the sampling frequency used by replacing parameter `FS_48000_HZ` with `FS_8000_HZ` on line 10 of file `L138_loop_poll.c`, that is,

```
L138_initialise_poll(FS_8000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
```

Rebuild the program by selecting *Project > Rebuild Active Project* and *Run* it again (if necessary, select *Target > Reset > System Reset* and then *Target > Restart* first). Music played through the system should now sound slightly less bright, corresponding to the lower sampling rate and bandwidth of the system.

Changing the gain of the AIC3106 codec

Change the output gain used in the program by replacing parameter `DAC_ATTEN_0DB` with `DAC_ATTEN_6DB` on line 10 of file `L138_loop_poll.c`, that is,

```
L138_initialise_poll(FS_48000_HZ, ADC_GAIN_0DB, DAC_ATTEN_6DB);
```

Rebuild the program by selecting *Project > Rebuild Active Project* and *Run* it again (if necessary, select *Target > Reset > System Reset* and then *Target > Restart* first). Music played through the system should now sound quieter, corresponding to the increased attenuation applied just before the DAC. A selection of DAC attenuation values are provided in the form of constants, for example, `DAC_ATTEN_9DB`, defined in support file `L138_aic3106_init.h`. These cover attenuation settings of 0 to 51 dB in increments of 3 dB. Finer resolution changes to DAC attenuation are possible and any integer value between 0 and 0x7F may be passed to function `L138_initialise_poll()` as a value of parameter `dac_atten`. Similarly, the programmable analog gain preceding the ADC is set according to the value of the parameter `adc_gain` passed to function `L138_initialise_poll()`. A selection of ADC gain settings are provided in the form of constants, for example, `ADC_GAIN_6DB`, defined in support file `L138_aic3106_init.h`. These correspond to gain settings of 0 to 51 dB in increments of 3 dB. Finer resolution changes to ADC gain are possible and any integer value between 0 and 0x7F may be passed to function `L138_initialise_poll()` as a value of parameter `adc_gain`. To use a dynamic microphone with the board, set the ADC gain to 24 dB.

EXAMPLE 2.2: Basic Input and Output Using Interrupts (`L138_loop_intr`)

Polling-based I/O is used in only a small number of the example programs in this book. Program `L138_loop_intr.c`, listed in Figure 2.8, is functionally equivalent to program `L138_loop_poll.c`, but uses interrupt-based I/O. `L138_loop_intr.c` uses the same interrupt-based technique for real-time I/O as program `L138_sine48_buf_intr.c`, described in Chapter 1.

This simple program is important because many of the example programs in this book are based on the same interrupt-driven model. Instead of simply copying the sequence of samples representing an input signal to the codec output, a digital filtering operation could be performed each time a new input sample is received, that is, a sample-by-sample processing algorithm could be inserted between the lines

```
// L138_loop_intr.c
//

#include "L138_aic3106_init.h"

interrupt void interrupt4(void) // interrupt service routine
{
    uint32_t sample;

    sample = input_sample(); // read L + R samples from ADC
    output_sample(sample); // write L + R samples to DAC
    return;
}

int main(void)
{
    L138_initialise_intr(FS_48000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);
}
```

Figure 2.8 Listing of loop program using interrupt-based I/O L138_loop_intr.c.

sample = input_sample();

and

output_sample(sample);

For this reason, it is worth taking time to ensure that you understand how program L138_loop_intr.c works.

In function `main()`, the initialization function `L138_initialise_intr()` is called. `L138_initialise_intr()` is very similar to `L138_initialise_poll()`, but in addition to initializing the eXperimenter board, codec, and McASP and not selecting polling mode (`poll = 0`), it sets up interrupts such that the AIC3106 codec will sample the analog input signal and the OMAP-L138 processor will be interrupted at the sampling frequency corresponding to the value of the parameter `fs`, for example, `FS_48000_HZ`. This is achieved by calling function `L138_init_mcasp_intr()` rather than function `L138_init_mcasp_poll()` from function `L138_initialise_intr()`. The only difference between these two routines is that in the former, McASP is configured to generate interrupts on transmit data ready.

In this example, a sampling rate of 48 kHz is used and therefore interrupts will occur every 20.833 μ s (sampling rates of 8, 9.6, 11.025, 16, 22.05, 24, 32, and 44.1 kHz are also possible). Constants `FS_8000_HZ`, `FS_48000_HZ`, and so on are defined in file `L138_aic3106_init.h`.

Following initialization, function `main()` enters an endless while loop effectively doing nothing but waiting for interrupts. The functions that act as interrupt service routines (ISRs) for different interrupts are specified in the interrupt service table (IST) contained in file `vectors_intr.asm`. This assembly language file differs from the file `vectors_poll.asm` (used in project `L138_loop_poll`) in that function `interrupt4()` is specified as the interrupt service routine for interrupt INT4.

When interrupt INT4 occurs, that interrupt service routine is called and it is within function `interrupt4()` that the most important program statements are executed. In this example,

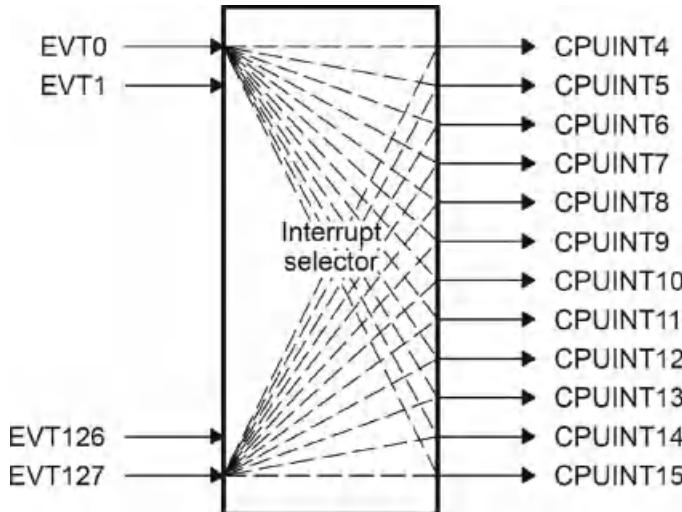


Figure 2.9 Block diagram of interrupt selector (IS) (courtesy of Texas Instruments).

function `output_sample()` is used to output a 32-bit value read from the codec using function `input_sample()`, that is, both left and right channel samples are passed directly from ADC to DAC.

Interrupts and events in the C6748

The C6748 CPU has four different types of interrupts: reset, nonmaskable, maskable, and exception. The reset interrupt has the highest priority of all and is used to halt the processor and return it to a known state. The nonmaskable interrupt has the second highest priority and in general is used to indicate hardware problems. Twelve maskable interrupts labeled INT4 to INT15 [4] have lower priorities. Within the C674x interrupt controller (INTC) [5], the interrupt selector (IS) routes any one of 128 different system events to any of the 12 maskable CPU interrupts. The routing implemented by the interrupt selector is determined by the contents of three Interrupt Mux Registers (INTMUX1 to INTMUX3). Figure 2.9 shows a block diagram of the interrupt selector.

System events are generated by various different parts of C6748 [6]. The example programs in this book make explicit use of the following two events only for interrupt- and DMA-based I/O, respectively:

- (a) Event #61, MCASP0_INT, MCASP0 combined RX/TX interrupt
- (b) Event #8, EDMA3_0_CC0_INT1, EDMA3 channel controller 0 shadow region 1 transfer completion interrupt

In either case, the event is routed to CPU interrupt INT4. This is achieved by writing the event number into the 7 LSBs of register INTMUX1. The statement

```
INTC_INTMUX1 = 0x3D; // write event no. #61 to INT4 field of INTMUX1
```

in function `L138_initialise_intr()` implements this.

In addition to a system event being routed to a CPU interrupt, interrupts must be generated and must be enabled.

In the case of event #61 (MCASP0_INT, MCASP0 combined RX/TX interrupt), interrupt generation is achieved by setting bit 5 (XDATA) in the McASP0 transmitter interrupt control register XINTCTL as part of the McASP0 configuration and initialization procedure, described in detail in Ref. [3]. The statement

```
MCASP->XINTCTL = 0x00000020; // interrupt on transmit
```

in function L138_init_mcasp_intr() does this.

Setting bit 4 (corresponding to INT4) in the CPU, interrupt enable register (IER) is implemented by the statement

```
IER |= 0x12; // enable INT4 and NMI
```

in function L138_initialise_intr(). For INT4 to interrupt the CPU, the Global Interrupt Enable (GIE) bit in the Control Status Register (CSR) must also be set and this is implemented by the statement

```
CSR |= 0x01; // enable interrupts globally
```

also in function L138_initialise_intr().

Interrupt generation for DMA-based I/O is described in detail in Chapter 5.

The interrupt service routines executed when interrupts occur are determined by the contents of the IST. IST comprises a 32-byte interrupt service fetch packet (ISFP) for each of the 16 CPU interrupts. An ISFP contains up to 14 instructions and in the case of the example programs in this book, the ISFP for INT4 causes a branch to function interrupt4(). The contents of the IST are set up in assembly language file vectors_intr.asm., listed in Figure 1.41. The same IST and therefore the same file vectors_intr.asm. is used for both interrupt- and DMA-based I/O example programs since in each case a different system event is routed to the same CPU interrupt.

The IST (excluding the reset ISFP which must always be located at address 0x00000000) may be located on any 256-word boundary in program memory pointed to by the interrupt service table pointer (ISTP). In function L138_initialise_intr(), the ISTP is configured by the statement

```
ISTP = (unsigned int)vectors;
```

where vectors is the symbol defined in vectors_intr.asm as the start of the IST. Alignment of vectors with a 256-word boundary in program memory is achieved by the statement

```
.align 1024
```

in program vectors_intr.asm.

At the start of the program, all maskable CPU interrupt flags (indicating that interrupts have occurred) are cleared. This is achieved by writing to the Interrupt Clear Register (ICR) in which bits 15 to 4 correspond to CPU interrupts INT15 to INT4. The statement

```
ICR = 0xFFFF0; // clear bits 15 to 4 in interrupt clear register
```

in function L138_initialise_intr() does this.

In the case of interrupt-based I/O, within the interrupt service routine `interrupt4()`, the event flag corresponding to event #61 is cleared using the statement

```
EVTCLR1 = 0x20000000; // clear event 61
```

however, since only one event is routed to CPU interrupt INT4 in this example program, it is not strictly necessary to do this [5].

Running the program

In the *C/C++* perspective, right-click on project `L138_loop_intr` in the *C/C++ Project View* window and *Set as Active Project*. Click on the *Debug Active Project* toolbar button and after the debugger has been launched and the program has been loaded, click on the *Run* button in the *Debug* window (*Debug* perspective). Use an mp3 player, signal generator, or PC sound card to provide an input to LINE IN and listen to the output on headphones or loudspeakers plugged into LINE OUT. Sampling frequency, ADC gain, and DAC attenuation may be changed in a way similar to that described in the previous example. In order to use a dynamic microphone as an input device, change the ADC gain specified in the call to function `L138_initialise_intr()` to 24 dB. The program may be halted and run again after selecting *Target > Reset > System Reset* and *Target > Restart*.

`EXAMPLE 2.3: Basic Input and Output Using DMA (L138_loop_edma)`

Program `L138_loop_edma.c`, listed in Figure 2.10, is functionally equivalent to programs `L138_loop_poll.c` and `L138_loop_intr.c`, but makes use of direct memory access. DMA-based I/O moves blocks or frames of data (samples) between codec and processor memory without CPU involvement, allowing the CPU to carry out other tasks at the same time.

Enhanced direct memory access (EDMA3) controller

In the OMAP-L138, DMA is implemented using the enhanced direct memory access (EDMA3) controller [7]. A typical use of the EDMA3 controller is to transfer data between memory-mapped, event-driven peripherals and memory. The multichannel audio serial port (McASP0) to which the codec is connected is an example of such a peripheral. The EDMA3 controller may be configured so that at the completion of a DMA transfer, an interrupt occurs. In addition, it may be configured so that the completion of one DMA transfer initiates another, using the EDMA3 linking mechanism.

The differences between using DMA-based I/O and using either polling- or interrupt-based I/O are evident in the `main()` function in program `L138_loop_edma.c`. Following initialization using function `L138_initialise_edma()`, an endless loop is entered. Within this endless loop, the variable `buffer_full` is tested and when its value indicates that a DMA transfer has been completed, the function `process_buffer()` is called. For successful real-time operation, function `process_buffer()` must complete execution before the next DMA transfer completes.

Function `L138_initialise_edma()` configures the EDMA3 controller such that the following occurs:

- (1) Two DMA transfers take place concurrently. One transfer moves data from an output buffer (array) to the DAC, while the other moves data from the ADC to an input buffer (array).

```

// L138_loop_edma.c
//
#include "L138_aic3106_init.h"

extern int16_t *pingIN, *pingOUT, *pongIN, *pongOUT;
volatile int buffer_full = 0;
int procBuffer;

interrupt void interrupt4(void) // interrupt service routine
{
    switch(EDMA_3CC_IPR)
    {
        case 1:                      // TCC = 0
            procBuffer = PING;        // process ping
            EDMA_3CC_ICR = 0x0001;    // clear EDMA3 IPR bit TCC
            break;
        case 2:                      // TCC = 1
            procBuffer = PONG;       // process pong
            EDMA_3CC_ICR = 0x0002;    // clear EDMA3 IPR bit TCC
            break;
        default:                     // may have missed an interrupt
            EDMA_3CC_ICR = 0x0003;    // clear EDMA3 IPR bits 0 and 1
            break;
    }
    EVTCLR0 = 0x000000100;
    buffer_full = 1;                // flag EDMA3 transfer
    return;
}
void process_buffer(void)
{
    int16_t *inBuf, *outBuf;        // pointers to process buffers
    int16_t left_sample, right_sample;
    int i;

    if (procBuffer == PING)         // use ping or pong buffers
    {
        inBuf = pingIN;
        outBuf = pingOUT;
    }
    if (procBuffer == PONG)
    {
        inBuf = pongIN;
        outBuf = pongOUT;
    }
    for (i = 0; i < (BUFCOUNT/2) ; i++)
    {
        left_sample = *inBuf++;
        right_sample = *inBuf++;

        *outBuf++ = left_sample;
        *outBuf++ = right_sample;
    }
    buffer_full = 0; // indicate that buffer has been processed
    return;
}
int main(void)
{
    L138_initialise_edma(FS_48000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1)
    {
        while (!buffer_full);
        process_buffer();
    }
}

```

Figure 2.10 Listing of loop program using DMA-based I/O L138_loop_edma.c.

- (2) The DMA transfers are synchronized with the sampling rate of the AIC3106 codec and the operation of McASP0.
- (3) Completion of one DMA transfer causes initiation of another. In the case of transfers from memory to the DAC, consecutive transfers alternate between using one output buffer (`pingOUT`) and another (`pongOUT`). In the case of transfers from ADC to memory, consecutive transfers alternate between using one input buffer (`pingIN`) and another (`pongIN`).
- (4) CPU interrupt INT4 occurs at the completion of each DMA transfer from ADC (via McASP0) to memory.

Interrupt service routine `interrupt4()`, the ISR for INT4, is executed after completion of each DMA transfer from ADC to memory. It carries out the following tasks:

- (1) Determine which pair of buffers (`pingIN` and `pingOUT` or `pongIN` and `pongOUT`) have most recently been used for I/O, setting the value of variable `proc-buffer` accordingly.
- (2) Clear the system event flag corresponding to event #8 by writing the value 0x00000100 to register `EVTCLR0`.
- (3) Set the value of variable `buffer_full` equal to 1 in order to indicate that a transfer has been completed.

More details of DMA-based I/O, including configuration of EDMA3 controller, are given in Chapter 5.

In function `main()`, once a DMA transfer has been completed and `buffer_full` has been set in function `interrupt4()`, function `process_buffer()` is called. A new block of input samples is waiting either in buffer `PINGin` or in buffer `PONGin` (as indicated by the value of `procbuffer`) and function `process_buffer()` must fill either buffer `PINGout` or buffer `PONGout` with new output samples. At the end of function `process_buffer()`, `buffer_full` is cleared. The size of the blocks of samples is set by the value of the constant `BUFCOUNT`, defined in file `L138_aic3106_init.h`. The number of 16-bit left channel samples per block is equal to `BUFCOUNT/2` and the number of 16-bit right channel samples per block is equal to `BUFCOUNT/2`.

Build and run program `L138_loop_edma.c` and verify its operation using an mp3 player, or a sound card and loudspeakers. When the program is halted, the DMA process may continue, but because the contents of input buffers `pingIN` and `pongIN` are no longer being transferred to output buffers `pingOUT` and `pongOUT` by function `process_buffer()`, the existing contents of the output buffers will be output repeatedly by the DAC.

The extra delay between input and output of `BUFCOUNT` sampling periods, introduced by DMA-based I/O, may be viewed using a signal generator and oscilloscope, connecting the signal generator output both to LINE IN on the eXperimenter and to channel 1 on the oscilloscope and LINE OUT on the eXperimenter to channel 2 on the oscilloscope. Use a low-frequency, for example, 100 Hz, sinusoid as a test signal so that the delay introduced will be less than one period of that signal and readily identified. Figure 2.11 shows the delays introduced if `BUFCOUNT` is equal to 128 or 32 and the sampling frequency is 48 kHz. The difference in the two delays shown is approximately equal to 96 sampling periods (2 ms). Verify that changing the value of `BUFCOUNT` in file `c:\eXperimenter\L138_support\L138_aic3106_init.h` changes the delay (note that the value of `BUFCOUNT` must be an even number). Setting the value of `BUFCOUNT` equal to 2 minimizes the delay (but rather defeats the object of using DMA).

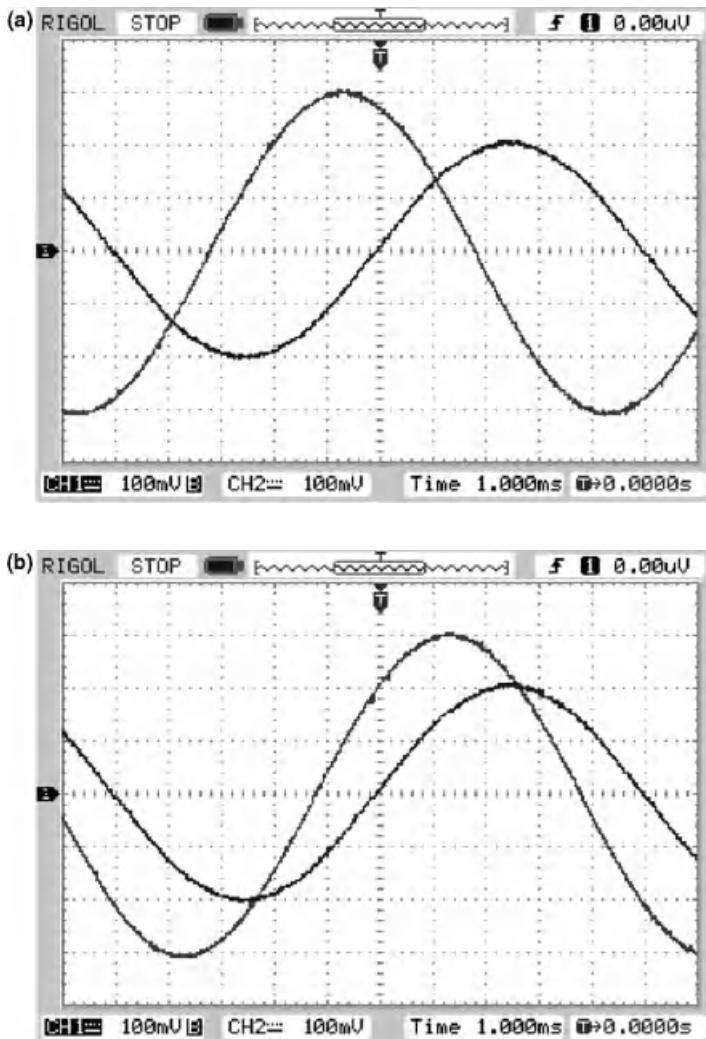


Figure 2.11 Input and output signals for program L138_loop_edma.c illustrating the delay introduced by DMA-based I/O. (a) BUFCOUNT = 128 (b) BUFCOUNT = 32.

EXAMPLE 2.4: Modifying Program L138_loop_intr.c to Create a Delay (L138_delay_intr)

Some simple, yet striking, effects can be achieved simply by delaying the samples as they pass from input to output. Program L138_delay_intr.c, listed in Figure 2.12, demonstrates this.

A delay line is implemented using the array `buffer` to store samples as they are read from the codec. Once the array is full, the program overwrites the oldest stored input sample with the current or newest input sample. Just prior to overwriting the oldest stored input sample in

```

// L138_delay_intr.c
//

#include "L138_aic3106_init.h"
#define BUF_SIZE 24000

uint16_t input,output,delayed;
uint16_t buffer[BUF_SIZE];
int i = 0;

interrupt void interrupt4(void) // interrupt service routine
{
    input = input_left_sample();
    delayed = buffer[i];
    output = delayed + input;
    buffer[i] = input;
    i = (i+1)%BUF_SIZE;
    output_left_sample(output);
    return;
}

int main(void)
{
    int i;

    for (i=0 ; i<BUF_SIZE ; i++)
    {
        buffer[i] = 0;
    }
    L138_initialise_intr(FS_48000_HZ,ADC_GAIN_24DB,DAC_ATTEN_0DB);
    while(1);
}

```

Figure 2.12 Listing of delay program L138_delay_intr.c.

buffer, that sample is retrieved, added to the current input sample and, written to the codec. Figure 2.13 shows a block diagram representation of the operation of program L138_delay_intr.c in which the block labeled T represents a delay of T seconds. The value of T is equal to the number of samples stored in buffer multiplied by the sampling period.

Run program L138_delay_intr.c, using a microphone and loudspeakers to verify its operation (the program is supplied with the ADC gain set for use with a dynamic microphone as an input device). A stereo version of the program L138_delay_intr_s.c is also supplied, with the ADC gain set to 0 dB for use with line level input.

To use the stereo version, right-click on the program name L138_delay_intr.c in the C/C++ perspective Project View window and select *Exclude File(s) from Build*. (excluding that source file from the project) and then right-click on the program name L138_delay_intr_s.c in the C/C++ perspective Project View window and select *Exclude File(s) from Build*. (including that source file in the project). The program names should appear in the Project View window as shown in Figure 2.14.

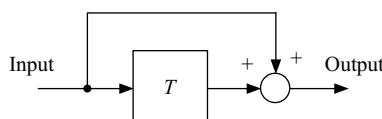


Figure 2.13 Block diagram representation of program L138_delay_intr.c.

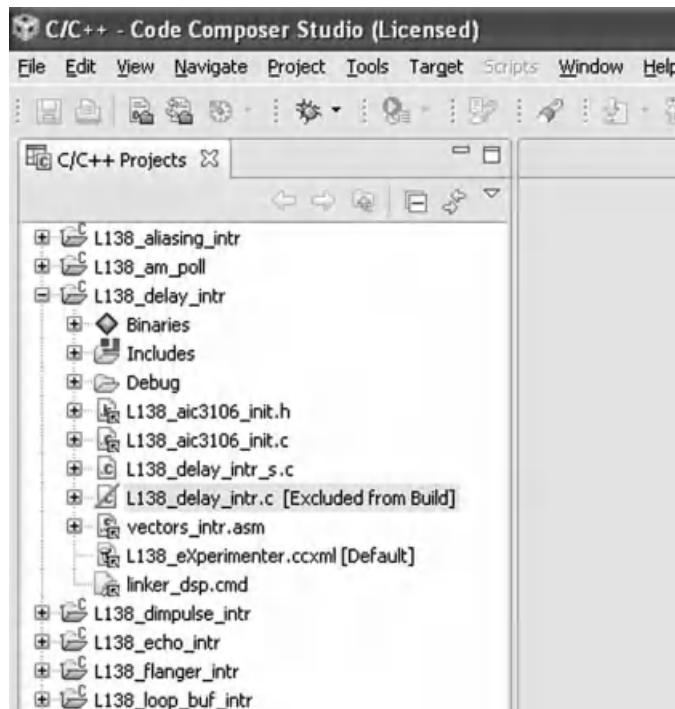


Figure 2.14 Project View window showing program L138_delay_intr_s.c included and program L138_delay_intr.c excluded from the project.

EXAMPLE 2.5: Modifying Program L138_loop_intr.c to Create an Echo (L138_echo_intr)

By feeding back a fraction of the output of the delay line to its input, a fading echo effect can be realized. Program L138_echo_intr.c, listed in Figure 2.15 and represented in block diagram form in Figure 2.16, does this.

The value of the constant BUF_SIZE in program L138_echo_intr.c determines the number of samples stored in the array buffer and hence the duration of the delay. The value of the constant GAIN determines the fraction of the output that is fed back into the delay line and hence the rate at which the echo effect fades away. Setting the value of GAIN equal to, or greater than, unity will cause instability of the loop.

Build and run this program. Experiment with different values of GAIN (between 0.0 and 1.0) and BUF_SIZE (between 100 and 8000). Source file L138_echo_intr.c must be edited and the project rebuilt in order to make these changes. The program is supplied with the ADC gain set for use with a dynamic microphone as an input device. A stereo version of the program L138_echo_intr_s.c, suitable for use with line level input, with ADC gain set to 0 dB is also supplied.

```

// L138_echo_intr.c
//

#include "L138_aic3106_init.h"
#define GAIN 0.4
#define BUF_SIZE 16000

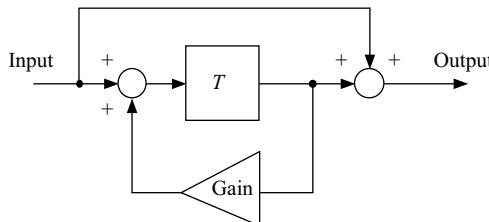
int16_t input,output,delayed;
int16_t buffer[BUF_SIZE];
int i = 0;

interrupt void interrupt4(void) // interrupt service routine
{
    input = input_left_sample();
    delayed = buffer[i];
    output = delayed + input;
    buffer[i] = input + delayed*GAIN;
    i = (i+1)%BUF_SIZE;
    output_left_sample(output);
    return;
}

int main(void)
{
    int i;

    for (i=0 ; i<BUF_SIZE ; i++)
    {
        buffer[i] = 0;
    }
    L138_initialise_intr(FS_48000_HZ,ADC_GAIN_24DB,DAC_ATTEN_0DB);
    while(1);
}

```

Figure 2.15 Listing of echo program L138_echo_intr.c.**Figure 2.16** Block diagram representation of program L138_echo_intr.c.

EXAMPLE 2.6: Modifying Program L138_loop_intr.c to Create a Flanging Effect (L138_flanger_intr)

Flanging is an audio effect used in recording studios (and live performances) that can add a whooshing sound like a jet aircraft passing overhead. It is a delay-based effect and can therefore be implemented as an extension of the previous two examples. The flanging effect is shown in block diagram form in Figure 2.17. The addition of a delayed and attenuated version of the input signal to itself creates a comb-like frequency response. If the frequency of the input signal is such that an integer multiple of its period is equal to the delay T , adding the delayed

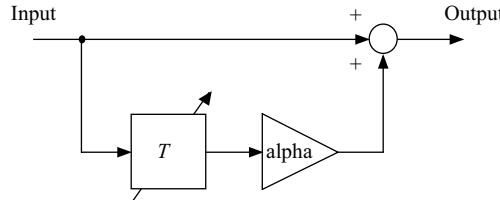


Figure 2.17 Block diagram representation of program L138_flanger_intr.c.

input signal will cancel out the input signal and this corresponds to a notch in the frequency response. The notches in the magnitude frequency response will therefore be spaced at regular intervals in frequency equal to $1/T\text{Hz}$ with the first notch located at frequency $1/2T\text{Hz}$. If the length of the delay is varied slowly, then the positions of the notches in the magnitude frequency response will vary accordingly to produce the flanging effect. As implemented in program L138_flanger_intr.c, the delay T varies sinusoidally between 200 and 1800 μs at a frequency of 0.1 Hz. Applied to music containing significant high-frequency content (e.g., drum sounds), the characteristic “jet aircraft passing overhead” type sound can be heard. Increasing the rate of change of delay to, say, 3 Hz gives an effect closer to that produced by a Leslie rotating speaker. Both mono and stereo versions of the program are provided.

If the delayed signal is subtracted from, rather than added to, the input signal, then the first notch in the magnitude frequency response will be located at 0 Hz, giving rise to a high-pass (very little bass response) effect overall. To subtract, rather than add, the delayed signal change lines 35 and 36 of program L138_flange_intr.c from

```
Lyn = Lxn+Lbuffer[out_ptr]*ALPHA;
Ryn = Rxn+Rbuffer[out_ptr]*ALPHA;
```

to

```
Lyn = Lxn-Lbuffer[out_ptr]*ALPHA;
Ryn = Rxn-Rbuffer[out_ptr]*ALPHA;
```

Figures 2.19 and 2.20 show experimentally measured examples of the instantaneous impulse and magnitude frequency responses of the flanger program for the two cases described above. Details of how these results were obtained are given later in this chapter.

The time-varying delay of the flanger may be illustrated using program L138_flange_dimpulse_intr.c and an oscilloscope. In this program, input samples are generated within the program rather than being read from the ADC. The sequence of input samples used is one nonzero value followed by 2047 zero values and this sequence is repeated periodically. The effect is to excite the flanger with a sequence of discrete-time impulses separated by 2048 sampling periods or 42.67 ms. The output from program L138_flange_dimpulse_intr.c comprises the input (which passes straight through) plus a delayed and attenuated version of the input. Figure 2.21 shows an example of the output from the program captured using a Rigol DS1052E oscilloscope. The pulse in the center of the screen corresponds to the input pulse and the pulse to the right corresponds to the delayed pulse. At the instant of capture shown in Figure 2.21, the delay is equal to approximately 400 μs . When the program is running, you should see the delay between the pulses varying slowly. The shape of the pulses in Figure 2.21 is explained in Example 2.15.

```

// L138_flanger_intr.c
//

#include "L138_aic3106_init.h"
#include "math.h"

#define TS 0.000020833333           // sampling rate 48 kHz
#define PERIOD 10                  // period of delay modulation
#define PI 3.14159
#define MEAN_DELAY 0.001            // mean delay in seconds
#define MODULATION_MAG 0.0008      // delay modulation magnitude
#define BUFSIZE 4096
#define ALPHA 0.9

AIC31_data_type codec_data;
uint16_t in_ptr = 0;                // pointer into buffers
uint16_t out_ptr;
float Lbuffer[BUFSIZE], Rbuffer[BUFSIZE];
float t = 0.0;
float Rxn, Ryn, Lxn, Lyn, delay_in_seconds;
uint16_t delay_in_samples;

interrupt void interrupt4(void) // interrupt service routine
{
    codec_data.uint = input_sample();
    Lxn = (float)codec_data.channel[LEFT];
    Rxn = (float)codec_data.channel[RIGHT];
    Lbuffer[in_ptr] = Lxn;
    Rbuffer[in_ptr] = Rxn;
    in_ptr = (in_ptr + 1) % BUFSIZE;
    t = t + TS;
    delay_in_seconds = MEAN_DELAY
                       + MODULATION_MAG * sin((2*PI/PERIOD)*t);
    delay_in_samples = (uint16_t)(delay_in_seconds * 48000.0);
    out_ptr = (in_ptr + BUFSIZE - delay_in_samples) % BUFSIZE;
    Lyn = Lxn + Lbuffer[out_ptr]*ALPHA;
    Ryn = Rxn + Rbuffer[out_ptr]*ALPHA;
    codec_data.channel[LEFT] = (uint16_t)Lyn;
    codec_data.channel[RIGHT] = (uint16_t)Ryn;;
    output_sample(codec_data.uint);
    return;
}

int main(void)
{
    L138_initialise_intr(FS_48000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
    while(1) ;
}

```

Figure 2.18 Listing of program L138_flanger_intr.c.

Figure 2.22 shows the output of program L138_flanger_intr.c modified so that the left channel outputs the sum of input and delayed signals and the right channel outputs the difference between input and delayed signals, displayed as spectrograms using *Goldwave*. The input to the flanger used to produce Figure 2.22 was pseudorandom noise. The dark bands in the spectrograms correspond to the notches in the magnitude frequency responses.

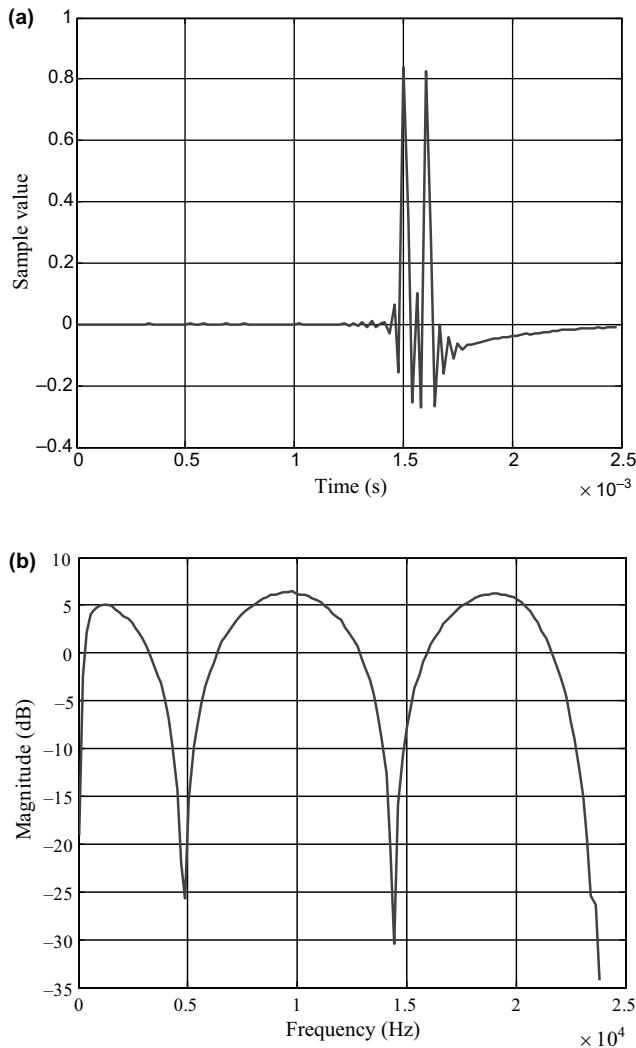


Figure 2.19 (a) Impulse response and (b) magnitude frequency response of flanger implemented using program L138_flanger_intr.c at an instant when delay T is equal to $104.2 \mu\text{s}$ and the delayed signal is added to input signal. The notches in the magnitude frequency response are at frequencies 4800 and 14400 Hz.

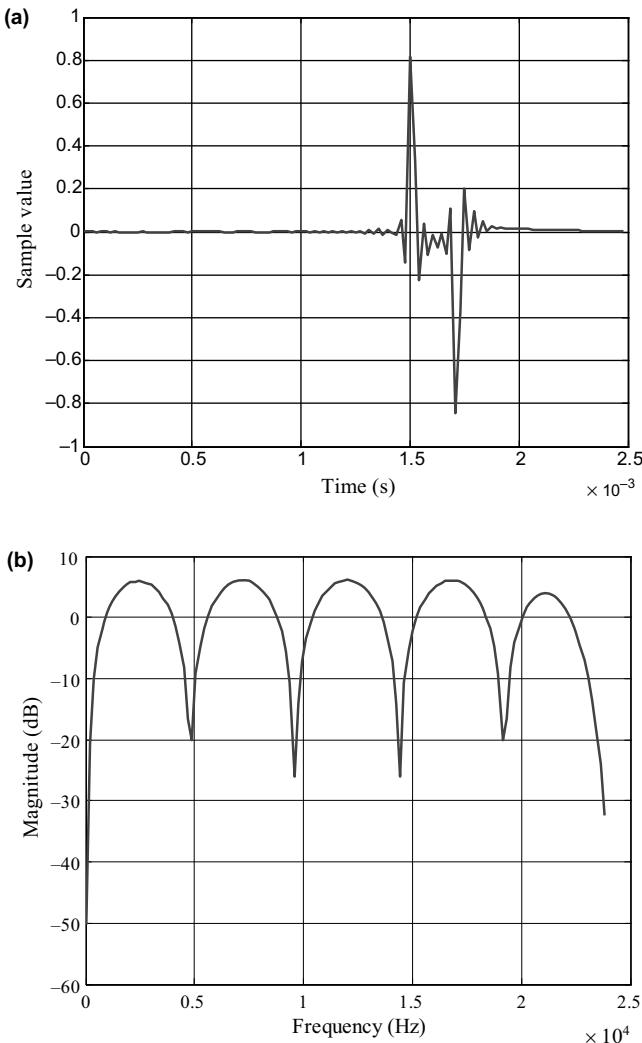


Figure 2.20 (a) Impulse response and (b) magnitude frequency response of flanger implemented using program L138_flanger_intr.c at an instant when delay T is equal to $208.3 \mu\text{s}$ and the delayed signal is subtracted from input signal. The notches in magnitude frequency response are at frequencies 0, 4800, 9600, 14,400, and 19,200 Hz.



Figure 2.21 Output waveform produced by program L138_flange_dimpulse_intr.c at an instant when delay T is approximately equal to $400\ \mu\text{s}$.

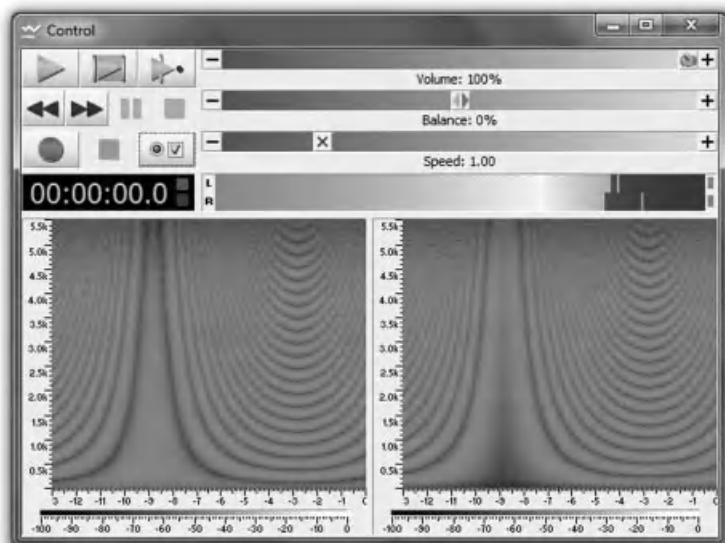


Figure 2.22 Spectrogram of flanger output for pseudorandom noise input. On the left channel, the delayed signal is added to the input signal. On the right channel, the delayed signal is subtracted from the input signal. The x -axis represents time in seconds and the y -axis represents frequency in hertz.

**EXAMPLE 2.7: Loop Program with Input Data Stored in a Buffer
(L138_loop_buf_intr)**

Program L138_loop_buf_intr.c listed in Figure 2.23 is similar to program L138_loop_intr.c, except that it maintains a circular buffer in array inbuffer containing the BUF_SIZE most recent input sample values. Consequently, it is possible to display this data after halting the program.

Build and run this program. Use a signal generator connected to the LINE IN socket to input a sinusoidal signal with a frequency between 100 and 3500 Hz. Halt the program after a short time and select *Tools > Graph > Single Time* in order to display the contents of array inbuffer. Figures 2.24 and 2.25 show the time domain representations of that data and the *Graph Properties* used. An input frequency of 550 Hz was used.

```
// L138_loop_buf_intr.c
//

#include "L138_aic3106_init.h"
#define BUFSIZE 512

int32_t inbuffer[BUFSIZE]; // int32_t for TI Data Format
int16_t buf_ptr = 0;

interrupt void interrupt4(void) // interrupt service routine
{
    int16_t sample_data;

    sample_data = input_left_sample();
    inbuffer[buf_ptr] = sample_data;
    buf_ptr = (buf_ptr+1)%BUFSIZE;
    output_left_sample(sample_data);

    return;
}

int main(void)
{
    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);
}
```

Figure 2.23 Listing of loop buffer program L138_loopbuf_intr.c.

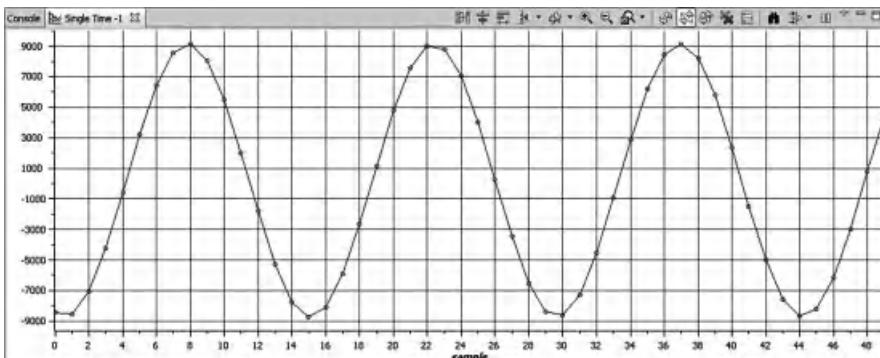


Figure 2.24 Contents of array inbuffer plotted following execution of program L138_loop_buf_intr.c.

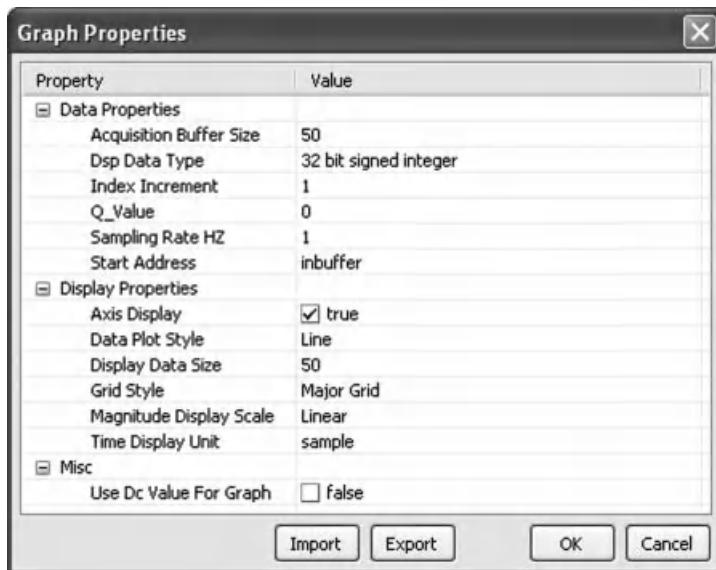


Figure 2.25 Graph Properties used to display contents of array inbuffer.

Alternatively, the data stored in array inbuffer can be exported from the Code Composer Studio IDE and plotted using MATLAB function L138_logfft().

Select View > Memory and enter inbuffer as the Address Text and 32 Bit Signed Integer as the Data Type. Then click on the Save button. Save 512 samples of type integer, starting at address inbuffer in a file of type TI Data Format (.dat) (Figure 2.26).

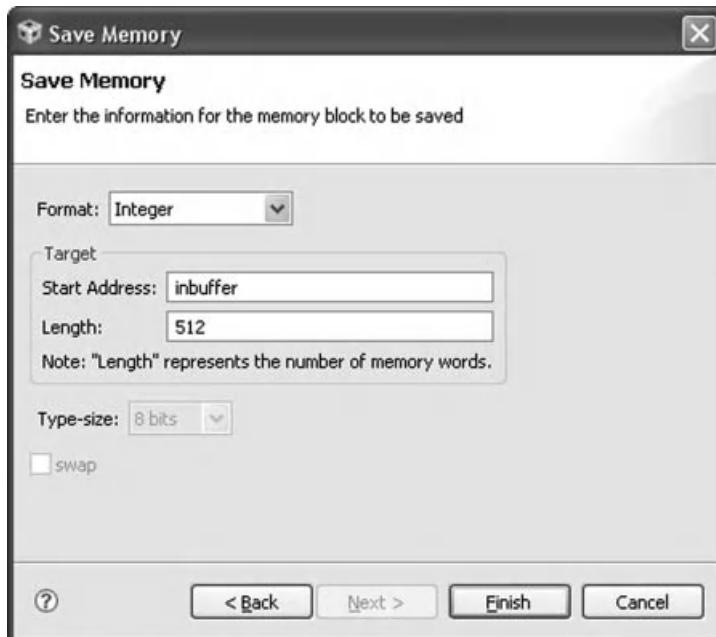


Figure 2.26 Save Memory window showing settings used with program L138_loop_buf_intr.c.

```
% L138_LOGFFT.M
%
% MATLAB function to read (sample) values saved from L138
% memory using CCSv4 and plot FFT magnitude on log scale
% CCSv4 data should be 32-bit floating point saved in
% TI Data Format .dat file
%

function L138_LOGFFT()
fname = input('enter filename ','s');
fs = input('enter sampling frequency in Hz ');
xscale = input('linear (0) or log (1) frequency scale? ');
fid = fopen(fname,'rt');
dummy = fscanf(fid,'%x',4)
N = fscanf(fid,'%x',1);
mydata=fscanf(fid,'%f',N);
fclose(fid);
g = abs(fft(mydata));
ff = 0:fs/N:(fs/2-fs/N);
figure(1)
plot(ff,20*log10(g(1:(N/2))),'LineWidth',2.0);
grid on
xlabel('frequency (Hz)', 'FontSize',12, 'FontName','times');
ylabel('magnitude (dB)', 'FontSize',12, 'FontName','times');
if xscale == 1
    set(gca,'FontSize',12,'XScale','log','FontName','times');
else
    set(gca,'FontSize',12,'XScale','lin','FontName','times');
end
tt = 0:1/fs:(N-1)/fs;
figure(2)
str = [num2str(N), ' sample values read from file'];
disp(str);
PTS = input('enter number of sample values to plot ');
plot(tt(1:PTS),mydata(1:PTS),'LineWidth',2.0);
grid on
xlabel('time (s)', 'FontSize',12, 'FontName','times');
ylabel('sample value', 'FontSize',12, 'FontName','times');
set(gca,'FontSize',12,'FontName','times');
```

Figure 2.27 MATLAB file L138_logfft.m

This data file may be read using MATLAB. The MATLAB function L138_logfft() (Figure 2.27) will prompt you for the name of the file you have saved and then plot the magnitude of the FFT of its contents. Figures 2.28 and 2.29 show the example data plotted using this method. Program L138_loop_buf_intr.c is used again later in this chapter in order to highlight the characteristics of the AIC3106 antialiasing filter.

2.3.2 Real-Time Sine Wave Generation

The following examples build on program L138_sine48_buf_intr.c, introduced in Chapter 1. By generating a variety of different analog output waveforms, including sinusoids of different frequencies, some important characteristics of the AIC3106 DAC may be demonstrated and the concepts of sampling, reconstruction, and aliasing illustrated. In addition, the use of the *Goldwave* shareware application is introduced. This virtual instrument is a useful alternative to an oscilloscope or a spectrum analyzer and is used again in later chapters.

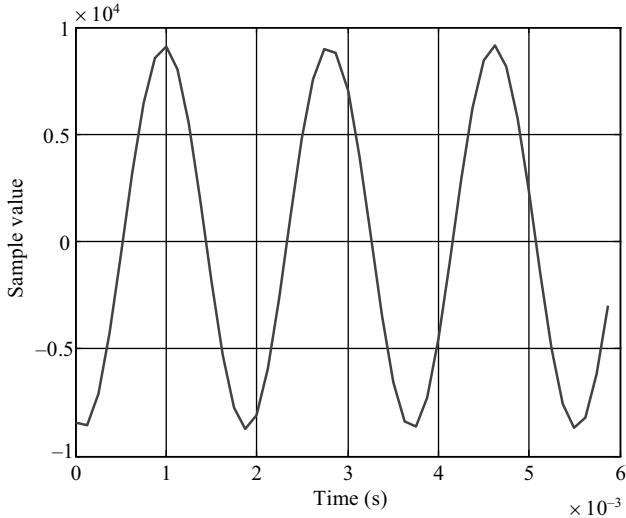


Figure 2.28 Contents of array `inbuffer` plotted using MATLAB function `L138_logfft()`.

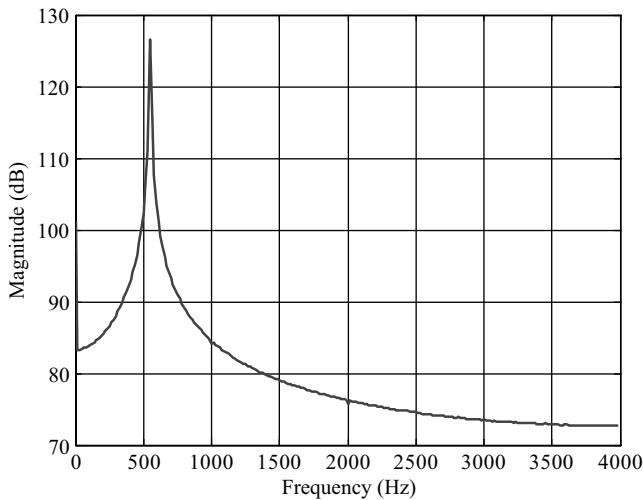


Figure 2.29 Magnitude of FFT of contents of array `inbuffer` plotted using MATLAB function `L138_logfft()`.

**EXAMPLE 2.8: Sine Wave Generation Using a Lookup Table
(`L138_sine48_intr`)**

Program `L138_sine48_intr.c` listed in Figure 2.30 generates a sinusoidal output signal using interrupt-based I/O and a lookup table method. Its operation is as follows. A 48 point lookup table is initialized in the array `sine_table` such that the value of `sine_table`

```

// L138_sine48_intr.c
//

#include "L138_aic3106_init.h"
#define LOOPLENGTH 48
int16_t sine_table[LOOPLENGTH] = {0, 1305, 2588, 3827,
    5000, 6088, 7071, 7934, 8660, 9239, 9659, 9914, 10000,
    9914, 9659, 9239, 8660, 7934, 7071, 6088, 5000, 3827,
    2588, 1305, 0, -1305, -2588, -3827, -5000, -6088, -7071,
    -7934, -8660, -9239, -9659, -9914, -10000, -9914, -9659,
    -9239, -8660, -7934, -7071, -6088, -5000, -3827, -2588,
    -1305};
int sine_ptr = 0; // pointer into lookup table

interrupt void interrupt4(void) // interrupt service routine
{
    uint16_t left_sample;

    left_sample = sine_table[sine_ptr];
    sine_ptr = (sine_ptr+1)%LOOPLENGTH;
    output_left_sample(left_sample);

    return;
}

int main(void)
{
    L138_initialise_intr(FS_48000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);
}

```

Figure 2.30 Listing of program L138_sine48_intr.c.

[i] is equal to

$$10,000 \sin(2\pi i/48), \quad \text{for } i = 0, 1, 2 \dots, 47. \quad (2.1)$$

In this example, a sampling rate of 48 kHz is used and therefore interrupts will occur every 20.833 μ s. Within ISR `interrupt4()`, the most important program statements are executed. Function `output_left_sample()` is used to output a value read from the array `sine_table` to the DAC and the index variable `sine_ptr` is incremented to point to the next value in the array. If the incremented value of `sine_ptr` is greater than, or equal to, the number of sample values in the table (`LOOPLENGTH`), it is reset to zero. The 1 kHz frequency of the sinusoidal output signal corresponds to the 48 samples per cycle output at a rate of 48 kHz. The DAC converts the output sample values into a sinusoidal analog output signal.

Build and run the program and verify a 1 kHz output tone using a loudspeaker. Program `L138_sine8_intr.c` is equivalent to program `L138_sine48_intr.c` in that it generates a 1 kHz sinusoid. However, it uses a sampling rate of 8 kHz and a lookup table containing just 8 sample values.

If you connect an oscilloscope to LINE OUT on the eXperimenter, you should see a waveform similar to that shown in Figure 2.31. There is a significant level of noise superimposed on the 1 kHz sine wave and this reveals something about the characteristics of the AIC3106 DAC. It is an oversampling sigma-delta DAC that deliberately moves noise out of the (audio) frequency band. The out-of-band noise present in the DAC output can be observed in the frequency domain using a spectrum analyzer or an FFT function on a digital oscilloscope.

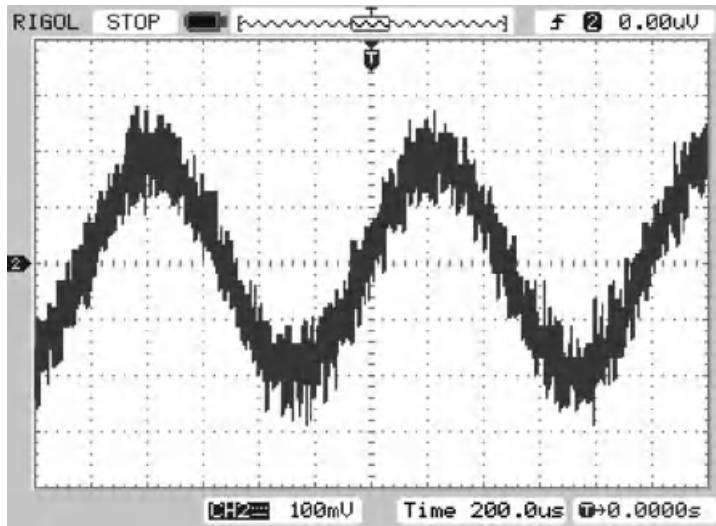


Figure 2.31 1 kHz sinusoid generated using program L138_sine48_intr.c viewed using *Rigol DS1052E* oscilloscope.

The plot shown in Figure 2.32 was obtained using a *Rigol DS1052E* digital oscilloscope to capture the output waveform and then the FFT of the data was plotted using MATLAB. It shows clearly the 1 kHz tone and the out-of-band noise above 100 kHz. The out-of-band noise is not a problem for listening to audio signals because headphones or loudspeakers will not usually have frequency responses that extend that high and neither does the human ear.

However, in order to see clearly the detail of audio signals using an oscilloscope, it is useful to add a first-order low-pass filter comprising a capacitor and a resistor to the LINE OUT signal path (Figure 2.33). Figures 2.34 and 2.35 show the filtered output signal from the program in both time and frequency domains for comparison with Figures 2.31 and 2.32.

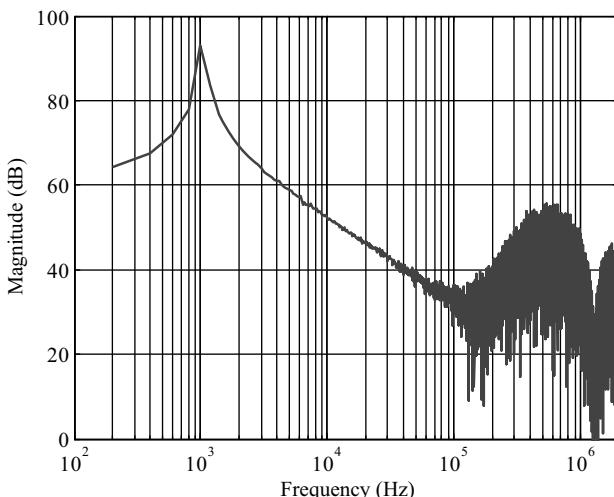


Figure 2.32 FFT of 1 kHz sinusoid generated by L138_sine48_intr.c plotted using MATLAB.

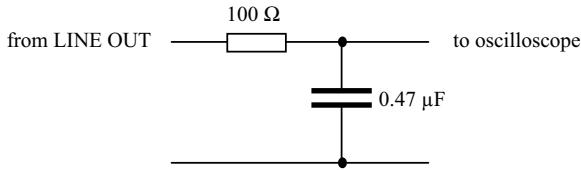


Figure 2.33 First-order CR filter used between LINE OUT on eXperimenter board and oscilloscope input.

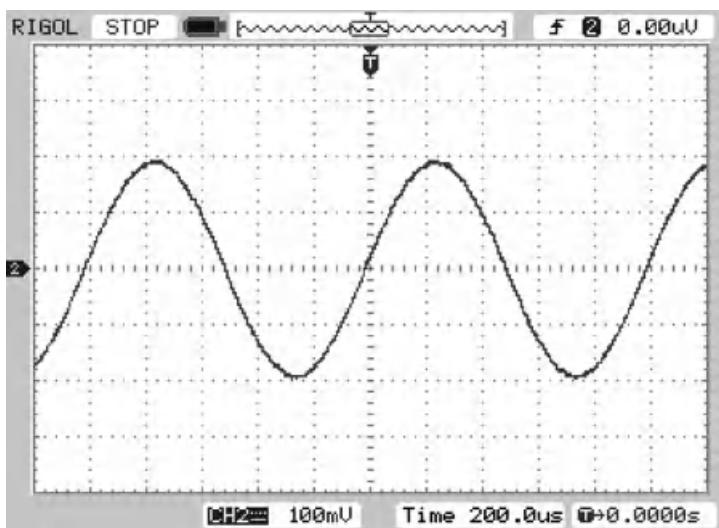


Figure 2.34 1 kHz sinusoid generated using program L138_sine48_intr.c and filtered using first-order CR filter, viewed using Rigol DS1052E oscilloscope.

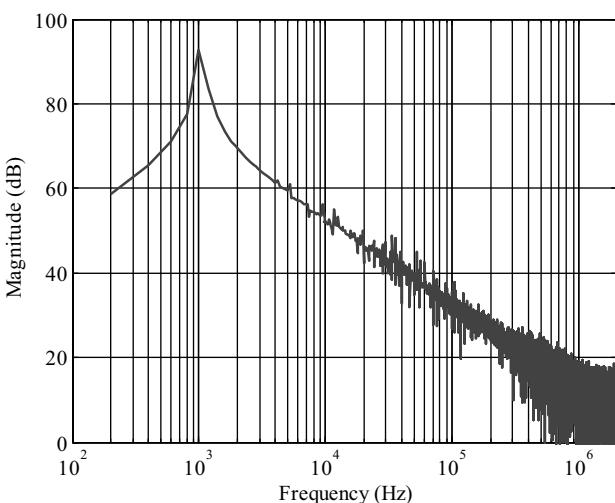


Figure 2.35 FFT of 1 kHz sinusoid generated by L138_sine48_intr.c and filtered using first-order CR filter, plotted using MATLAB.

**EXAMPLE 2.9: Sine Wave Generation Using `sin()` Function Call
(L138_sine_intr)**

Sine waves of different frequencies can be generated using the lookup table method employed by programs L138_sine8_intr.c and L138_sine48_intr.c. For example, a 3 kHz sine wave can be generated using program L138_sine8_intr.c by changing the line that reads

```
int16_t sine_table[N] = {0,7071,10000,7071,0,-7071,-10000,-7071};  
to read
```

```
int16_t sine_table[N] = {0,7071,-10000,7071,0,-7071,10000,-7071};
```

However, changing the contents and/or size of the lookup table is not a flexible way of generating sinusoids of arbitrary frequencies (Examples 2.10 and 2.11 demonstrate how different frequency sinusoids may be generated from a fixed lookup table). Program L138_sine_intr.c, listed in Figure 2.36, takes a different approach. At each sampling instant, that is, within function interrupt4(), a new output sample value is calculated using a call to the math library function sin(). The floating point parameter, theta, passed to that function is incremented at each sampling instant by the value theta_increment = 2*PI*frequency/SAMPLING_FREQ and when value of theta exceeds 2π , the value 2π is subtracted from it.

While program L138_sine_intr.c has the advantage of flexibility, it also has the disadvantage, relative to program L138_sine8_intr.c, that it requires far greater computational effort, which is important in real-time applications.

Build and run this program and experiment by changing the value assigned to the variable frequency within the range of 100 to 3800. Note that the sampling frequency used by this program is 8 kHz.

```
// L138_sine_intr.c  
  
#include "L138_aic3106_init.h"  
#include "math.h"  
  
#define SAMPLING_FREQ 8000  
#define PI 3.14159265358979  
  
float frequency = 1000.0;  
float amplitude = 20000.0;  
float theta_increment;  
float theta = 0.0;  
  
interrupt void interrupt4(void) // interrupt service routine  
{  
    theta_increment = 2*PI*frequency/SAMPLING_FREQ;  
    theta += theta_increment;  
    if (theta > 2*PI) theta -= 2*PI;  
    output_left_sample((int16_t)(amplitude*sin(theta)) );  
    return;  
}  
  
int main(void)  
{  
    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);  
    while(1);  
}
```

Figure 2.36 Listing of program L138_sine_intr.c.

EXAMPLE 2.10: Sine Generation with DIP Switches for Amplitude and Frequency Control (L138_sine_DIP_intr)

```
// L138_sine_DIP_intr.c
//

#include "L138_aic3106_init.h"
#include "evmomap1138_dip.h"

#define LOOPLENGTH 32

int16_t sine_ptr = 0;
int16_t sine_table[LOOPLENGTH]
    = {0,195,383,556,707,831,924,981,1000,
      981,924,831,707,556,383,195,0,-195,
      -383,-556,-707,-831,-924,-981,-1000,
      -981,-924,-831,-707,-556,-383,-195};

int16_t gain = 10;
int16_t frequency = 2;

interrupt void interrupt4(void) // interrupt service routine
{
    uint16_t left_sample;

    left_sample = (sine_table[sine_ptr]*gain);
    sine_ptr += frequency;
    sine_ptr = sine_ptr % LOOPLENGTH;
    output_left_sample(left_sample);

    return;
}

int main(void)
{
    uint32_t rtn;
    uint32_t DIP_value;

    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    rtn = DIP_init();
    if (rtn != ERR_NO_ERROR)
    {
        printf("error initializing dip switches!\r\n");
        return (rtn);
    }
    while(1)
    {
        rtn = DIP_getAll(&DIP_value);
        gain = (((DIP_value>>2)%4)+1)*8;
        frequency = ((DIP_value % 4) + 1)*2;
    }
}
```

Figure 2.37 Listing of program L138_sine_DIP_intr.c.

The interrupt-based program L138_sine_DIP_intr.c, listed in Figure 2.37, outputs a sine wave on the left channel of LINE OUT. DIP switches #1 through #4 are used to vary both the amplitude (gain) and the frequency of the sinusoid generated. Using a lookup table containing 32 samples of a single cycle of a sine wave, the frequency of the output waveform is varied by selecting different numbers of points per cycle. DIP switches #3 and #4 select one of four different amplitudes of the waveform.

The 32 sine data values in the array `sine_table` correspond to

$$1000 \sin(2\pi i/32), \quad \text{for } 0, 1, 2, \dots, 31. \quad (2.2)$$

User DIP switches #1 and #2 set the value of the variable `frequency` to 2, 4, 6, or 8. This value is added to the index into the lookup table, `sine_ptr`, at each sampling instant. The modulo operator is used to test when the end of the lookup table is reached. When the loop index exceeds `LOOPLENGTH`, it is reset to zero. For example, with DIP switches #1 and #2 both ON, the loop or frequency index steps through every other value in the table. This corresponds to 16 data values within a single cycle.

Verify that initially the frequency generated is 500 Hz (DIP switches #1 and #2 ON). Change the positions of DIP switches #1 and #2 and verify that the signal frequencies generated are 1000, 1500, and 2000 Hz. Changing the positions of DIP switches #3 and #4, you should be able to set the amplitude of the sine wave to approximately 150, 300, 450, or 600 mV.

EXAMPLE 2.11: Sweep Sinusoid Using Table with 8000 Points
(L138_sweep_poll)

Figure 2.38 shows a listing of the program L138_sweep_poll.c, which generates a swept frequency sinusoidal signal using a lookup table containing 8000 sample values. This program illustrates a method of waveform generation used in many laboratory signal generators. The header file `sine8000_table.h` generated using the MATLAB command

```
>> x = 1000*sin(2*pi*[0:7999]/8000);
```

contains 8000 sample values that represent a single cycle of a sine wave. Figure 2.39 shows a partial listing of the file `sine8000_table.h`.

At each sampling instant, program L138_sweep_poll.c reads an output sample value from the array `sine8000`, using the value of `float_index` converted to an integer, as an index, and increments the value of `float_index` by the value `float_incr`. With `N` points in the lookup table representing a single cycle of a sinusoid, the frequency of the output waveform is equal to `SAMPLING_FREQ*float_incr/N`.

A fixed value of `float_incr` would result in a fixed output frequency. In program L138_sweep_poll.c, the value of `float_incr` itself is incremented at each sampling instant by the value `DELTA_INCR` and hence the frequency of the output waveform increases gradually from `START_FREQ` to `STOP_FREQ`. The output waveform generated by the program can be altered by changing the values of the constants `START_FREQ`, `STOP_FREQ`, and `SWEEPTIME`, from which the value of `DELTA_INCR` is calculated.

Build and run this program. Verify the output to be a sweeping sinusoid taking `SWEEPTIME` seconds to increase in frequency from `START_FREQ` to `STOP_FREQ`. Note that the source program uses the polling I/O method (and uses the interrupt vector file `vectors_poll.asm`).

```
// L138_sweep_poll.c
//

#include "L138_aic3106_init.h"
#include "sine8000_table.h" //one cycle with 8000 points
#define SAMPLING_FREQ 8000.0
#define N 8000
#define START_FREQ 1000.0
#define STOP_FREQ 2000.0
#define START_INCR START_FREQ*N/SAMPLING_FREQ
#define STOP_INCR STOP_FREQ*N/SAMPLING_FREQ
#define SWEPTIME 2
#define DELTA_INCR (STOP_INCR - START_INCR) / (N*SWEPTIME)

int16_t amplitude = 10;
float float_index = 0.0;
float float_incr = START_INCR;
int i;

int main(void)
{
    L138_initialise_poll(FS_8000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
    while(1)
    {
        float_incr += DELTA_INCR;
        if (float_incr > STOP_INCR) float_incr = START_INCR;
        float_index += float_incr;
        if (float_index > N) float_index -= N;
        i = (int16_t)(float_index);
        output_left_sample(amplitude*sine8000[i]);
    }
}
```

Figure 2.38 Listing of program L138_sweep_poll.c.

```
// sine8000_table.h Sine table with 8000 points

short sine8000[8000] =
{0, 1, 2, 2, 3, 4, 5, 5,
6, 7, 8, 9, 9, 10, 11, 12,
13, 13, 14, 15, 16, 16, 17, 18,
19, 20, 20, 21, 22, 23, 24, 24,
25, 26, 27, 27, 28, 29, 30, 31,
31, 32, 33, 34, 35, 35, 36, 37,
38, 38, 39, 40, 41, 42, 42, 43,
44, 45, 46, 46, 47, 48, 49, 49,
50, 51, 52, 53, 53, 54, 55, 56,
57, 57, 58, 59, 60, 60, 61, 62,
63, 64, 64, 65, 66, 67, 67, 68,
69, 70, 71, 71, 72, 73, 74, 75,
.
.
.
-19, -18, -17, -16, -16, -15, -14, -13,
-13, -12, -11, -10, -9, -9, -8, -7,
-6, -5, -5, -4, -3, -2, -2, -1};
```

Figure 2.39 Fragment of file sine8000_table.h.

**EXAMPLE 2.12: Generation of DTMF Tones Using a Lookup Table
(L138_sineDTMF_intr)**

Program L138_sineDTMF_intr.c, listed in Figure 2.40, uses a lookup table containing 512 samples of a single cycle of a sinusoid together with two independent pointers to generate a dual tone multifrequency (DTMF) waveform. DTMF waveforms are used in telephone networks to indicate key presses. A DTMF waveform is the sum of two sinusoids of different frequencies. A total of 16 different combinations of frequencies, each comprising 1 of 4 low-frequency components (697, 770, 852, or 941 Hz) and 1 of 4 high-frequency

```
// L138_sineDTMF_intr.c
//

#include "L138_aic3106_init.h"
#include <math.h>
#define PI 3.14159265358979

#define TABLESIZE 512           // size of look up table
#define SAMPLING_FREQ 16000
#define STEP_770 (float)(770 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1336 (float)(1336 * TABLESIZE)/SAMPLING_FREQ
#define STEP_697 (float)(697 * TABLESIZE)/SAMPLING_FREQ
#define STEP_852 (float)(852 * TABLESIZE)/SAMPLING_FREQ
#define STEP_941 (float)(941 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1209 (float)(1209 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1477 (float)(1477 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1633 (float)(1633 * TABLESIZE)/SAMPLING_FREQ

int16_t sine_table[TABLESIZE];
float loopindexlow = 0.0;
float loopindexhigh = 0.0;
int16_t i;

interrupt void interrupt4(void) // interrupt service routine
{
    output_left_sample(sine_table[(int16_t)loopindexlow]
                        + sine_table[(int16_t)loopindexhigh]);
    loopindexlow += STEP_770;
    if (loopindexlow > (float)TABLESIZE)
        loopindexlow -= (float)TABLESIZE;
    loopindexhigh += STEP_1477;
    if (loopindexhigh > (float)TABLESIZE)
        loopindexhigh -= (float)TABLESIZE;

    return;
}

int main(void)
{
    L138_initialise_intr(FS_16000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
    for (i=0 ; i< TABLESIZE ; i++)
        sine_table[i] = (short)(10000.0*sin(2*PI*i/TABLESIZE));
    while(1);
}
```

Figure 2.40 Listing of program L138_sinedtmf_intr.c.

components (1209, 1336, 1477, or 1633 Hz), are used. Program L138_sineDTMF_intr.c uses two independent pointers into a single lookup table, each updated at the same rate (16 kHz) but each stepping through the values in the table at a different rate.

A pointer that stepped through every single one of the TABLESIZE samples stored in the lookup table at a sampling rate of 16 kHz would generate a sinusoidal tone with a frequency equal to $(16000 / \text{TABLESIZE})$. A pointer that stepped through the samples stored in the lookup table, incremented by a value STEP, would generate a sinusoidal tone with a frequency equal to $(16000 * \text{STEP} / \text{TABLESIZE})$.

From this it is possible to calculate the required step size for any desired frequency. For example, in order to generate a sinusoid with frequency 770 Hz, the required step size is $\text{STEP} = \text{TABLESIZE} * 770 / 16000 = 24.64$.

In other words, at each sampling instant, the pointer into the lookup table should be incremented by 24.64. The pointer value, or index, into the lookup table must be an integer value ((short)loopindexlow), but a floating-point value of the pointer, or index, loopindexlow is maintained by the program and incremented by STEP_770, wrapping around 0.0 when its value exceeds 512.0 using the statements

```
loopindexlow += STEP_770;
if(loopindexlow>(float)TABLESIZE) loopindexlow-=(float)TABLESIZE;
```

In program L138_sineDTMF_intr.c, the floating-point values by which the lookup table indices are incremented are predefined using, for example, line

```
#define STEP_770 (float) (770 * TABLESIZE) / SAMPLING_FREQ
```

In order to change the DTMF tone generated and simulate a different key press, edit the file L138_sineDTMF_intr.c and change the lines that read

```
loopindexlow += STEP_770;
loopindexhi += STEP_1477;
```

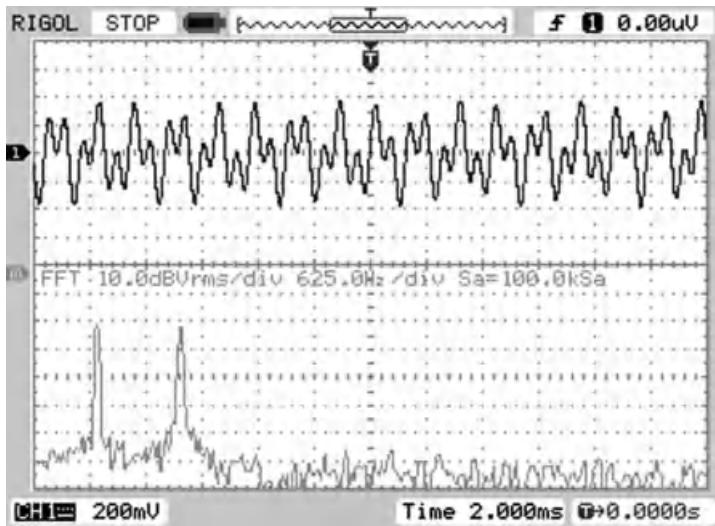


Figure 2.41 Output from program L138_sineDTMF_intr.c viewed using a *Rigol DS1052E* oscilloscope.

to, for example,

```
loopindexlow += STEP_697;
loopindexhi += STEP_1209;
```

An example of the output generated by program L138_sineDTMF_intr.c is shown in Figure 2.41.

EXAMPLE 2.13: Signal Reconstruction, Aliasing, and the Properties of AIC23 Codec (L138_sine_intr.c)

Generating analog output signals using, for example, program L138_sine_intr.c is a useful means of investigating the characteristics of the AIC3106 codec.

Change the value of the variable `frequency` in program L138_sine_intr.c to an arbitrary value between 100.0 and 3500.0 and you should find that a sine wave of that frequency (in Hz) is generated. Change the value of the variable `frequency` to 7000.0 however, and you will find that a 1 kHz sine wave is generated. The same is true if the value of `frequency` is changed to 9000.0 or 15000.0. These effects are due to the phenomenon of aliasing.

Since the reconstruction (digital-to-analog conversion) process is one of low-pass filtering, it follows that the bandwidth of signals output by the codec is limited. This can be demonstrated in a number of different ways.

For example, run program L138_sine_intr.c with the value of the variable `frequency` set to 3500.0 and verify that the output waveform generated has a frequency of 3500 Hz. Change the value of the variable `frequency` to 4500.0. The frequency of the output waveform should again be equal to 3500 Hz. Try any value for the variable `frequency`. You should find that it is impossible to generate an output waveform with a frequency greater than 4000 Hz (assuming a sampling frequency of 8 kHz). This is consistent with viewing the DAC as a low-pass filter.

EXAMPLE 2.14: Square Wave Generation Using a Lookup Table (L138_squarewave_intr)

Program L138_squarewave_intr.c, listed in Figure 2.42, differs from program L138_sine8_intr.c only in that it uses a lookup table containing 64 samples of a single cycle of a square wave of frequency 125 Hz rather than 8 samples of a single cycle of a sine wave.

Build and run the program and using an oscilloscope, you should see an output waveform similar to that shown in Figure 2.43. This waveform is equivalent to a square wave (represented by the samples in the lookup table) passed through a low-pass filter (the DAC). The symmetrical ringing at each edge of the square wave is indicative of the presence of a digital FIR filter, which is how the AIC3106 DAC implements the low-pass reconstruction filter. The drooping of the level of the waveform between transients seen in the oscilloscope trace of Figure 2.43 is due to the AC-coupling of the codec to the LINE OUT socket.

The low-pass characteristic of the reconstruction filter can further be highlighted by looking at the frequency content of the output waveform. Although the Fourier series representation of a square wave is the sum of an infinite series of harmonic components, only harmonic components with frequencies below 3.8 kHz are present in the analog output waveform, as shown in the lower trace of Figure 2.44. Note that the detail of the ringing at the edges of the square wave is visible because of the use of a low-pass CR filter between LINE OUT and the oscilloscope.

The following examples demonstrate a number of alternative approaches to observing the low-pass characteristic of the DAC reconstruction filter.

```
// L138_squarewave_intr.c
//

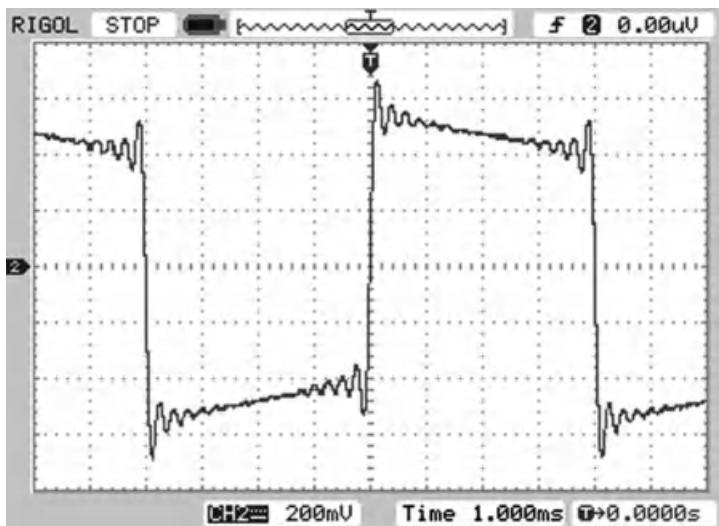
#include "L138_aic3106_init.h"
#define LOOPLENGTH 64

int16_t square_table[LOOPLENGTH] =
{10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
-10000,-10000,-10000,-10000,-10000,-10000,-10000,-10000,
-10000,-10000,-10000,-10000,-10000,-10000,-10000,-10000,
-10000,-10000,-10000,-10000,-10000,-10000,-10000,-10000,
-10000,-10000,-10000,-10000,-10000,-10000,-10000,-10000};

int16_t loopindex = 0;

interrupt void interrupt4(void) // interrupt service routine
{
    output_left_sample(square_table[loopindex++]);
    if (loopindex >= LOOPLENGTH)
        loopindex = 0;
    return;
}

int main(void)
{
    L138_initialise_intr(FS_8000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
    while(1);
}
```

Figure 2.42 Listing of program L138_squarewave_intr.c.**Figure 2.43** Output from program L138_squarewave_intr.c plotted using a Rigol DS1052E oscilloscope.

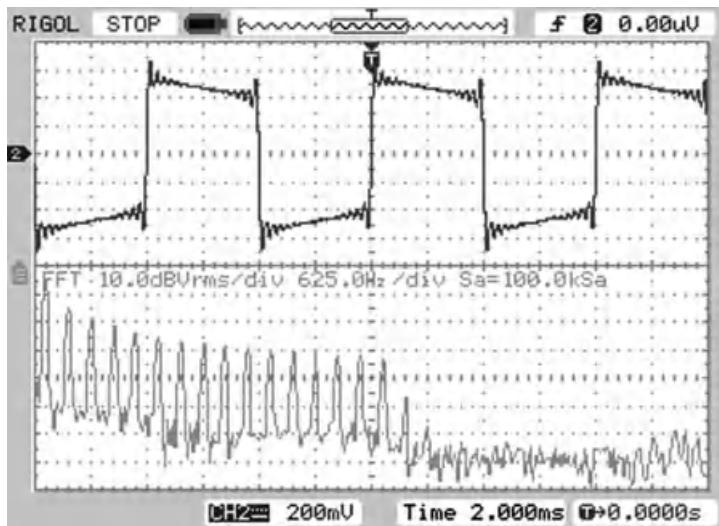


Figure 2.44 Magnitude of FFT of output from program L138_squarewave_intr.c plotted using a Rigol DS1052E oscilloscope.

EXAMPLE 2.15: Impulse Response of AIC3106 DAC Reconstruction Filter (L138_dimpulse_intr)

The frequency of the output waveform generated by program L138_squarewave_intr.c is sufficiently low that it may be considered as illustrating the step response of the

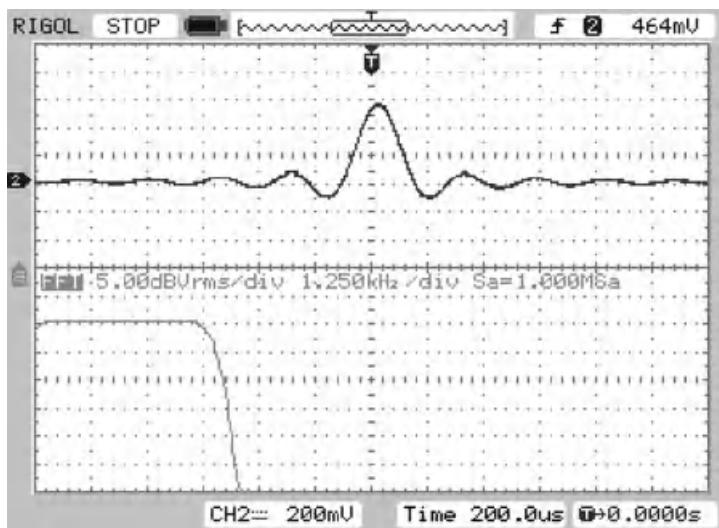


Figure 2.45 Output from program L138_dimpulse_intr.c plotted using a Rigol DS1052E oscilloscope. The horizontal scale of the lower trace is 1.25kHz per division and the vertical scale is 5 dB per division.

reconstruction filter in the DAC. The impulse response of the filter can be illustrated by running program L138_dimpulse_intr.c. This program replaces the samples of a square wave in the lookup table with a discrete-time impulse sequence. Figure 2.45 shows the output waveform generated by L138_dimpulse_intr.c and its magnitude FFT calculated using a *Rigol DS1052E* oscilloscope. The Fourier transform of the impulse response of a linear time invariant system is equal to its frequency response.

EXAMPLE 2.16: Frequency Response of DAC Reconstruction Filter Using a Pseudorandom Binary Sequence (L138_prbs_intr)

Program L138_prbs_intr.c, listed in Figure 2.46, generates a pseudorandom binary sequence and writes this to the DAC. Function prbs(), defined in file L138_aic3106_int.c, uses a software-based implementation of a maximal-length sequence technique for generating a pseudorandom binary sequence. An initial 16-bit seed is assigned to a register. Bits b0, b1, b11, and b13 are XORed and the result is placed into a feedback variable. The register with the initial seed value is then shifted 1 bit to the left. The feedback variable is then assigned to bit b0 of the register. A scaled minimum or maximum is assigned to prnseq, depending on whether the register's bit b0 is 0 or 1. This scaled value corresponds to the noise-level amplitude.

Build and run the program. Figure 2.47 shows the output waveform displayed using an oscilloscope and using *Goldwave*. The output spectrum is relatively flat until the cutoff frequency of approximately 4000 Hz, which represents the bandwidth of the reconstruction filter on the AIC3106 codec.

Programmable de-emphasis in the AIC3106 codec

The AIC3106 codec includes a programmable first-order de-emphasis filter that may optionally be switched into the signal path just before the DAC. Program L138_prbs_deemph_

```
// L138_prbs_intr.c
//

#include "L138_aic3106_init.h"

AIC31_data_type codec_data;

interrupt void interrupt4(void) // interrupt service routine
{
    codec_data.channel[0] = prbs();
    codec_data.channel[1] = codec_data.channel[0];
    output_sample(codec_data.uint);
    return;
}

int main(void)
{
    L138_initialise_intr(FS_8000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
    while(1);
}
```

Figure 2.46 Listing of program L138_prbs_intr.c.

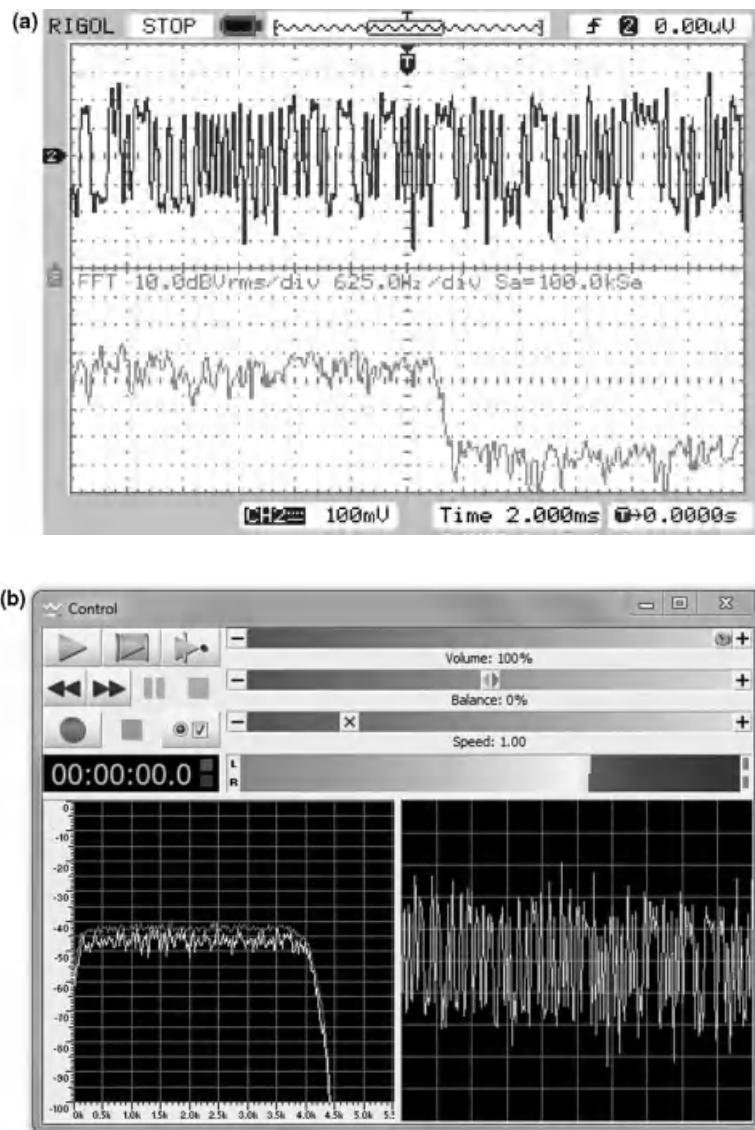


Figure 2.47 Output from program L138_prbs_intr.c plotted using a Rigol DS1052E oscilloscope and using Goldwave. The horizontal scale of the lower trace is 625 Hz per division and the vertical scale is 10 dB per division.

intr.c, located in project folder L138_prbs_intr, demonstrates its use (Figure 2.48). DIP switch #1 is used to enable or disable the de-emphasis and DIP switch #2 is used to choose between two different sets of filter coefficients written to page 1 control registers 21 through 26 (left channel) and 47 through 52 (right channel) in the AIC3106 [1]. One set of filter coefficients

```

// L138_prbs_deemph_intr.c
//

#include "L138_aic3106_init.h"
#include "evmomap1138_dip.h"
#include "evmomap1138_led.h"

AIC31_data_type codec_data;

interrupt void interrupt4(void) // interrupt service routine
{
    codec_data.channel[0] = prbs();
    codec_data.channel[1] = codec_data.channel[0];
    output_sample(codec_data.uint);
    return;
}

int main(void)
{
    uint32_t rtn;
    uint8_t newDIP_value;

    L138_initialise_intr(FS_8000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
    rtn = LED_init();
    if (rtn != ERR_NO_ERROR)
    {
        printf("error initializing leds!\r\n");
        return (rtn);
    }

    rtn = DIP_init();
    if (rtn != ERR_NO_ERROR)
    {
        printf("error initializing dip switches!\r\n");
        return (rtn);
    }

    while(1)
    {
        rtn = DIP_get(0, &newDIP_value);
        if(newDIP_value == 0)
        {
            LED_turnOn(0);
            AIC3106_writeRegister( 0, 0x00 );
            AIC3106_writeRegister( 12, 0x00 );
        }
        else
        {
            LED_turnOff(0);
            AIC3106_writeRegister( 0, 0x00 );
            AIC3106_writeRegister( 12, 0x05 );
        }
        rtn = DIP_get(1, &newDIP_value);
        if(newDIP_value == 0)
        {
            LED_turnOn(1); // default de-emph
            AIC3106_writeRegister( 0, 0x01 );
            AIC3106_writeRegister( 21, 0x39 );
            AIC3106_writeRegister( 22, 0x55 );
            AIC3106_writeRegister( 23, 0xF3 );
            AIC3106_writeRegister( 24, 0x2D );
        }
    }
}

```

Figure 2.48 Listing of program L138_prbs_deemph_intr.c.

```

        AIC3106_writeRegister( 25, 0x53 );
        AIC3106_writeRegister( 26, 0x7E );
        AIC3106_writeRegister( 47, 0x39 );
        AIC3106_writeRegister( 48, 0x55 );
        AIC3106_writeRegister( 49, 0xF3 );
        AIC3106_writeRegister( 50, 0x2D );
        AIC3106_writeRegister( 51, 0x53 );
        AIC3106_writeRegister( 52, 0x7E );
    }
}
else
{
    LED_turnOff(1); //HPF on L and R
    AIC3106_writeRegister( 0, 0x01 );
    AIC3106_writeRegister( 21, 0x5F );
    AIC3106_writeRegister( 22, 0xFF );
    AIC3106_writeRegister( 23, 0xa0 );
    AIC3106_writeRegister( 24, 0x00 );
    AIC3106_writeRegister( 25, 0x10 );
    AIC3106_writeRegister( 26, 0x00 );
    AIC3106_writeRegister( 47, 0x5F );
    AIC3106_writeRegister( 48, 0xFF );
    AIC3106_writeRegister( 49, 0xa0 );
    AIC3106_writeRegister( 50, 0x00 );
    AIC3106_writeRegister( 51, 0x10 );
    AIC3106_writeRegister( 52, 0x00 );
}
}

```

Figure 2.48 (Continued)

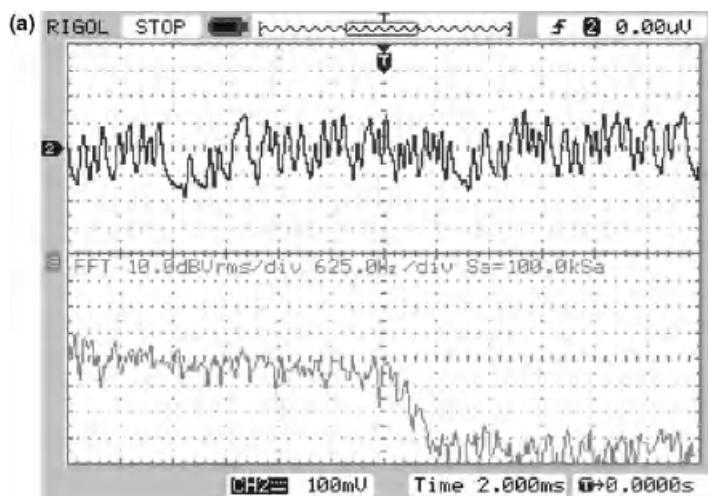


Figure 2.49 Output from program L138_prbs_deemph_intr.c with DIP switch #1 ON (de-emphasis enabled) and DIP switch #2 OFF (default de-emphasis characteristic selected). The default de-emphasis filter attenuates higher frequencies by approximately 6 dB.

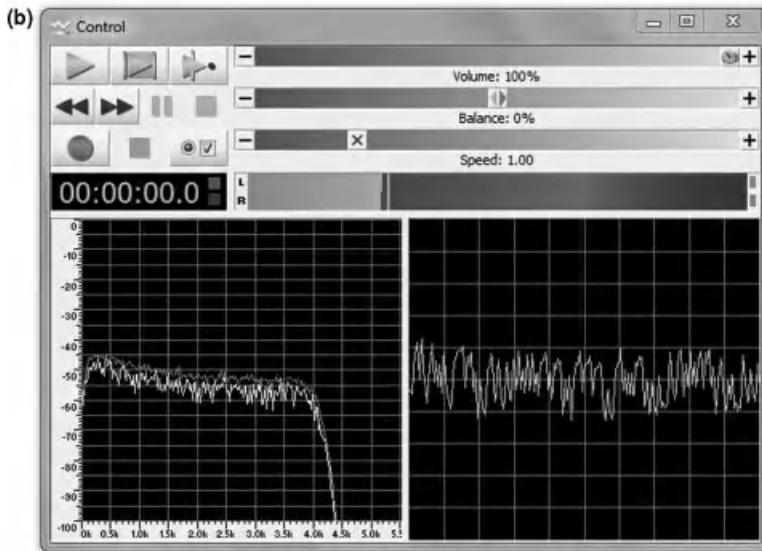


Figure 2.49 (Continued)

represents the default de-emphasis characteristic used, that is, the reset values for the page 1 control registers 21 through 26 and 47 through 52. The other set of filter coefficients are those of a high-pass filter. The de-emphasis filter can be used to implement any first-order IIR filter. The de-emphasis filters are enabled and disabled by writing to bits 0 (right channel) and 2 (left channel) in page 0 control register 12 (Audio Codec Digital Filter Control Register).

Examples of the output from program L138_prbs_deemph_intr.c are shown in Figures 2.49 and 2.50. The program may be run after excluding source file L138_prbs_intr.c

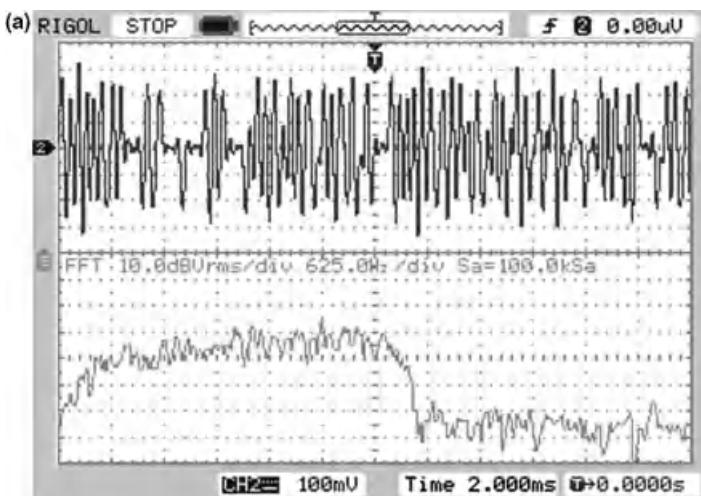
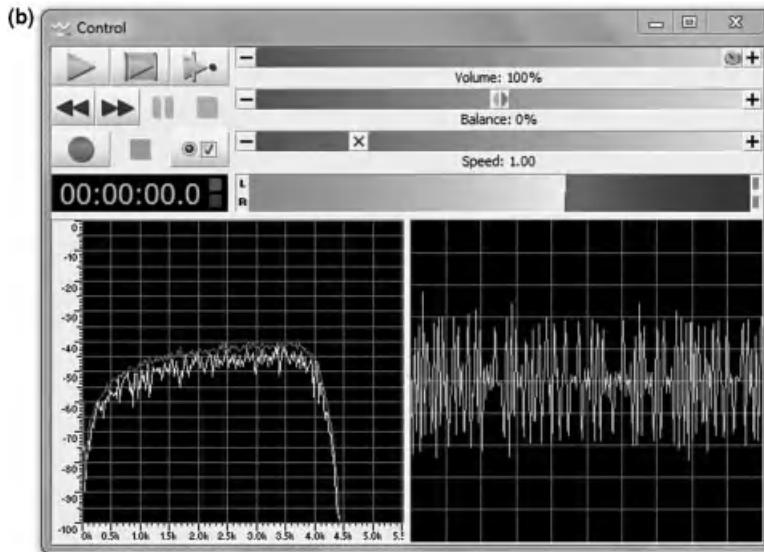


Figure 2.50 Output from program L138_prbs_deemph_intr.c with DIP switch #1 ON (de-emphasis enabled) and DIP switch #2 ON (high-pass characteristic selected).

**Figure 2.50 (Continued)**

from the project build and including source file L138_prbs_deemph_intr.c, as shown in Figure 2.51.

Programmable digital effect filters in the AIC3106 codec

The AIC3106 codec also includes two fourth-order IIR filters that may optionally be switched into the left and right channel signal paths just before the DAC. Program L138_prbs_biquad_intr.c, also stored in project folder L138_prbs_intr,

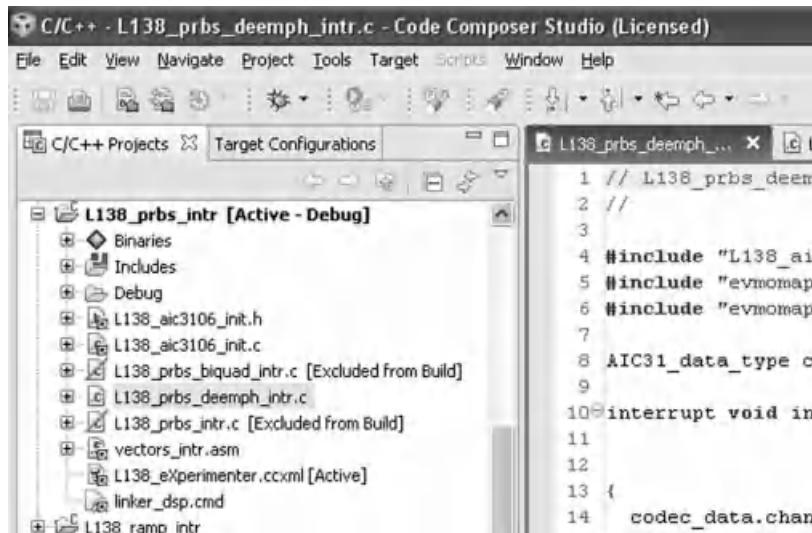


Figure 2.51 Project View window showing source file L138_prbs_deemph_intr.c included in build.

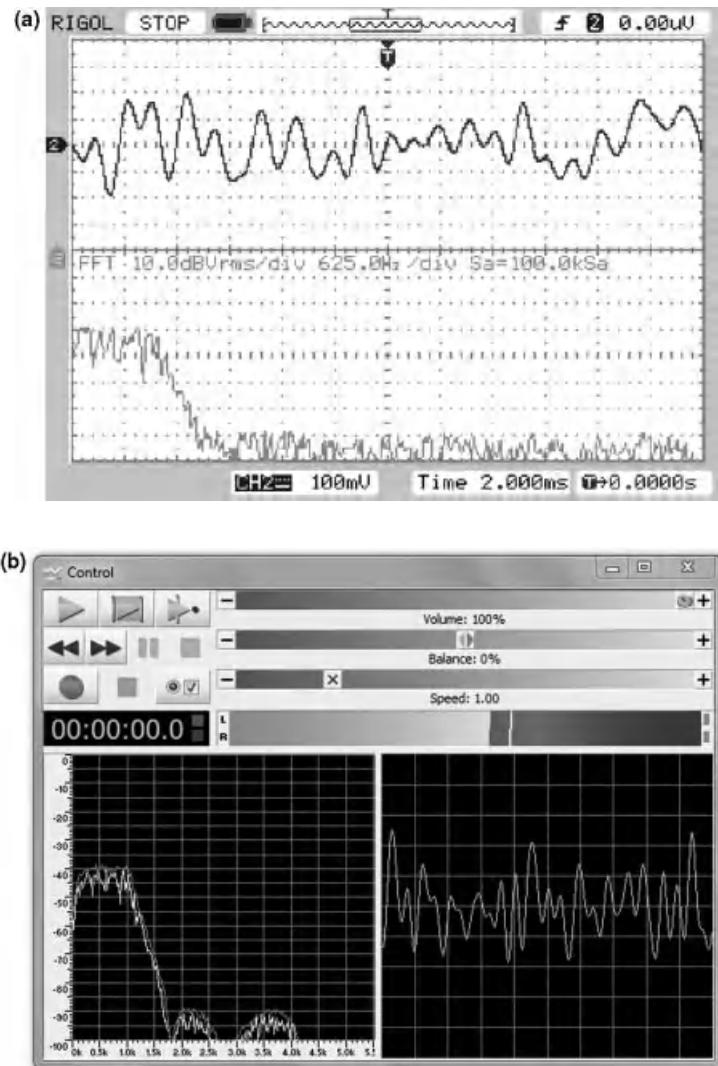


Figure 2.52 Output from program L138_prbs_biquad_intr.c viewed using an oscilloscope and using *Goldwave*.

demonstrates how the characteristics of these filters can be programmed by writing filter coefficients to AIC3106 page 1 control registers 1 through 20 (left channel) and 27 through 46 (right channel). In this example, a fourth-order elliptic low-pass filter is implemented. Figure 2.52 shows the filtered PRBS signal viewed using an oscilloscope and using *Goldwave*. The IIR filters are enabled and disabled by writing to bits 1 (right channel) and 3 (left channel) in page 0 control register 12 (Audio Codec Digital Filter Control Register). The characteristics of these filters and how to program them are described in greater detail in Chapter 5.

EXAMPLE 2.17: Frequency Response of DAC Reconstruction Filter Using Pseudorandom Noise (`L138_prandom_intr`)

Program `L138_prandom_intr.c` is similar to program `L138_prandom_intr.c`, except that it uses a Parks–Miller algorithm to generate a pseudorandom noise sequence. This may be of use as an alternative to PRBS in some applications. Function `prandom()` is defined in file `L138_aic3106_init.c`.

Build and run this program. Figure 2.53 shows the output waveform displayed using an oscilloscope.

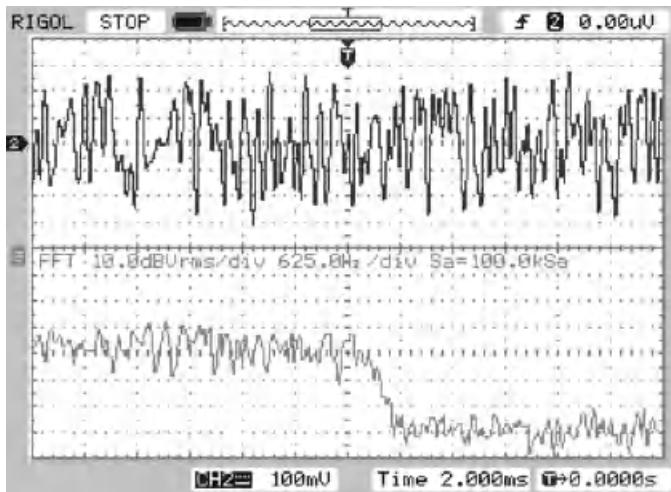


Figure 2.53 Output signal generated by program `L138_prandom_intr.c`.

Aliasing

Examples 2.12 through–2.17 demonstrate that the AIC3106 codec cannot generate signal components having frequencies greater than half the sampling frequency. It follows that it is inadvisable to allow analog input signal components having frequencies greater than half the sampling frequency to be sampled at the input to the DSP system. This can be prevented by passing analog input signals through a low-pass antialiasing filter. A digital antialiasing filter with characteristics similar to those of the reconstruction filter in the digital-to-analog converter is incorporated into the oversampling delta–sigma analog-to-digital converter in the AIC3106 codec. In addition, a second-order analog low-pass filter with 20 dB of attenuation at 1 MHz is built into the device.

EXAMPLE 2.18: Step Response of AIC3106 Codec Antialiasing Filter (`L138_loopbuf_intr`)

In order to investigate the step response of the antialiasing filter on the AIC3106, connect a signal generator to the LINE IN socket. Adjust the signal generator to give a square wave output of frequency 270 Hz and amplitude 0.2 V. Build and run program `L138_loopbuf_intr.c` on the eXperimenter, halting the program after a few seconds. View the most recent 128 input

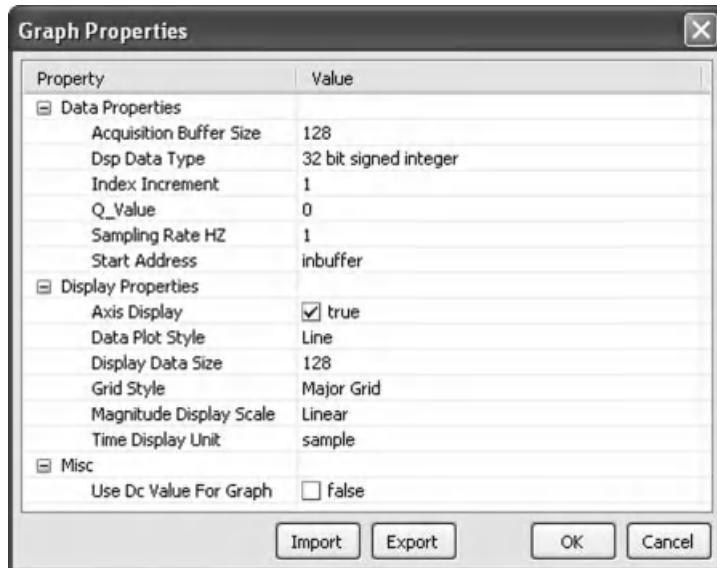


Figure 2.54 *Graph Properties* for use with program `L138_loop_buf_intr.c`.

sample values by selecting *Tools > Graph > Single Time* and setting the *Graph Properties*, as shown in Figure 2.54. You should see something similar to that shown in Figure 2.55b. Figure 2.55a shows the square wave input signal that produced the display of Figure 2.55b. The ringing on either side of edges of the square wave and the drooping of the level between transients are due to the antialiasing filter and the AC-coupling of the codec to the LINE IN socket on the eXperimenter.

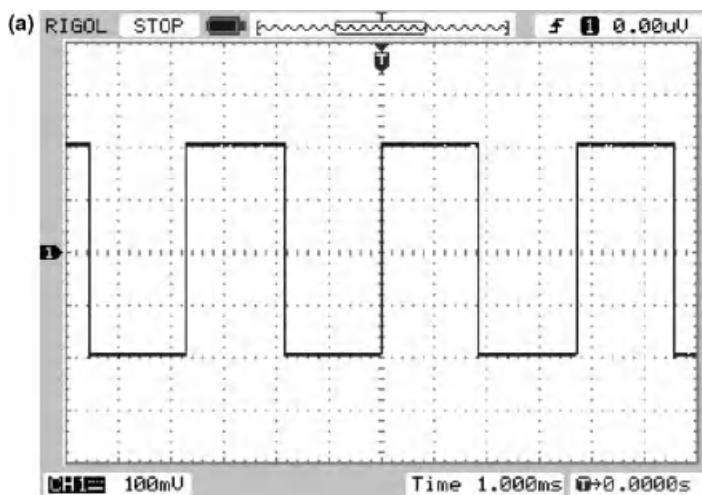
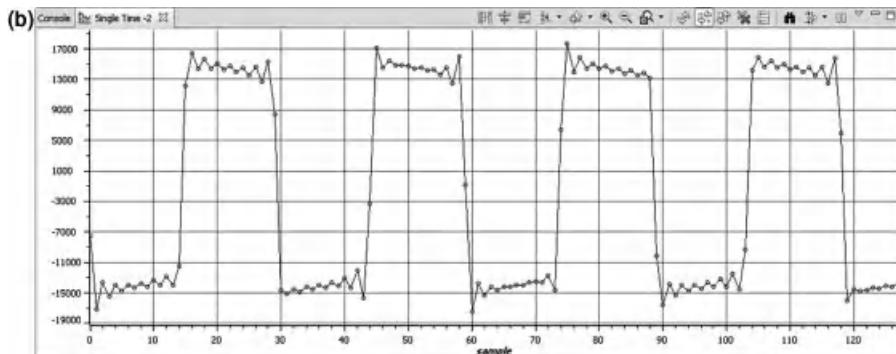


Figure 2.55 (a) Square wave input to program `L138_loop_buf_intr.c`. (b) Input samples stored in array `inbuffer` by program `L138_loop_buf_intr.c`.

**Figure 2.55** (Continued)

```
// L138_sine48_loop_intr.c
//

#include "L138_aic3106_init.h"

#define BUFSIZE 128
#define LOOPLENGTH 48

AIC31_data_type codec_data;

int16_t inbuffer[BUFSIZE];
int16_t sine_table[LOOPLENGTH] = {0, 1305, 2588, 3827,
    5000, 6088, 7071, 7934, 8660, 9239, 9659, 9914, 10000,
    9914, 9659, 9239, 8660, 7934, 7071, 6088, 5000, 3827,
    2588, 1305, 0, -1305, -2588, -3827, -5000, -6088, -7071,
    -7934, -8660, -9239, -9659, -9914, -10000, -9914, -9659,
    -9239, -8660, -7934, -7071, -6088, -5000, -3827, -2588,
    -1305};
int16_t inbuf_ptr = 0;
int sine_ptr = 0;

interrupt void interrupt4(void) // interrupt service routine
{
    uint16_t left_sample;

    inbuffer[inbuf_ptr] = input_left_sample();
    inbuf_ptr = (inbuf_ptr+1) % BUFSIZE;
    left_sample = sine_table[sine_ptr];
    sine_ptr = (sine_ptr+1)%LOOPLENGTH;
    codec_data.channel[RIGHT]=left_sample;
    codec_data.channel[LEFT]=left_sample;
    output_sample(codec_data.uint);
    return;
}

int main(void)
{
    L138_initialise_intr(FS_48000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);
}
```

Figure 2.56 Listing of program L138_sine48_loop_intr.c.

**EXAMPLE 2.19: Demonstration of AIC3106 Codec Antialiasing Filter
(L138_sine48_loop_intr)**

Program L138_sine48_loop_intr.c, listed in Figure 2.56, is similar to program L138_sin48_intr.c in that it generates a 1 kHz sine wave output using samples read from a lookup table. It differs in that it reads input samples from the ADC and stores the 128 most recent ones in array inbuffer. Connect LINE OUT on the eXperimenter to LINE IN using a 3.5 mm jack to 3.5 mm jack cable and run the program for a short length of time. Plot the contents of inbuffer using the *Graph Properties* shown in Figure 2.57 and you should see something similar to the plot shown in Figure 2.58. The out-of-band noise that could be seen using an oscilloscope, in Example 2.8, has been filtered out by the antialiasing filters in the AIC3106 with the result that the samples stored in inbuffer form a smooth sine wave.

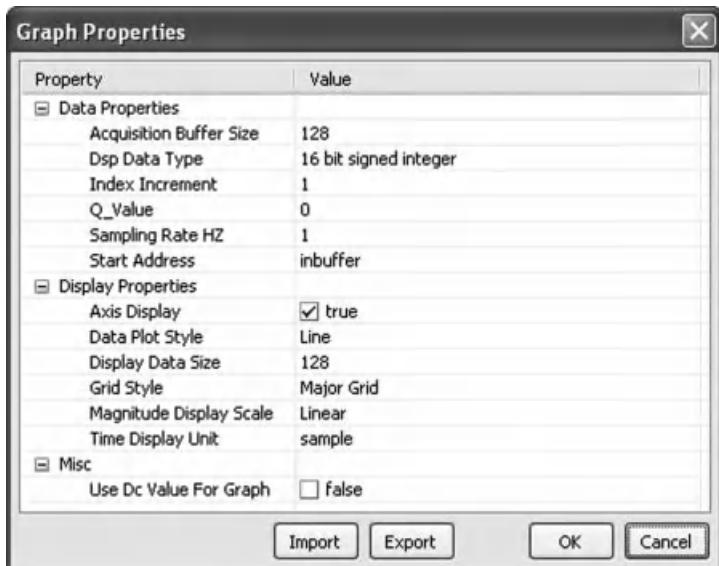


Figure 2.57 *Graph Properties* for use with program L138_sine48_loop_intr.c.

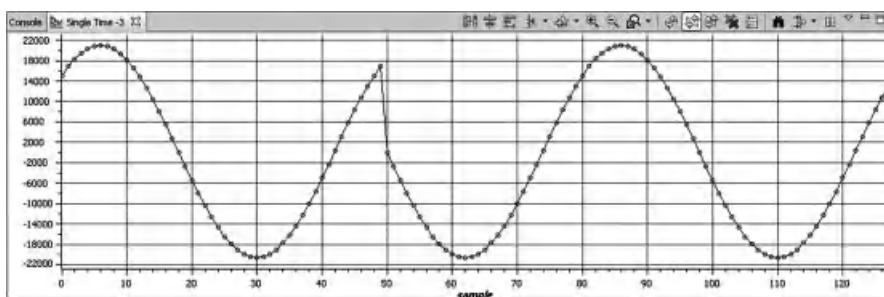


Figure 2.58 Input samples stored in array inbuffer by program L138_sine_48_loop_intr.c.

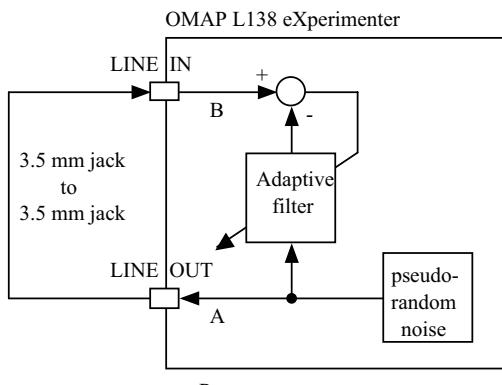
EXAMPLE 2.20: Demonstration of Aliasing (L138_aliasing_intr)

The analog and digital antialiasing filters in the AIC3106 codec cannot be bypassed or disabled. However, aliasing can be demonstrated by downsampling a digital signal within a program without appropriate antialiasing measures. Program L138_aliasing_intr.c uses a sampling rate of 16 kHz for the codec, but then resamples the sequence of samples produced by the ADC at the lower rate of 8 kHz (downsampling). The sequence of samples generated at a rate of 16 kHz by the ADC may contain frequency components at frequencies greater than 4 kHz and therefore if that sample sequence is downsampled to a rate of 8 kHz simply by discarding every second sample, aliasing may occur. To avoid aliasing, the 16 kHz sample sequence output by the ADC must be passed through a digital antialiasing filter before downsampling. Program L138_aliasing_intr.c uses an FIR filter (see Chapter 4) for this task. For the purposes of this example, it is sufficient to consider that the program demonstrates the effect of sampling at a frequency of 8 kHz with and without using an antialiasing filter.

Build, load, and run program L138_aliasing_intr.c. Connect a signal generator to LINE IN and an oscilloscope, *Goldwave*, or headphones to LINE OUT and vary the frequency of a sinusoidal input signal between 0 and 8 kHz. With DIP switch #1 OFF, the antialiasing filter is enabled and signals with frequencies greater than 4.0 kHz do not pass from LINE IN to LINE OUT. Switching DIP switch #1 ON disables the antialiasing filter in program L138_aliasing_intr.c and by varying the frequency of the input signal, you should be able to verify that signals with frequencies between 4 and 8 kHz are “folded back” into the frequency range 0–4 kHz.

EXAMPLE 2.21: Identification of AIC3106 Codec Bandwidth Using an Adaptive Filter (L138_sysid_intr)

Another way of observing the limited bandwidth of the codec is to measure its magnitude frequency response using program L138_sysid_intr.c. This program uses an adaptive FIR filter and is described in more detail in Chapter 6. However, you need not understand exactly how program L138_sysid_intr.c works in order to use it. Effectively, it identifies the characteristics of the path between its discrete-time output and its discrete-time input (points A and B in Figure 2.59).



Program L138_sysid_intr.c

Figure 2.59 Connection diagram for program L138_sysid_intr.c.

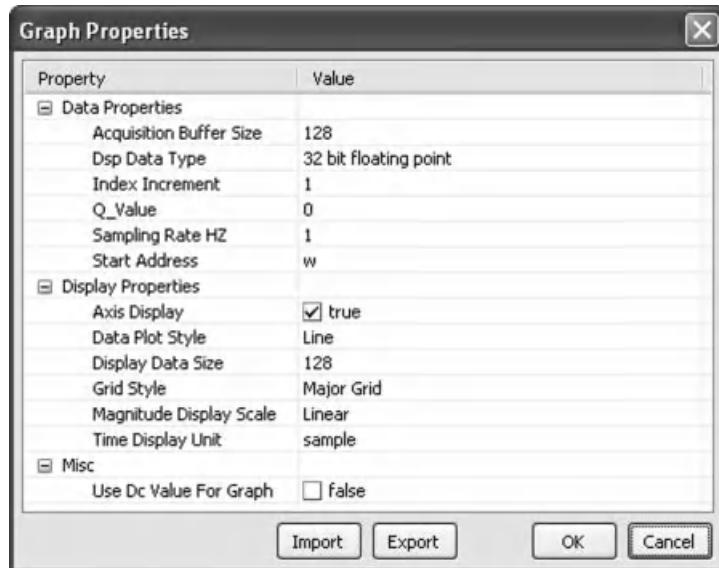


Figure 2.60 *Graph Properties* for viewing the impulse response identified using program L138_sysid_intr.c.

Connect the LINE OUT output on the eXperimenter to its LINE IN input using a 3.5 mm jack to 3.5 mm jack cable, as shown in Figure 2.59. The signal path that will be identified by program L138_sysid_intr.c comprises the series combination of the digital-to-analog and analog-to-digital converters and the AC-coupling circuits between converters and jack sockets.

Run the program for a few seconds and then halt it and observe the values of the weights of the adaptive filter (the identified impulse response of the signal path) by selecting *Tools > Graph > Single Time* and setting the *Graph Properties*, as shown in Figure 2.60. Compare the resultant graph, shown in Figure 2.61, with the output of program L138_dimpulse_intr.c shown in Figure 2.45.

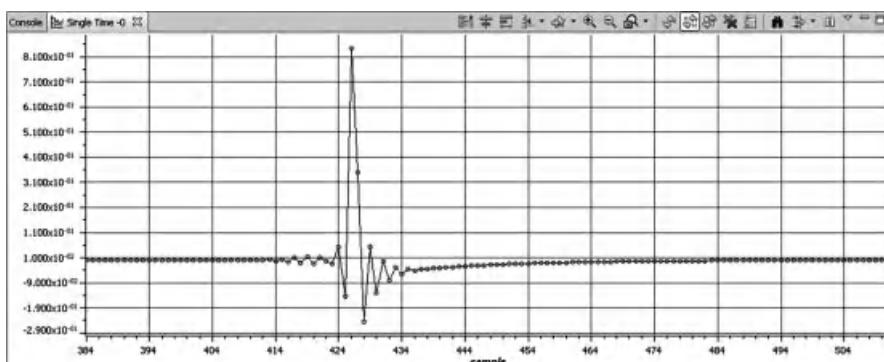


Figure 2.61 The impulse response identified using program L138_sysid_intr.c with connections as shown in Figure 2.60.

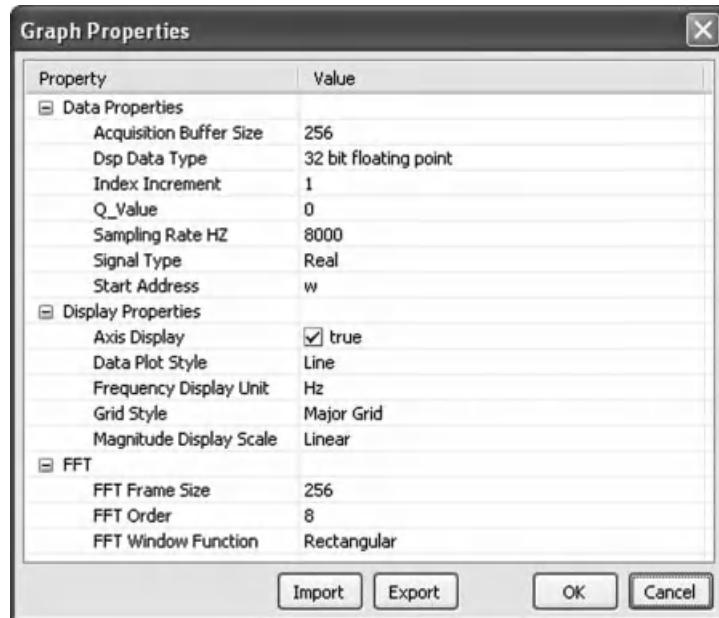


Figure 2.62 Graph Properties for viewing the magnitude frequency response identified using program L138_sysid_intr.c.

To view the magnitude frequency response of the signal path, select *Tools > Graph > FFT Magnitude* and set the *Graph Properties*, as shown in Figure 2.62. The roll-off of the frequency response at very low frequencies is due to the AC-coupling between the codec and the 3.5 mm LINE IN and LINE OUT sockets. The roll-off of the frequency response at frequencies above 3500 Hz is due to the antialiasing and reconstruction filters in the ADC and DAC (Figure 2.63).

Alternatively, the impulse and magnitude frequency responses identified using program L138_sysid_intr.c can be displayed using MATLAB function L138_logfft() after first saving the 256 adaptive filter coefficients, stored in array w, to a *TI Data Format* file using the parameters shown in Figure 2.64.

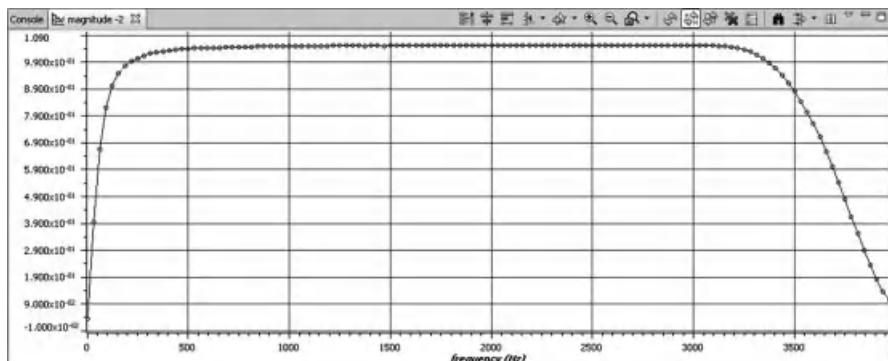


Figure 2.63 The magnitude frequency response identified using program L138_sysid_intr.c with connections as shown in Figure 2.60.

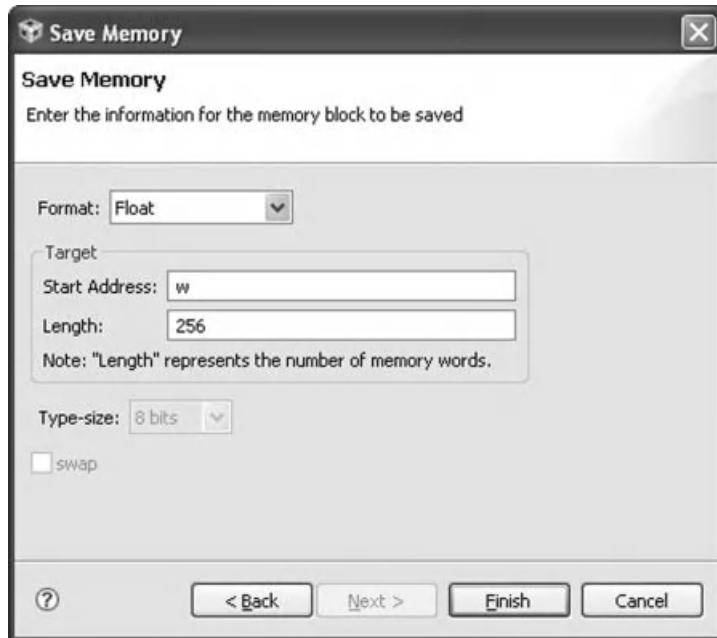


Figure 2.64 Save Memory parameters used to save adaptive filter coefficients to a TI Data Format file (.dat).

Program L138_sysid_intr.c may be used to identify the characteristics of another system (rather than just a cable) connected between LINE OUT and LINE IN (Figure 2.65). Figure 2.66 shows the result when a first-order low-pass filter comprising a capacitor and resistor was used.

Program L138_sysid_deemph_intr.c is similar to L138_sysid_intr.c, but enables the digital de-emphasis filter located just before the DAC in the AIC3106 as in Example 2.16. Figure 2.67 shows the result of running the program with DIP switch #1 ON. Compare this with Figure 2.49.

Another programmable feature of the AIC3106 codec is two first-order high-pass filters (one per channel) located just after the analog-to-digital converter. Using bits 4 through 7 of page 0 control register 12, each of these can be set to have one of three different cutoff frequencies or can be disabled. In addition, they may be programmed to exhibit different characteristics by writing filter coefficients to page 1 control registers 65 through 76 [1]. Figure 2.68 shows the impulse and magnitude frequency responses identified using program L138_sysid_hpf_intr.c in which high-pass filters with a cutoff frequency of 0.025 times the sampling frequency are enabled by writing the value 0xF0 to control register 12.

The plots illustrating examples of the instantaneous frequency response of program L138_flanger_intr.c in Example 2.6 were obtained using program L138_sysid_flange_intr.c, a slightly modified version of program L138_sysid_intr.c that includes a flanger (Figure 2.17), with a fixed delay, in the signal path identified. Program L138_sysid_flange_intr.c is stored in project folder L138_sysid_intr.

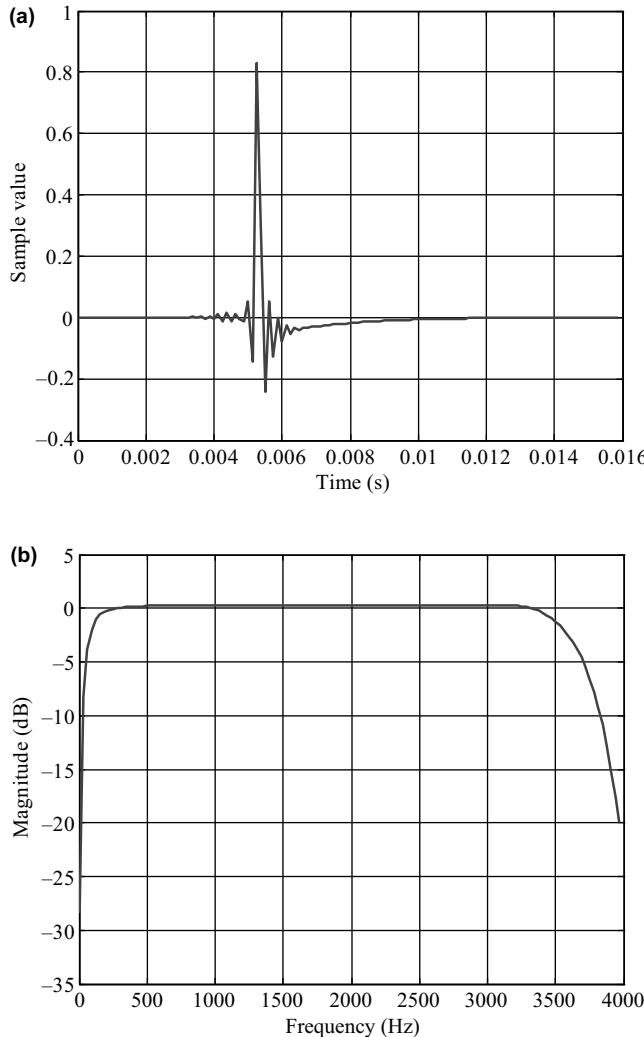


Figure 2.65 The impulse response and magnitude frequency response identified using program L138_sysid_intr.c with connections as shown in Figure 2.59, displayed using MATLAB function L138_logfft().

EXAMPLE 2.22: Identification of AIC3106 Codec Bandwidth Using Two eXperimenters

Program L138_sysid_intr.c can identify frequency response characteristics in the range 0 to half the sampling frequency (in this case, the sampling frequency is equal to 8 kHz), but the antialiasing and reconstruction filters in the codec have a bandwidth only slightly less than this. Hence, in Figures 2.61–2.68, only the pass band of those filters is displayed. The following example uses two eXperimenters, one running program L138_loop_intr.c

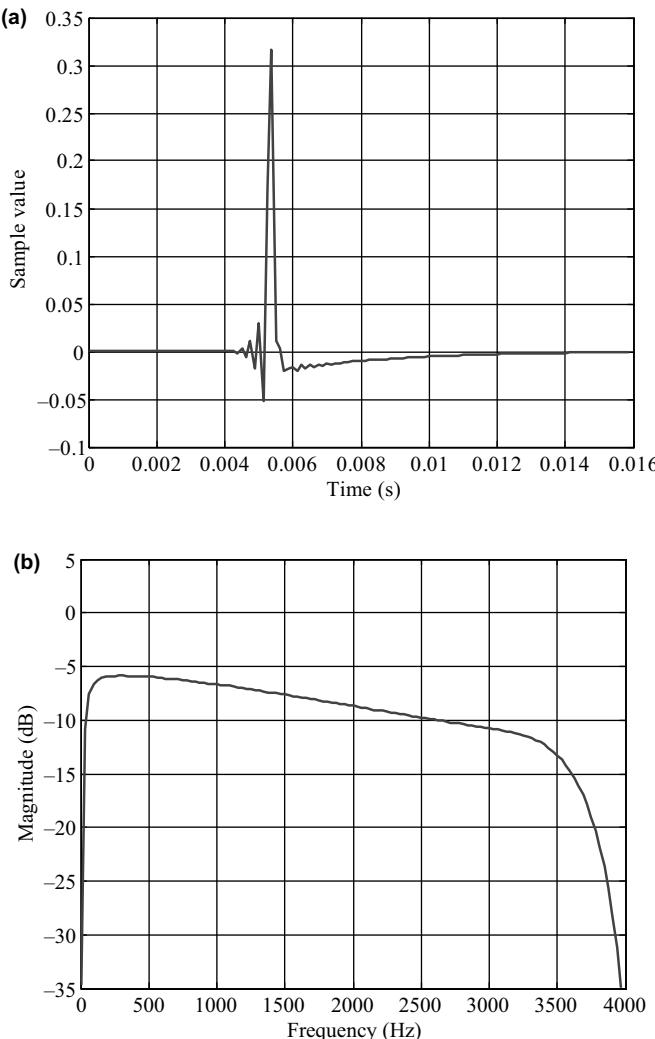


Figure 2.66 The impulse response and magnitude frequency response identified using program L138_sysid_intr.c with a first-order analog filter connected between LINE IN and LINE OUT sockets, displayed using MATLAB function L138_logfft().

with a sampling rate of 8 kHz and the other running program L138_sysid_intr.c using a sampling frequency of 16 kHz, allowing it to identify frequency response characteristics in the range of 0 to 8 kHz and to give a better idea of the pass band, stop band, and transition band of the antialiasing and reconstruction filters. In order to set the sampling frequency in program L138_sysid_intr.c to 16 kHz, change the line that reads

```
L138_initialise_intr(FS_8000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
```

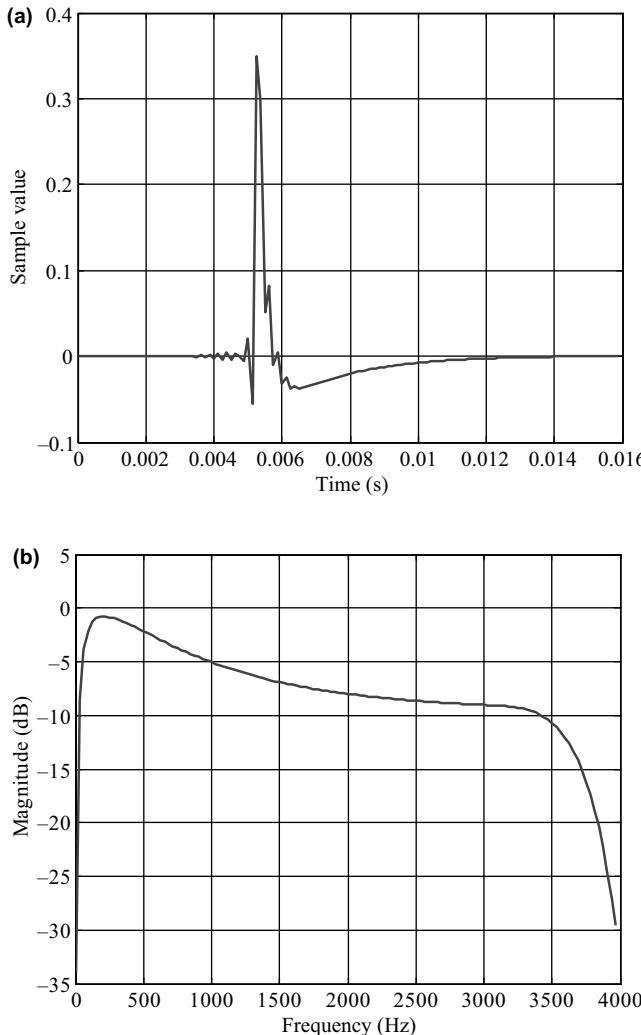


Figure 2.67 The impulse response and magnitude frequency response identified using program L138_sysid_intr.c with connections as shown in Figure 2.59 and de-emphasis enabled, displayed using MATLAB function L138_logfft().

to read

```
L138_initialise_intr(FS_16000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
```

Connect two eXperimenters together, as shown in Figure 2.69. Ensure that program L138_loop_intr.c is running on one eXperimenter before running program L138_sysid_intr.c for a short time on the other. After running and halting the program, select *Tools > Graph > FFT Magnitude* and set the *Graph Properties* as shown in Figure 2.70 in order

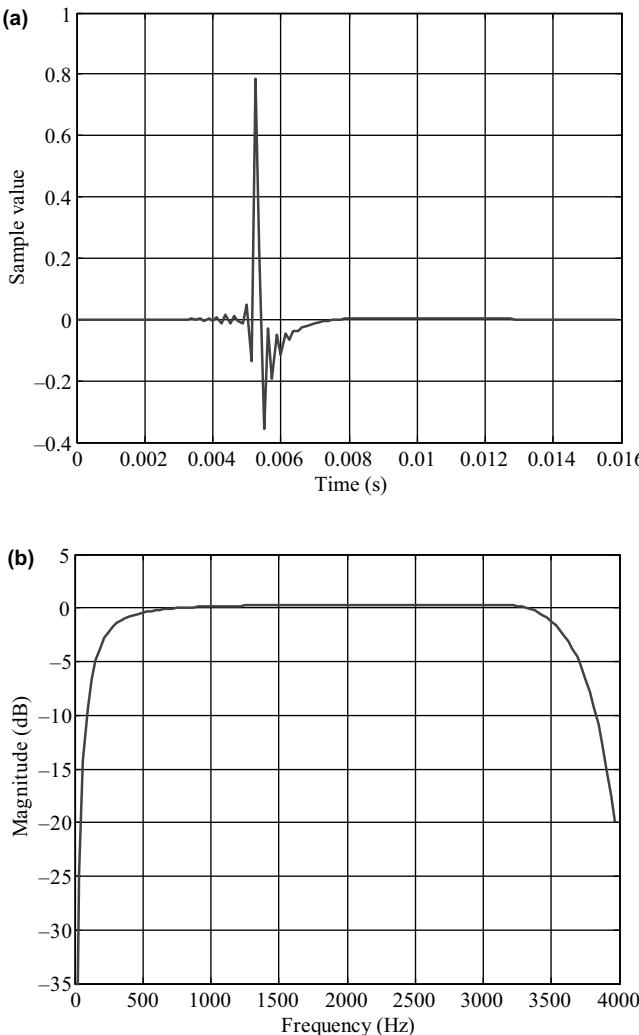


Figure 2.68 The impulse response and magnitude frequency response identified using program L138_sysid_intr.c with connections as shown in Figure 2.59 and ADC high-pass filter enabled and set to a cutoff frequency equal to 0.025 times the sampling frequency, displayed using MATLAB function L138_logfft().

to view the magnitude frequency response of the eXperimenter running program L138_loop_intr.c. Alternatively, save the adaptive filter coefficients in program L138_sysid_intr.c to a .dat file and plot using MATLAB function L138_logfft().

Note in Figure 2.71 that the delay in the identified signal path is greater than that in Figure 2.65. Compared with the connections shown in Figure 2.59, the path identified by the adaptive filter contains another ADC and another DAC.

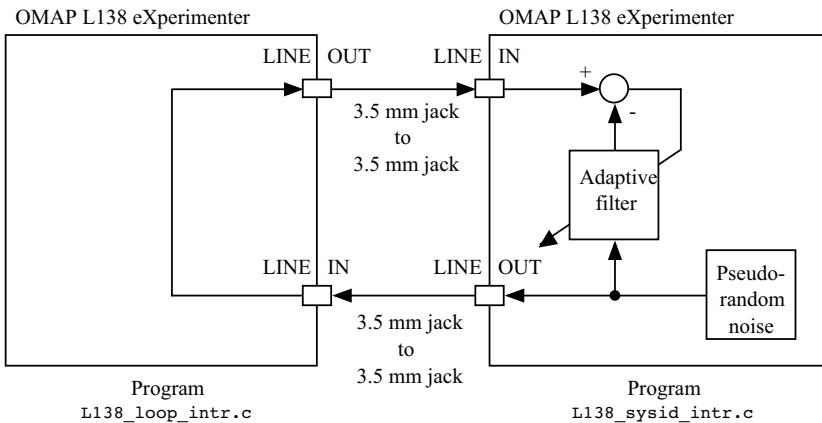


Figure 2.69 Connections for the identification of the bandwidth of the AIC3106 codec using two eXperimenters.

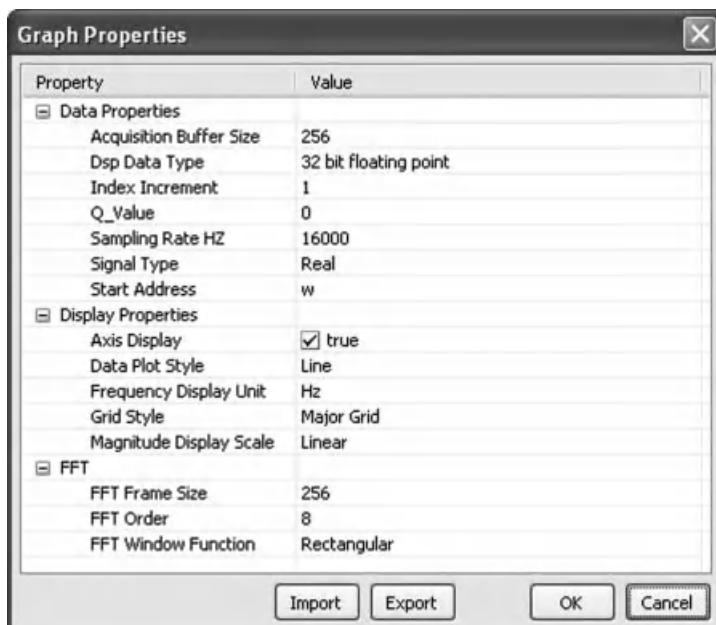


Figure 2.70 *Graph Properties* for viewing the impulse response identified using program L138_sysid_intr.c in Example 2.20.

EXAMPLE 2.23: Ramp Generation (L138_ramp_intr)

Figure 2.72 shows a listing of program L138_ramp_intr.c, which generates a ramp, or sawtooth, output waveform. The value of the output sample is incremented by 2000 every sampling instant until it reaches the value 30000, at which point it is reset to the value -30000.

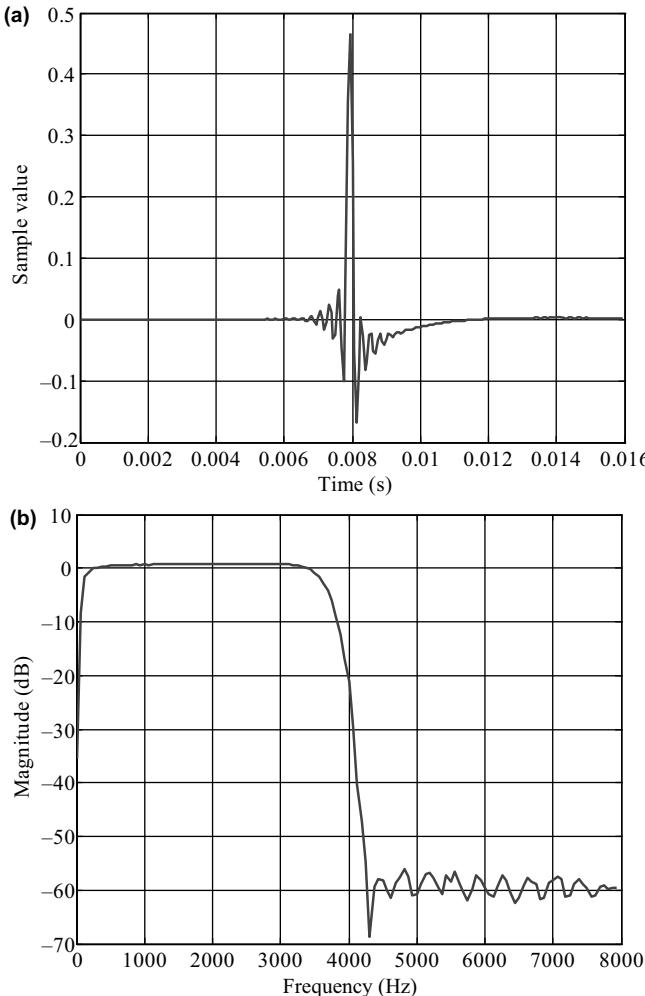


Figure 2.71 The impulse response and magnitude frequency response identified using program L138_sysid_intr.c with connections as shown in Figure 2.69 in Example 2.20, displayed using MATLAB function L138_logfft().

Build and run this program. Figure 2.73 shows the output waveform captured using an oscilloscope.

EXAMPLE 2.24: Amplitude Modulation (L138_am_p011)

This example illustrates an amplitude modulation (AM) scheme. Figure 2.74 shows a listing of program L138_am_p011.c, which generates an AM signal. The array `baseband` holds 20 samples of a single cycle of a cosine waveform with a frequency of $fs/20 = 400$ Hz. The array

```

// L138_ramp_intr.c
//

#include "L138_aic3106_init.h"
#define LOOPLENGTH 64

int16_t output = 0;

interrupt void interrupt4(void) // interrupt service routine
{
    output_left_sample(output); // output to L DAC
    output += 2000; // increment output value
    if (output >= 32000) // if peak is reached
        output = -32000; // reinitialize
    return;
}

int main(void)
{
    L138_initialise_intr(FS_8000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
    while(1);
}

```

Figure 2.72 Listing of program L138_ramp_intr.c.

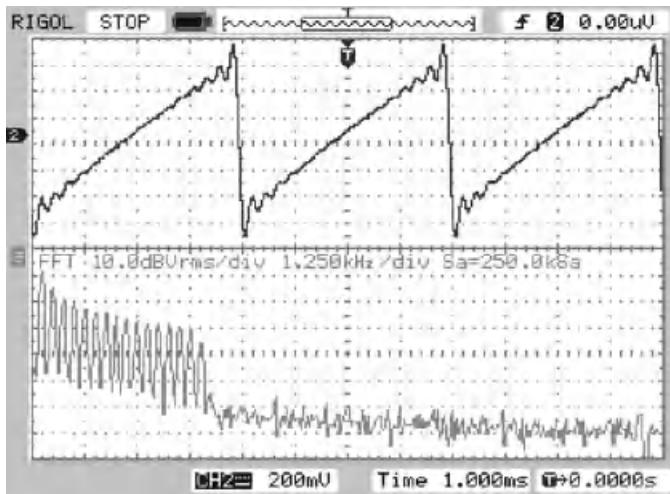


Figure 2.73 Output waveform generated by program L138_ramp_intr.c.

carrier holds 20 samples of 5 cycles of a sinusoidal carrier signal with a frequency of $5 \text{ fs}/20 = 2000 \text{ Hz}$. Output sample values are calculated by multiplying the baseband signal by the carrier signal. In this way, the baseband signal modulates the carrier signal. The variable `amp` is used to set the modulation index. Program L138_am_poll.c uses the polling method for input and output.

Build and run this program. Verify that the output consists of the 2 kHz carrier signal and two sideband signals, as shown in Figure 2.75. The sideband signals are at the frequency of the

```

// L138_am_poll.c
//

#include "L138_aic3106_init.h"

short amp = 20;                                //index for modulation

int main(void)
{
    int16_t baseband[20]={1000,951,809,587,309,0,-309,
                          -587,-809,-951,-1000,-951,-809,
                          -587,-309,0,309,587,809,951}; // 400 Hz
    int16_t carrier[20] ={1000,0,-1000,0,1000,0,-1000,
                          0,1000,0,-1000,0,1000,0,-1000,
                          0,1000,0,-1000,0}; // 2 kHz
    int16_t output[20];
    int16_t k;

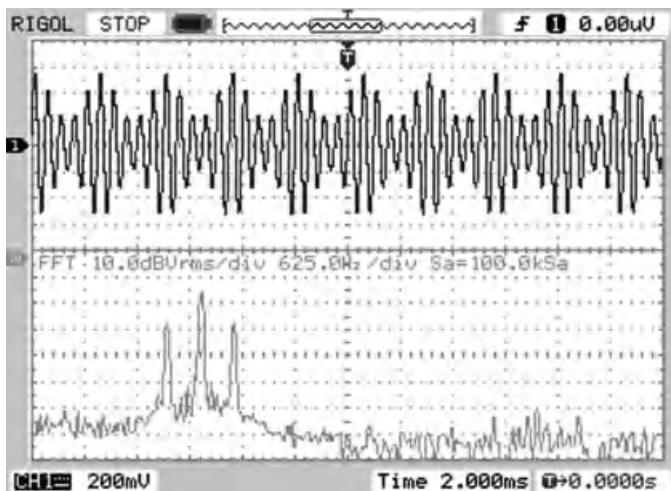
    L138_initialise_poll(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);

    while(1)
    {
        for (k=0; k<20; k++)
        {
            output[k]= carrier[k] + ((amp*baseband[k]*carrier[k]/10)>>12);
            output_left_sample(20*output[k]);
        }
    }
}

```

Figure 2.74 Listing of program L138_am_poll.c.

carrier signal $+/-$ the frequency of the sideband signal or at 1600 and 2400 Hz. The magnitude of the sidebands relative to the carrier signal may be altered by changing the value of the variable amp in the source file. The program will need to be recompiled for such a change to take effect.

**Figure 2.75** Output from program L138_am_poll.c viewed on a Rigol DS1052E oscilloscope.

```

// L138_record_poll.c
//

#include "L138_aic3106_init.h"

#define SAMPLING_FREQ      48000
#define RECORD_SECONDS      5
#define SILENT_SECONDS      2
#define RECORD_SAMPLES      RECORD_SECONDS * SAMPLING_FREQ
#define SILENT_SAMPLES      SILENT_SECONDS * SAMPLING_FREQ

uint32_t i;
uint32_t buffer[RECORD_SAMPLES];
#pragma DATA_SECTION(buffer,".EXT_RAM") // locate buffer
                                         // in external memory

int main()
{
    uint32_t sample;

    L138_initialise_poll(FS_48000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);

    while(1)
    {
        for (i=0; i<RECORD_SAMPLES; i++) // record from LINE IN
        {                                // to buffer
            sample = input_sample();      // for RECORD_TIME_SECONDS
            buffer[i] = sample;
            output_sample(sample);
        }

        for (i=0;i<SILENT_SAMPLES;i++) output_sample(0);

        for (i=0;i<RECORD_SAMPLES;i++)      // play back to LINE OUT
        {                                // from buffer
            sample = buffer[i];
            output_sample(sample);
        }
        for (i=0;i<SILENT_SAMPLES;i++) output_sample(0);
    }
}

```

Figure 2.76 Listing of program L138_record_poll.c.

EXAMPLE 2.25: Use of External Memory to Record Music
(L138_record_poll)

This example illustrates the use of the pragma directive in a C source program to store data in external memory. The C6748 processor contains a total of over 256 kB of internal memory, but the eXperimenter board includes a further 128 MB of SDRAM external memory. Figure 2.76 shows a listing of program L138_record_poll.c that implements this project example. It declares a buffer of size RECORD_SAMPLES 32-bit integers, allowing RECORD_TIME seconds of stereo input signal to be recorded and stored in external memory.

The pragma directive, used in the line

```
#pragma DATA_SECTION(buffer, ".EXT_RAM")
```

specifies that the array `buffer` is allocated to a memory section named `.EXT_RAM`. Within the linker command file `linker_dsp.cmd`, this section is mapped into the external SDRAM on the eXperimenter (starting at address `0x0800000000`). Without the use of the pragma directive, array `buffer` would have been allocated to the memory section `.bss` along with the other variables declared in program `L138_record_poll.c`. Build and run the program. Connect a music source to the LINE IN socket and loudspeakers to the LINE OUT socket. The program loops continuously, recording `RECORD_TIME` seconds of input signal, waiting for `SILENT_TIME` seconds, playing back the `RECORD_TIME` seconds of recorded signal, and waiting for `SILENT_TIME` seconds again before repeating.

REFERENCES

1. *Low-Power Stereo Audio Codec for Portable Audio/Telephony*, SLAS509E, Texas Instruments, Dallas, TX, 2008.
2. *TMS320 DSP/BIOS v5.41 User's Guide*, SPRU42H, Texas Instruments, Dallas, TX, 2009.
3. *TMS320C674x/OMAP-L1x Processor Multichannel Audio Serial Port (McASP) User's Guide*, SPRUFM1, Texas Instruments, Dallas, TX, 2009.
4. *TMS320C674x DSP CPU and Instruction Set Reference Guide*, SPRUFE8B, Texas Instruments, Dallas, TX, 2010.
5. *TMS320C674x DSP Megamodule Reference Guide*, SPRUFK5A, Texas Instruments, Dallas, TX, 2010.
6. *OMAP-L138 C6-Integra™ DSP + ARM® Processor*, SPRS586C, Texas Instruments, Dallas, TX, 2011.
7. *TMS320C6748/46/42 and OMAP-L138 Processor Enhanced Direct Memory Access (EDMA3) Controller User's Guide*, SPRUGP9B, Texas Instruments, Dallas, TX, 2010.

Chapter 3

Finite Impulse Response Filters

- Introduction to digital filters
- The z -transform
- Design and implementation of finite impulse response (FIR) filters
- Programming examples using C and ASM code

3.1 INTRODUCTION TO DIGITAL FILTERS

Filtering is fundamental to digital signal processing. Commonly it refers to processing a sequence of samples representing a time domain signal so as to alter its frequency domain characteristics, and often this consists of attenuating or filtering out selected frequency components. Digital filters are classified according to their structure as nonrecursive FIR filters or as recursive infinite impulse response (IIR) filters. This chapter is concerned with FIR filters. IIR filters are described in Chapter 4.

3.1.1 FIR Filter

A generic FIR filter is shown in block diagram form in Figure 3.1. The components of the filter are:

- (i) A delay line, or buffer, in which a number of previous input samples $x(n - i)$ are stored. At each sampling instant, the contents of the delay line are updated such that samples are shifted one position (to the right in the diagram) and a new input sample $x(n)$ is introduced at the start of the delay line.
- (ii) A number of blocks (multipliers) that multiply the samples stored in the delay line by a set of filter coefficients $h(n)$.
- (iii) A summing junction that sums the multiplier outputs to form the current filter output sample $y(n)$.

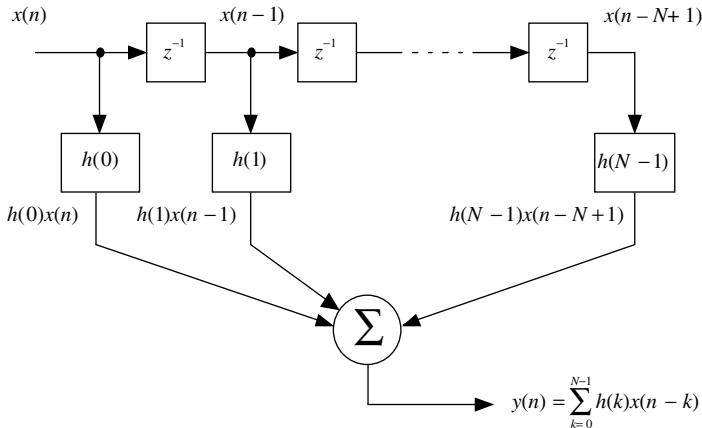


Figure 3.1 Block diagram representation of a generic FIR filter.

In Figure 3.1, the delay line is represented by a series of blocks, each acting as a delay of one sampling period, that is, the output of each block in the delay line is its input, delayed by one sampling period. The z -transfer function of a delay of one sample is z^{-1} .

The multipliers and filter coefficients are represented by blocks, the outputs of which are their inputs multiplied by the filter coefficient with which they are labeled.

At the n th sampling instant, the samples stored in the delay line are $x(n), x(n-1), x(n-2), \dots$ and the output of the filter, $y(n)$, is described by the difference equation:

$$y(n) = \sum_{i=0}^{N-1} h(i)x(n-i). \quad (3.1)$$

This equation is an example of a convolution sum representing the input-output relationship of a discrete-time, linear time invariant system having a finite impulse response $h(n)$ of length N samples. The output of any LTI system (continuous- or discrete-time) is formed by convolving its input signal with its impulse response. For continuous-time LTI systems, there exists a corresponding convolution integral.

Equivalence of an FIR filter's coefficients and its impulse response

The impulse response of an FIR filter is equal to its coefficients. This is straightforward to visualize using the block diagram representation of the filter. Consider a unit impulse input sequence, that is, a sequence containing just one nonzero (unit) sample. This sample enters the delay line on the left-hand side of the block diagram shown in Figure 3.1 and is shifted right at each sampling instant. At any particular sampling instant, the output of the filter will comprise the unit sample multiplied by just one of the filter coefficients. All other filter coefficients will multiply zero sample values

from the delay line and will contribute nothing to the output. Hence, the output sequence $y(n)$ (the unit impulse response of the filter) will comprise the filter coefficients $h(n)$.

Advantages and disadvantages of FIR filters

Although it is possible for FIR filters to approximate the characteristics of continuous-time analog filters, one of their advantages is that they may be used to implement arbitrary filter characteristics that it is impossible to implement using analog circuits. For this reason, and unlike the IIR filters described in Chapter 4, their design is not based on the theory of analog filters. A disadvantage of FIR filters is that their implementation may be computationally expensive. Obtaining an arbitrary filter characteristic, to the required accuracy, may require a large number of filter coefficients.

FIR filter implementation

The structure and operation of an FIR filter is simple (and is a fundamental part of many DSP applications). Typically, the internal architecture of a digital signal processor is optimized (single instruction cycle multiply accumulate units) for efficient computation of the convolution sum. For the convolution sum of Equation 3.1 to be computed, a DSP must have sufficient memory to store N previous input samples and N filter coefficients and should also have sufficient computational power to execute the required number of multiplications and additions within one sampling period.

3.1.2 Introduction to the z-Transform

The z -transform is the discrete-time counterpart of the Laplace transform. The Laplace transform is used to solve continuous-time, linear differential equations, representing them as algebraic expressions in the complex Laplace variable s , and to represent continuous-time LTI systems as s -transfer functions. The Laplace variable s may be viewed as an operator, representing differentiation with respect to time.

The z -transform is used to solve discrete-time difference equations, representing them as algebraic expressions in the complex variable z , and to represent discrete-time LTI systems as z -transfer functions. The variable z may be viewed as an operator, representing a shift of one sample in a sequence.

The z -transform $H(z)$ of a (discrete-time) sequence $h(n)$ is defined as

$$Z\{h(n)\} = H(z) = \sum_{n=0}^{\infty} h(n)z^{-n}, \quad (3.2)$$

where z is a complex variable.

$H(z)$ is therefore a polynomial in z containing as many terms as there are sample values in the sequence $h(n)$. For each term in the polynomial $H(z)$, the coefficient of a power of z corresponds to a sample value in the sequence $h(n)$ and the power to which z is raised corresponds to the position of the sample in the sequence. In a discrete-time system, the factor z^{-n} corresponds to the time $t = nT$ [1].

EXAMPLE 3.1: **z -Transform of Exponential Function** $x(n) = e^{nk}$

The z -transform of the sequence $x(n) = e^{nk}$, where $n \geq 0$ and k is a constant, is

$$X(z) = \sum_{n=0}^{\infty} e^{nk} z^{-n} = \sum_{n=0}^{\infty} (e^k z^{-1})^n. \quad (3.3)$$

Comparing this with the Taylor series approximation,

$$\sum_{n=0}^{\infty} u^n = \frac{1}{1-u}, \quad \text{for } |u| < 1, \quad (3.4)$$

Equation 3.3 becomes

$$X(z) = \frac{1}{1 - e^k z^{-1}} = \frac{z}{z - e^k}, \quad \text{for } |e^k z^{-1}| < 1 \quad \text{or} \quad |z| > |e^k|. \quad (3.5)$$

EXAMPLE 3.2: **z -Transform of Unit Step Function** $x(n) = 1$

The unit step function $x(n) = 1$ may be viewed as an instance of the previous example, where $k = 0$. Hence,

$$X(z) = \frac{1}{1 - z^{-1}} = \frac{z}{z - 1}, \quad |z| > 1. \quad (3.6)$$

EXAMPLE 3.3: **z -Transform of a Sinusoid** $x(n) = \sin(n\omega T)$

A sinusoidal function may be represented by complex exponentials according to Euler's formula $e^{jx} = \cos(x) + j \sin(x)$, that is,

$$\sin(n\omega T) = \frac{e^{jn\omega T} - e^{-jn\omega T}}{2j}, \quad (3.7)$$

and hence the z -transform of the sequence $x(n) = \sin(n\omega T)$ is

$$\begin{aligned} X(z) &= \frac{1}{2j} \sum_{n=0}^{\infty} (e^{jn\omega T} z^{-n} - e^{-jn\omega T} z^{-n}) \\ &= \frac{1}{2j} \sum_{n=0}^{\infty} e^{jn\omega T} z^{-n} - \frac{1}{2j} \sum_{n=0}^{\infty} e^{-jn\omega T} z^{-n}. \end{aligned} \quad (3.8)$$

Using the result (3.5) with $k = j\omega T$ in the first summation and $k = -j\omega T$ in the second,

$$\begin{aligned} X(z) &= \frac{1}{2j} \left(\frac{z}{z - e^{j\omega T}} - \frac{z}{z - e^{-j\omega T}} \right) \\ &= \frac{1}{2j} \left(\frac{z^2 - ze^{-j\omega T} - z^2 + ze^{j\omega T}}{z^2 - z(e^{-j\omega T} + e^{j\omega T}) + 1} \right) \\ &= \frac{z \sin(\omega T)}{z^2 - 2z \cos(\omega T) + 1}, \quad \text{for } |z| > 1. \end{aligned} \quad (3.9)$$

Similarly, using Euler's formula for $\cos(n\omega T)$ as the sum of two complex exponentials, one can find the z -transform of the sequence $x(n) = \cos(n\omega T)$ as

$$X(z) = \frac{z^2 - z \cos(\omega T)}{z^2 - 2z \cos(\omega T) + 1}, \quad \text{for } |z| > 1. \quad (3.10)$$

3.1.3 Properties of the z -Transform

Linearity

The z -transform obeys the laws of superposition.

If

$$Z\{x(n)\} = X(z) \quad \text{and} \quad Z\{y(n)\} = Y(z),$$

then

$$Z\{ax(n) + by(n)\} = aX(z) + bY(z),$$

where $x(n)$ and $y(n)$ are arbitrary sequences and a and b are arbitrary constants.

Shifting property

For a shifted sequence $x[n - m]$ where m is any integer,

$$Z\{x(n - m)\} = z^{-m}X(z). \quad (3.11)$$

From the definition of the z -transform,

$$Z\{x(n-m)\} = \sum_{n=0}^{\infty} x(n-m)z^{-n}. \quad (3.12)$$

Substituting $l = n - m$,

$$\begin{aligned} Z\{x(n-m)\} &= \sum_{l=0}^{\infty} x(l)z^{-(l+m)} \\ &= \sum_{l=0}^{\infty} x(l)z^{-l}z^{-m} \\ &= z^{-m} \sum_{l=0}^{\infty} x(l)z^{-l}, \end{aligned} \quad (3.13)$$

which is recognizable as

$$Z\{x(n-m)\} = z^{-m}X(z). \quad (3.14)$$

Time delay property

Quantity z^{-n} in the z -domain corresponds to a shift of n sampling instants in the time domain.

This is also known as the unit delay property of the z -transform.

Convolution property

The forced output $y(n)$ of an LTI system having impulse response $h(n)$ and input $x(n)$ is (as implemented explicitly by an FIR filter)

$$y(n) = \sum_{m=0}^{\infty} h(m)x(n-m). \quad (3.15)$$

This is the *convolution sum*. Taking its z -transform

$$\begin{aligned} Z\{y[n]\} &= Z\left\{ \sum_{m=0}^{\infty} h[m]x[n-m] \right\}, \\ Y(z) &= \sum_{n=0}^{\infty} \left[\sum_{m=0}^{\infty} h[m]x[n-m] \right] z^{-n}. \end{aligned} \quad (3.16)$$

Changing the order of summation

$$\begin{aligned} Y(z) &= \sum_{m=0}^{\infty} \left[\sum_{n=0}^{\infty} h[m]x[n-m] \right] z^{-n} \\ &= \sum_{m=0}^{\infty} h[m] \left[\sum_{n=0}^{\infty} x[n-m] \right] z^{-n}. \end{aligned} \quad (3.17)$$

Letting $l = n - m$,

$$\begin{aligned} Y(z) &= \sum_{m=0}^{\infty} h[m] \sum_{l=0}^{\infty} x[l] z^{-l} z^{-m} \\ &= \sum_{m=0}^{\infty} h[m] z^{-m} \sum_{l=0}^{\infty} x[l] z^{-l} \\ &= H(z)X(z). \end{aligned} \quad (3.18)$$

Hence,

$$Z\{h[n] * x[n]\} = H(z)X(z), \quad (3.19)$$

that is, the z -transform of the linear convolution of sequences $h(n)$ and $x(n)$ is equivalent to the product of the z -transforms, $H(z)$ and $X(z)$, of $h(n)$ and $x(n)$.

3.1.4 **z -Transfer Functions**

The convolution property of the z -transform is related closely to the concept of the z -transfer function. The z -transfer function of a discrete-time LTI system is defined as the ratio of the z -transform of its output sequence to the z -transform of its input sequence. The z -transform of a system output sequence is therefore the z -transform of its input sequence multiplied by its z -transfer function. Since the z -transform of a unit impulse sequence is equal to unity, the z -transfer function of a system is also equivalent to the z -transform of its impulse response (Figure 3.2).

3.1.5 **Mapping from the s -Plane to the z -Plane**

The Laplace transform can be used to determine the stability of a continuous-time LTI system. If the poles of a system are to the left of the imaginary axis in the s -plane, they correspond to exponentially decaying components of that system's response in the time domain and hence to stability. Poles located in the right-hand half of the s -plane correspond to components of that system's response in the time domain that increase

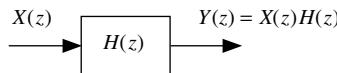
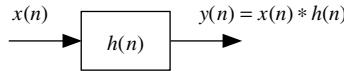


Figure 3.2 Time domain and z -domain block diagram representations of a discrete-time LTI system.

exponentially and hence to instability. Purely imaginary poles correspond to oscillatory (sinusoidal) system response components.

The relationship between the s -plane and the z -plane is represented by the equation $z = e^{sT}$. Substituting for s according to $s = \sigma + j\omega$,

$$z = e^{\sigma T} e^{j\omega T}. \quad (3.20)$$

The magnitude of z is given by $e^{\sigma T}$ and its phase by $e^{j\omega T}$. Consider the three regions of the s -plane that determine system stability.

$\sigma < 0$

The left-hand half of the s -plane represents values of s that have negative real parts and this corresponds to values of z that have magnitudes less than unity ($e^{\sigma T} < 1$). In other words, the left-hand half of the s -plane maps to a region of the complex z -plane inside the unit circle, as shown in Figure 3.3. If the poles of a z -transfer function lie inside that unit circle, then the system represented by that transfer function will be stable.

$\sigma > 0$

The right-hand half of the s -plane represents values of s that have positive real parts and this corresponds to values of z that have magnitudes greater than unity ($e^{\sigma T} > 1$).

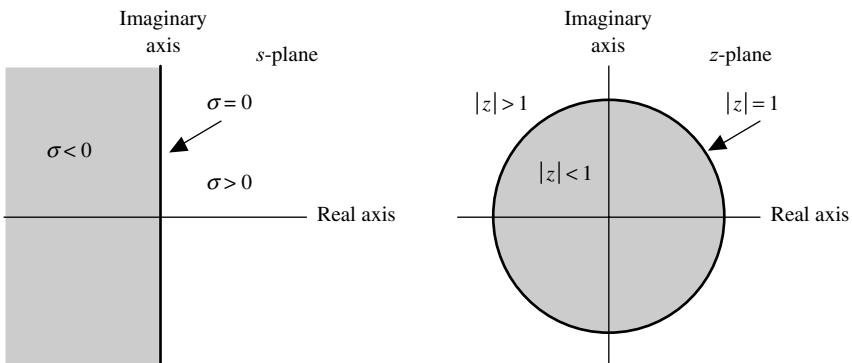


Figure 3.3 Mapping from the s -plane to the z -plane.

In other words, the right-hand half of the s -plane maps to a region of the complex z -plane outside the unit circle, as shown in Figure 3.3. If the poles of a z -transfer function lie outside that unit circle, then the system represented by that transfer function will be unstable.

$$\sigma = 0$$

The imaginary axis $s = j\omega$ in the s -plane maps to the unit circle $z = e^{j\omega T}$ in the z -plane. If the poles of a z -transfer function lie on the unit circle in the z -plane, then the system will have an oscillatory response. The imaginary axis in the s -plane maps to the unit circle in the z -plane.

3.1.6 Difference Equations

A digital filter is represented by a difference equation in a way similar to that in which an analog filter is represented by a differential equation. In a way similar to that in which a differential equation may be solved using Laplace transforms, a difference equation may be solved using z -transforms. In order to do this, the z -transforms of expressions such as $x(n - k)$, which corresponds to the k th derivative $d^k x(t)/dt^k$ of an analog signal $x(t)$, must be found. From the definition of the z -transform,

$$X(z) = \sum_{n=0}^{\infty} x(n)z^{-n} = x(0) + x(1)z^{-1} + x(2)z^{-2} + \dots \quad (3.21)$$

The z -transform of $x(n - 1)$, which corresponds to a first-order derivative $dx(t)/dt$, is

$$\begin{aligned} Z\{x(n - 1)\} &= \sum_{n=0}^{\infty} x(n - 1)z^{-n} \\ &= x(-1) + x(0)z^{-1} + x(1)z^{-2} + x(2)z^{-3} + \dots \\ &= x(-1) + z^{-1}[x(0) + x(1)z^{-1} + x(2)z^{-2} + \dots] \\ &= x(-1) + z^{-1}X(z), \end{aligned} \quad (3.22)$$

where $x(-1)$ represents the initial condition associated with a first-order difference equation. Similarly, the z -transform of $x(n - 2)$, which corresponds to a second-order derivative $d^2x(t)/dt^2$, is

$$\begin{aligned} Z\{x(n - 2)\} &= \sum_{n=0}^{\infty} x(n - 2)z^{-n} \\ &= x(-2) + x(-1)z^{-1} + x(0)z^{-2} + x(1)z^{-3} + \dots \\ &= x(-2) + x(-1)z^{-1} + z^{-2}[x(0)z^{-1} + x(1)z^{-2} + \dots] \\ &= x(-2) + x(-1)z^{-1} + z^{-2}X(z), \end{aligned} \quad (3.23)$$

where $x(-2)$ and $x(-1)$ represent the two initial conditions associated with a second-order difference equation. In general,

$$Z\{x(n-k)\} = z^{-k} \sum_{m=1}^k x(-m)z^m + z^{-k}X(z). \quad (3.24)$$

If the initial conditions are all zero, then $x(-m) = 0$ for $m = 1, 2, \dots, k$ and Equation 3.24 reduces to

$$Z\{x(n-k)\} = z^{-k}X(z). \quad (3.25)$$

3.1.7 Frequency Response and the z-Transform

The frequency response of a discrete-time system can be found by evaluating its z -transfer function for $z = e^{j\omega T}$, where ω represents frequency in radians per second and T represents sampling period in seconds. In other words, the frequency response is found by evaluation of a z -transfer function for values of z that lie on the unit circle in the z -plane and therefore it is periodic in ωT . It is common to express the frequency response of a discrete-time system as a function of normalized frequency $\hat{\omega} = \omega T$ over a range of 2π radians. Evaluating a z -transform for $z = e^{j\hat{\omega}T}$, that is, around the unit circle in the z -plane, is equivalent to evaluating a Laplace transform for $s = j\omega$, that is, along the imaginary axis in the s -plane.

If $H(z)$ represents the transfer function of a discrete-time LTI system, then evaluation of that expression using $z = e^{j\hat{\omega}T}$ yields the system's frequency response.

$$H(\hat{\omega}) = \sum_{n=0}^{N-1} h(n)e^{-jn\hat{\omega}}. \quad (3.26)$$

This equation is equivalent to applying the discrete-time Fourier transform (DTFT)

$$H(\hat{\omega}) = \sum_{n=-\infty}^{\infty} h(n)e^{-jn\hat{\omega}} \quad (3.27)$$

to the finite set of filter coefficients $h(n)$.

3.1.8 Ideal Filter Response Classifications: LP, HP, BP, and BS

Shown in Figure 3.4 are the magnitude frequency responses of ideal low-pass, high-pass, band-pass, and band-stop filters. These are some of the most common filter characteristics used in a range of applications. In later sections, it will be shown that high-pass, band-pass, and band-stop filters may easily be derived from low-pass designs.

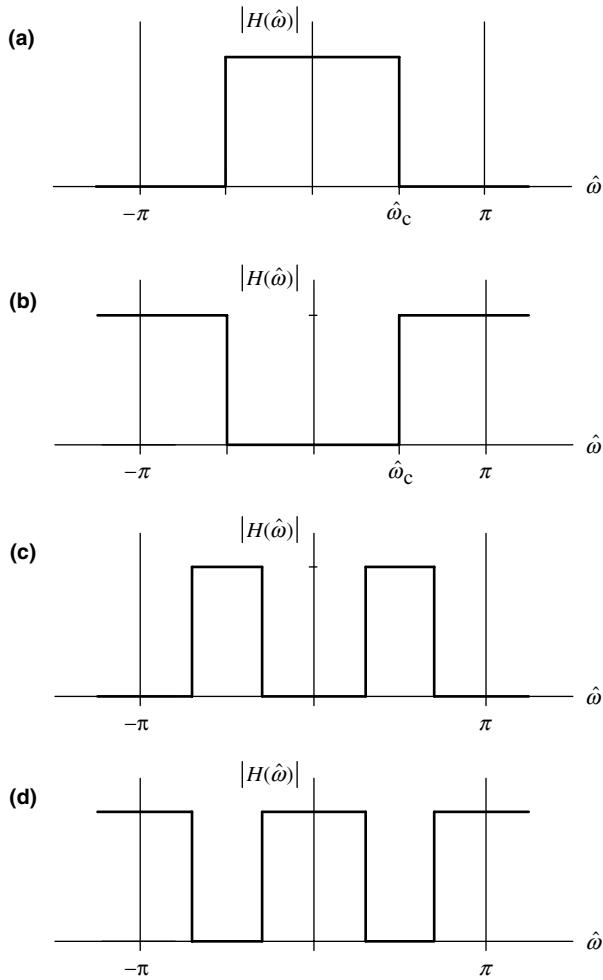


Figure 3.4 Ideal filter magnitude frequency responses. (a) Low-pass (LP). (b) High-pass (HP). (c) Band-pass (BP). (d) Band-stop (BS).

3.1.9 Window Method of Filter Design

The window, or Fourier series, approach to FIR filter design comprises three basic steps:

- (1) Specify a desired (continuous) frequency response.
- (2) Use inverse Fourier transformation to obtain a corresponding (discrete) impulse response.
- (3) Multiply that impulse response (the FIR filter coefficients) by a finite, tapered window function.

The third step is necessary because a practical FIR filter must have a finite number of coefficients. If the inverse discrete-time Fourier transform is applied to a continuous frequency response, it will, in general, yield an infinite sequence in the time domain. A finite window function will truncate that sequence. A tapered window function will reduce ripple (gain variation) in the resulting frequency response.

Since the impulse response of an FIR filter is discrete, its frequency response will be periodic and, therefore, its desired frequency response need be specified over just one period (2π radians of normalized frequency $\hat{\omega}$). As stated in Section 3.1.7, the coefficients of an FIR filter (its impulse response) and its continuous periodic frequency response are related by the discrete-time Fourier transform. In other words, FIR filter coefficients may be computed from a desired continuous frequency response using the inverse DTFT. This can be relatively straightforward for some analytic frequency responses (expressed as algebraic functions of $\hat{\omega}$), including the ideal low-pass filter characteristic shown in Figure 3.4, but for arbitrary frequency responses, applying the inverse DTFT may be problematic.

3.1.10 Window Functions

A number of different tapered window functions are used in FIR filter design. All have the effect of reducing the magnitude of gain variations in the filter frequency response at the expense of a less sharp transition between pass- and stopbands. These properties are related to the difference in magnitude between the peak and the first sidelobe and the width of the central lobe of the DTFT of the (discrete) window function itself.

For example, a rectangular window has the narrowest central lobe (corresponding to the sharpest transition between pass- and stopbands), but its first side lobe is only 13 dB down from the peak of its central main lobe.

The following window functions are among the most commonly used.

Hamming window

The Hamming window function is

$$w_{\text{HAMM}}(n) = \begin{cases} 0.54 + 0.46 \cos(n\pi/L), & -L \leq n \leq L, \\ 0, & \text{otherwise.} \end{cases} \quad (3.28)$$

The magnitude of the first side lobe is approximately 43 dB less than that of the main lobe.

Hanning window

The Hanning window function is

$$w_{\text{HANN}}(n) = \begin{cases} 0.5 + 0.5 \cos(n\pi/L), & -L \leq n \leq L, \\ 0, & \text{otherwise.} \end{cases} \quad (3.29)$$

The magnitude of the first side lobe is approximately 31 dB less than that of the main lobe.

Blackman window

The Blackman window function is

$$w_{\text{BLACK}}(n) = \begin{cases} 0.42 + 0.5 \cos(n\pi/L) + 0.08 \cos(2n\pi/L), & -L \leq n \leq L, \\ 0, & \text{otherwise.} \end{cases} \quad (3.30)$$

The magnitude of the first side lobe is approximately 58 dB less than that of the main lobe. Although the Blackman window produces a greater reduction in side lobe magnitude than the Hamming or the Hanning window, it has a significantly wider main lobe. Used in the design of a low-pass filter, the Blackman window will reduce ripple in the magnitude frequency response significantly, but will result in a relatively gradual transition from passband to stopband.

Kaiser window

The Kaiser window is very popular for use in FIR filter design. It has a variable parameter to control the size of the side lobe relative to the main lobe. The Kaiser window function is

$$w_{\text{KAISER}}(n) = \begin{cases} I_0(b)/I_0(a), & |n| \leq L, \\ 0, & \text{otherwise,} \end{cases} \quad (3.31)$$

where a is an empirically determined variable and $b = a[1 - (n/L)^2]^{1/2}$. $I_0(x)$ is a modified Bessel function of the first kind defined by

$$I_0(x) = 1 + \frac{0.25x^2}{(1!)^2} + \frac{(0.25x^2)^2}{(2!)^2} + \dots = 1 + \sum_{n=1}^{\infty} \left[\frac{(x/2)^n}{n!} \right]^2, \quad (3.32)$$

which converges rapidly. A trade-off between the magnitude of the side lobe and the width of the main lobe can be achieved by changing the length of the window, n , and the value of the parameter a .

EXAMPLE 3.4: Design of an Ideal Low-Pass FIR Filter Using the Window Method

The ideal low-pass filter characteristic shown in Figure 3.5 is described by

$$H(\hat{\omega}) = \begin{cases} 1, & |\hat{\omega}| < \hat{\omega}_c, \\ 0, & \text{otherwise,} \end{cases} \quad (3.33)$$

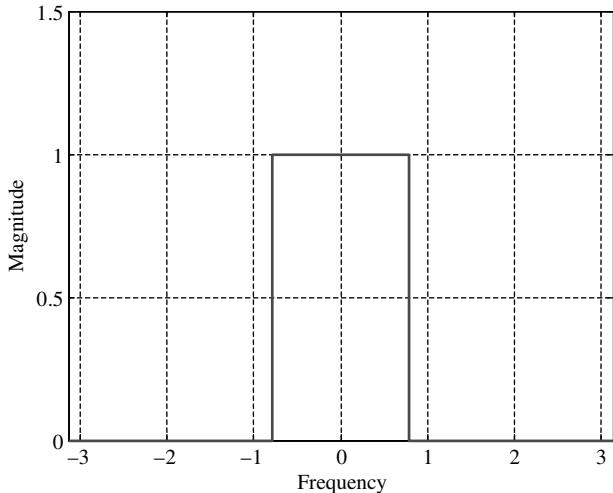


Figure 3.5 Ideal low-pass frequency response defined over normalized frequency range $-\pi \leq \hat{\omega} \leq \pi$, for $\hat{\omega}_c = \pi/4$.

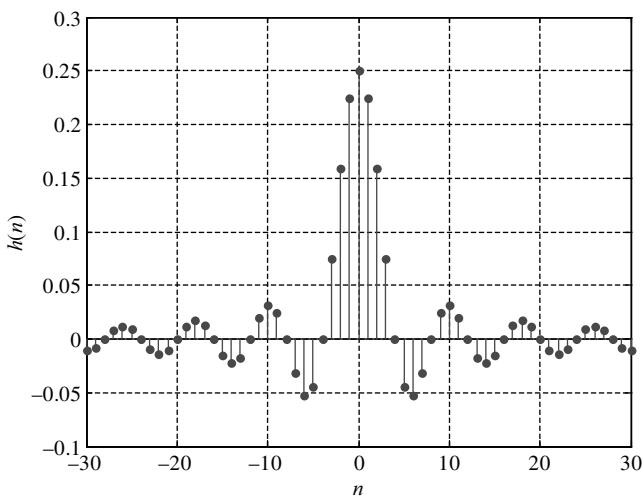


Figure 3.6 Sixty-one of the infinite number of values in the discrete-time impulse response obtained by applying the inverse DTFT to the ideal low-pass filter frequency response of Figure 3.5.

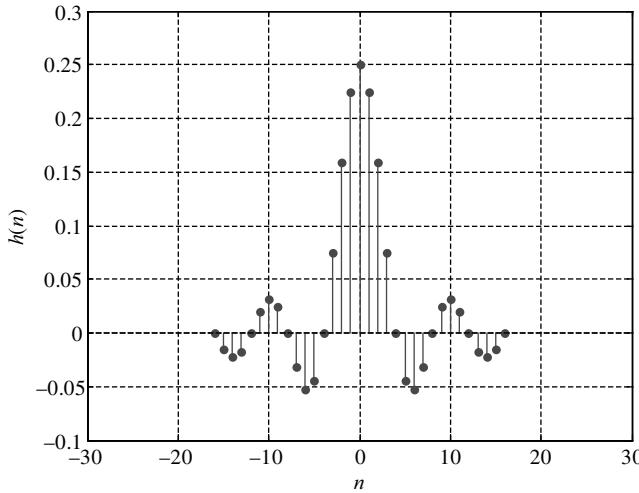


Figure 3.7 The discrete-time impulse response of Figure 3.6 truncated to $N=33$ values.

and the inverse DTFT of Equation 3.33 is

$$\begin{aligned} h(n) &= \int_{-\hat{\omega}_c}^{\hat{\omega}_c} e^{jn\hat{\omega}} d\hat{\omega} \\ &= \frac{\sin(\hat{\omega}_c n)}{\pi n} = \frac{\hat{\omega}_c}{\pi} \operatorname{sinc}\left(\frac{\hat{\omega}_c n}{\pi}\right), \quad -\infty \leq n \leq \infty. \end{aligned} \quad (3.34)$$

This result is illustrated in Figure 3.6, for $\hat{\omega}_c = \pi/4$, over the range $-30 \leq n \leq 30$. In order to implement an FIR filter, impulse response $h(n)$ must be truncated by multiplication with a window function of finite extent. Figure 3.7 shows the result of truncating $h(n)$ using a rectangular window of length 33. Truncation has the effect of introducing gain variations (ripple) into the corresponding frequency response as shown in Figure 3.8. This continuous (periodic) frequency response is found by taking the (forward) DTFT of the truncated impulse response shown in Figure 3.7.

Multiplying the impulse response of Figure 3.6 by a tapered window function has the effect of reducing the ripple in the magnitude frequency response at the expense of making the transition from passband to stopband less sharp. Figures 3.9–3.11 show a 33-point Hanning window function, the result of multiplying the impulse response (filter coefficients) of Figure 3.6 by that window function, and the magnitude frequency response corresponding to the filter coefficients shown in Figure 3.10, respectively.

Figure 3.12 shows the magnitude frequency responses of Figures 3.8 and 3.11 plotted together on a logarithmic scale. This emphasizes the wider main lobe and suppressed side lobes associated with the Hanning window.

Finally, it is necessary to shift the time domain filter coefficients. The preceding figures show magnitude frequency responses that are even functions of frequency and for which zero phase shift is assumed. These correspond to real-valued filter coefficients that are even functions

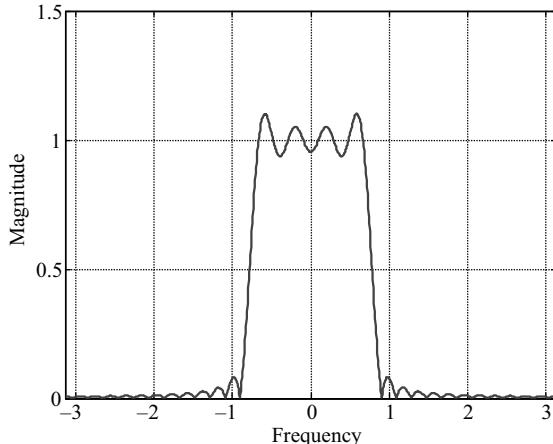


Figure 3.8 The continuous (periodic) frequency response obtained by taking the DTFT of the truncated impulse response shown in Figure 3.7 (plotted against normalized frequency $\hat{\omega}$).

of time, but are also noncausal (filter coefficient $h(0)$ is assumed to correspond to time $t = 0$). This can be changed by introducing a delay and indexing the coefficients from 0 to 32 rather than from -16 to 16 . This has no effect on the magnitude, but introduces a linear phase shift to the frequency response of the filter.

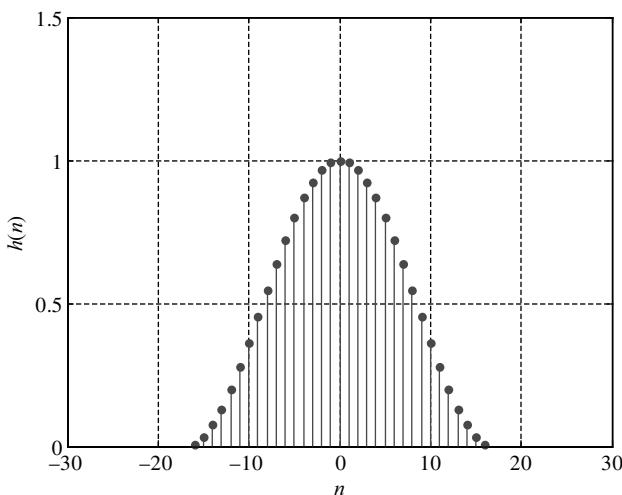


Figure 3.9 A 33-point Hanning window function.

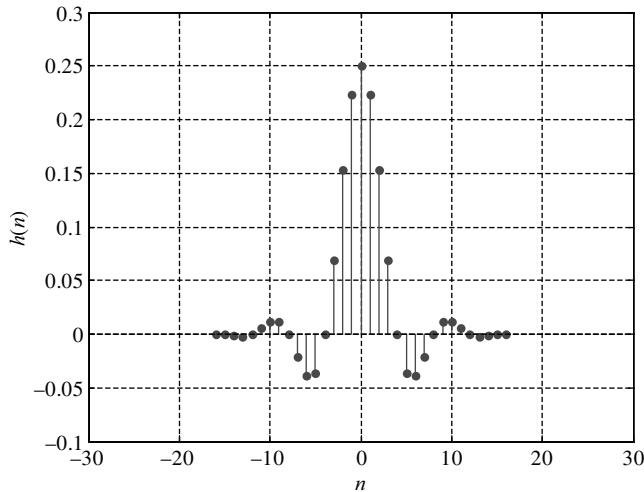


Figure 3.10 The filter coefficients of Figure 3.6 multiplied by the Hanning window of Figure 3.9.

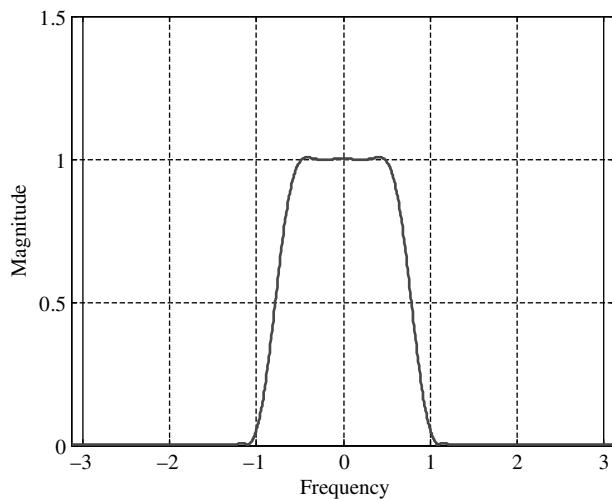


Figure 3.11 The magnitude frequency response corresponding to the filter coefficients of Figure 3.10 (plotted against normalized frequency $\hat{\omega}$).

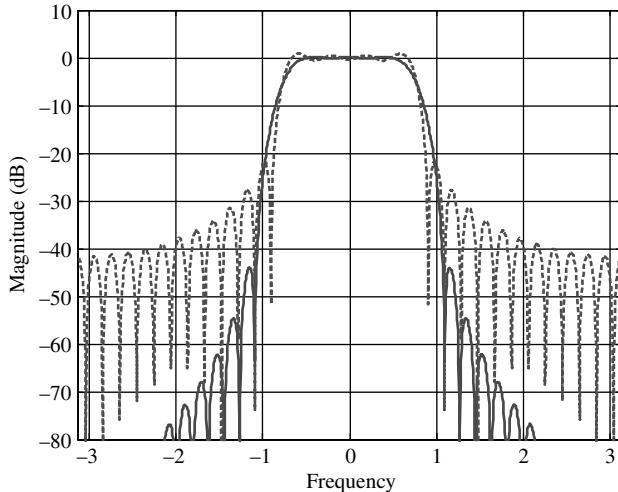


Figure 3.12 The magnitude frequency responses of Figures 3.8 (dotted) and 3.11 plotted on a logarithmic scale, against normalized frequency $\hat{\omega}$.

3.1.11 Design of Band-Pass and High-Pass Filters Using Frequency Shifting

The frequency shifting property of the Fourier transform may be used to derive band-pass and high-pass filter designs from low-pass filter designs. For example, the low-pass filter design of Example 3.4 may be transformed into a band-pass filter with bandwidth $\pi/2$, centered on frequency $3\pi/8$, simply by multiplying its coefficients by samples of a cosine wave with frequency $3\pi/8$, as shown in Figures 3.13–3.15.

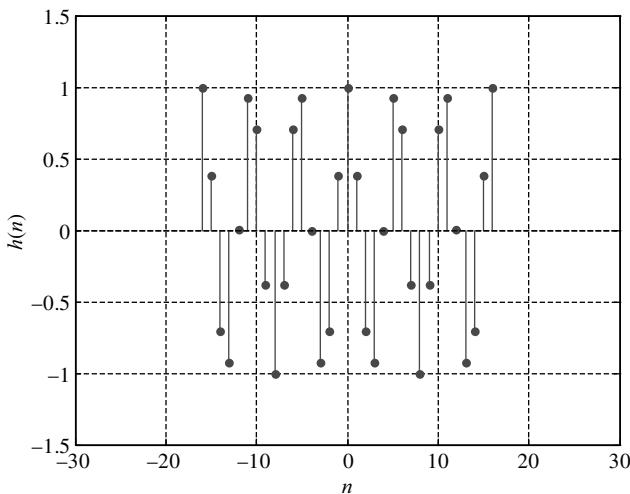


Figure 3.13 Thirty-three samples of a cosine wave of frequency $\hat{\omega} = 3\pi/8$.

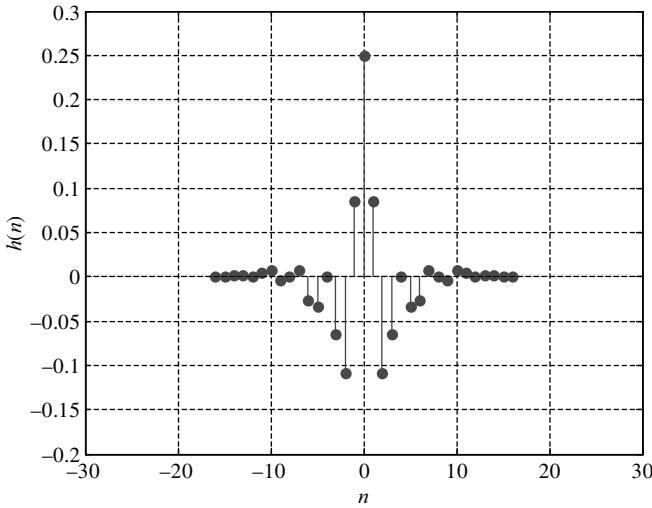


Figure 3.14 The low-pass filter coefficients derived in Example 3.4 multiplied by the samples shown in Figure 3.13.

Similarly, the low-pass filter design of Example 3.4 may be transformed into a high-pass filter with bandwidth $\hat{\omega} = \pi/2$, centered on frequency π by multiplying its coefficients by samples of a cosine wave with frequency $\hat{\omega} = \pi$, as shown in Figures 3.16–3.18.

The samples of a cosine of frequency $\hat{\omega} = \pi$ are such that the transformation of a low-pass filter to a high-pass filter is achieved simply by changing the polarity of every other coefficient.

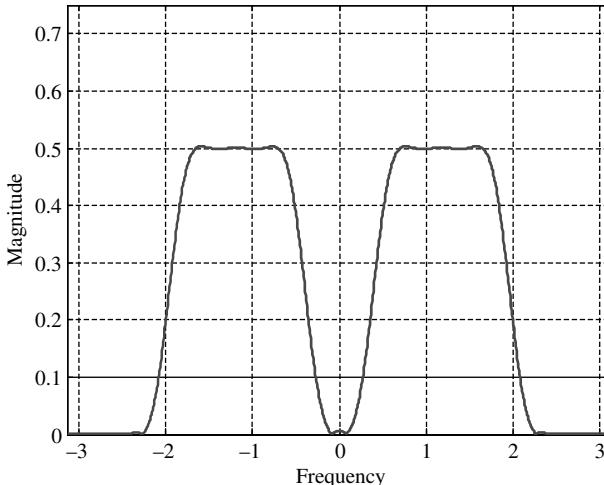


Figure 3.15 The magnitude frequency response corresponding to the band-pass filter coefficients shown in Figure 3.14 (plotted against normalized frequency $\hat{\omega}$).

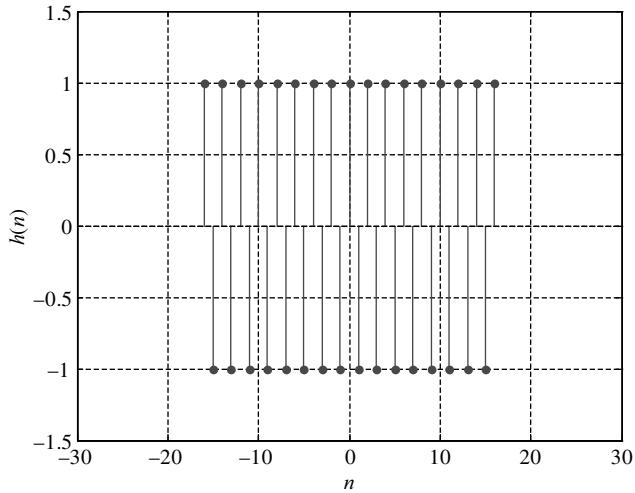


Figure 3.16 Thirty-three samples of a cosine wave of frequency $\hat{\omega} = \pi$.

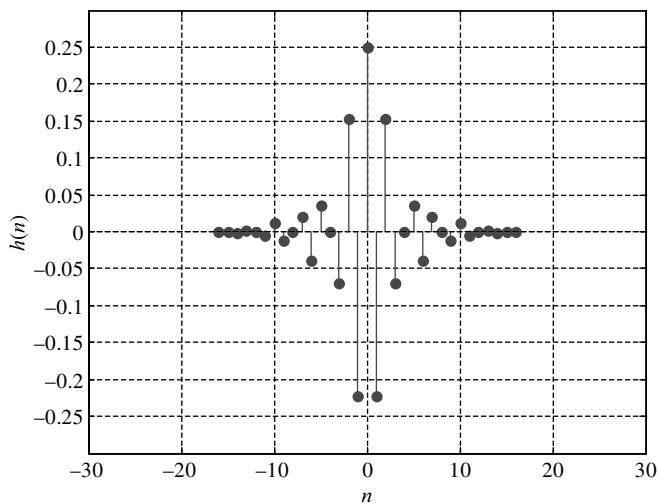


Figure 3.17 The low-pass filter coefficients derived in Example 3.4 multiplied by the samples shown in Figure 3.16.

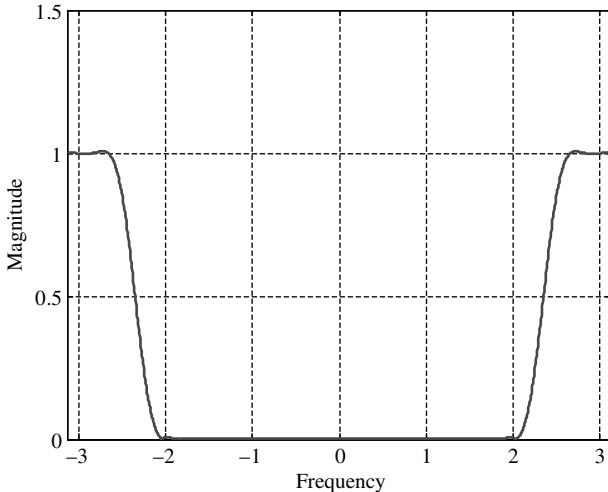


Figure 3.18 The magnitude frequency response, plotted against normalized frequency $\hat{\omega}$, corresponding to the filter coefficients shown in Figure 3.17.

3.2 PROGRAMMING EXAMPLES USING C AND ASM CODE

The following examples illustrate the implementation of FIR filters using C and assembly language. The use of an optimized FIR filtering routine from the c674x DSPLIB library [2] is demonstrated and several different methods of displaying or estimating the magnitude frequency response of a filter are presented.

EXAMPLE 3.5: Moving Average Filter (L138_average_intr)

The moving average filter is widely used in DSP and arguably is the easiest of all digital filters to understand. It is particularly effective at removing (high frequency) random noise from a signal or at smoothing a signal.

The moving average filter operates by taking the arithmetic mean of a number of past input samples in order to produce each output sample. This may be represented by the equation

$$y(n) = \frac{1}{N} \sum_{i=0}^{N-1} x(n-i), \quad (3.35)$$

where $x(n)$ represents the n th sample of an input signal and $y(n)$ the n th sample of the filter output. The moving average filter is an example of convolution using a very simple filter kernel, or impulse response, comprising N coefficients each of value $1/N$. Equation 3.35 may be thought of as a particularly simple case of the more general convolution sum implemented by a finite impulse response filter, and introduced in Section 3.1, that is,

$$y(n) = \sum_{i=0}^{N-1} h(i)x(n-i), \quad (3.36)$$

where the FIR filter coefficients $h(i)$ are samples of the filter impulse response and in the case of the moving average filter, each is equal to $1/N$. As far as implementation is concerned, at the n th sampling instant, we could

- (i) multiply N past input samples individually by $1/N$ and sum the N products,
- (ii) sum N past input samples and multiply the sum by $1/N$, or
- (iii) maintain a moving average by adding a new input sample (multiplied by $1/N$) to and subtracting the $(n - N + 1)$ th input sample (multiplied by $1/N$) from a running total.

The third method of implementation is recursive, that is, calculation of the output $y(n)$ makes use of a previous output value $y(n - 1)$. The recursive expression

$$y(n) = \frac{1}{N}x(n) - \frac{1}{N}x(n-N) + y(n-1) \quad (3.37)$$

conforms to the general expression for a recursive or infinite impulse response filter

$$y(n) = \sum_{k=0}^M b_k x(n-k) - \sum_{l=1}^N a_l y(n-l) \quad (3.38)$$

that is examined in Chapter 4.

Program L138_average_intr.c, listed in Figure 3.19, uses the first of these options, even though it is not the most computationally efficient. The value of N defined near the start of the source file determines the number of previous input samples to be averaged. Launch the Code Composer Studio IDE selecting workspace L138_chapter3 and make project L138_average_intr active. Build and run the program.

Several different methods exist by which the characteristics of the five-point moving average filter may be demonstrated. A test file mefsin.wav, stored in folder L138_average_intr, contains a recording of speech corrupted by the addition of a sinusoidal tone. Listen to this file using *Goldwave*, *Windows Media Player*, or similar. Then connect the PC sound card line output to the LINE IN socket on the eXperimenter and listen to the filtered test signal (LINE OUT). You should find that the sinusoidal tone has been blocked and that the voice sounds muffled. Both observations are consistent with the filter having a low-pass frequency response.

```

// L138_average_intr.c
//

#include "L138_aic3106_init.h"

#define N 5           // number of points averaged
float x[N];          // filter input delay line
float h[N];          // filter coefficients

interrupt void interrupt4(void) // interrupt service routine
{
    short i;
    float yn = 0.0;           // filter output

    x[0] = (float)(input_left_sample()); // input from ADC
    for (i=0 ; i<N ; i++)           // calculate output
        yn += h[i]*x[i];
    for (i=(N-1) ; i>0 ; i--)       // shift delay line
        x[i] = x[i-1];
    output_left_sample((uint16_t)(yn)); // output to DAC
    return;
}

int main(void)
{
    short i;

    for (i=0 ; i<N ; i++)           // initialise coeffs
        h[i] = 1.0/N;
    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);                      // loop forever
}

```

Figure 3.19 Listing of program L138_average_intr.c.

A more rigorous method of assessing the magnitude frequency response of the filter is to use a signal generator and an oscilloscope or spectrum analyzer to measure its gain at different individual frequencies. Using this method, it is straightforward to identify two distinct notches in the magnitude frequency response at 1600 Hz (corresponding to the tone in test file mefsin.wav) and at 3200 Hz.

The theoretical frequency response of the filter can be found by taking the discrete-time Fourier transform of its coefficients

$$H(\hat{\omega}) = \sum_{n=0}^{N-1} h[n]e^{-j\hat{\omega}n} \quad (3.39)$$

evaluated over the frequency range $0 \leq \hat{\omega} < 2\pi$, where $\hat{\omega} = \omega T_s$ and T_s is the sampling period.

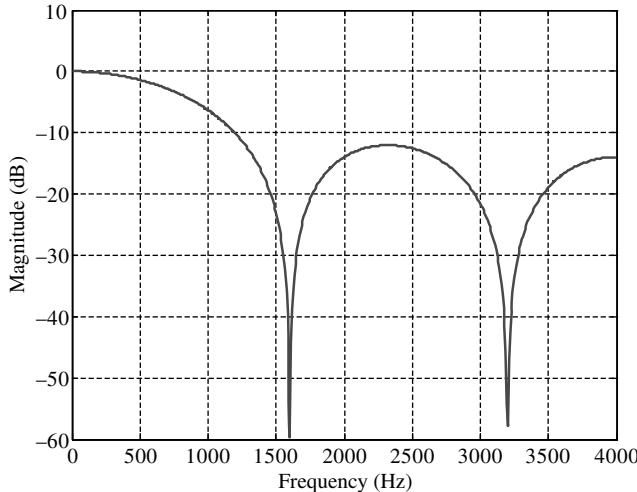


Figure 3.20 Theoretical magnitude frequency response of five-point moving average filter (sampling rate 8 kHz).

In this case,

$$\begin{aligned}
 |H(\hat{\omega})| &= \left| \sum_{n=0}^4 0.2e^{-j\hat{\omega}n} \right| \\
 &= \left| \sum_{n=-2}^2 0.2e^{-j\hat{\omega}n} \right| \\
 &= |0.2(e^{j2\hat{\omega}} + e^{j\hat{\omega}} + 1 + e^{-j\hat{\omega}} + e^{-j2\hat{\omega}})| \\
 &= 0.2(1 + 2\cos(\hat{\omega}) + 2\cos(2\hat{\omega})).
 \end{aligned} \tag{3.40}$$

Changing the summation limits from $0 \leq n \leq 4$ to $-2 \leq n \leq 2$ changes the phase but not the magnitude of the frequency response of the filter. The theoretical magnitude frequency response of the filter is illustrated in Figure 3.20. This is the magnitude of a Dirichlet, or periodic sinc, function.

EXAMPLE 3.6: Moving Average Filter with Internally Generated Pseudorandom Noise as Input (`L138_average_prn_intr`)

Another method of assessing the magnitude frequency response of a filter is to use wideband noise as an input signal. Program `L138_average_prn_intr.c` demonstrates this technique. A pseudorandom binary sequence (PRBS) is generated within the program (see program `L138_prbs_intr.c` in Chapter 2) and used as an input to the filter in lieu of samples read from the ADC. The filtered noise can be viewed on a spectrum analyzer and whereas the frequency content of the PRBS input is uniform across all frequencies, the frequency content of

the filtered noise corresponds to the frequency response of the filter. Many modern oscilloscopes are equipped with FFT functions and one of these, or the *Goldwave* application, provide low-cost alternatives to using a dedicated spectrum analyzer. Figure 3.21 shows the output of program L138_averagen_intr.c captured using the FFT function of a *Rigol DS1052E* oscilloscope and *Goldwave*. Compare these plots with that of Figure 3.20.

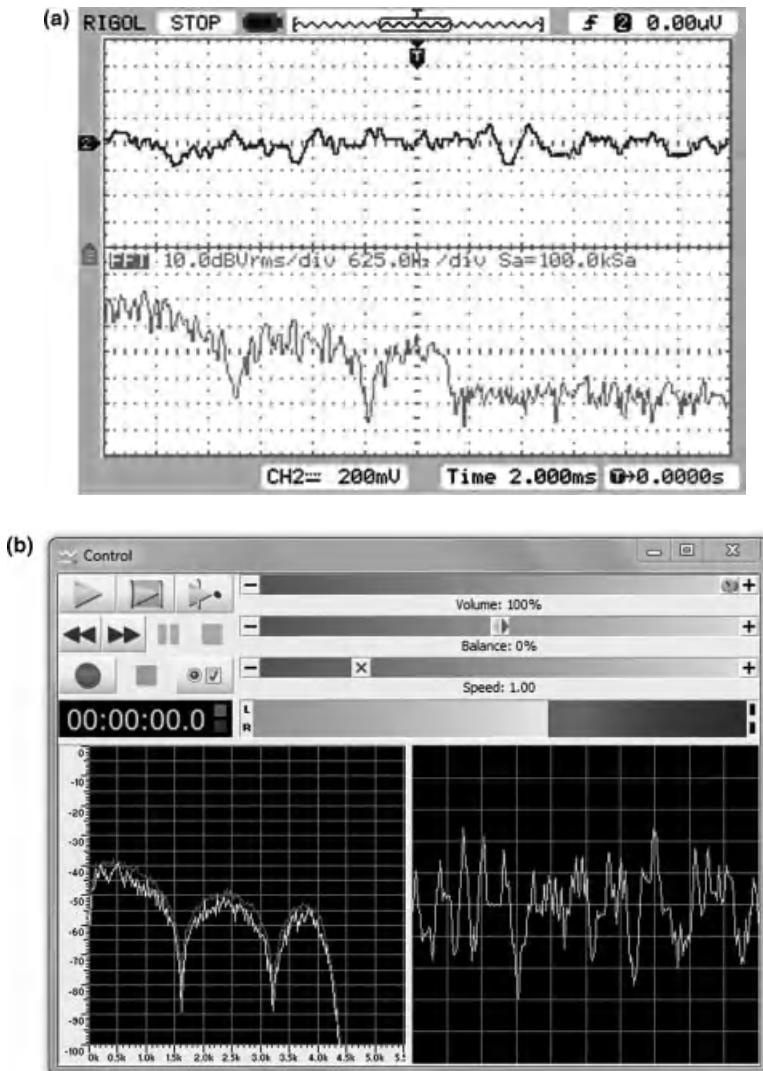


Figure 3.21 Magnitude frequency response of the five-point moving average filter demonstrated using program L138_averagen_intr.c and displayed using (a) *Rigol DS1052E* oscilloscope (lower trace) and (b) *Goldwave* (left-hand window).

EXAMPLE 3.7: Identification of Moving Average Filter Frequency Response Using a Second eXperimenter Board (L138_sysid_intr)

In Chapter 2, program L138_sysid_intr.c was used to identify the characteristics of the antialiasing and reconstruction filters of the AIC3106 codec. Here, the same program is used to identify the characteristics of the moving average filter. For this example, two eXperimenter boards connected as shown in Figure 3.22 are required. On one of the boards run program L138_average_intr.c and on the other run program L138_sysid_intr.c. After

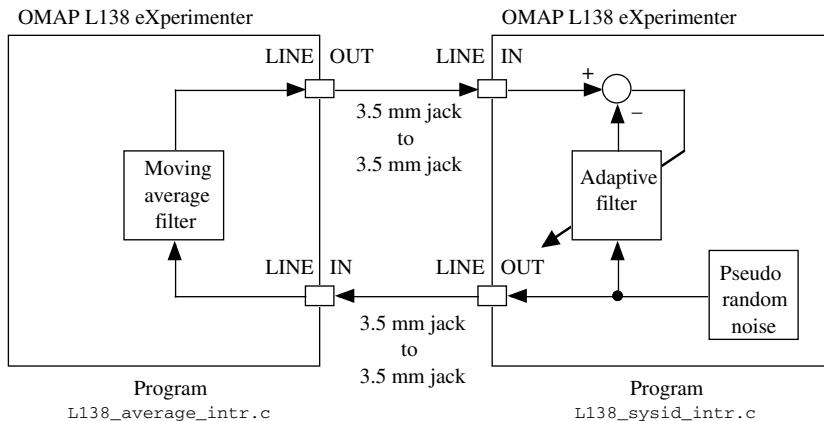


Figure 3.22 Connection diagram for the use of program L138_sysid_intr.c to identify the characteristics of a moving average filter implemented using a second eXperimenter board.

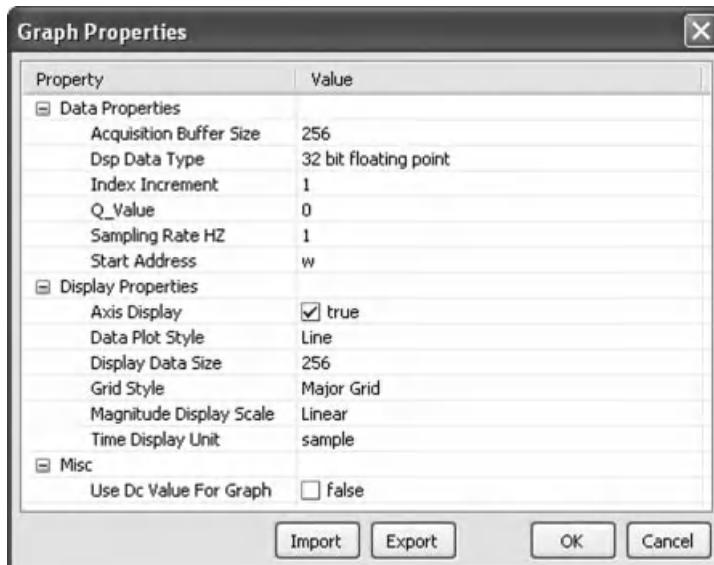


Figure 3.23 Graph Properties settings used to view the impulse response identified by program L138_sysid_intr.c.

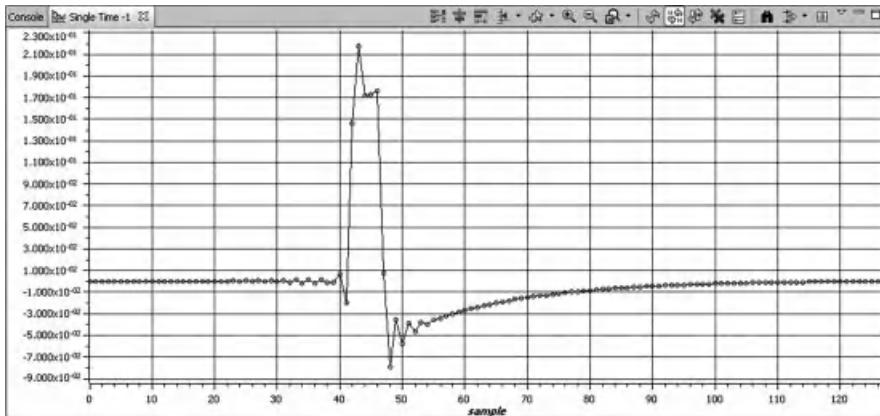


Figure 3.24 Impulse response of the five-point moving average filter identified by program L138_sysid_intr.c.

program L138_sysid_intr.c has run for a few seconds, halt the program and select *Tools > Graph > Single Time*. Using the *Graph Properties* shown in Figure 3.23, you should see something similar to that shown in Figure 3.24, that is, the impulse response identified by the adaptive filter. Using *Tools > Graph > FFT Magnitude* and the *Graph Properties* shown in Figure 3.25, the magnitude frequency response of the moving average filter, identified by the adaptive filter, may be displayed (Figure 3.26). Figure 3.27 shows the result of saving the

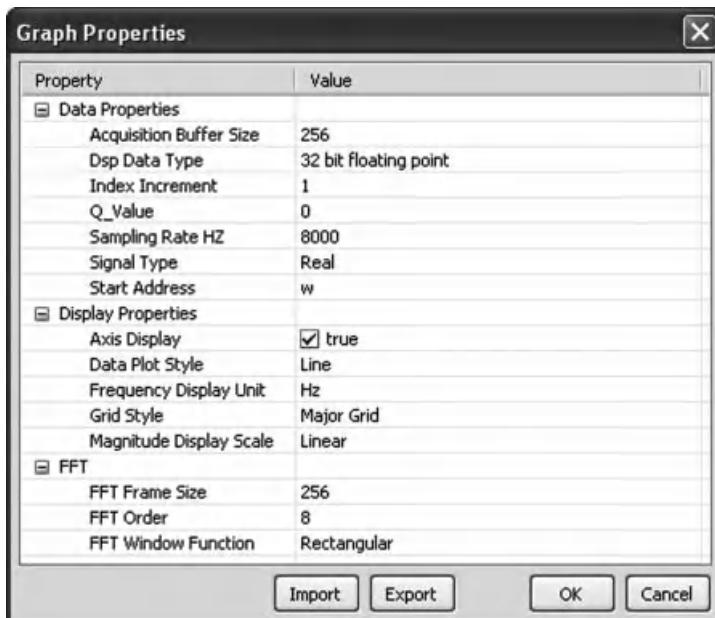


Figure 3.25 *Graph Properties* settings used to view the magnitude frequency response identified by program L138_sysid_intr.c.

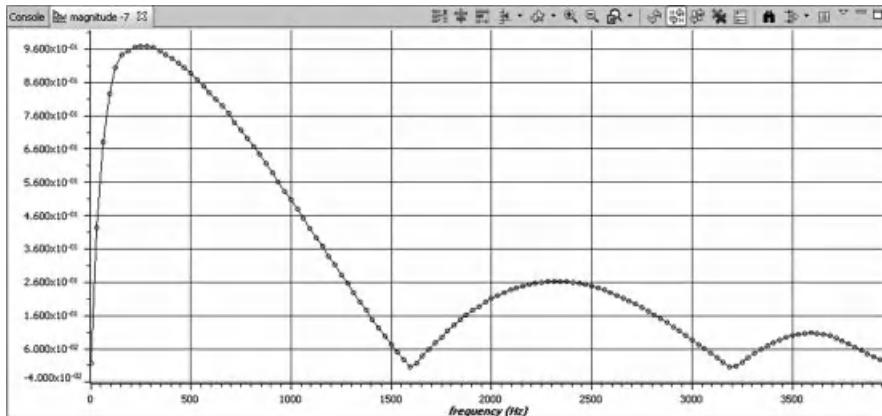


Figure 3.26 Magnitude frequency response of the five-point moving average filter identified by program L138_sysid_intr.c.

adaptive filter weights (array w) to a *TI Data Format (.dat)* file, and using MATLAB function L138_logfft() to plot its contents. Plotted on the same axes is the theoretical magnitude frequency response of the five-point moving average filter.

Program L138_sysid_intr.c gives a reasonably accurate indication of the magnitude frequency response of the filter. The discrepancies between theoretical and identified responses at frequencies greater than 3.5 kHz and at very low frequencies are due to the characteristics of the antialiasing and reconstruction filters in the AIC3106 codec and to the AC-coupling of its LINE IN and LINE OUT connections, respectively.

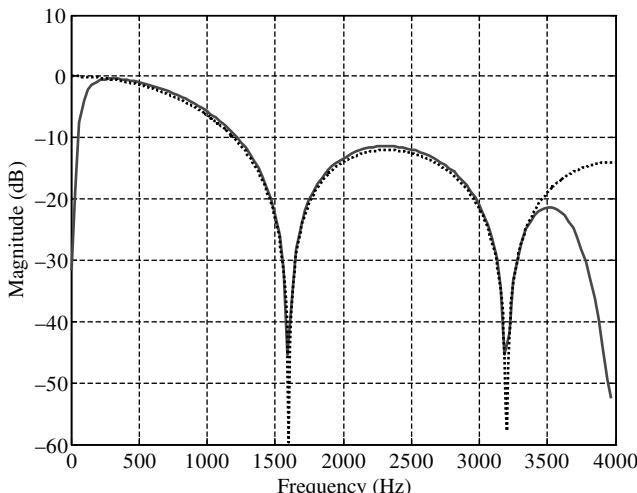


Figure 3.27 Magnitude frequency response of five-point moving average filter identified using program L138_sysid_intr.c, plotted on the same axes as the theoretical magnitude frequency response (dotted).

EXAMPLE 3.8: Identification of Moving Average Filter Frequency Response Using a Single eXperimenter Board (L138_sysid_average_intr)

Program L138_sysid_average_intr.c, listed in Figure 3.28, collapses the signal path considered in the previous example onto just one eXperimenter board, as shown in Figure 3.29. Build and run the program and plot the values of the adaptive filter coefficients w using the *Graph Properties* shown in Figure 3.25. The result should be only subtly different from that shown in Figure 3.26.

Altering the coefficients of the moving average filter

The frequency response of the moving average filter can be changed by altering the number of previous input samples that are averaged. Modify program L138_averagen_intr.c so that it implements an eleven-point moving average filter by changing the line that reads

```
#define N 5
```

to read

```
#define N 11
```

Build and run the program and verify that the frequency response of the filter has changed to that shown in Figure 3.30. Alternatively, you can make a similar change to the number of points in the moving average filter in program L138_sysid_average_intr.c.

The frequency response of the eleven-point moving average filter has the same basic form as that of the five-point moving average filter, but the notches in the frequency response occur at integer multiples of (8000/11) Hz, that is, at 727, 1455, 2182, 2909, and 3636 Hz.

The frequency response of the filter can also be changed by altering the relative values of the coefficients. Modify program L138_averagen_intr.c or L138_sysid_average_intr.c again, changing the lines that read

```
#define N 11
float h[N];
```

to read

```
#define N 5
float h[N] = {0.0833, 0.2500, 0.3333, 0.2500, 0.0833};
```

and comment out the following program statements

```
for (i=0 ; i<N ; i++)
h[i] = 1.0/N;
```

Build and run the program and observe the frequency response of the filter using *Goldwave* (in the case of program L138_averagen_intr.c) or by plotting a graph (in the case of program L138_sysid_average_intr.c).

You should find that the high-frequency components of the input signal (pseudorandom noise) have been attenuated more than before and also that the notches at 1600 and 3200 Hz

```

// L138_sysid_average_intr.c
//

#include "L138_aic3106_init.h"

#define beta 1E-12           // adaptive filter learning rate
#define WLENGTH 256          // no of adaptive filter weights
#define N 5                  // no of fixed filter coeffs

float h[N];                // filter coeffs
float w[WLENGTH+1];        // adaptive filter coeffs
float dly_adapt[WLENGTH+1]; // adaptive filter delay line
float dly_fix[N+1];        // fixed filter delay line
AIC31_data_type codec_data;

interrupt void interrupt4(void) // interrupt service routine
{
    int i;
    float fir_out, E, adaptfir_out;

    dly_fix[0] = prbs();           // input prbs to both filters
    dly_adapt[0] = dly_fix[0];

    fir_out = 0.0;                // compute fixed filter output
    for (i = N-1; i >= 0; i--)
    {
        fir_out +=(h[i]*dly_fix[i]);
        dly_fix[i+1] = dly_fix[i];
    }
    codec_data.channel[LEFT] = (uint16_t)(fir_out);
    codec_data.channel[RIGHT] = (uint16_t)(fir_out);
    output_sample(codec_data.uint);
    fir_out = (float)(input_left_sample());

    adaptfir_out = 0.0;           //compute adaptive filter output
    for (i = 0; i < WLENGTH; i++)
        adaptfir_out +=(w[i]*dly_adapt[i]);

    E = fir_out - adaptfir_out;   // compute error

    for (i = WLENGTH-1; i >= 0; i--)
    {
        w[i] = w[i]+(beta*E*dly_adapt[i]); // adapt weights
        dly_adapt[i+1] = dly_adapt[i];       // update delay line
    }
    return;
}

int main(void)
{
    int i;

    for (i=0; i <= WLENGTH; i++) w[i] = 0.0;
    for (i=0 ; i<N ; i++) h[i] = 1.0/N;
    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1) ;
}

```

Figure 3.28 Listing of program L138_sysid_average_intr.c.

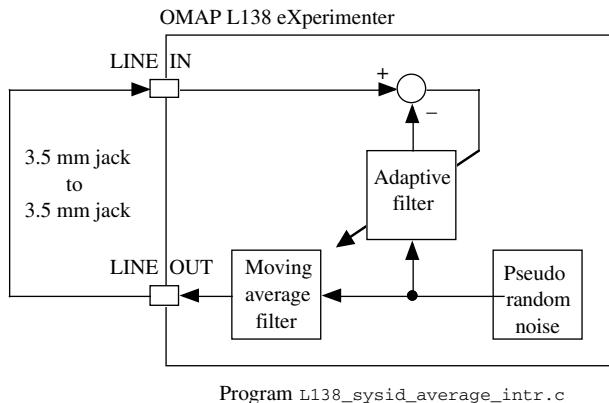


Figure 3.29 Connection diagram for program L138_sysid_average_intr.c.

have disappeared. You have effectively applied a Hanning window to the coefficients of the five-point moving average filter (Figure 3.31).

The N -point Hanning window is described by the equation

$$w(n) = 0.5 \left(1 - \cos\left(2\pi \frac{n}{N-1}\right) \right), \quad 0 \leq n < N, \quad (3.41)$$

and hence for $n = 0$ and $n = N$, $w(n) = 0$. Since there is no point in including two zero value coefficients in the FIR filtering operation, in this example the five nonzero values of a seven-point Hanning window function, rather than the five values, including two zero values, of a five-

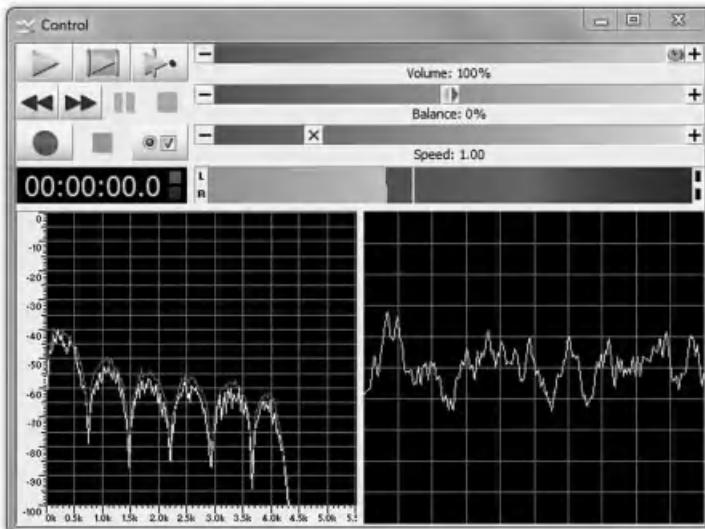


Figure 3.30 Magnitude frequency response of eleven-point moving average filter implemented using program L138_averagen_intr.c and displayed using Goldwave.

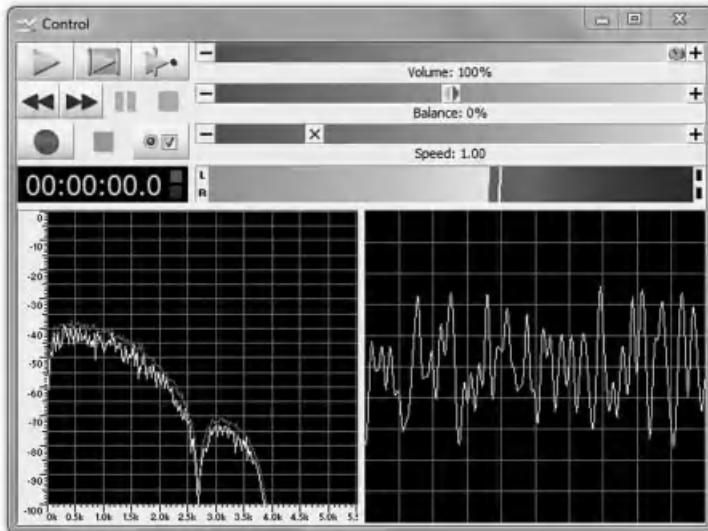


Figure 3.31 Magnitude frequency response of five-point moving average filter with Hanning window implemented using program L138_averagen_intr.c and displayed using *Goldwave*.

point Hanning window function, have been used. The wider point illustrated by this example is that a five-coefficient FIR filter can have a variety of different frequency response characteristics.

The theoretical magnitude frequency response of the filter may be found by taking the DTFT of its coefficients:

$$\begin{aligned}
 |H(\hat{\omega})| &= \left| \sum_{n=-2}^2 0.2e^{-jn\hat{\omega}} \right| \\
 &= |0.0833e^{j2\hat{\omega}} + 0.25e^{j\hat{\omega}} + 0.3333 + 0.25e^{-j\hat{\omega}} + 0.0833e^{-j2\hat{\omega}}| \\
 &= 0.3333 + 0.5 \cos(\hat{\omega}) + 0.1666 \cos(2\hat{\omega}).
 \end{aligned} \tag{3.42}$$

The measured frequency responses of the five-point moving average filter and of its windowed version may be interpreted as demonstrating the frequency domain characteristics of rectangular and Hanning windows, as discussed in Section 3.1.10. Specifically, the Hanning window has a wider main lobe and relatively smaller side lobe(s).

EXAMPLE 3.9: FIR Filter with Moving Average, Low-Pass, Band-Stop, and Band-Pass Characteristics Defined in Separate Coefficient Files (L138_fir_intr.c)

The mechanism used by program L138_fir_intr.c (Figure 3.32) to calculate each output sample is identical to that employed by program L138_average_intr.c. Function interrupt4() has exactly the same definition in both programs. However, whereas program L138_average_intr.c calculated the values of its coefficients in function main(), program L138_fir_intr.c reads the values of its coefficients from a separate file.

```

// L138_fir_intr.c
//

#include "L138_aic3106_init.h"

#include "ave5.cof"

float x[N]; // filter delay line

interrupt void interrupt4(void) // interrupt service routine
{
    short i;
    float yn = 0.0;

    x[0] = (float)(input_left_sample()); // input from ADC
    for (i=0 ; i<N ; i++) // compute filter output
        yn += h[i]*x[i];
    for (i=(N-1) ; i>0 ; i--) // shift delay line
        x[i] = x[i-1];
    output_left_sample((uint16_t)(yn)); // output to DAC
    return;
}

int main(void)
{
    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);
}

```

Figure 3.32 Listing of program L138_fir_intr.c.

Five-point moving average (ave5.cof)

Coefficient file ave5.cof is listed in Figure 3.33. Using that file, program L138_fir_intr.c implements the same five-point moving average filter implemented by program L138_average_intr.c in Example 3.5. The number of filter coefficients is specified by the value of the constant N , defined in the .cof file, and the coefficients are specified as the initial values in an N element array h of type float. Build and run program L138_fir_intr.c and verify that it implements a five-point moving average filter.

Low-pass, cutoff at 2000 hz (lp55.cof)

Edit file L138_fir_intr.c, changing the line that reads

```
#include "ave5.cof"
```

to read

```
#include "lp55.cof"
```

Build and run the program. Input a sinusoidal signal and verify that this is attenuated significantly if its frequency is greater than 2 kHz.

```
// ave5.cof
// this file was generated using function L138_fir_coeffs.m

#define N 5

float h[N] = {
2.0000E-001,2.0000E-001,2.0000E-001,2.0000E-001,2.0000E-001
};
```

Figure 3.33 Listing of coefficient file ave5.cof.***Band-stop, centered at 2700 hz (bs2700.cof)***

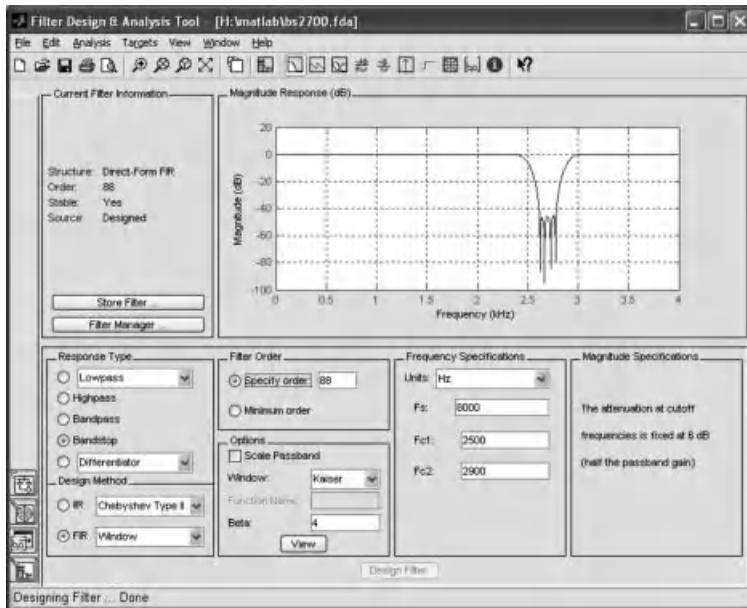
Edit file L138_fir_intr.c, changing the line that reads

```
#include "ave5.cof"
```

to read

```
#include "bs2700.cof"
```

Build and run program L138_fir_intr.c. Input a sinusoidal signal and vary the input frequency slightly below and above 2700 Hz. Verify that the output is a minimum at 2700 Hz. The values of the coefficients for this filter were calculated using MATLAB's filter design and analysis tool, fdatool, as shown in Figure 3.34.

**Figure 3.34** MATLAB fdatool window corresponding to design of FIR band-stop filter centered at 2700 Hz.

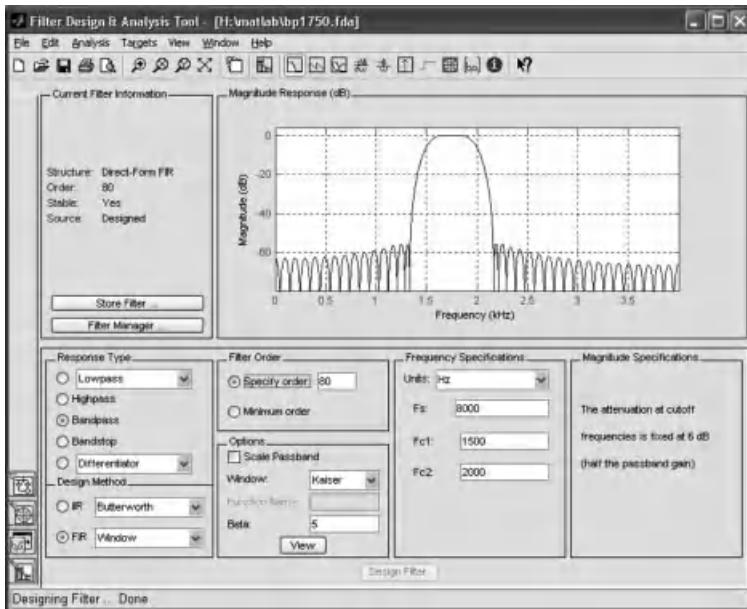


Figure 3.35 MATLAB `fdatool` window corresponding to design of FIR band-stop filter centered at 1750 Hz.

Band-pass, centered at 1750 hz (bp1750.coф)

Edit program `L138_fir_intr.c` again to include the coefficient file `bp1750.coф` in place of `bs2700.coф`. File `bp1750.coф` represents an FIR band-pass filter (81 coefficients) centered at 1750 Hz, as shown in Figure 3.35. Again, this filter was designed using MATLAB's `fdatool`. Build and run the program again and verify a band-pass filter centered at 1750 Hz.

Generating FIR filter coefficient (.coф) files using MATLAB

If the number of filter coefficients is small, a coefficient (.coф) file can be edited by hand. To be compatible with program `L138_fir_intr.c`, a coefficient file must define constant N and declare and initialize the contents of an array h, containing N floating-point values. For larger numbers of coefficients, the MATLAB function `L138_fir_coeffs()`, supplied to accompany this book as file `L138_fir_coeffs.m`, in folder `L138_support`, can be used. This function, listed in Figure 3.36, should be passed a MATLAB vector of coefficient values and will prompt the user for an output filename.

For example, the coefficient file `ave5.coф`, listed in Figure 3.33, was created by typing the following at the MATLAB command prompt:

```
>> x = [0.2, 0.2, 0.2, 0.2, 0.2];
>> L138_fir_coeffs(x)
enter filename for coefficients ave5.coф
```

Note that the coefficient filename must be entered in full, including the suffix `.coф`.

```
% L138_FIR_COEFFS.M
% MATLAB function to write FIR filter coefficients
% in format suitable for use in L138 eXperimenter programs
% L138_fir_intr.c and L138_firprn_intr.c
% written by Donald Reay
%
function L138_fir_coeffs(coeff)
coefflen=length(coeff);
fname = input('enter filename for coefficients ','s');
fid = fopen(fname,'wt');
fprintf(fid,'// %s\n',fname);
fprintf(fid,'// this file was generated using ');
fprintf(fid,'function L138_fir_coeffs.m\n');
fprintf(fid,'#define N %d\n',coefflen);
fprintf(fid,'float h[N] = { ');
for i=1:coefflen
    if six coeffs have been written to current line
    % then start new line
    if j>5
        j=0;
        fprintf(fid,'\n');
    end
    % if this is the last coefficient then simply write
    % its value to the current line
    % else write coefficient value, followed by comma
    if i==coefflen
        fprintf(fid,'%2.4E',coeff(i));
    else
        fprintf(fid,'%2.4E,',coeff(i));
        j=j+1;
    end
end
fprintf(fid,'};\n');
fclose(fid);
```

Figure 3.36 Listing of MATLAB m-file L138_fir_coeffs.m.

Alternatively, the MATLAB filter design and analysis tool `fdatool` can be used to calculate FIR filter coefficients and to export them to the MATLAB workspace. Then function `L138_fir_coeffs()` can be used to create a coefficient file compatible with program `L138_fir_intr.c`. It is recommended that the filter coefficient values passed to function `L138_fir_coeffs()` are normalized such that their sum is unity.

EXAMPLE 3.10: FIR Implementation with a Pseudorandom Noise Sequence as Input (`L138_firprn_intr`)

Program `L138_firprn_intr.c` (Figure 3.37) implements an FIR filter and uses an internally generated pseudorandom noise sequence as input. In all other respects, it is similar to program `L138_fir_intr.c`. The coefficient file `bs2700.cof` is used initially.

```

// L138_firprn_intr.c
//

#include "L138_aic3106_init.h"
#include "bs2700.cof"

float x[N];                                // filter delay line
AIC31_data_type codec_data;

interrupt void interrupt4(void) // interrupt service routine
{
    short i;
    float yn = 0.0;

    x[0] = (float)(prbs());           // input from prbs function
    for (i=0 ; i<N ; i++)           // compute filter output
        yn += h[i]*x[i];
    for (i=(N-1) ; i>0 ; i--)       // shift delay line
        x[i] = x[i-1];
    codec_data.channel[LEFT] = (uint16_t)(yn);
    codec_data.channel[RIGHT] = (uint16_t)(yn);
    output_sample(codec_data.uint); // output to L and R DAC
    return;
}

int main(void)
{
    L138_initialise_intr(FS_8000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
    while(1);
}

```

Figure 3.37 Listing of program L138_firprn_intr.c.

Build and run the program and verify that the output signal is pseudo random noise filtered by an FIR band stop filter centered at 2700 Hz.

The output signal is shown using *Goldwave* and the FFT function of a *Rigol DS1052E* oscilloscope in Figure 3.38.

Testing different FIR filters

Halt the program and edit the C source file L138_firprn_intr.c to include and test different coefficient files that represent different FIR filters. Each of the following coefficient files contains 55 coefficients (except comb14.cof).

- (1) bp55.cof: band-pass with center frequency $F_S/4$
- (2) bs55.cof: band-stop with center frequency $F_S/4$
- (3) lp55.cof: low-pass with cutoff frequency $F_S/4$
- (4) hp55.cof: high-pass with bandwidth $F_S/4$
- (5) pass2b.cof: band-pass with two passbands
- (6) pass3b.cof: band-pass with three passbands
- (7) pass4b.cof: band-pass with four passbands
- (8) comb14.cof: multiple notches (comb filter)

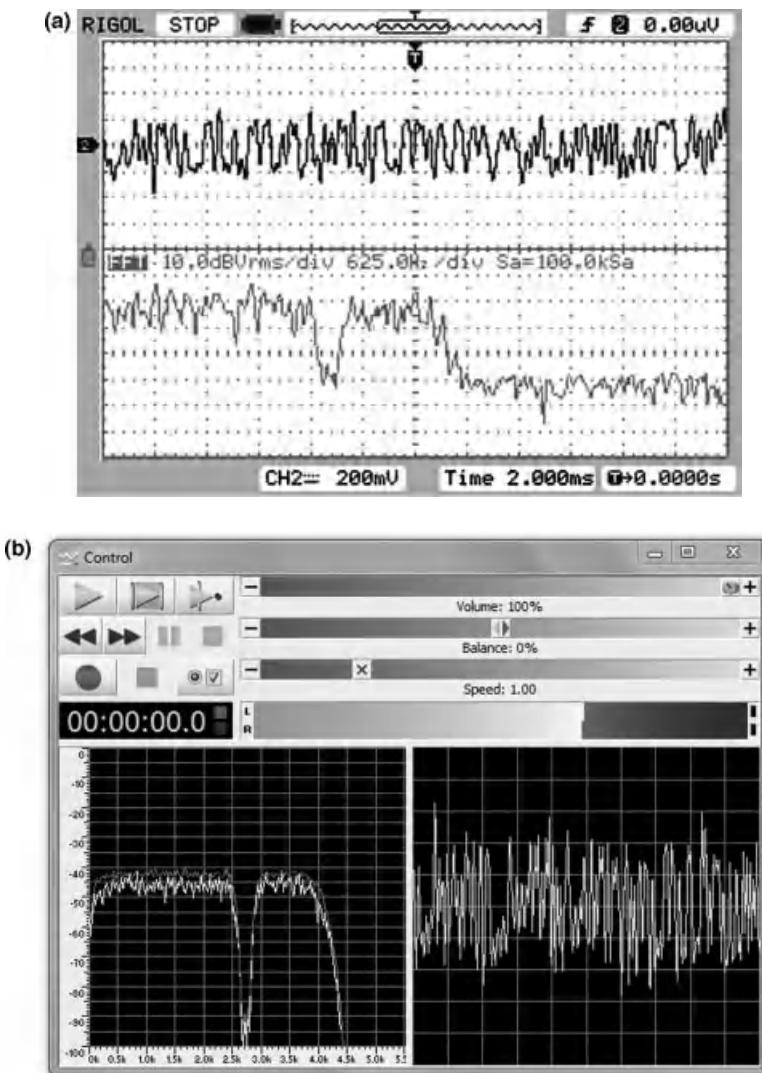


Figure 3.38 Output generated using program L138_firprn_intr.c and coefficient file bs2700.coef displayed using (a) Rigol DS1052E oscilloscope and (b) Goldwave.

Figure 3.39a shows FFT of the output of an FIR filter with two passbands, using the coefficient file pass2b.coef. Figure 3.39b shows FFT of the output of a high-pass FIR filter using the coefficient file hp55.coef. These plots were obtained using the FFT function of a *Rigol DS1052E* oscilloscope.

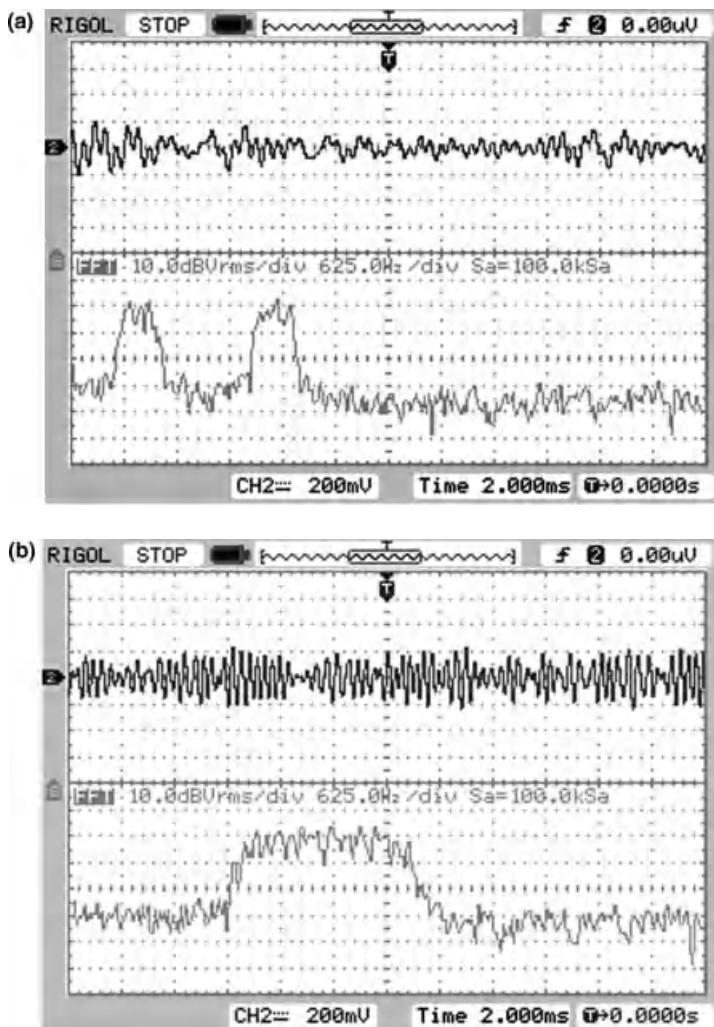


Figure 3.39 Output from program L138_firprn_intr.c using coefficient files (a) pass2b.coef and (b) hp55.coef.

EXAMPLE 3.11: FIR Filter with Internally Generated Pseudorandom Noise as Input and Output Stored in Memory (L138_firprnbuf_intr)

This example extends the previous one by storing the 1024 most recent output samples in memory. Program L138_firprnbuf_intr.c is listed in Figure 3.40. The coefficient file bp41.coef represents a 41-coefficient FIR band-pass filter centered at 1000 Hz.

```

// L138_firprnbuf_intr.c
//

#include "L138_aic3106_init.h"
#include "bp41.cof"

#define YNBUFSIZE 1024

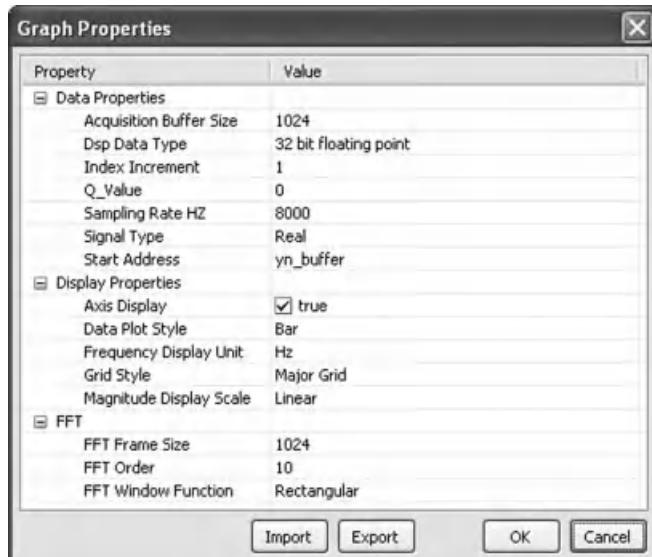
float x[N];           // filter delay line
float yn_buffer[YNBUFSIZE]; // output buffer
short ynbuflength = 0;

interrupt void interrupt4(void) // interrupt service routine
{
    short i;
    float yn = 0.0;

    x[0] = (float)(prbs()); // input from prbs function
    for (i=0 ; i<N ; i++) // compute filter output
        yn += h[i]*x[i];
    for (i=(N-1) ; i>0 ; i--) // shift delay line
        x[i] = x[i-1];
    yn_buffer[ynbuflength++] = yn; // buffer output samples
    if(ynbuflength >= YNBUFSIZE) ynbuflength = 0;
    output_left_sample((uint16_t)(yn)); // output to L DAC
    return;
}

int main(void)
{
    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);
}

```

Figure 3.40 Listing of program L138_firprnbuf_intr.c.**Figure 3.41** Graph Properties settings for use with program L138_firprnbuf_intr.c.

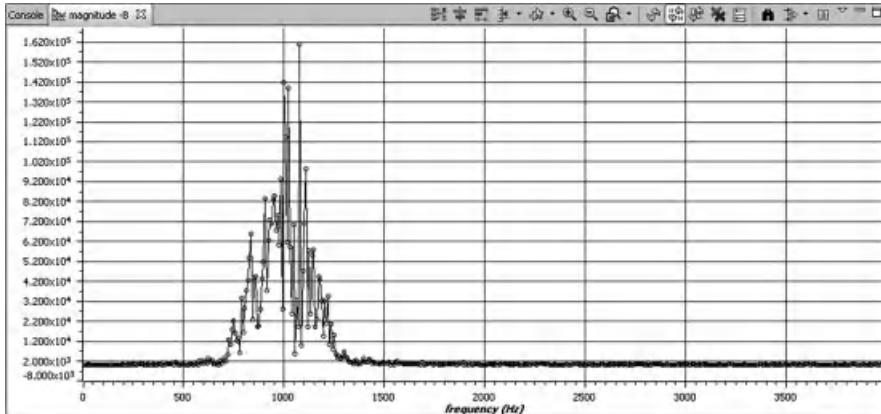


Figure 3.42 Magnitude of the FFT of the output from program L138_firprnbuf_intr.c.

Build and run the program. Verify that the output signal is bandlimited noise. Then halt the program, select *Tools > Graph > FFT Magnitude*, and set the *Graph Properties* as shown in Figure 3.41 in order to look at the frequency content of the 1024 stored output samples (Figure 3.42).

Figure 3.43 shows the FFT magnitude of the filter coefficients for comparison.

An alternative version of program L138_firprnbuf_intr.c has been supplied that allows the user to switch the signal written to the AIC3106 ADC between filtered and unfiltered PRBS so as to emphasize the action of the filter. Figure 3.44 shows the output of program L138_firprnbuf_DIP_intr.c with DIP switch #1 OFF and ON.

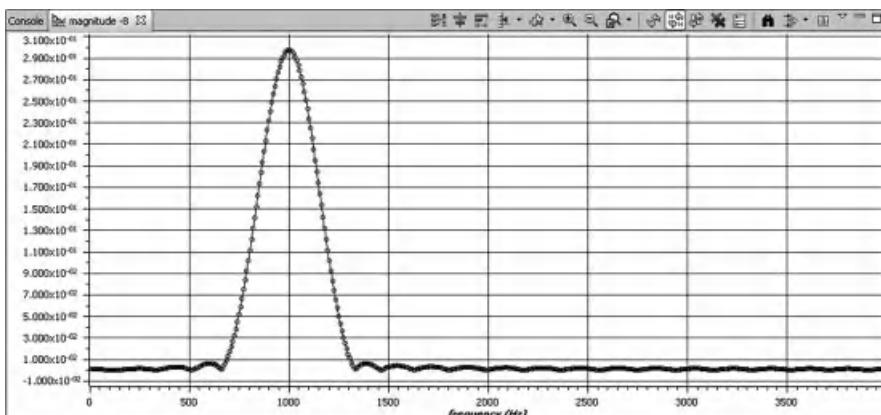


Figure 3.43 Magnitude of the FFT of the filter coefficients used by program L138_firprnbuf_intr.c.

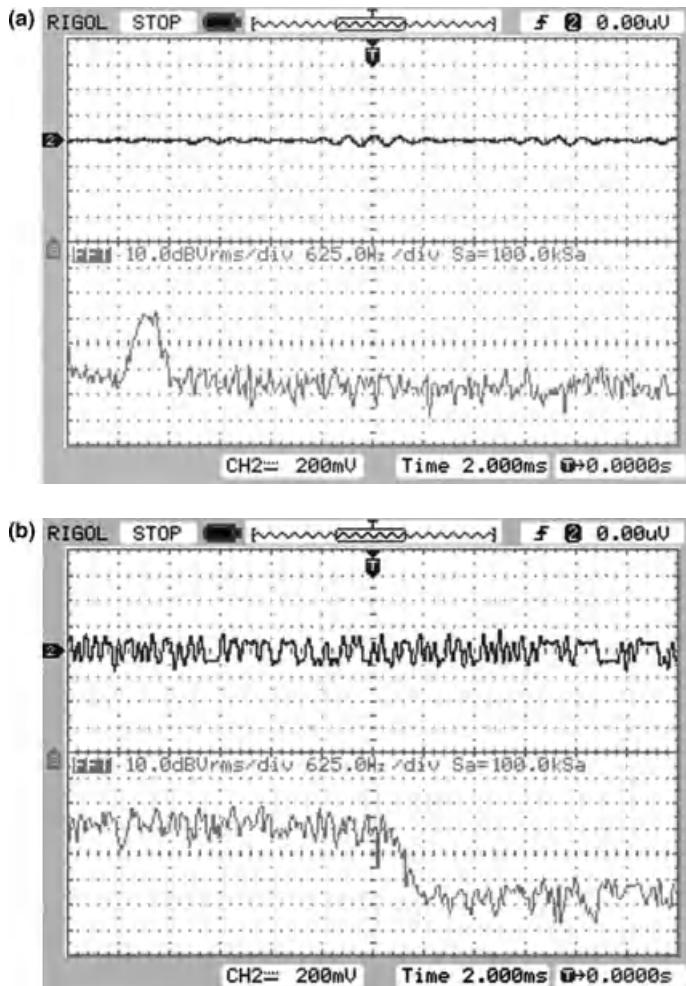


Figure 3.44 Output from program L138_firprnbuf_DIP_intr.c. (a) Filtered noise (DIP switch #1 OFF). (b) Unfiltered noise (DIP switch #1 ON).

EXAMPLE 3.12: Effects on Voice or Music Using Three FIR Low-Pass Filters
(L138_fir3lp_intr)

Figure 3.45 shows a listing of the program L138_fir3lp_intr.c which implements three FIR low-pass filters with cutoff frequencies at 600, 1500, and 3000Hz, respectively. Filter coefficients designed using MATLAB are read from file L138_fir3lp_coeffs.h and, during initialization, copied into a single, two-dimensional array h. While the program is running, variable LP_number selects the desired low-pass filter to be implemented. For

```

// L138_fir3lp_intr.c
//

#include "L138_aic3106_init.h"
#include "evmomapl138_dip.h"
#include "L138_fir3lp_coeffs.h"

uint32_t FIR_number = 0;
float dly[N];                                // filter delay line
float h[3][N];                                // filter coefficients

interrupt void interrupt4(void) // interrupt service routine
{
    int i;
    float yn = 0.0;

    dly[0] = (float)(input_left_sample()); // input from ADC
    for (i = 0; i < N; i++)                // compute output
        yn +=(h[FIR_number][i]*dly[i]);
    for (i = N-1; i > 0; i--)              // shift delay line
        dly[i] = dly[i-1];
    output_left_sample((uint16_t)(yn));     // output to DAC
    return;
}

int main()
{
    uint32_t DIP_value, rtn;
    int i;

    for (i=0; i<N; i++)
    {
        dly[i] = 0.0;
        h[0][i] = hlp600[i];
        h[1][i] = hlp1500[i];
        h[2][i] = hlp3000[i];
    }
    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    rtn = DIP_init();                      // initialise user DIP switches
    if (rtn != ERR_NO_ERROR)
    {
        printf("error initializing dip switches!\r\n");
        return (rtn);
    }
    while(1)
    {
        rtn = DIP_getAll(&DIP_value); // read user DIP switches
        FIR_number = (DIP_value % 3); // switch filter coeffs
    }
}

```

Figure 3.45 Listing of program L138_fir3lp_intr.c.

example, if LP_number is set to 0, $h[0][i]$ is set equal to $hlp600[i]$, that is, the set of coefficients representing a low-pass filter with a cutoff frequency of 600 Hz. The value of LP_number can be set to 0, 1, or 2 to implement 600, 1500, or 3000 Hz low-pass filter, respectively, using DIP switches #1 and #2 while the program is running.

As supplied, the program sets the AIC3106 ADC gain to 0 dB and is suited to input signals from a sound card or a signal generator. In order to test the effect of the filters using a dynamic microphone as an input device, change the line that reads

```
L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
```

to read

```
L138_initialise_intr(FS_8000_HZ,ADC_GAIN_24DB,DAC_ATTEN_0DB);
```

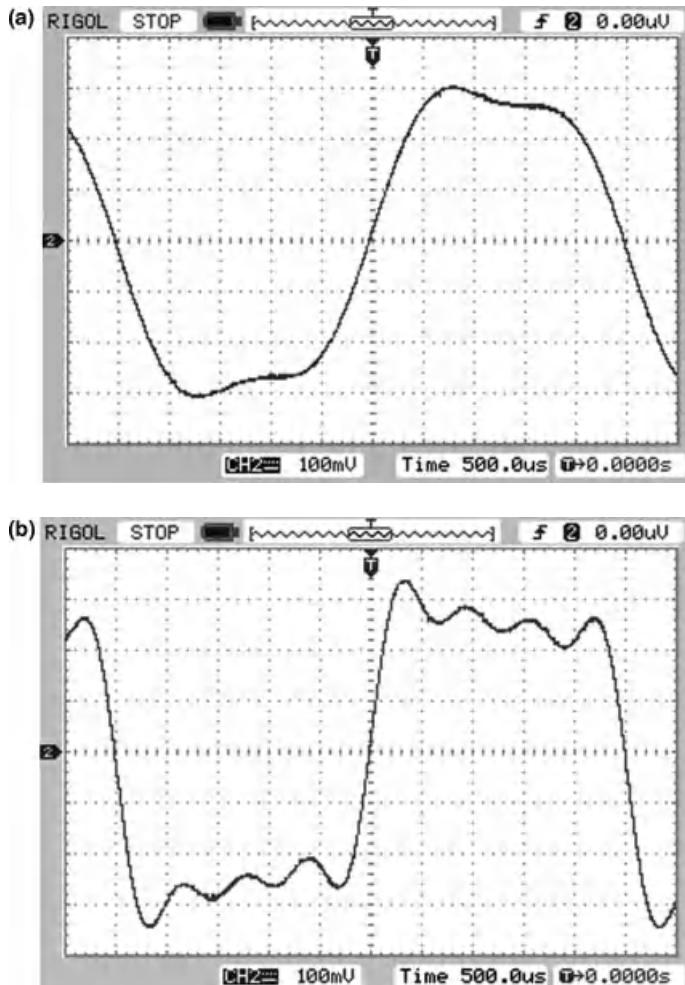


Figure 3.46 A 200 Hz square wave passed through three different low-pass filters implemented using program L138_fir3lp_intr.c.

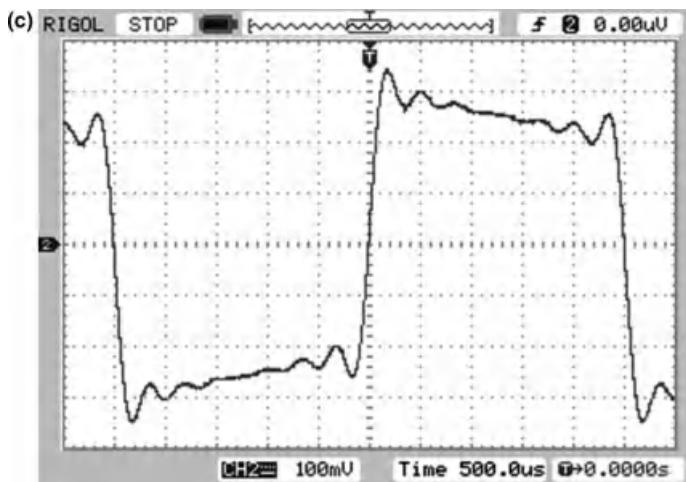


Figure 3.46 (Continued)

The effect of the filters is particularly striking if applied to musical input. Alternatively, the effects of the filters can be illustrated using an oscilloscope and a signal generator. Figure 3.46 shows a 200 Hz square wave that has been passed through the three low-pass filters.

EXAMPLE 3.13: Implementation of Four Different Filters: Low-Pass, High-Pass, Band-Pass, and Band-Stop (`L138_fir4types_intr`)

This example is very similar to the previous one but illustrates the effects of low-pass, high-pass, band-pass, and band-stop FIR filters. The filter type may be changed while program `L138_fir4types_intr.c` is running using DIP switches #1 and #2. The following four 81-coefficient filters were designed using MATLAB:

- (1) Low-pass filter, with bandwidth of 1500 Hz
- (2) High-pass filter, with bandwidth of 2200 Hz
- (3) Band-pass filter, with center frequency at 1750 Hz
- (4) Band-stop filter, with center frequency at 790 Hz

As in the previous example, the effects of the four different filters on musical input are particularly striking.

Figure 3.47 shows the magnitude frequency response of the FIR band-stop filter centered at 790 Hz, tested using the file `stereonoise.wav` played through a PC sound card as input.

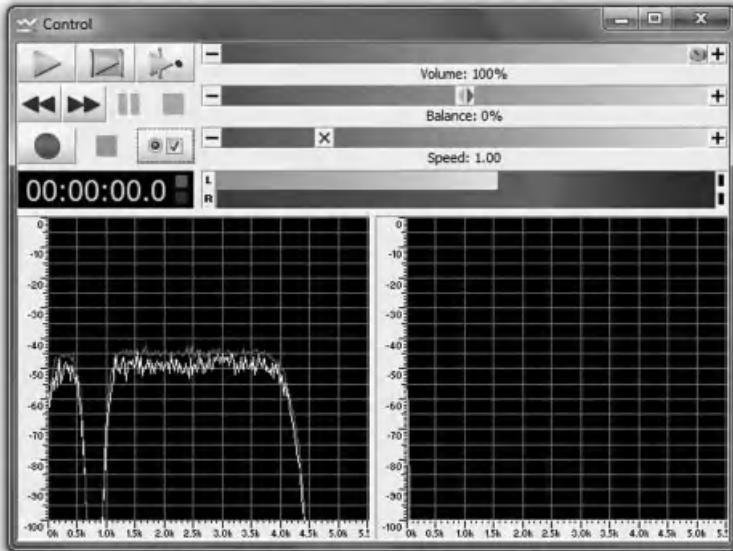


Figure 3.47 Output generated using program L138_fir4types_intr.c using file stereonoise.wav as input and displayed using *Goldwave*.

EXAMPLE 3.14: Two Notch Filters to Recover a Corrupted Speech Recording
(L138_notch2_intr)

This example illustrates the use of two notch (band-stop) FIR filters in series to recover a speech recording corrupted by the addition of two sinusoidal signals at frequencies of 900 and 2700 Hz. Program L138_notch2_intr.c is listed in Figure 3.48. Header file notch2_coeffs.h contains the coefficients for two FIR notch (band-stop) filters in arrays h900 and h2700. The output of the first notch filter, centered at 900 Hz, is used as the input to the second notch filter, centered at 2700 Hz.

Build and run the program. The file corrupt.wav, stored in folder L138_notch2_intr contains a recording of speech corrupted by the addition of 900 and 2700 Hz sinusoidal tones. Listen to this file using *Goldwave*, *Windows Media Player*, or similar. Then connect the PC sound card output to the LINE IN socket on the eXperimentator and listen to the filtered test signal on LINE OUT. DIP switch #1 can be used to select either the output of the first or the second of the notch filters.

Compare the results of this example with those obtained in Example 3.5 in which a notch in the magnitude frequency response of a moving average filter was exploited in order to filter out an unwanted sinusoidal tone. In this case, the filtered speech sounds brighter because the notch filters used here do not have an overall low pass characteristic.

```

// L138_notch2_intr.c
//

#include "L138_aic3106_init.h"
#include "evmomapl138_dip.h"
#include "notch2_coeffs.h"

float dly1[N]={0};           // filter1 delay line
float dly2[N]={0};           // filter2 delay line
volatile short out_type = 0;

interrupt void interrupt4() // interrupt service routine
{
    short i;
    float ylout, y2out;

    dly1[0] = (float)(input_left_sample());
    ylout = 0.0;           // compute filter 1 output
    for (i = 0; i< N; i++)
        ylout += h900[i]*dly1[i];
    dly2[0]=(ylout);
    y2out = 0.0;           // compute filter 2 output
    for (i = 0; i< N; i++)
        y2out += h2700[i]*dly2[i];
    for (i = N-1; i > 0; i--) // shift delay lines
    {
        dly1[i] = dly1[i-1];
        dly2[i] = dly2[i-1];
    }
    if (out_type==0)         // select output
        output_left_sample((short)(y2out));
    if (out_type==1)
        output_left_sample((short)(ylout));
    return;
}

int main()
{
    uint32_t rtn;
    uint8_t DIP_value;

    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    rtn = DIP_init();
    if (rtn != ERR_NO_ERROR)
    {
        printf("error initializing dip switches!\r\n");
        return (rtn);
    }
    while(1)
    {
        rtn = DIP_get(0, &DIP_value);
        if(DIP_value == 0) out_type = 0; else out_type = 1;
    }
}

```

Figure 3.48 Listing of program, L138_notch2_intr.c.

**EXAMPLE 3.15: Voice Scrambling Using Filtering and Modulation
(L138_scrambler_intr)**

This example illustrates a voice scrambling and descrambling scheme. The approach makes use of basic algorithms for filtering and modulation. Modulation was introduced in the AM example in Chapter 2. With voice as input, the resulting output is scrambled voice. The original descrambled voice is recovered when the output of the eXperimenter is used as the input to a second eXperimenter running the same program.

The scrambling method used is commonly referred to as frequency inversion. It takes an audio range, in this case 300 Hz–3 kHz, and “folds” it about a 3.3 kHz carrier signal. The frequency inversion is achieved by multiplying (modulating) the audio input by a carrier signal, causing a shift in the frequency spectrum with upper and lower sidebands. In the lower sideband that represents the audible speech range, the low tones are high tones, and vice versa.

Figure 3.49 shows a block diagram of the scrambling scheme. At point A, we have an input signal, bandlimited to 3 kHz. At point B, we have a double-sideband signal with suppressed carrier. At point C, the upper sideband and the section of the lower sideband between 3 and 3.3 kHz are filtered out. The scheme is attractive because of its simplicity. Only simple DSP algorithms, namely, filtering, sine wave generation, and amplitude modulation are required for its implementation.

Figure 3.50 shows a listing of program L138_scrambler_intr.c which operates at a sampling rate f_s of 16 kHz. The input signal is first low-pass filtered using an FIR filter with 65 coefficients, stored in array h, and defined in the file lp3k64.coef. The filtering algorithm used is identical to that used in program L138_fir_intr.c. The filter delay line is implemented using array x1 and the output is assigned to variable yn1. The filter output (at point A in Figure 3.49) is multiplied (modulated) by a 3.3 kHz sinusoid stored as 160 samples (exactly 33 cycles) in array sine160 (read from file sine160.h). Finally, the modulated signal (at point B) is low-pass filtered again, using the same set of filter coefficients h (lp3k64.coef), but a different filter delay line implemented using array x2 and the output variable yn2. The output is a scrambled signal (at point C). Using this scrambled signal as the input to a second eXperimenter running the same algorithm, the original descrambled input is recovered as the output of the second eXperimenter.

Build and run the program. First, test the program using a 2 kHz sine wave as input. The resulting output is a lower sideband signal at 1.3 kHz. The upper sideband signal at 5.3 kHz is filtered out by the second low-pass filter. By varying the frequency of the sinusoidal input, you should be able to verify that input frequencies in the range of 300–3000 Hz appear as output frequencies in the inverted range of 3000–300 Hz.

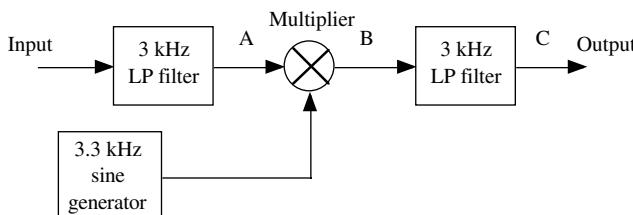


Figure 3.49 Block diagram representation of scrambler implemented using program L138_scrambler_intr.c.

A second eXperimenter running the same program can be used to recover the original signal (simulating the receiving end). Use the output of the first board as the input to the second.

In order to test the scrambler and descrambler using speech from a microphone as the input, change the ADC gain setting on the first board (the board that will use a dynamic microphone as an input device) from 0 to 24 dB by altering the statement that reads

```
L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
```

to read

```
L138_initialise_intr(FS_8000_HZ,ADC_GAIN_24DB,DAC_ATTEN_0DB);
```

Connect LINE OUT on the first eXperimenter (scrambler) to LINE IN on the second eXperimenter (descrambler).

Interception of the speech signal could be made more difficult by changing the modulation frequency dynamically and by including (or omitting) the carrier frequency according to a predefined sequence.

```
// L138_scrambler_intr.c
//

#include "L138_aic3106_init.h"

#include "sine160.h"
#include "lp3k64.cof"           // filter coefficient file
float yn1, yn2;                // filter outputs
float x1[N],x2[N];            // filter delay lines
int index = 0;

interrupt void interrupt4(void) // interrupt service routine
{
    short i;

    x1[0] = (float)(input_left_sample()); // input from ADC
    yn1 = 0.0;                         // compute filter 1
    for (i=0 ; i<N ; i++) yn1 += h[i]*x1[i];
    for (i=(N-1) ; i>0 ; i--) x1[i] = x1[i-1];
    yn1 *= sine160[index++];           // mix with 3300 Hz
    if (index >= NSINE) index = 0;
    x2[0] = yn1;                      //get new input into delay line
    yn2 = 0.0;                         // compute filter 2
    for (i=0 ; i<N ; i++) yn2 += h[i]*x2[i];
    for (i=(N-1) ; i>0 ; i--) x2[i] = x2[i-1];
    output_left_sample((short)(yn2));   //output to DAC
    return;
}

void main()
{
    L138_initialise_intr(FS_16000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);
}
```

Figure 3.50 Listing of program L138_scrambler_intr.c.

EXAMPLE 3.16: FIR Filter Implemented Using DMA-Based I/O (L138_fir_edma)

The C programs L138_fir_edma.c (Figure 3.51) and L138_firprn_edma.c have similar functionality to programs L138_fir_intr.c and L138_firprn_intr.c, but

```
// L138_fir_edma.c
//

#include "L138_aic3106_init.h"
#include "bp1750.cof"

extern int16_t *pingIN, *pingOUT, *pongIN, *pongOUT;
volatile int buffer_full = 0;
int procBuffer;

float x[N];
float y[(BUFCOUNT/2)];

interrupt void interrupt4(void) // interrupt service routine
{
    switch(EDMA_3CC_IPR)
    {
        case 1:                      // TCC = 0
            procBuffer = PING;       // process ping
            EDMA_3CC_ICR = 0x0001;   // clear EDMA3 IPR bit TCC
            break;
        case 2:                      // TCC = 1
            procBuffer = PONG;      // process pong
            EDMA_3CC_ICR = 0x0002;   // clear EDMA3 IPR bit TCC
            break;
        default:                     // may have missed an interrupt
            EDMA_3CC_ICR = 0x0003;   // clear EDMA3 IPR bits 0 and 1
            break;
    }
    EVTCLR0 = 0x000000100;
    buffer_full = 1;              // flag EDMA3 transfer
    return;
}

void process_buffer(void)
{
    int16_t *inBuf, *outBuf;      // pointers to process buffers
    int16_t left_sample, right_sample;
    int i,j;

    if (procBuffer == PING)       // use ping or pong buffers
    {
        inBuf = pingIN;
        outBuf = pingOUT;
    }
    if (procBuffer == PONG)
    {
        inBuf = pongIN;
        outBuf = pongOUT;
    }
}
```

Figure 3.51 Listing of program L138_fir_edma.c.

```

for (i = 0; i < (BUFCOUNT/2) ; i++)
{
    left_sample = *inBuf++;
    right_sample = *inBuf++;

    y[i] = 0.0;
    x[0] = (float)(left_sample); // input from ADC
    for (j=0 ; j<N ; j++)          // compute filter output
        y[i] += h[j]*x[j];

    for (j=(N-1) ; j>0 ; j--)      // shift delay line
        x[j] = x[j-1];
}
for (i = 0; i < (BUFCOUNT/2) ; i++)
{
    *outBuf++ = (int16_t)(y[i]); // output to L and R DAC
    *outBuf++ = (int16_t)(y[i]);
}
buffer_full = 0;                  // flag buffer processed
return;
}

int main(void)
{
    L138_initialise_edma(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1)
    {
        while (!buffer_full);
        process_buffer();
    }
}

```

Figure 3.51 (Continued)

use DMA-based as opposed to interrupt-based I/O, as illustrated previously in Example 2.6. As supplied, they implement band-pass filters centered on 1750Hz. The filter characteristics implemented by the programs can be changed by including different .cof files.

**EXAMPLE 3.17: FIR Filter Implemented Using a DSPLIB Function
(L138_fir_dsplib_edma)**

Function `DSPF_sp_fir_gen()` supplied as part of the c674x DSPLIB library is an optimized floating-point implementation of an FIR filter. Because it acts on a block of input samples, computing a corresponding block of output samples, it is suited to DMA-based I/O. Apart from using the library function, program `L138_fir_dsplib_edma.c`, listed in Figure 3.52, is similar to program `L138_fir_edma.c`.

For this program to work, due to the nature of the library function `DSPF_sp_fir_gen()`, both the number of samples in each DMA block (`BUFCOUNT/2`, set in file `L138_aic3106_init.h`) and the number of filter coefficients (`N`, set in file `lp55zp56.cof`) must be divisible by 4 and greater than 0 [2]. Before calling function `DSPF_sp_fir_gen()` `BUFCOUNT/2`, new input samples are written into array `x` starting at `x[N-1]`. The first `N-1`

```

// L138_fir_dsplib_edma.c
//

#include "L138_aic3106_init.h"
#include "dsplib674x.h"
#include "lp55zp56.cof"

extern int16_t *pingIN, *pingOUT, *pongIN, *pongOUT;
volatile int buffer_full = 0;
int procBuffer;

float x[(BUFCOUNT/2)+N-1];
float y[(BUFCOUNT/2)];

interrupt void interrupt4(void) // interrupt service routine
{
    switch(EDMA_3CC_IPR)
    {
        case 1:                      // TCC = 0
            procBuffer = PING;       // process ping
            EDMA_3CC_ICR = 0x0001;   // clear EDMA3 IPR bit TCC
            break;
        case 2:                      // TCC = 1
            procBuffer = PONG;      // process pong
            EDMA_3CC_ICR = 0x0002;   // clear EDMA3 IPR bit TCC
            break;
        default:                     // may have missed an interrupt
            EDMA_3CC_ICR = 0x0003;   // clear EDMA3 IPR bits 0 and 1
            break;
    }
    EVTCLR0 = 0x00000100;
    buffer_full = 1;              // flag EDMA3 transfer
    return;
}

void process_buffer(void)
{
    int16_t *inBuf, *outBuf;      // pointers to process buffers
    int16_t left_sample, right_sample;
    int i;

    if (procBuffer == PING)      // use ping or pong buffers
    {
        inBuf = pingIN;
        outBuf = pingOUT;
    }
    if (procBuffer == PONG)
    {
        inBuf = pongIN;
        outBuf = pongOUT;
    }

    for (i = 0; i < (BUFCOUNT/2) ; i)
    {
        left_sample = *inBuf;
        right_sample = *inBuf;
        x[N+i-1] = left_sample;
    }

    DSPF_sp_fir_gen(x,h,y,N,(BUFCOUNT/2));
}

```

Figure 3.52 Listing of program L138_fir_dsplib_edma.c.

```

for (i=0 ; i<N ; i) x[i] = x[(BUFCOUNT/2)];
for (i = 0; i < (BUFCOUNT/2) ; i)
{
    *outBuf = (int16_t)(y[i]); // output to L and R DAC
    *outBuf = (int16_t)(y[i]);
}
buffer_full = 0; // flag buffer processed
return;
}

int main(void)
{
    L138_initialise_edma(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1)
    {
        while (!buffer_full);
        process_buffer();
    }
}

```

Figure 3.52 (Continued)

elements of the array should contain the previous N-1 input samples. When function `DSPF_sp_fir_gen()` is called, a block of `BUFCOUNT/2` output samples is computed and returned in array `y`. After calling function `DSPF_sp_fir_gen()`, the N-1 most recent input samples starting at element `BUFCOUNT/2` in array `x` are shifted `BUFCOUNT/2` positions into elements `x[0]` to `x[N-2]`. In this way, and due to the nature of the function `DSPF_sp_fir_gen()`, the overlap-save method of block processing convolution is implemented by the program.

**EXAMPLE 3.18: FIR Implementation Using C Calling an ASM Function
(`L138_FIRcasm_intr.c`)**

This example is included in order to demonstrate that the OMAP-L138 processor is capable of running programs written in assembly language for older members of the Texas Instruments TMS320C6x family of digital signal processors. The following programs were published originally in Ref. [3]. The C program `L138_FIRcasm_intr.c` (Figure 3.53) calls the assembly language function `_fircasmfunc()` defined in file `FIRCASMfunc.asm` (Figure 3.54) and which implements an FIR filter.

Build and run the program and use a signal generator and oscilloscope to verify that it implements a 1kHz FIR band-pass filter. Two buffers are used by program `L138_FIRcasm_intr.c`. Array `dly` is used to store N previous input samples and array `h` stores N filter coefficients. The value of constant N is defined in the filter coefficient (.cof) file. On each interrupt, a new input sample is acquired and stored at the end (higher memory address) of the buffer `dly`. The delay samples are stored in memory starting with the oldest sample stored at the lowest memory address. The newest sample is at the end of the buffer.

156 Chapter 3 Finite Impulse Response Filters

```
// L138_FIRcasm_intr.c
//

#include "L138_aic3106_init.h"
#include "bp41.cof"

int yn = 0;                                // filter output
short dly[N];                             // filter delay line

interrupt void interrupt4(void) // interrupt service routine
{
    dly[N-1] = input_left_sample();          // input from ADC
    yn = fircasmfunc(dly,h,N);             // call ASM function
    output_left_sample((short)(yn>>15)); // output to DAC
    return;
}

void main()
{
    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);
}
```

Figure 3.53 Listing of program L138_FIRcasm_intr.c.

;**FIRCASMfunc.asm** ASM function called from C to implement FIR
;A4 = Samples address, B4 = coeff address, A6 = filter order
;Delays organized as:x(n-(N-1))...x(n);coeff as h[0]...h[N-1]

```
.def _fircasmfunc
_fircasmfunc:                         ;ASM function called from C
    MV A6,A1           ;setup loop count
    MPY A6,2,A6         ;since dly buffer data as byte
    ZERO A8            ;init A8 for accumulation
    ADD A6,B4,B4        ;since coeff buffer data as byte
    SUB B4,1,B4        ;B4=bottom coeff array h[N-1]
loop:
    LDH *A4++,A2        ;A2=x[n-(N-1)+i] i=0,1,...,N-1
    LDH *B4--,B2        ;B2=h[N-1-i] i=0,1,...,N-1
    NOP 4              ;using data move "up"
    MPY A2,B2,A6        ;A6=x[n-(N-1)+i]*h[N-1-i]
    NOP
    ADD A6,A8,A8        ;accumulate in A8
    LDH *A4,A7          ;A7=x[(n-(N-1)+i+1]update delays
    NOP 4              ;using data move "up"
    STH A7,*-A4[1]      ;-->x[(n-(N-1)+i] update sample
    SUB A1,1,A1          ;decrement loop count
    [A1] B loop          ;branch to loop if count # 0
    NOP 5
    MV A8,A4            ;result returned in A4
    B B3                ;return addr to calling routine
    NOP 4
```

Figure 3.54 Listing of program FIRCASMfunc.asm.

The coefficients are arranged in memory with $h(0)$ at the beginning of the coefficient buffer and $h(N - 1)$ at the end.

The addresses of the delay sample buffer, the filter coefficient buffer, and the size of each buffer are passed to the ASM function through registers A4, B4, and A6, respectively. The size of each buffer through register A6 is doubled since data in each memory location are stored as bytes. The pointers A4 and B4 are incremented or decremented every 2 bytes (two memory locations). The end address of the coefficients' buffer is in B4, which is at $2N - 1$.

The two 16-bit load (LDH) instructions load the content in memory pointed to (whose address is specified) by A4 and the content in memory at the address specified by B4. This loads the oldest sample and last coefficient, $x(n - (N - 1))$ and $h(N - 1)$, respectively. A4 is then postincremented to point at $x(n - (N - 2))$, and B4 is postdecremented to point at $h(N - 2)$. After the first accumulation, the oldest sample is updated. The content in memory at the address specified by A4 is loaded into A7, and then stored at the preceding memory location. This is because A4 is postdecremented without modification to point at the memory location containing the oldest sample. As a result, the oldest sample $x(n - (N - 1))$ is replaced (updated) by $x(n - (N - 2))$. The updating of the delay samples is for the next unit of time. As the output at time n is being calculated, the samples are updated or “primed” for time $(n + 1)$. At time n , the filter's output is

$$y(n) = h(N - 1)x(n - (N - 1)) + h(N - 2)x(n - (N - 2)) + \dots + h(0)x(n).$$

The loop is processed N times. For each time n , $n + 1$, $n + 2$, ..., an output value is calculated, with each sample updated for the next unit of time. The newest sample is also updated in this process, with an invalid data value residing at the memory location beyond the end of the buffer. But this is remedied since for each unit of time, the newest sample, acquired through the ADC of the codec, overwrites it.

Accumulation is in A8 and the result, for each unit of time, is moved to A4 to be returned to the calling function. The address of the calling function is in B3.

EXAMPLE 3.19: FIR Implementation Using C Calling a Faster ASM Function (FIRcasmfast**)**

The same C calling program L138_FIRcasm_intr.c is used in this example as in Example 3.18. It calls the ASM function `_fircasmfunc()`, defined in the file `FIRCASMfuncfast.asm`, as shown in Figure 3.55. This ASM function executes faster than the function in the previous example by having parallel instructions and rearranging the sequence of instructions. There are two parallel instructions: LDH/LDH and SUB/LDH.

- (1) The number of NOPs is reduced from 19 to 11.
- (2) The SUB instruction to decrement the loop count is moved up the program.
- (3) The sequence of some instructions is changed to fill some of the NOP slots. For example, the conditional branch instruction executes after the ADD instruction to accumulate in A8, since branching has five delay slots. Additional changes to make it

```

;FIRCASMfuncfast.asm C-called faster function to implement FIR
.def _fircasmfunc
_fircasmfunc:
    MV A6,A1           ;ASM function called from C
    MPY A6,2,A6         ;since dly buffer data as byte
    ZERO A8             ;init A8 for accumulation
    ADD A6,B4,B4        ;since coeff buffer data as byte
    SUB B4,1,B4         ;B4=bottom coeff array h[N-1]
loop:
    LDH *A4++,A2        ;start of FIR loop
    || LDH *B4--,B2      ;A2=x[n-(N-1)+i] i=0,1,...,N-1
    || SUB A1,1,A1       ;B2=h[N-1-i] i=0,1,...,N-1
    || LDH *A4,A7        ;decrement loop count
    || NOP 4              ;A7=x[(n-(N-1)+i+1]update delays
    [A1] STH A7,*-A4[1]  ;A7-->x[(n-(N-1)+i] update sample
    B loop               ;branch to loop if count # 0
    NOP 2
    MPY A2,B2,A6        ;A6=x[n-(N-1)+i]*h[N-1-i]
    NOP
    ADD A6,A8,A8        ;accumlate in A8
    B B3                 ;return addr to calling routine
    MV A8,A4             ;result returned in A4
    NOP 4

```

Figure 3.55 Listing of program FIRCASMfuncfast.asm.

faster would also make it less comprehensible due to further resequencing of the instructions. Exclude file L138_FIRCASMfunc.asm from project and include file FIRCASMfuncfast.asm. Build and run the program and verify implementation of the same 1 kHz band-pass filter as in the previous example.

REFERENCES

1. R. D. Strum and D. E. Kirk, *First Principles of Discrete Systems and Digital Signal Processing*, Addison-Wesley, Reading, MA, 1988.
2. http://processors.wiki.ti.com/index.php/C674x_DSPLIB
3. R. Chassaing, *DSP Applications Using C and the TMS320C6x DSK*, John Wiley & Sons, Inc., New York, NY, 2002.

Chapter 4

Infinite Impulse Response Filters

- Infinite impulse response (IIR) filter structures: direct form I, direct form II, cascade, and parallel
- Impulse invariant and bilinear transform methods of filter design
- Sinusoidal waveform generation using difference equations
- MATLAB filter design and analysis package
- Programming examples using C and ASM code

FIR filter discussed in Chapter 3 has no analog counterpart. In this chapter, we discuss the infinite impulse response filter that makes use of the vast knowledge already acquired with analog filters. The design procedure described in this chapter involves the conversion of an analog filter to an equivalent discrete filter. Either the impulse invariance or bilinear transformation (BLT) technique may be used. Both procedures convert the transfer function of an analog filter in the s -domain into an equivalent discrete-time transfer function in the z -domain.

4.1 INTRODUCTION

Consider a general input–output equation of the form

$$y(n) = \sum_{k=0}^M b_k x(n-k) - \sum_{l=1}^N a_l y(n-l) \quad (4.1)$$

or, equivalently,

$$\begin{aligned} y(n) = & b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) + \cdots + b_M x(n-M) \\ & - a_1 y(n-1) - a_2 y(n-2) - \cdots - a_N y(n-N) \end{aligned} \quad (4.2)$$

This recursive difference equation represents an IIR filter. The output $y(n)$, at time n , depends not only on the current input $x(n)$, at time n , and on past inputs $x(n-1), x(n-2), \dots, x(n-M)$ but also on past outputs $y(n-1), y(n-2), \dots, y(n-N)$.

If we assume all initial conditions to be zero in Equation 4.2, its z -transform is

$$\begin{aligned} Y(z) &= (b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_N z^{-N}) X(z) \\ &\quad - (a_1 z^{-1} + a_2 z^{-2} + \dots + a_M z^{-M}) Y(z). \end{aligned} \quad (4.3)$$

Letting $N = M$ in Equation 4.3, the transfer function $H(z)$ of the IIR filter is

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_N z^{-N}} = \frac{N(z)}{D(z)}, \quad (4.4)$$

where $N(z)$ and $D(z)$ represent the numerator and denominator polynomials, respectively. Multiplying and dividing by z^n , $H(z)$ becomes

$$H(z) = \frac{b_0 z^N + b_1 z^{N-1} + b_2 z^{N-2} + \dots + b_N}{z^N + a_1 z^{N-1} + a_2 z^{N-2} + \dots + a_N} = C \prod_{i=1}^N \frac{z - z_i}{p - p_i}, \quad (4.5)$$

where C is a constant, which is a transfer function with N zeros and N poles. If all the coefficients a_j in Equation 4.5 are equal to zero, this transfer function reduces to a transfer function with N poles at the origin in the z -plane representing the FIR filter discussed in Chapter 3. For a discrete-time system to be stable, all the poles of its z -transfer function must lie inside the unit circle, as discussed in Chapter 3. Hence, for an IIR filter to be stable, the magnitude of each of its poles must be less than 1 or

- (1) If $|p_i| < 1$, then $h(n) \rightarrow 0$, as $n \rightarrow \infty$, yielding a stable system.
- (2) If $|p_i| > 1$, then $h(n) \rightarrow \infty$, as $n \rightarrow \infty$, yielding an unstable system.
- (3) If $|p_i| = 1$, the system is marginally stable, yielding an oscillatory response.

Furthermore, multiple order poles on the unit circle yield an unstable system. Note again that with all the coefficients a_j equal to zero, the system reduces to a nonrecursive and stable FIR filter.

4.2 IIR FILTER STRUCTURES

Several different structures may be used to represent an IIR filter.

4.2.1 Direct Form I Structure

With the direct form I structure shown in Figure 4.1, the filter in Equation 4.2 can be realized. For an N th-order filter, this structure contains $2N$ delay elements, each

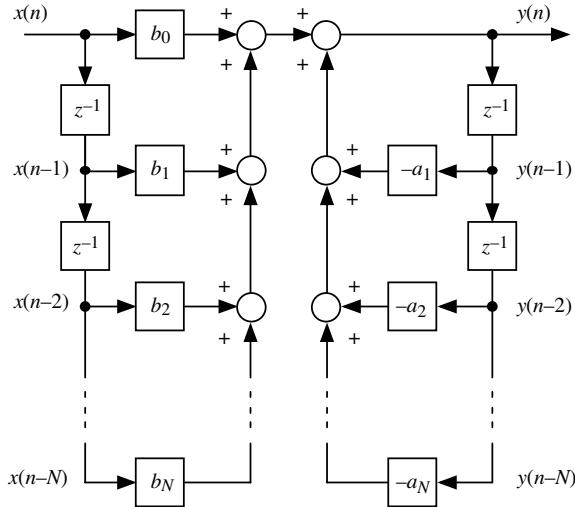


Figure 4.1 Direct form I IIR filter structure.

represented by z^{-1} . For example, a second-order filter with $N = 2$ will contain four delay elements.

4.2.2 Direct Form II Structure

The direct form II structure shown in Figure 4.2 is one of the most commonly used structures. It requires half as many delay elements as the direct form I. For example, a second-order filter requires two delay elements z^{-1} , as opposed to four with the direct form I.

From the block diagram of Figure 4.2, it can be seen that

$$w(n) = x(n) - a_1 w(n-1) - a_2 w(n-2) - \cdots - a_N w(n-N) \quad (4.6)$$

and that

$$y(n) = b_0 w(n) + b_1 w(n-1) + b_2 w(n-2) + \cdots + b_N w(n-N). \quad (4.7)$$

Taking z -transforms of Equations 4.6 and 4.7,

$$W(z) = X(z) - a_1 z^{-1} W(z) - a_2 z^{-2} W(z) - \cdots - a_N z^{-N} W(z) \quad (4.8)$$

and hence,

$$X(z) = (1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_N z^{-N}) W(z) \quad (4.9)$$

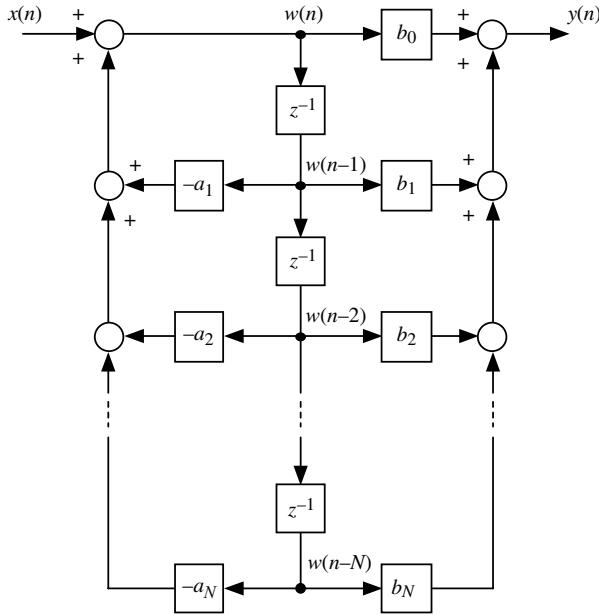


Figure 4.2 Direct form II IIR filter structure.

and

$$Y(z) = (b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_N z^{-N}) W(z). \quad (4.10)$$

Thus,

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_N z^{-N}}, \quad (4.11)$$

which is the same as Equation 4.4.

The direct form II structure can be represented by difference Equations 4.6 and 4.7 taking the place of Equation 4.2.

Equations 4.6 and 4.7 may be used to implement an IIR filter in a computer program. Initially, $w(n-1), w(n-2), \dots$ are set to zero. At time n , a new sample $x(n)$ is acquired, Equation 4.6 is used to solve for $w(n)$, and then the output $y(n)$ is calculated using Equation 4.7.

4.2.3 Direct Form II Transpose

The direct form II transpose structure shown in Figure 4.3 is a modified version of the direct form II and requires the same number of delay elements.

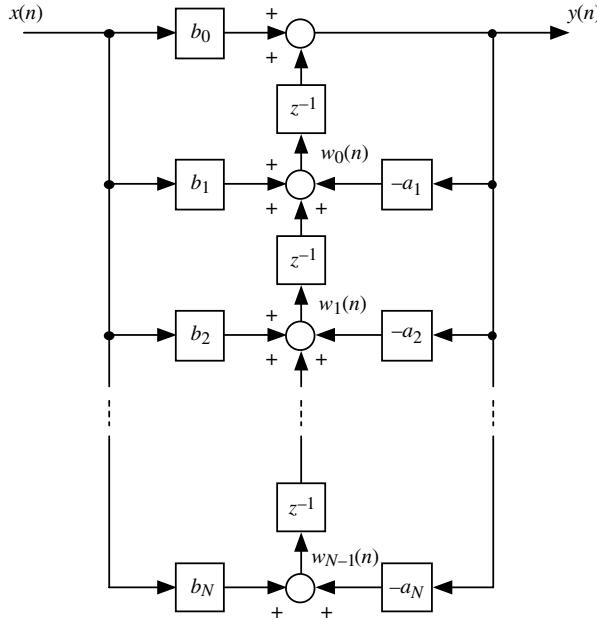


Figure 4.3 Direct form II transpose IIR filter structure.

From inspection of the block diagram, the filter output can be computed using

$$y(n) = b_0x(n) + w_0(n-1). \quad (4.12)$$

Subsequently, the contents of the delay line can be updated using

$$w_0(n) = b_1x(n) + w_1(n-1) - a_1y(n), \quad (4.13)$$

$$w_1(n) = b_2x(n) + w_2(n-1) - a_2y(n), \quad (4.14)$$

and so on until finally

$$w_{N-1}(n) = b_Nx(n) - a_Ny(n). \quad (4.15)$$

Using Equation 4.13 to find $w_0(n-1)$,

$$w_0(n-1) = b_1x(n-1) + w_1(n-2) - a_1y(n-1),$$

Equation 4.12 becomes

$$y(n) = b_0x(n) + [b_1x(n-1) + w_1(n-2) - a_1y(n-1)].$$

Similarly, using Equation 4.14 to find $w_1(n - 2)$,

$$w_1(n - 2) = b_2x(n - 2) + w_2(n - 3) - a_2y(n - 2),$$

Equation 4.12 becomes

$$y(n) = b_0x(n) + [b_1x(n - 1) + [b_2x(n - 2) + w_2(n - 3) - a_2y(n - 2)] - a_1y(n - 1)]. \quad (4.16)$$

Continuing this procedure until Equation 4.15 has been used, it can be shown that Equation 4.12 is equivalent to Equation 4.2 and hence that the block diagram of Figure 4.3 is equivalent to those of Figures 4.1 and 4.2. The transposed structure implements the zeros first and then the poles, whereas the direct form II structure implements the poles first.

4.2.4 Cascade Structure

The transfer function in (4.5) can be factorized as

$$H(z) = CH_1(z)H_2(z) \cdots H_r(z) \quad (4.17)$$

in terms of first- or second-order transfer functions. The cascade (or series) structure is shown in Figure 4.4. An overall transfer function can be represented with cascaded transfer functions. For each section, the direct form II structure or its transpose version can be used. Figure 4.5 shows a fourth-order IIR structure in terms of two direct form II second-order sections in cascade. The transfer function $H(z)$, in terms of cascaded second-order transfer functions, can be written as

$$H(z) = \prod_{i=1}^{N/2} \frac{b_{0i} + b_{1i}z^{-1} + b_{2i}z^{-2}}{1 + a_{1i}z^{-1} + a_{2i}z^{-2}}, \quad (4.18)$$

where the constant C in (4.17) is incorporated into the coefficients. For example, $N = 4$ for a fourth-order transfer function, and (4.18) becomes

$$H(z) = \frac{(b_{01} + b_{11}z^{-1} + b_{21}z^{-2})(b_{02} + b_{12}z^{-1} + b_{22}z^{-2})}{(1 + a_{11}z^{-1} + a_{21}z^{-2})(1 + a_{12}z^{-1} + a_{22}z^{-2})}, \quad (4.19)$$

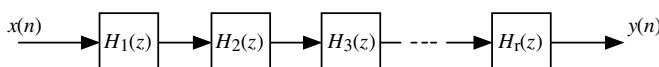


Figure 4.4 Cascade form IIR filter structure.

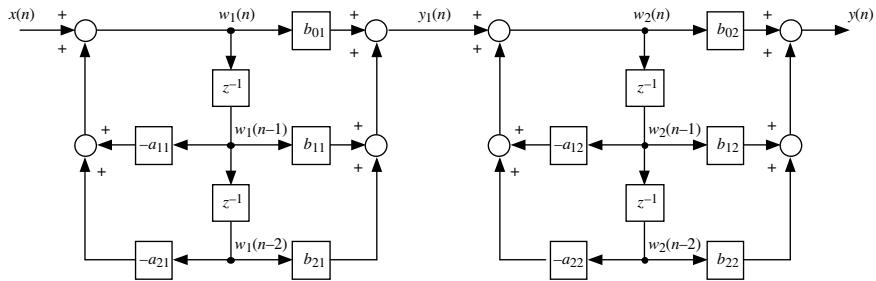


Figure 4.5 Fourth-order IIR filter with two direct form II sections in cascade.

as can be verified in Figure 4.5. From a mathematical standpoint, proper ordering of the numerator and denominator factors does not affect the output result. However, from a practical standpoint, proper ordering of each second-order section can minimize quantization noise. Note that the output of the first section, $y_1(n)$, becomes the input to the second section. With an intermediate output result stored in one of the registers, a premature truncation of the intermediate output becomes negligible. A programming example later in this chapter will illustrate the implementation of an IIR filter using cascaded second-order direct form II sections.

4.2.5 Parallel Form Structure

The transfer function in (4.5) can be represented as

$$H(z) = C + H_1(z) + H_2(z) + \dots + H_{r1}(z), \quad (4.20)$$

which can be obtained using a partial fraction expansion (PFE) of (4.5). This parallel form structure is shown in Figure 4.6. Each of the transfer functions $H_1(z)$, $H_2(z)$, ... can be either first- or second-order functions.

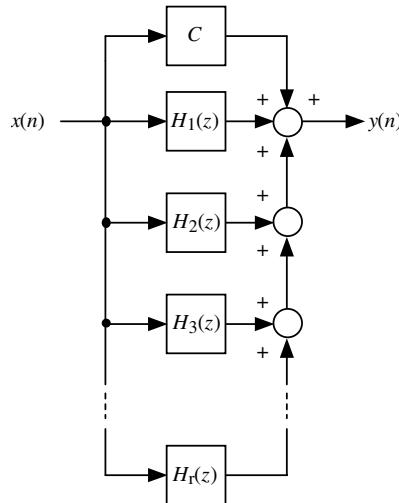
As with the cascade structure, the parallel form can efficiently be represented in terms of second-order direct form II structure sections. $H(z)$ can be expressed as

$$H(z) = C + \sum_{i=1}^{N/2} \frac{b_{0i} + b_{1i}z^{-1} + b_{2i}z^{-2}}{1 + a_{1i}z^{-1} + a_{2i}z^{-2}}. \quad (4.21)$$

For example, for a fourth-order transfer function, $H(z)$ in (4.21) becomes

$$H(z) = C + \frac{b_{01} + b_{11}z^{-1} + b_{21}z^{-2}}{1 + a_{12}z^{-1} + a_{22}z^{-2}} + \frac{b_{02} + b_{12}z^{-1} + b_{22}z^{-2}}{1 + a_{12}z^{-1} + a_{22}z^{-2}}. \quad (4.22)$$

This fourth-order parallel structure is represented in terms of two direct form II sections, as shown in Figure 4.7. From that figure, the output $y(n)$ can be expressed in terms of the output of each section, or

**Figure 4.6** Parallel form IIR filter structure.

$$y(n) = Cx(n) + \sum_{i=1}^{N/2} y_i(n). \quad (4.23)$$

The quantization error associated with the coefficients of an IIR filter depends on the amount of shift in the position of its poles and zeros in the complex plane. This implies that the shift in the position of a particular pole depends on the positions of all the other poles. To minimize this dependency of the poles, an N th-order IIR filter is typically implemented as cascaded second-order sections.

4.3 IMPULSE INVARIANCE

This method of IIR filter design is based on the concept of mapping each s -plane pole of a continuous-time filter to a corresponding z -plane pole using the substitution $(1 - e^{-p_k t_s} z^{-1})$ for $(s + p_k)$ in $H(s)$. This can be achieved by several different means. Partial fraction expansion of $H(s)$ and substitution of $(1 - e^{-p_k t_s} z^{-1})$ for $(s + p_k)$ can involve a lot of algebraic manipulation.

An equivalent method of making the transformation is to use tables of Laplace and z -transforms. Generally, tables of Laplace transforms list s -domain transfer functions and their corresponding impulse responses. Tables of z -transforms may be used to find the z -transfer function corresponding to an impulse response. The method is referred to as impulse invariance because of the equivalence of the impulse responses of the digital filter (described by z -transfer function) and of the analog prototype (described by s -transfer function). The specific relationship between the two impulse responses is that one comprises samples of a scaled version of the other. The performance of the two filters may differ, however, depending on how well the

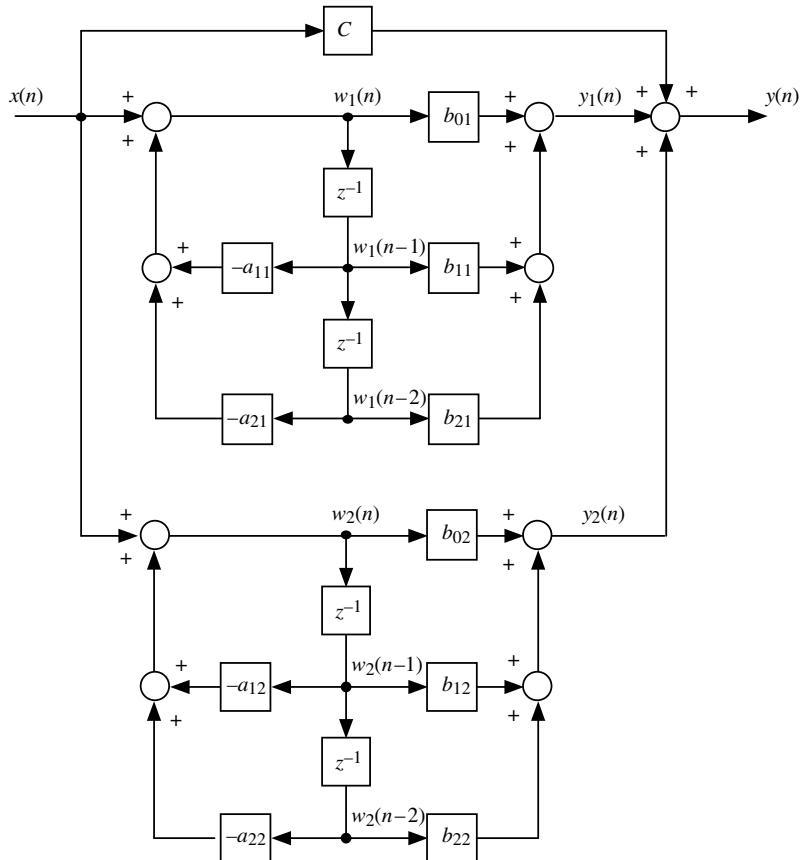


Figure 4.7 Fourth-order IIR filter with two direct form II sections in parallel.

detail of the continuous impulse response of the analog prototype is represented by its sampled form. As will be illustrated in Section 4.5, if the sampling rate of the digital filter is not sufficiently high to capture the details of continuous-time impulse response $h(t)$, then the high-frequency characteristics of the prototype filter may not be reproduced in the digital implementation.

4.4 BILINEAR TRANSFORMATION

The BLT is the most commonly used technique for transforming an analog filter into a digital filter. It provides one-to-one mapping from the analog s -plane to the digital z -plane, using

$$s = K \frac{(z - 1)}{(z + 1)}. \quad (4.24)$$

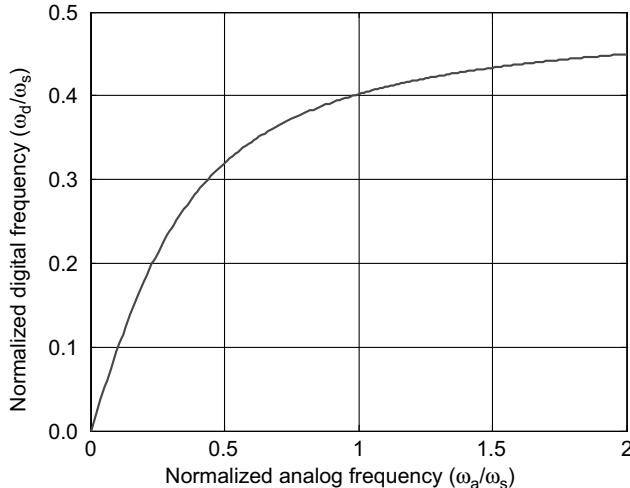


Figure 4.8 Relationship between analog and digital frequencies ω_A and ω_D due to frequency warping in the bilinear transform.

The constant K in (4.24) is commonly chosen as $K = 2/T$, where T represents the sampling period in seconds, of the digital filter. Other values for K can be selected, as described in Section 4.4.1.

The bilinear transformation allows the following:

- (1) The left region in the s -plane, corresponding to $\sigma < 0$, maps *inside* the unit circle in the z -plane.
- (2) The right region in the s -plane, corresponding to $\sigma > 0$, maps *outside* the unit circle in the z -plane.
- (3) The imaginary $j\omega$ axis in the s -plane maps *on* the unit circle in the z -plane.

Let ω_A and ω_D represent analog and digital frequencies, respectively. With $s = j\omega_A$ and $z = e^{j\omega_D T}$, (4.24) becomes

$$j\omega_A = K \frac{(e^{j\omega_D T} - 1)}{(e^{j\omega_D T} + 1)} = K \frac{e^{j\omega_D T/2}(e^{j\omega_D T/2} - e^{-j\omega_D T/2})}{e^{j\omega_D T/2}(e^{j\omega_D T/2} + e^{-j\omega_D T/2})}. \quad (4.25)$$

Using Euler's expressions for sine and cosine in terms of complex exponential functions, ω_A from (4.25) becomes

$$\omega_A = K \tan\left(\frac{\omega_D T}{2}\right), \quad (4.26)$$

which relates the analog frequency ω_A to the digital frequency ω_D . This relationship is plotted in Figure 4.8 for positive values of ω_A . The nonlinear compression of the entire

analog frequency range into the digital frequency range from zero to $\omega_s/2$ is referred to as frequency warping ($\omega_s = 2\pi/T$).

4.4.1 Bilinear Transform Design Procedure

The BLT design procedure for transforming an analog filter design expressed as a transfer function $H(s)$ into a z -transfer function $H(z)$ representing a discrete-time IIR filter is described by

$$H(z) = H(s) \Big|_{s=\frac{2(z-1)}{T(z+1)}} . \quad (4.27)$$

$H(s)$ can be chosen according to well-documented analog filter design theory, for example, Butterworth, Chebyshev, Bessel, or elliptic.

It is common to choose $K = 2/T$. Alternatively, it is possible to prewarp the analog filter frequency response in such a way that the bilinear transform maps an analog frequency $\omega_A = \omega_c$, in the range 0 to $\omega_s/2$, to exactly the same digital frequency $\omega_D = \omega_c$. This is achieved by choosing

$$K = \frac{\omega_c}{\tan(\pi\omega_c/\omega_s)} . \quad (4.28)$$

4.5 PROGRAMMING EXAMPLES USING C AND ASM CODE

The examples in this section introduce and illustrate the implementation of infinite impulse response filtering. Many different approaches to IIR filter design are possible and most often IIR filters are designed with the aid of software tools. Before using such a design package and in order to appreciate better what such design packages do, a simple example will be used to illustrate some of the basic principles of IIR filter design.

4.5.1 Design of a Simple IIR Low-Pass Filter

Traditionally, IIR filter design is based on the concept of transforming a continuous-time, or analog, design into the discrete-time domain. Butterworth, Chebyshev, Bessel, and elliptic classes of analog filter are widely used. In this example, a second-order, type 1 Chebyshev low-pass filter with 2 dB of passband ripple and a cutoff frequency of 1500 Hz (9425 rad/s) is used.

The continuous-time transfer function of this filter is

$$H(s) = \frac{58072962}{s^2 + 7576s + 73109527} \quad (4.29)$$

and its frequency response is shown in Figure 4.9.

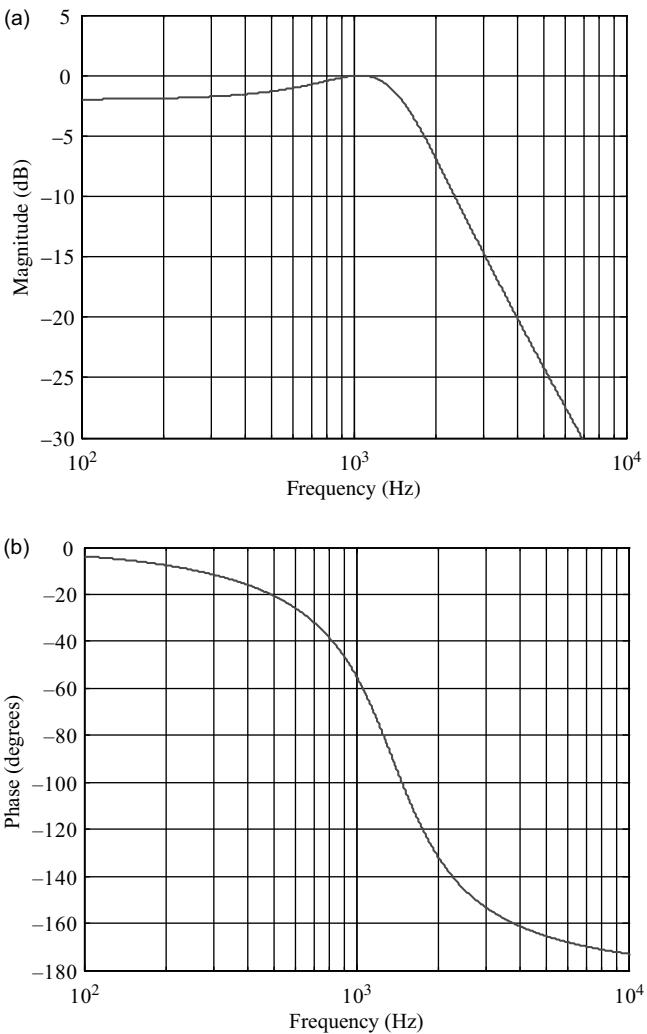


Figure 4.9 (a) Magnitude frequency response of filter $H(s)$. (b) Phase response of filter $H(s)$.

Using MATLAB, the coefficients of this s -transfer function can be generated by typing

```
>> [b, a] = cheby1(2, 2, 2*pi*1500, 's');
```

at the command line.

Our task is to transform this design into the discrete-time domain. One method of achieving this is the impulse invariance method.

Impulse invariance method

Starting with the filter transfer function (4.29), we can make use of the Laplace transform pair

$$L\{Ae^{-\alpha t} \sin(\omega t)\} = \frac{A\omega}{s^2 + 2\alpha s + (\alpha^2 + \omega^2)} \quad (4.30)$$

(the filter's transfer function is equal to the Laplace transform of its impulse response) and use the values

$$\begin{aligned}\alpha &= 7576/2 = 3787.9 \\ \omega &= \sqrt{73109527 - 3787.9^2} = 7665.6 \\ A &= 58072962/7665.6 = 7575.8.\end{aligned}$$

Hence, the impulse response of the filter in this example is given by

$$h(t) = 7575.8e^{-3788t} \sin(7665.6t). \quad (4.31)$$

The z -transform pair

$$Z\{Ae^{-\alpha t} \sin(\omega t)\} = \frac{Ae^{-\alpha t_s} \sin(\omega t_s)z^{-1}}{1 - 2(Ae^{-\alpha t_s} \cos(\omega t_s))z^{-1} + e^{-2\alpha t_s}z^{-2}} \quad (4.32)$$

yields the following discrete-time transfer function when we substitute for ω, A, α , and set $t_s = 0.000125$ in Equation 4.32.

$$H(z) = \frac{Y(z)}{X(z)} = \frac{0.48255z^{-1}}{1 - 0.71624315z^{-1} + 0.38791310z^{-2}}. \quad (4.33)$$

From $H(z)$, the following difference equation may be derived:

$$y(n) = 0.48255x(n-1) + 0.71624315y(n-1) - 0.38791310y(n-2). \quad (4.34)$$

With reference to (4.1), we can see that $a_1 = 0.71624315$, $a_2 = -0.38791310$, $b_0 = 0.0000$, and $b_1 = 0.48255$.

In order to apply the impulse invariant method using MATLAB, type

```
>> [bz, az] = impinvar(b, a, 8000);
```

This discrete-time filter has the property that its discrete-time impulse response $h(n)$ is equal to samples of the continuous-time impulse response $h(t)$ (scaled by the sampling period), as shown in Figure 4.10.

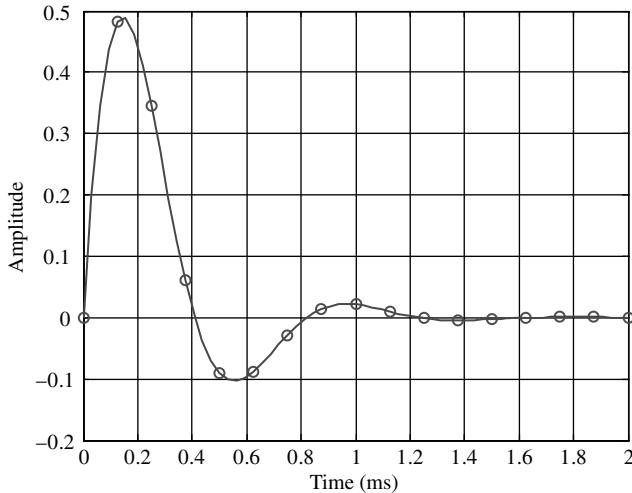


Figure 4.10 Impulse responses $h(t)$ (scaled by sampling period t_s) and $h(n)$ of continuous-time filter and its impulse invariant digital implementation.

Although it is evident from Figure 4.10 that the discrete-time impulse response $h(n)$ decays almost to zero, this sequence is not finite. Whereas the impulse response of an FIR filter is given explicitly by its finite set of coefficients, the coefficients of an IIR filter are used in a recursive Equation 4.1 to determine its impulse response $h(n)$.

EXAMPLE 4.1: **Implementation of an IIR Filter Using Cascaded Second-Order Direct Form II Sections (L138_iirssos_intr)**

Program L138_iirssos_intr.c, listed in Figure 4.11, implements a generic IIR filter using cascaded direct form II second-order sections and coefficient values stored in a separate file. Each section of the filter is implemented using the following two program statements

```
wn = input - a[section][1]*w[section][0] - a[section][2]*w[section][1];
yn = b[section][0]*wn + b[section][1]*w[section][0] +
    b[section][2]*w[section][1];
```

which correspond to the equations

$$w(n) = x(n) - a_1 w(n-1) - a_2 w(n-2)$$

and

$$y(n) = b_0 w(n) + b_1 w(n-1) + b_2 y(n-2).$$

With reference to Figure 4.5 and to Equation 4.18, the coefficients a_{0i} , a_{1i} , a_{2i} , b_{0i} , b_{1i} , and b_{2i} are stored as

$a[i][0]$, $a[i][1]$, $a[i][2]$, $b[i][0]$, $b[i][1]$, and $b[i][2]$, respectively. $w[i][0]$ and $w[i][1]$ correspond to $w_i(n-1)$ and $w_i(n-2)$.

```

// L138_iirssos_intr.c
//
// IIR filter implemented using second order sections
// floating point coefficients read from file
//

#include "L138_aic3106_init.h"
#include "impinv.cof"

float w[NUM_SECTIONS][2] = {0};

interrupt void interrupt4(void) // interrupt service routine
{
    int section; // index for section number
    float input; // input to each section
    float wn,yn; // intermediate and output values

    input = ((float)input_left_sample()); // input from ADC
    for (section=0 ; section< NUM_SECTIONS ; section++)
    {
        wn = input - a[section][1]*w[section][0]
            - a[section][2]*w[section][1];
        yn = b[section][0]*wn + b[section][1]*w[section][0]
            + b[section][2]*w[section][1];
        w[section][1] = w[section][0];
        w[section][0] = wn;
        input = yn; // output of current section is input to next
    }
    output_left_sample((int16_t)(yn)); // output to L DAC
    return;
}

int main(void)
{
    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);
}

```

Figure 4.11 IIR filter program using second-order sections in cascade (L138_iirssos_intr.c).

The impulse invariant filter is implemented using program L138_iirssos_intr.c by including the coefficient file impinv.cof, listed in Figure 4.12. The number of cascaded second-order sections is defined as NUM_SECTIONS in that file.

```

// impinv.cof
// second order type 1 Chebyshev LPF with 2dB passband ripple
// and cutoff frequency 1500Hz

#define NUM_SECTIONS 1

float b[NUM_SECTIONS][3]={ {0.0, 0.48255, 0.0} };
float a[NUM_SECTIONS][3]={ {1.0, -0.71624, 0.387913} };

```

Figure 4.12 Listing of coefficient file impinv.cof.

```

// L138_iirssoprn_intr.c
//
// IIR filter implemented using second order sections
// floating point coefficients read from file
//

#include "L138_aic3106_init.h"
#include "impinv.cof"

float w[NUM_SECTIONS][2] = {0};
AIC31_data_type codec_data;

interrupt void interrupt4(void) // interrupt service routine
{
    int section;    // index for section number
    float input;    // input to each section
    float wn,yn;    // intermediate and output values

    input =((float)prbs());           // pseudo random input
    for (section=0 ; section< NUM_SECTIONS ; section++)
    {
        wn = input - a[section][1]*w[section][0]
            - a[section][2]*w[section][1];
        yn = b[section][0]*wn + b[section][1]*w[section][0]
            + b[section][2]*w[section][1];
        w[section][1] = w[section][0];
        w[section][0] = wn;
        input = yn; // output of current section is input to next
    }
    codec_data.channel[LEFT] = (int16_t)yn;
    codec_data.channel[RIGHT] = (int16_t)yn;
    output_sample(codec_data.uint); // output to L and R DAC
    return;
}

int main(void)
{
    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);
}

```

Figure 4.13 IIR filter program using second-order sections in cascade, and internally generated pseudorandom noise as input (L138_iirssoprn_intr.c).

Build and run the program. Using a signal generator and oscilloscope to measure the magnitude frequency response of the filter, you should find that the attenuation of frequencies above 2500 Hz is not very pronounced. This is due to both the low order of the filter and the inherent shortcomings of the impulse invariant design method.

A number of alternative methods of assessing the magnitude frequency response of the filter will be described in the next few examples. In common with most of the example programs described in this chapter, program L138_iirssoprn_intr.c uses interrupt-based I/O (Figure 4.13). A DMA-based I/O version of the program L138_iirssoprn_edma.c is also provided in the same project folder. It can be used by excluding program L138_iirssoprn_intr.c from the project and including program L138_iirssoprn_edma.c before rebuilding the project.

EXAMPLE 4.2: **Implementation of IIR Filter Using Cascaded Second-Order Transposed Direct Form II Sections (L138_iirsostr_intr)**

A transposed direct form II structure can be implemented using program L138_iirsostr.c by replacing the lines that read

```
wn = input - a[section][1]*w[section][0]
    - a[section][2]*w[section][1];
yn = b[section][0]*wn + b[section][1]*w[section][0]
    + b[section][2]*w[section][1];
w[section][1] = w[section][0];
w[section][0] = wn;
```

with the following:

```
yn = b[section][0]*input + w[section][0];
w[section][0] = b[section][1]*input + w[section][1]
    - a[section][1]*yn;
w[section][1] = b[section][2]*input - a[section][2]*yn;
```

(Variable wn is not required in the latter case.)

This substitution has been made in program L138_iirsostr_intr.c, stored in project folder L138_iirsostr_intr. You should not notice any difference in the characteristics of the filters implemented using programs L138_iirsostr.c and L138_iirsostr_intr.c.

EXAMPLE 4.3: **Estimating the Frequency Response of an IIR Filter Using Pseudorandom Noise as Input (L138_iirsosprn_intr)**

Program L138_iirsosprn_intr.c is closely related to program L138_firprn_intr.c, described in Chapter 3. In real-time, it generates a pseudorandom binary sequence and uses this wideband noise signal as the input to an IIR filter. The output of the filter is written to the DAC in the AIC3106 codec and the resulting analog signal (filtered noise) may be analyzed using an oscilloscope, spectrum analyzer, or *Goldwave*. The frequency content of the filter output gives a good indication of the filter's magnitude frequency response. Figures 4.14 and 4.15 show the output of the example filter (using coefficient file impinv.cof) displayed using the FFT function of a *Rigol DS1052E* oscilloscope and using *Goldwave*.

In Figure 4.14, the vertical scale is 5 dB per division and the horizontal scale is 625 Hz per division. The low pass characteristic of the example filter is evident in the left-hand half of the figures between 0 and 2500 Hz. Between 2500 and 4000 Hz, the low pass characteristic is less pronounced and the steeper roll-off beyond 4000 Hz is due not to the IIR filter but to the reconstruction filter in the AIC3106 codec.

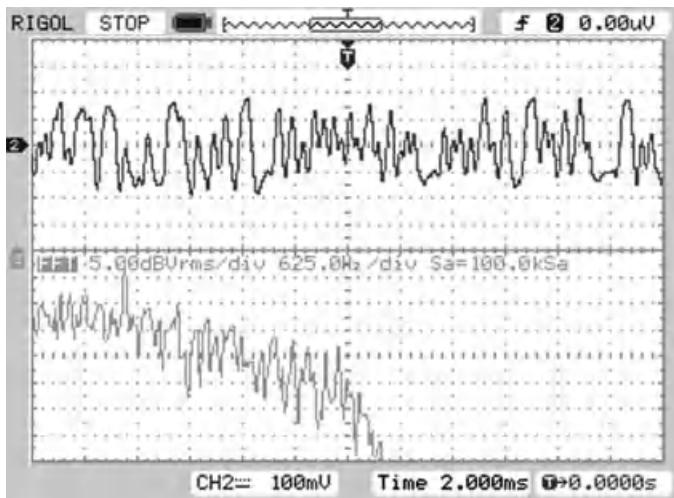


Figure 4.14 Output from program L138_iirssosprn_intr.c, using coefficient file impinv.coef, viewed using the FFT function of a *Rigol DS1052E* oscilloscope.

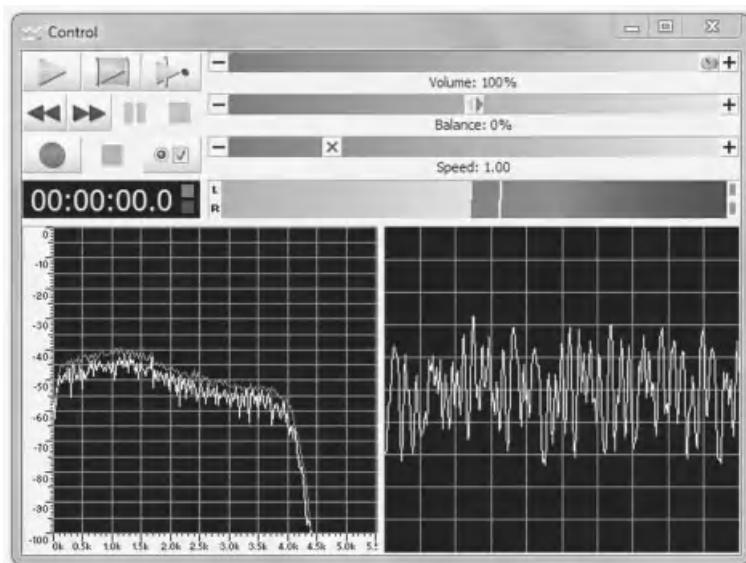


Figure 4.15 Output from program L138_iirssosprn_intr.c, using coefficient file impinv.coef, viewed using *Goldwave*.

EXAMPLE 4.4: Estimating the Frequency Response of an IIR Filter Using a Sequence of Impulses as Input (L138_iirssosdelta_intr)

Instead of a pseudorandom binary sequence, program L138_iirssosdelta_intr.c uses a sequence of discrete-time impulses as the input to an IIR filter (Figure 4.16). The resultant

```
// L138_iirssosdelta_intr.c
//
// IIR filter implemented using second order sections
// floating point coefficients read from file
//

#include "L138_aic3106_init.h"
#define BUFSIZE 256
#define AMPLITUDE 60000
#include "impinv.cof"

float w[NUM_SECTIONS][2] = {0};
float dimpulse[BUFSIZE];
float response[BUFSIZE];
int index = 0;

interrupt void interrupt4(void) // interrupt service routine
{
    int section; // index for section number
    float input; // input to each section
    float wn,yn; // intermediate and output values

    input = dimpulse[index]; // input from array dimpulse
    for (section=0 ; section< NUM_SECTIONS ; section++)
    {
        wn = input - a[section][1]*w[section][0]
            - a[section][2]*w[section][1];
        yn = b[section][0]*wn + b[section][1]*w[section][0]
            + b[section][2]*w[section][1];
        w[section][1] = w[section][0];
        w[section][0] = wn;
        input = yn; // output of current section is input to next
    }
    response[index++] = yn; // save output value
    if (index >= BUFSIZE) index = 0;
    output_left_sample((int16_t)(yn*AMPLITUDE));
    return;
}

int main(void)
{
    int i;

    for (i=0 ; i< BUFSIZE ; i++) dimpulse[i] = 0.0;
    dimpulse[0] = 1.0;
    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);
}
```

Figure 4.16 IIR filter program using second-order sections in cascade and internally generated impulses as input (L138_iirssosdelta_intr.c).

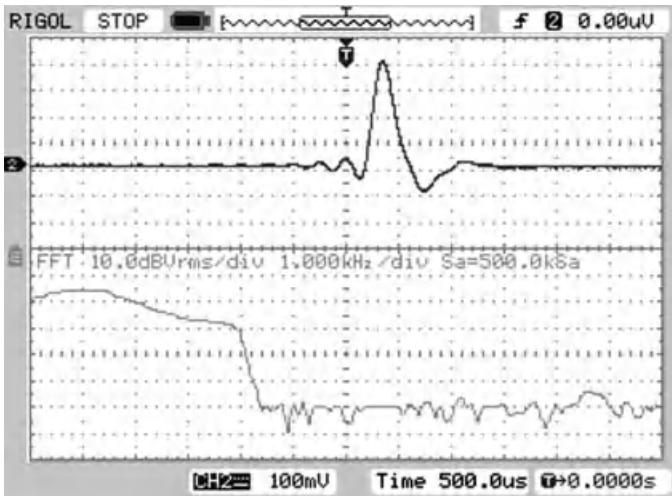


Figure 4.17 Output from program L138_iirssosdelta_intr.c, using coefficient file impinv.coef, viewed using the FFT function of a *Rigol DS1052E* oscilloscope.

output is an approximation to a repetitive sequence of filter impulse responses. This relies on the filter impulse response decaying practically to zero within the period between successive input impulses. The filter output is written to the DAC in the AIC3106 codec and the resulting analog signal may be analyzed using an oscilloscope, spectrum analyzer, or *Goldwave*. In addition, program L138_iirssosdelta_intr.c stores the BUFSIZE most recent samples of the filter output yn in array response and either by selecting *Tools > Graph* in the Code Composer Studio IDE or by using the MATLAB function L138_logfft(), the response of the filter may be viewed in both time and frequency domains.

Build and run the program after excluding and including the appropriate files in project c:\eXperimenter\L138_Chapter5\L138_iirssosdelta_intr.

Figure 4.17 shows the analog output signal generated by the program, captured using a *Rigol DS1052E* oscilloscope. The upper trace shows the time domain impulse response of the filter (500 μ s per division) and the lower trace shows the FFT of that impulse response over a frequency range of 0–12 kHz. The output waveform is shaped by both the IIR filter and the AIC3106 codec reconstruction filter. In the frequency domain, the codec reconstruction filter is responsible for the steep roll-off of gain at frequencies above 4000 Hz. In the time domain, the characteristics of the codec reconstruction filter are evident in the slight ringing that precedes the greater part of the impulse response waveform.

Halt the program and select *Tools > Graph > FFT Magnitude*. Set the *Graph Properties* as indicated in Figure 4.18 and you should see something similar to the graph shown in Figure 4.19.

Figure 4.20 shows the contents of buffer response plotted using MATLAB function L138_logfft().

Aliasing in the impulse invariant method

There are significant differences between the magnitude frequency response of the analog prototype filter used in this example (Figure 4.9) and that of its impulse invariant digital

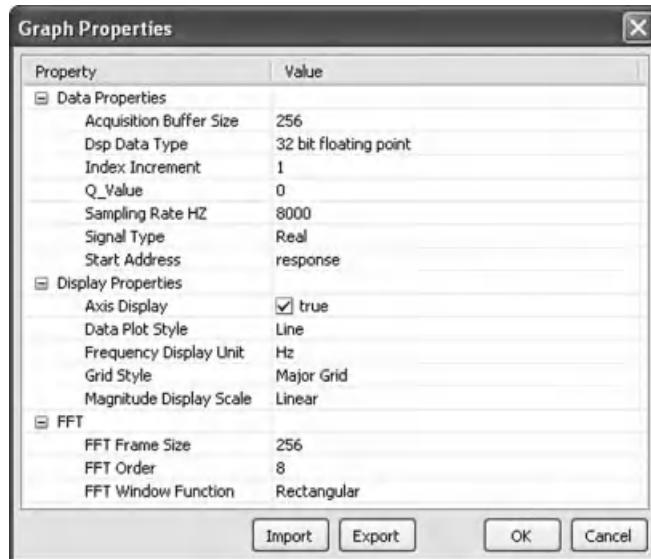


Figure 4.18 Graph Properties for use with program L138_iirssosdelta_intr.c.

implementation (Figure 4.20). The gain of the analog prototype has a magnitude of -15 dB at 3000 Hz, whereas, according to Figure 4.20, the gain of the digital filter at that frequency has a magnitude closer to -11 dB. This difference is due to aliasing. Whenever a signal is sampled, the problem of aliasing should be addressed and in order to avoid aliasing, the signal to be sampled should not contain any frequency components at frequencies greater than or equal to half the sampling frequency. The impulse invariant transformation yields a discrete-time impulse response equivalent to the continuous-time impulse response of the analog prototype $h(t)$ at the sampling instants, but this is not sufficient to ensure that the continuous-time response of a discrete-time implementation of the filter is equivalent to that of the analog prototype. The impulse invariant method will be completely free of aliasing

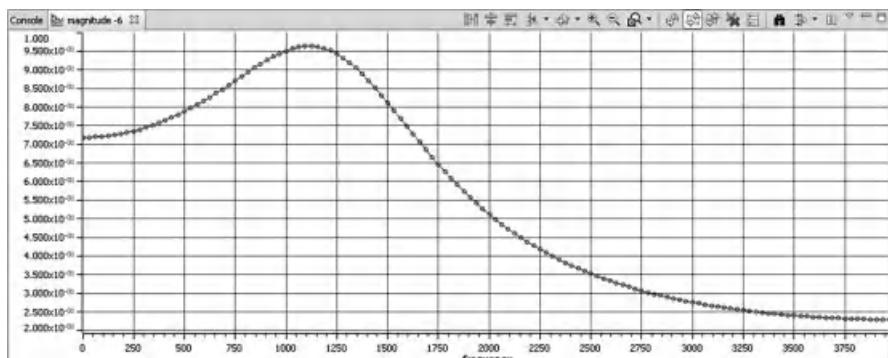


Figure 4.19 The magnitude frequency response of the filter implemented by program L138_iirssosdelta_intr.c, using coefficient file impinv.cof, plotted using Code Composer Studio software.

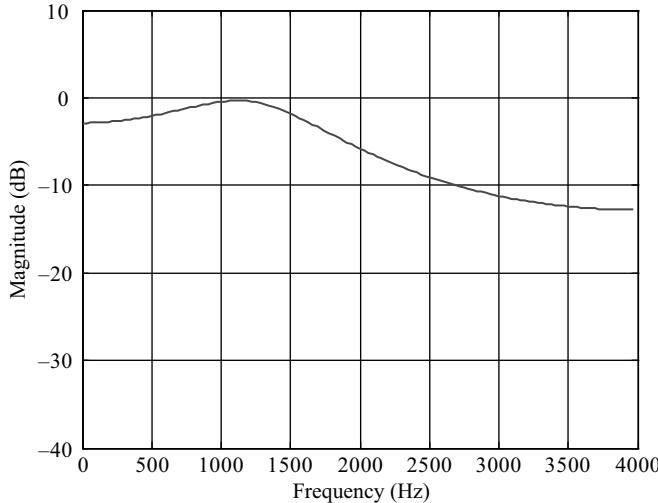


Figure 4.20 The magnitude frequency response of the filter implemented by program L138_iirssosdelta_intr.c, using coefficient file impinv.cof, plotted using MATLAB function L138_logfft().

effects only if the impulse response $h(t)$ contains no frequency components at frequencies greater than or equal to half the sampling frequency.

In our example, the magnitude frequency response of the analog prototype filter will be folded back on itself about the 4000 Hz point and this can be verified using MATLAB function freqz() which assesses the frequency response of a digital filter. Type

```
>> freqz(bz, az);
```

at the MATLAB command line in order to view the theoretical frequency response of the filter and compare this with Figure 4.20.

An alternative method of transforming an analog filter design to a discrete-time implementation, that eliminates this effect, is the use of the bilinear transform.

Bilinear transform method of digital filter implementation

The bilinear transform method of converting an analogue filter design to discrete-time is relatively straightforward, often involving less algebraic manipulation than the impulse invariant method. It is achieved by making the substitution

$$s = \frac{2(z-1)}{T(z+1)} \quad (4.35)$$

in $H(s)$, where T is the sampling period of the digital filter, that is,

$$H(z) = H(s)|_{s=\frac{2(z-1)}{T(z+1)}} \quad (4.36)$$

Applying this to the s -transfer function of (4.29) results in the following z -transfer function:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{0.12895869 + 0.25791738z^{-1} + 0.12895869z^{-2}}{1 - 0.81226498z^{-1} + 0.46166249z^{-2}}. \quad (4.37)$$

From $H(z)$, the following difference equation may be derived:

$$y(n) = 0.1290x(n) + 0.2579x(n-1) + 0.1290x(n-2) + 0.8123y(n-1) - 0.4617y(n-2). \quad (4.38)$$

This can be achieved in MATLAB by typing

```
>> [bd,ad] = bilinear(b,a,8000);
```

The characteristics of the filter can be examined by changing the coefficient file used by programs L138_iirssos_intr.c, L138_iirssosprn_intr.c and L138_iirssosdelta_intr.c from impinv.coef to bilinear.coef. In each case, change the line that reads

```
#include "impinv.coef"
```

to read

```
#include "bilinear.coef"
```

before building, loading, and running the programs.

Figures 4.21–4.24 show results obtained using programs L138_iirssosprn_intr.c and L138_iirssosdelta_intr.c with coefficient file bilinear.coef. The attenuation

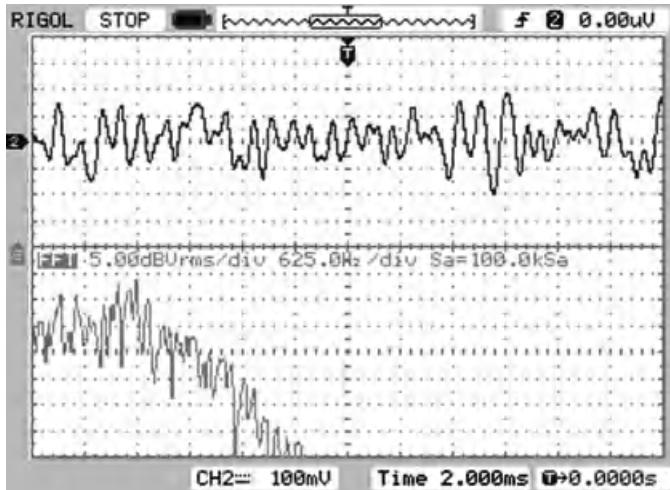


Figure 4.21 Output from program L138_iirssosprn_intr.c, using coefficient file bilinear.coef, viewed using the FFT function of a Rigol DS1052E oscilloscope.

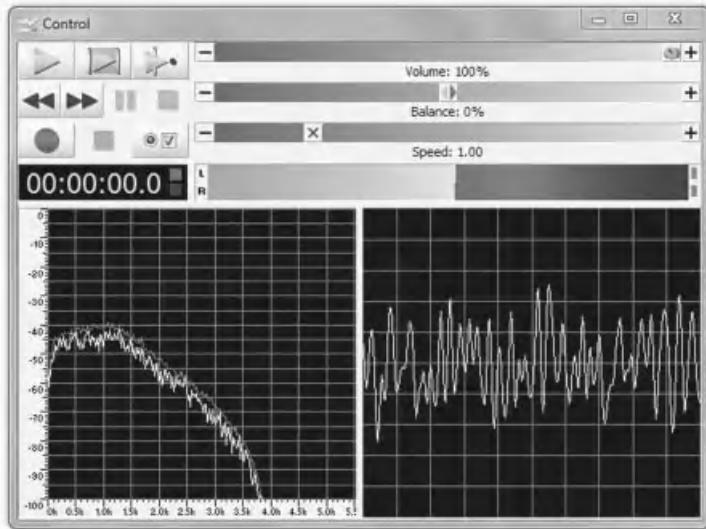


Figure 4.22 Output from program L138_irsosprn_intr.c, using coefficient file bilinear.cof, viewed using *Goldwave*.

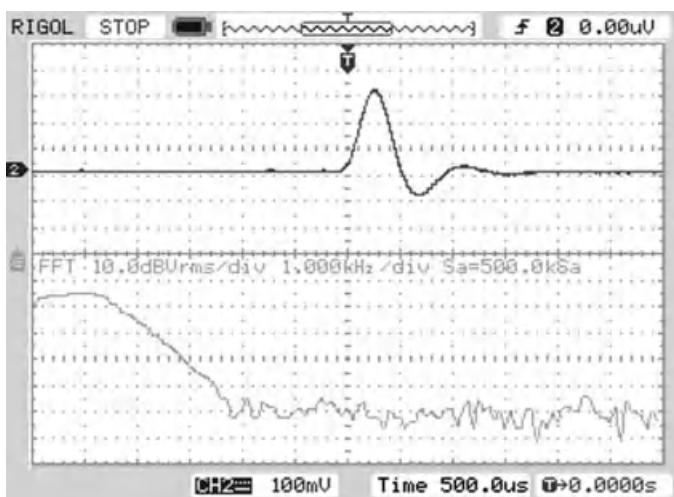


Figure 4.23 Output from program L138_irsosdelta_intr.c, using coefficient file bilinear.cof, viewed using the FFT function of a *Rigol DS1052E* oscilloscope.

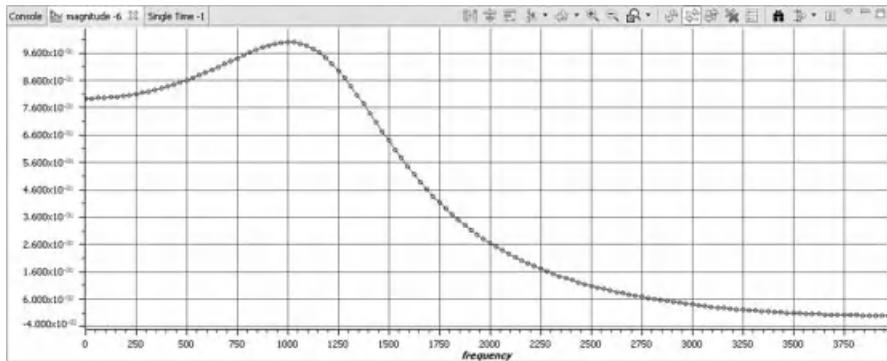


Figure 4.24 The magnitude frequency response of the filter implemented by program L138_iirssosdelta_intr.c, using coefficient file bilinear.cof, plotted using Code Composer Studio software.

provided by this filter at high frequencies is much greater than in the impulse invariant case. In fact, the attenuation at frequencies higher than 2000 Hz is significantly greater than that of the analog prototype filter (Figure 4.25).

Frequency warping in the bilinear transform

The concept behind the bilinear transform is that of compressing the frequency response of an analog filter design such that its response over the entire range of frequencies from zero to

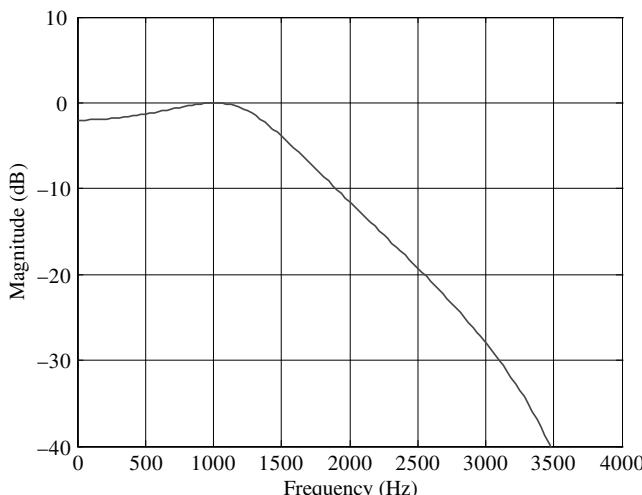


Figure 4.25 The magnitude frequency response of the filter implemented by program L138_iirssosdelta_intr.c, using coefficient file bilinear.cof, plotted using MATLAB function L138_logfft().

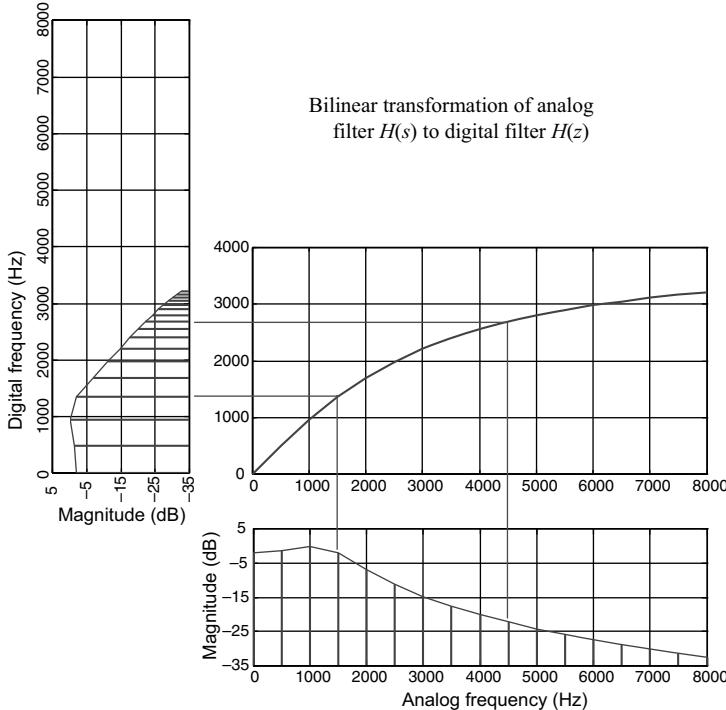


Figure 4.26 The effect of the bilinear transform on the magnitude frequency response of the example filter.

infinity is mapped onto the frequency range zero to half the sampling frequency of the digital filter. This may be represented by

$$f_D = \frac{\arctan(\pi f_A T_s)}{\pi T_s} \quad \text{or} \quad \omega_D = \frac{2}{T_s} \arctan\left(\frac{\omega_A T_s}{2}\right) \quad (4.39)$$

and

$$f_A = \frac{\tan(\pi f_D T_s)}{\pi T_s} \quad \text{or} \quad \omega_A = \frac{2}{T_s} \tan\left(\frac{\omega_D T_s}{2}\right) \quad (4.40)$$

where ω_D is the frequency at which the complex gain of the digital filter is equal to the complex gain of the analog filter at frequency ω_A . This relationship between ω_D and ω_A is illustrated in Figure 4.26. Consequently, there is no problem with aliasing, as seen in the case of impulse invariant transformation.

However, as a result of the frequency warping inherent in the bilinear transform, in this example, the cutoff frequency of the discrete-time filter obtained is not 1500 Hz but 1356 Hz. Figure 4.26 also shows that the gain of the analog filter at a frequency of 4500 Hz is equal to the gain of the digital filter at a frequency of 2428 Hz and that the digital frequency 1500 Hz corresponds to an analog frequency of 1702 Hz.

If we had wished to create a digital filter having a cutoff frequency of 1500 Hz, we could have applied the bilinear transform of Equation 4.35 to an analog prototype having a cutoff frequency of 1702 Hz.

This technique is referred to as prewarping the prototype analog design and is used by default in the MATLAB filter design and analysis tool `fdatool`, described in the next section. A digital filter with a cutoff frequency of 1500 Hz may be obtained by applying the bilinear transform to the analog filter:

$$H(s) = \frac{74767106}{s^2 + 8596s + 94126209}, \quad (4.41)$$

that is,

$$H(z) = H(s)|_{s=\frac{2(z-1)}{T(z+1)}} = \frac{0.15325778 + 0.30651556z^{-1} + 0.15325778z^{-2}}{1 - 0.66423178z^{-1} + 0.43599223z^{-2}}. \quad (4.42)$$

The analog filter represented by Equation 4.41 can be produced using the MATLAB command

```
>> [bb,aa] = cheby1(2,2,2*pi*1702,'s');
```

and the bilinear transformation applied by typing

```
>> [bbd,aad] = bilinear(bb,aa,8000);
```

to yield the result given by Equation 4.42.

Alternatively, prewarping of the analog filter design considered previously can be combined with application of the bilinear transform by typing

```
>> [bbd,aad]=bilinear(b,a,8000,1500);
```

at the MATLAB command line.

Coefficient file `bilinearw.cof`, stored in project folders `L138_iirssos_intr`, `L138_iirssosprn_intr`, and `L138_iirssosdelta_intr`, contains the coefficients obtained as described above.

Using MATLAB's filter design and analysis tool

MATLAB provides a filter design and analysis tool `fdatool` that makes the calculation of IIR filter coefficient values simple. Coefficients can be exported in direct form II, second-order section format and a MATLAB function `L138_iirssos_coeffs()`, supplied with this book as file `L138_iirssos_coeffs.m`, can be used to generate coefficient files compatible with the programs in this chapter.

EXAMPLE 4.5: Fourth-Order Elliptic Low-Pass IIR Filter Designed Using fdatool

To invoke the *Filter Design and Analysis* Tool window, type

```
>> fdatool
```

in the MATLAB command window. Enter the parameters for a fourth-order elliptic low-pass IIR filter with a cutoff frequency of 800 Hz, 1 dB of ripple in the passband, and 50 dB of stopband attenuation. Click on *Design Filter* and then look at the characteristics of the filter using options from the *Analysis* menu (Figure 4.27).

This example illustrates the steep transition from pass to stopbands possible even with relatively few filter coefficients.

Select *Filter Coefficients* from the *Analysis* menu in order to list the coefficient values designed. fdatool automatically designs filters as cascaded second-order sections. Each section is similar to those shown in block diagram form in Figure 4.5 and each section is characterized by six parameter values a_0 , a_1 , a_2 , b_0 , b_1 , and b_2 .

By default, fdatool uses the bilinear transform method of designing a digital filter starting from an analog prototype. Figure 4.28 shows the use of fdatool to design the Chebyshev filter considered in the preceding examples. Note that the magnitude frequency response decreases more and more rapidly with frequency approaching half the sampling frequency, and compare this with Figure 4.25. This is characteristic of filters designed using the bilinear transform.

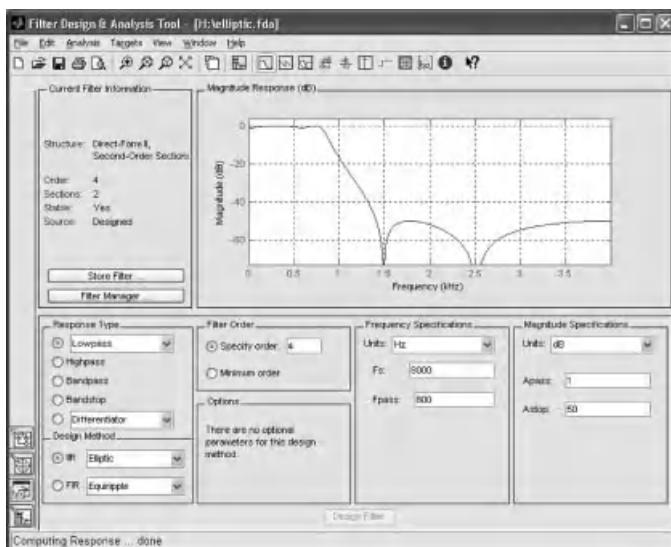


Figure 4.27 MATLAB fdatool window showing the magnitude frequency response of a fourth-order elliptic low-pass filter.

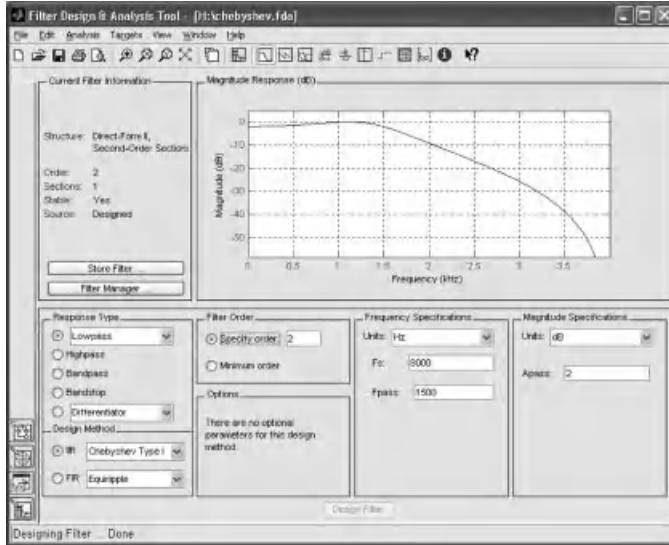


Figure 4.28 MATLAB fdatool window showing the magnitude frequency response of a second-order Chebyshev low-pass filter.

Implementing a filter designed using `fdatool` on the eXperimenter

In order to implement a filter designed using `fdatool` on the eXperimenter, carry out the following four steps:

- (1) Design the IIR filter using `fdatool`.
- (2) Click *Export* in the `fdatool` *File* menu.
- (3) Select *Workspace*, *Coefficients*, *SOS*, and *G* and click *Export*.
- (4) At the MATLAB command line, type `L138_iirssos_coeff(SOS, G)` and enter a filename, for example, `elliptic.cof`.

```
// elliptic.cof
// this file was generated using function L138_iirssos_coeff.m

#define NUM_SECTIONS 2

float b[NUM_SECTIONS][3] = {
{3.46359750E-002, 2.72500874E-002, 3.46359750E-002},
{2.90182959E-001, -2.25444662E-001, 2.90182959E-001} };

float a[NUM_SECTIONS][3] = {
{1.0000000E+000, -1.52872987E+000, 6.37029381E-001},
{1.0000000E+000, -1.51375731E+000, 8.68678568E-001} };
```

Figure 4.29 Listing of coefficient file `elliptic.cof`.

```
% L138_IIRSO_S_COEFFS.M
%
% MATLAB function to write SOS IIR filter coefficients
% in format suitable for use in L138 eXperimenter programs
% including L138_iirssos_intr.c, L138_iirssosprn_intr.c and
% L138_iirssosdelta_intr.c
% assumes that coefficients have been exported from
% fdatool as two matrices
% first matrix has format
% [ b10 b11 b12 a10 a11 a12
%   b20 b21 b22 a20 a21 a22
% ...
% ]
% where bij is the bj coefficient in the ith stage
% second matrix contains gains for each stage
%
function L138_iirssos_coeffs(coeff,gain)
%
num_sections=length(gain)-1;
fname = input('enter filename for coefficients ','s');
fid = fopen(fname,'wt');
fprintf(fid,'// %s\n',fname);
fprintf(fid,'// this file was generated using');
fprintf(fid,'\\n// function L138_iirssos_coeffs.m\\n',fname);
fprintf(fid,'\\n#define NUM_SECTIONS %d\\n',num_sections);
% first write the numerator coefficients b
% i is used to count through sections
fprintf(fid,'\\nfloat b[NUM_SECTIONS][3] = { \\n');
for i=1:num_sections
    if i==num_sections
        fprintf(fid,'{%.2.8E, %.2.8E, %.2.8E} };\\n',...
        coeff(i,1)*gain(i),coeff(i,2)*gain(i),coeff(i,3)*gain(i));
    else
        fprintf(fid,'{%.2.8E, %.2.8E, %.2.8E},\\n',...
        coeff(i,1)*gain(i),coeff(i,2)*gain(i),coeff(i,3)*gain(i));
    end
end
% then write the denominator coefficients a
% i is used to count through sections
fprintf(fid,'\\nfloat a[NUM_SECTIONS][3] = { \\n');
for i=1:num_sections
    if i==num_sections
        fprintf(fid,'{%.2.8E, %.2.8E, %.2.8E} };\\n',...
        coeff(i,4),coeff(i,5),coeff(i,6));
    else
        fprintf(fid,'{%.2.8E, %.2.8E, %.2.8E},\\n',...
        coeff(i,4),coeff(i,5),coeff(i,6));
    end
end
fclose(fid);
```

Figure 4.30 Listing of MATLAB file L138_iirssos_coeffs.m.

Figure 4.29 shows an example of a coefficient file produced using MATLAB function L138_iirssos_coeffs().

Program L138_iirssos_intr.c, introduced in Example 4.1, can be used to implement the filter. Edit the line in program L138_iirssos_intr.c that reads

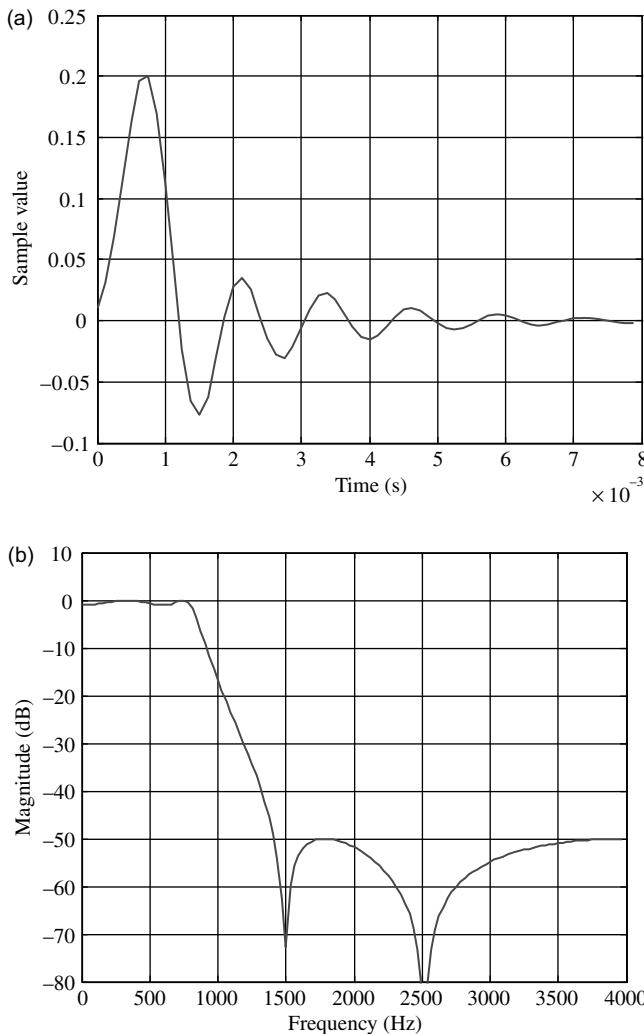


Figure 4.31 Impulse response and magnitude frequency response of the filter implemented by program L138_iirssosdelta_intr.c, using coefficient file elliptic.cof, plotted using MATLAB function L138_logfft().

```
#include "bilinear.cof"
```

to read

```
#include "elliptic.cof"
```

and build and run the program. The coefficients are also compatible with programs L138_iirssosdelta_intr.c and L138_iirssosdelta_intr.c.

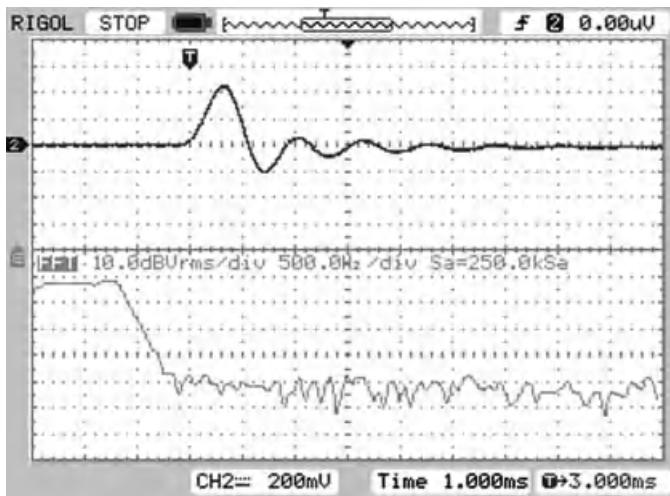


Figure 4.32 Output from program L138_iirssosdelta_intr.c, using coefficient file elliptic.cof, viewed using the FFT function of a Rigol DS1052E oscilloscope.

Figures 4.31 and 4.32 show results obtained using program L138_iirssosdelta_intr.c and coefficient file elliptic.cof.

EXAMPLE 4.6: Band-Pass Filter Design Using fdatool

Figure 4.33 shows fdatool used to design an 18th-order Chebyshev type 2 IIR band-pass filter centered at 2000 Hz. The filter coefficient file bp2000.cof, stored in folder

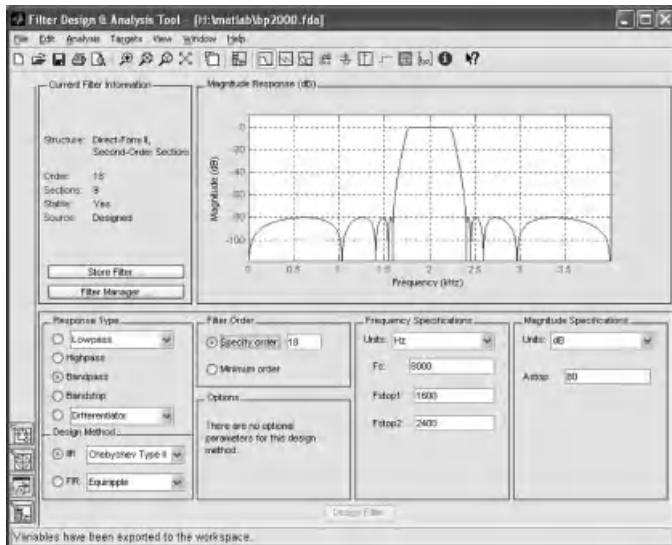


Figure 4.33 MATLAB fdatool window showing the magnitude frequency response of an 18th-order band-pass filter centered on 2000 Hz.

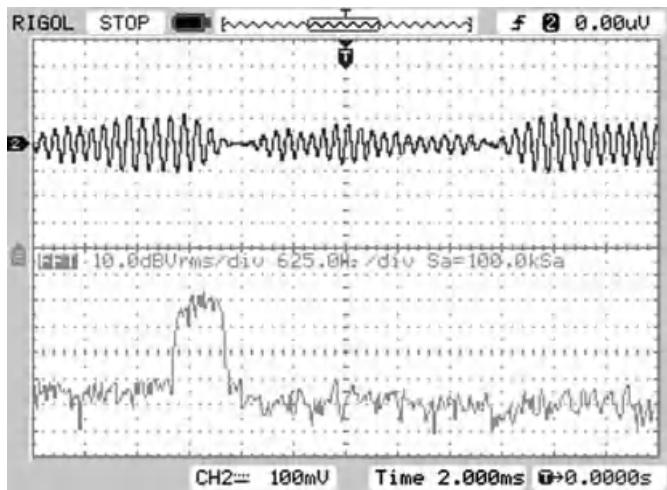


Figure 4.34 Output from program L138_iirssosprn_intr.c, using coefficient file bp2000.coef, viewed using FFT function of a *Rigol DSI052E* oscilloscope.

L138_Chapter4, is compatible with programs L138_iirssos_intr.c, L138_iirssosdelta_intr.c, and L138_iirssosprn_intr.c. Figures 4.34 and 4.35 show the output from program L138_iirssosprn_intr.c using these coefficients.

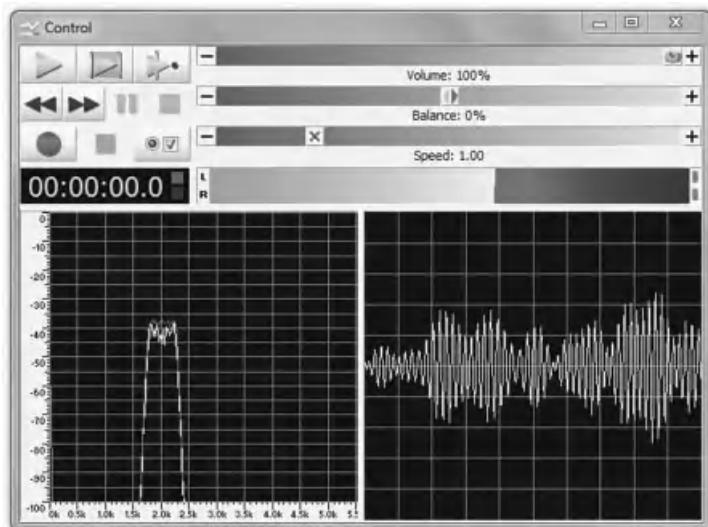


Figure 4.35 Output from program L138_iirssosprn_intr.c, using coefficient file bp2000.coef, viewed using *Goldwave*.

**EXAMPLE 4.7: Implementation of IIR Filter Using DSPLIB Function
DSPF_sp_biquad() (L138_iirssos_DSPLIB_edma)**

The c674x digital signal processing library (DSPLIB) comprises a number of functions, including correlation, convolution, filtering, and FFT. These functions may be called from a C program and their performance is optimized for speed of execution. This example demonstrates the use of the IIR filtering function `DSPFsp_biquad()`.

Function `DSPF_sp_biquad()` implements a second-order IIR filter section (a biquad) using single precision floating-point arithmetic. By calling the function more than once, higher order filters may be implemented as cascaded second-order sections. The function processes a block of input samples to produce a corresponding block of output samples and is therefore suited to the use of DMA-based I/O. Program L138_iirssos_DSPLIB_edma.c uses that I/O method.

The c674x DSPLIB package, which may be downloaded from the TI website, includes equivalent C language function definitions that indicate the inner workings of the library functions. Part of file `DSPF_sp_biquad_cn.c` is listed in Figure 4.36 and it is evident from this that it implements a direct form II filter structure.

In program L138_iirssos_DSPLIB_edma.c, listed in Figure 4.37, the line `#include "dsplib674x.h"` ensures that the library can be used. The paths to that file and to the library file `dsplib674x.lib` are included in the *Build Properties* for project L138_iirssos_DSPLIB_edma. The program as supplied implements the fourth-order elliptic low-pass filter used in Example 4.5, reading filter coefficients from the file `elliptic.cof`.

Within function `process_buffer()`, left channel input samples are first copied from the appropriate ping-pong input buffer into array `x`.

Function `DSPF_sp_biquad()` expects to be passed pointers to the following:

- (1) An array of input samples.
- (2) An array containing filter coefficients b_0 , b_1 , and b_2 .
- (3) An array containing filter coefficients a_0 , a_1 , and a_2 .
- (4) An array containing the two values stored in the delay line of the filter structure.
- (5) An array of output samples.

Two buffers, `x` and `y`, each of length `BUFCOUNT/2` are used to hold input and output samples. Pointers `inptr` and `outptr` are used to access these arrays and are passed to the function. Before the function is called for the first time, `inptr` is initialized to point to array `x`, `outptr` is initialized to point to array `y`, and input data is copied into array `x`.

```
#include "DSPF_sp_biquad_cn.h"
void DSPF_sp_biquad_cn(float *x, float *b, float *a,
    float *delay, float *y, const int n)
{
    int i;

    for (i = 0; i < n; i++)
    {
        y[i] = b[0] * x[i] + delay[0];
        delay[0] = b[1] * x[i] - a[1] * y[i] + delay[1];
        delay[1] = b[2] * x[i] - a[2] * y[i];
    }
}
```

Figure 4.36 Listing of `DSPF_sp_biquad_cn.c` (courtesy of Texas Instruments).

```

// L138_iirsoS_DSPLIB_edma.c
//

#include "L138_aic3106_init.h"
#include "dsplib674x.h"

#include "elliptic.cof"

extern int16_t *pingIN, *pingOUT, *pongIN, *pongOUT;
volatile int buffer_full = 0;
int procBuffer;
float delay[NUM_SECTIONS][2] = {0.0};

float x[BUFCOUNT/2];
float y[BUFCOUNT/2];

interrupt void interrupt4(void) // interrupt service routine
{
    switch(EDMA_3CC_IPR)
    {
        case 1:                      // TCC = 0
            procBuffer = PING;       // process ping
            EDMA_3CC_ICR = 0x0001;   // clear EDMA3 IPR bit TCC
            break;
        case 2:                      // TCC = 1
            procBuffer = PONG;      // process pong
            EDMA_3CC_ICR = 0x0002;   // clear EDMA3 IPR bit TCC
            break;
        default:                     // may have missed an interrupt
            EDMA_3CC_ICR = 0x0003;   // clear EDMA3 IPR bits 0 and 1
            break;
    }
    EVTCLR0 = 0x00000100;
    buffer_full = 1;              // flag EDMA3 transfer
    return;
}

void process_buffer(void)
{
    int16_t *inBuf, *outBuf;      // pointers to process buffers
    int i, section;
    float *inptr, *outptr, *tempptr;

    if (procBuffer == PING)       // use ping or pong buffers
    {
        inBuf = pingIN;
        outBuf = pingOUT;
    }
    if (procBuffer == PONG)
    {
        inBuf = pongIN;
        outBuf = pongOUT;
    }
    for (i = 0; i < (BUFCOUNT/2) ; i++)
    {
        x[i] = (float)(*inBuf++);
        inBuf++;

```

Figure 4.37 IIR filter implemented using DSPLIB function DSPF_sp_biquad() (L138_iirsoS_DSPLIB_edma.c).

```

}
inptr = x;
outptr = y;
for(section = 0 ; section < NUM_SECTIONS ; section++)
{
    DSPF_sp_biquad(inptr,&b[section][0],&a[section][0],&delay[section][0],outptr,
(BUFCOUNT/2));
    tempptr = outptr;
    outptr = inptr;
    inptr = tempptr;
}
for (i = 0; i < (BUFCOUNT/2) ; i++)
{
    *outBuf++ = (int16_t)(x[i]);
    *outBuf++ = (int16_t)(x[i]);
}
buffer_full = 0;           // flag buffer processed
return;
}

int main(void)
{
    L138_initialise_edma(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1)
    {
        while (!buffer_full);
        process_buffer();
    }
}
}

```

Figure 4.37 (Continued)

The first time the function is called, the output of the biquad filter is stored in the array pointed to by `outptr`, that is, `y`. The function is called `NUM_SECTIONS` times and after each call, the values of pointers `inptr` and `outptr` are swapped. In this way, the buffer of previously calculated output samples is used as input for the next section. After the function has been called `NUM_SECTIONS` times, and the values of `inptr` and `outptr` have been swapped for the last time, it is pointer `inptr` that points to the filter output samples.

The filter output samples are then copied into the appropriate ping-pong output buffer. Copying values in and out of the ping-pong buffers is relatively inefficient but is necessary in this instance because they contain interleaved left and right channel samples, and function `DSPF_sp_biquad()` processes just one channel.

Change the line in program `L138_iirsos_DSPLIB_edma.c` that reads

```
x[i] = (float) (*inBuf++);
```

to read

```
x[i] = (float) (prbs());
```

in order to use internally generated pseudorandom noise as an input and enable the filter characteristics to be observed using an oscilloscope or *Goldwave* without the need for an externally applied input signal.

EXAMPLE 4.8: Fixed-Point Implementation of an IIR Filter (L138_iir_intr)

Program L138_iir_intr.c, listed in Figure 4.38, implements a generic IIR filter using cascaded second-order sections and fixed-point (integer) coefficients. The program implements each second-order section as a direct form II structure using the following two program statements:

```

wn = input - ((a[section][1]*w[section][0])>>15)
    - ((a[section][2]*w[section][1])>>15);
yn = ((b[section][0]*wn)>>15)
    + ((b[section][1]*w[section][0])>>15)
    + ((b[section][2]*w[section][1])>>15);

// L138_iir_intr.c
//
// IIR filter implemented using second order sections
// integer coefficients read from file
//

#include "L138_aic3106_init.h"
#include "bs1800int.cof"

int w[NUM_SECTIONS][2] = {0};

interrupt void interrupt4() //interrupt service routine
{
    int section;    // index for section number
    int input;      // input to each section
    int wn,yn;      // intermediate and output values

    input = input_left_sample(); // input from ADC
    for (section=0 ; section< NUM_SECTIONS ; section++)
    {
        wn = input - ((a[section][1]*w[section][0])>>15)
            - ((a[section][2]*w[section][1])>>15);
        yn = ((b[section][0]*wn)>>15)
            + ((b[section][1]*w[section][0])>>15)
            + ((b[section][2]*w[section][1])>>15);
        w[section][1] = w[section][0];
        w[section][0] = wn;
        input = yn; // output of current section is input to next
    }
    output_left_sample((int16_t)(yn)); // output to DAC
    return;
}

int main(void)
{
    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);
}

```

Figure 4.38 IIR filter implemented using fixed-point arithmetic (L138_iir_intr.c).

These statements correspond to the equations

$$w(n) = x(n) - a_1 w(n-1) - a_2 w(n-2)$$

and

$$y(n) = b_0 w(n) + b_1 w(n-1) + b_2 y(n-2).$$

The values of the coefficients in the files `bs1800int.cof` and `ellipint.cof` were calculated using MATLAB's `fdatool` and function `L138_iirssos_coeffs_int()`, an integer version of function `L138_iirssos_coeffs()` that multiplies the filter coefficients generated using `fdatool` by 32768 and represents them as 16-bit signed integers. Build and run this program. Verify that an IIR band-stop filter centered at 1800 Hz is implemented, if coefficient file `bs1800int.cof` is used. This program could be used to investigate the effect of quantization of IIR filter coefficients.

EXAMPLE 4.9: Implementation of a Fourth-Order IIR Filter Using AIC3106 Digital Effects Filter (`L138_sysid_biquad_intr`)

The AIC3106 codec contains two fourth-order IIR filters (one for each channel) just before the DAC. Their coefficients can be programmed using page 1 control registers 1–20 (Left Channel Audio Effects Filter Coefficient Registers) and 27–46 (Right Channel Audio Effects Filter Coefficient Registers). They are enabled by setting bit 3 (left channel) and/or bit 1 (right channel) in page 0 control register 12 (Audio Codec Digital Filter Control Register). Each filter is implemented as two second-order (biquad) sections with an overall z -transfer function:

$$H(z) = \left(\frac{N_0 + 2N_1 z^{-1} + N_2 z^{-2}}{32768 - 2D_1 z^{-1} - D_2 z^{-2}} \right) \left(\frac{N_3 + 2N_4 z^{-1} + N_5 z^{-2}}{32768 - 2D_4 z^{-1} - D_5 z^{-2}} \right), \quad (4.43)$$

where coefficients N_i and D_i are 16-bit signed integers. Full details of these filters are given in the AIC3106 data sheet [1].

Program `L138_sysid_biquad_intr.c` is almost identical to program `L138_sysid_intr.c`, introduced in Chapter 2. It uses an adaptive FIR filter in order to measure the response of a signal path including the codec. It differs only in that the codec is programmed to include the fourth-order IIR filter described above.

Board support library (BSL) function `AIC3106_writeRegister()` is used to program the 8-bit control registers of the AIC3106 and each of the 16-bit coefficients of the filters must therefore be split into two 8-bit bytes. MATLAB function `aic3106_biquad()`, listed in Figure 4.39, has been provided to automate the generation of these program statements following calculation of filter coefficients using `fdatool`.

```

% AIC3106_BIQUAD.M
%
% MATLAB function to write C program statements
% to program left and right channel biquads in AIC3106 codec.
% Assumes that coefficients of a fourth order IIR filter
% have been designed using fdatool and exported to workspace
% as two matrices. These are passed to function as coeff and
% gain.
%
% First matrix coeff has format
%
% [ b10 b11 b12 a10 a11 a12
%   b20 b21 b22 a20 a21 a22
% ]
%
% where bij is the bj coefficient in the ith stage.
%
% Second matrix gain contains gains for the two stages of the
% fourth order filter. fdatool generates three gains - one for
% each second order stage and an output gain.
%
% This function calls function l138_make_hex_str() defined in
% file L138_MAKE_HEX_STR.M.
%
% Details of AIC3106 registers in TI document SLAS509E.
%

function aic3106_biquad(coeff,gain)
fname = input('enter filename for C program statements ','s');
fid = fopen(fname,'wt');
fprintf(fid,'// %s\n',fname);
fprintf(fid,'// this file was generated using');
fprintf(fid,' function aic3106_biquad.m\n',fname);
fn_name = 'AIC3106_writeRegister';
% left channel audio effects filter coefficients
coeff(1,1)
gain(1)
round(coeff(1,1)*gain(1)*2^15)
a = l138_make_hex_str(round(coeff(1,1)*gain(1)*(2^15 - 1)));
fprintf(fid,'%s( 1,0x%s);\n',fn_name,a(1),a(2));
fprintf(fid,'%s( 2,0x%s);\n',fn_name,a(3),a(4));
a = l138_make_hex_str(round(coeff(1,2)*gain(1)*2^14));
fprintf(fid,'%s( 3,0x%s);\n',fn_name,a(1),a(2));
fprintf(fid,'%s( 4,0x%s);\n',fn_name,a(3),a(4));
a = l138_make_hex_str(round(coeff(1,3)*gain(1)*(2^15 - 1)));
fprintf(fid,'%s( 5,0x%s);\n',fn_name,a(1),a(2));
fprintf(fid,'%s( 6,0x%s);\n',fn_name,a(3),a(4));
a = l138_make_hex_str(round(coeff(2,1)*gain(2)*(2^15 - 1)));
fprintf(fid,'%s( 7,0x%s);\n',fn_name,a(1),a(2));
fprintf(fid,'%s( 8,0x%s);\n',fn_name,a(3),a(4));
a = l138_make_hex_str(round(coeff(2,2)*gain(2)*2^14));
fprintf(fid,'%s( 9,0x%s);\n',fn_name,a(1),a(2));
fprintf(fid,'%s(10,0x%s);\n',fn_name,a(3),a(4));
a = l138_make_hex_str(round(coeff(2,3)*gain(2)*(2^15 - 1)));
fprintf(fid,'%s(11,0x%s);\n',fn_name,a(1),a(2));
fprintf(fid,'%s(12,0x%s);\n',fn_name,a(3),a(4));
a = l138_make_hex_str(round(-coeff(1,5)*2^14));

```

Figure 4.39 MATLAB function aic3106_biquad().

```

fprintf(fid,'%s(13,0x%s$); \n',fn_name,a(1),a(2));
fprintf(fid,'%s(14,0x%s$); \n',fn_name,a(3),a(4));
a = 1138_make_hex_str(round(-coeff(1,6)*(2^15 - 1)));
fprintf(fid,'%s(15,0x%s$); \n',fn_name,a(1),a(2));
fprintf(fid,'%s(16,0x%s$); \n',fn_name,a(3),a(4));
a = 1138_make_hex_str(round(-coeff(2,5)*2^14));
fprintf(fid,'%s(17,0x%s$); \n',fn_name,a(1),a(2));
fprintf(fid,'%s(18,0x%s$); \n',fn_name,a(3),a(4));
a = 1138_make_hex_str(round(-coeff(2,6)*(2^15 - 1)));
fprintf(fid,'%s(19,0x%s$); \n',fn_name,a(1),a(2));
fprintf(fid,'%s(20,0x%s$); \n',fn_name,a(3),a(4));
% right channel audio effects filter coefficients
a = 1138_make_hex_str(round(coeff(1,1)*gain(1)*(2^15 - 1)));
fprintf(fid,'%s(27,0x%s$); \n',fn_name,a(1),a(2));
fprintf(fid,'%s(28,0x%s$); \n',fn_name,a(3),a(4));
a = 1138_make_hex_str(round(coeff(1,2)*gain(1)*2^14));
fprintf(fid,'%s(29,0x%s$); \n',fn_name,a(1),a(2));
fprintf(fid,'%s(30,0x%s$); \n',fn_name,a(3),a(4));
a = 1138_make_hex_str(round(coeff(1,3)*gain(1)*(2^15 - 1)));
fprintf(fid,'%s(31,0x%s$); \n',fn_name,a(1),a(2));
fprintf(fid,'%s(32,0x%s$); \n',fn_name,a(3),a(4));
a = 1138_make_hex_str(round(coeff(2,1)*gain(2)*(2^15 - 1)));
fprintf(fid,'%s(33,0x%s$); \n',fn_name,a(1),a(2));
fprintf(fid,'%s(34,0x%s$); \n',fn_name,a(3),a(4));
a = 1138_make_hex_str(round(coeff(2,2)*gain(2)*2^14));
fprintf(fid,'%s(35,0x%s$); \n',fn_name,a(1),a(2));
fprintf(fid,'%s(36,0x%s$); \n',fn_name,a(3),a(4));
a = 1138_make_hex_str(round(coeff(2,3)*gain(2)*(2^15 - 1)));
fprintf(fid,'%s(37,0x%s$); \n',fn_name,a(1),a(2));
fprintf(fid,'%s(38,0x%s$); \n',fn_name,a(3),a(4));
a = 1138_make_hex_str(round(-coeff(1,5)*2^14));
fprintf(fid,'%s(39,0x%s$); \n',fn_name,a(1),a(2));
fprintf(fid,'%s(40,0x%s$); \n',fn_name,a(3),a(4));
a = 1138_make_hex_str(round(-coeff(1,6)*(2^15 - 1)));
fprintf(fid,'%s(41,0x%s$); \n',fn_name,a(1),a(2));
fprintf(fid,'%s(42,0x%s$); \n',fn_name,a(3),a(4));
a = 1138_make_hex_str(round(-coeff(2,5)*2^14));
fprintf(fid,'%s(43,0x%s$); \n',fn_name,a(1),a(2));
fprintf(fid,'%s(44,0x%s$); \n',fn_name,a(3),a(4));
a = 1138_make_hex_str(round(-coeff(2,6)*(2^15 - 1)));
fprintf(fid,'%s(45,0x%s$); \n',fn_name,a(1),a(2));
fprintf(fid,'%s(46,0x%s$); \n',fn_name,a(3),a(4));
fclose(fid);

```

Figure 4.39 (Continued)

Connect LINE OUT on the eXperimenter to LINE IN as shown in Figure 4.40 and build and run the program as supplied. Initially, the AIC3106 biquad filters are neither programmed nor enabled. Halt the program after a few seconds and use *Tools > Graph > FFT Magnitude* in order to examine the frequency response of the signal path including DAC, ADC, and LINE OUT and LINE IN AC-coupling circuits. You should see something similar to the graph shown in Figure 4.41.

Figure 4.42 lists the contents of file `elliptic_coeffs.h`. This file was generated using MATLAB function `aic3106_biquad()` to process filter coefficients of a fourth-order

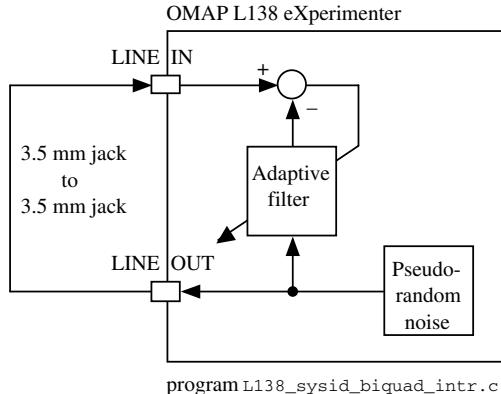


Figure 4.40 Connection diagram for program L138_sysid_biquad_intr.c.

elliptic LPF designed using fdatool. Cut and paste these statements into program L138_sysid_biquad_intr.c just after the statement

```
L138_initialise_intr(FS_8000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
```

Add the statement

```
AIC3106_writeRegister(0,0x01); // select page 1
```

immediately preceding the statements cut and pasted from file elliptic_coeffs.h and add the statements

```
AIC3106_writeRegister(0,0x00); //select page 0
AIC3106_writeRegister(12,0x0A); //enable DAC effects filters
```

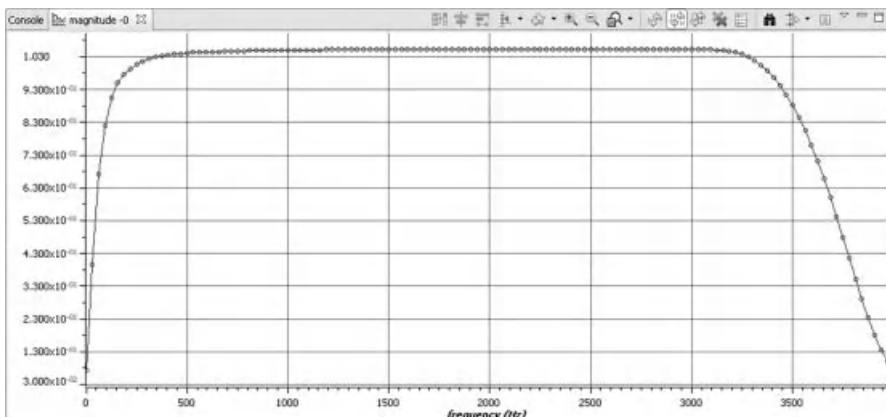


Figure 4.41 Frequency response of signal path through DAC, connecting cable, and ADC shown in Figure 4.40.

```
// elliptic_coefffs.h
// this file was generated using function aic3106_biquad.m
AIC3106_writeRegister( 1,0x01);
AIC3106_writeRegister( 2,0xec);
AIC3106_writeRegister( 3,0x01);
AIC3106_writeRegister( 4,0x1c);
AIC3106_writeRegister( 5,0x01);
AIC3106_writeRegister( 6,0xec);
AIC3106_writeRegister( 7,0x7f);
AIC3106_writeRegister( 8,0xff);
AIC3106_writeRegister( 9,0xea);
AIC3106_writeRegister(10,0xcd);
AIC3106_writeRegister(11,0x7f);
AIC3106_writeRegister(12,0xff);
AIC3106_writeRegister(13,0x59);
AIC3106_writeRegister(14,0xae);
AIC3106_writeRegister(15,0xb7);
AIC3106_writeRegister(16,0x86);
AIC3106_writeRegister(17,0x53);
AIC3106_writeRegister(18,0xa2);
AIC3106_writeRegister(19,0x93);
AIC3106_writeRegister(20,0xf7);
AIC3106_writeRegister(27,0x01);
AIC3106_writeRegister(28,0xec);
AIC3106_writeRegister(29,0x01);
AIC3106_writeRegister(30,0x1c);
AIC3106_writeRegister(31,0x01);
AIC3106_writeRegister(32,0xec);
AIC3106_writeRegister(33,0x7f);
AIC3106_writeRegister(34,0xff);
AIC3106_writeRegister(35,0xea);
AIC3106_writeRegister(36,0xcd);
AIC3106_writeRegister(37,0x7f);
AIC3106_writeRegister(38,0xff);
AIC3106_writeRegister(39,0x59);
AIC3106_writeRegister(40,0xae);
AIC3106_writeRegister(41,0xb7);
AIC3106_writeRegister(42,0x86);
AIC3106_writeRegister(43,0x53);
AIC3106_writeRegister(44,0xa2);
AIC3106_writeRegister(45,0x93);
AIC3106_writeRegister(46,0xf7);
```

Figure 4.42 Listing of file `elliptic_coefffs.h`.

immediately following the previously pasted statements. The last of these statements will enable the biquad filters on both left- and right-hand channels.

Once again, build and load the program and run it for a few seconds. You should now see the magnitude frequency response of the biquad filter in the Code Composer Studio *Graph* window, as shown in Figure 4.43. Figure 4.44 shows the same response plotted after saving the contents of array `w` to a *TI Data Format* file and using MATLAB function `L138_logfft()`.

Changing the response of AIC3106 digital effects filter

In order to program an alternative filter response into the digital effects filter,

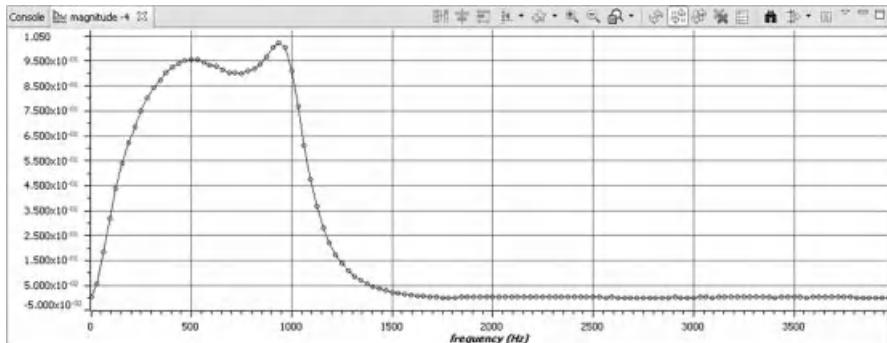


Figure 4.43 Frequency response of signal path through DAC, connecting cable and ADC shown in Figure 4.40 with biquad filter programmed as an elliptic low-pass filter and enabled.

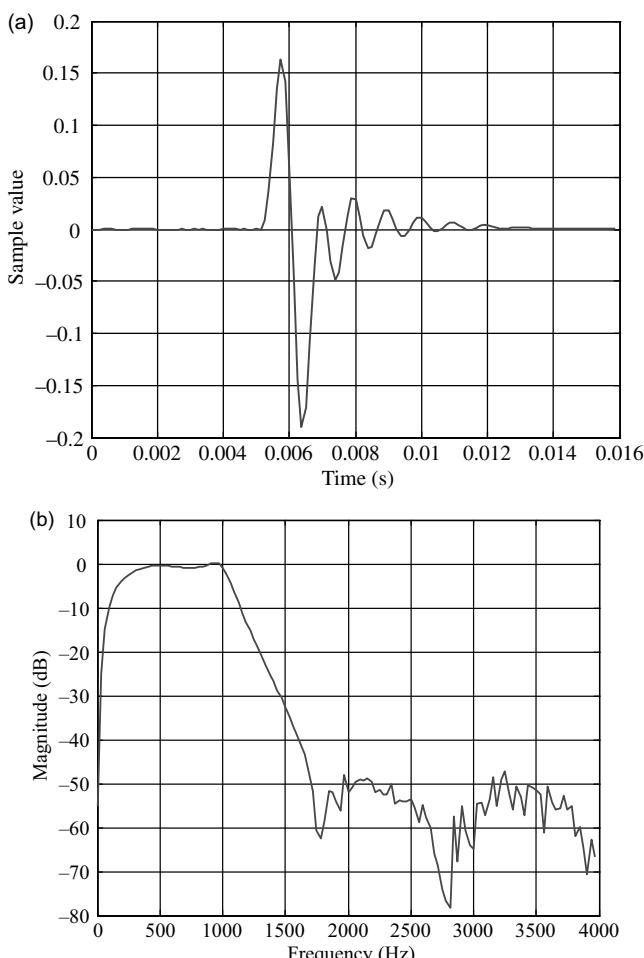


Figure 4.44 Impulse response and magnitude frequency response of signal path through DAC, connecting cable, and ADC shown in Figure 4.40 with AIC3106 biquad filter programmed as an elliptic low-pass filter and enabled, plotted using MATLAB function `L138_logfft()`.

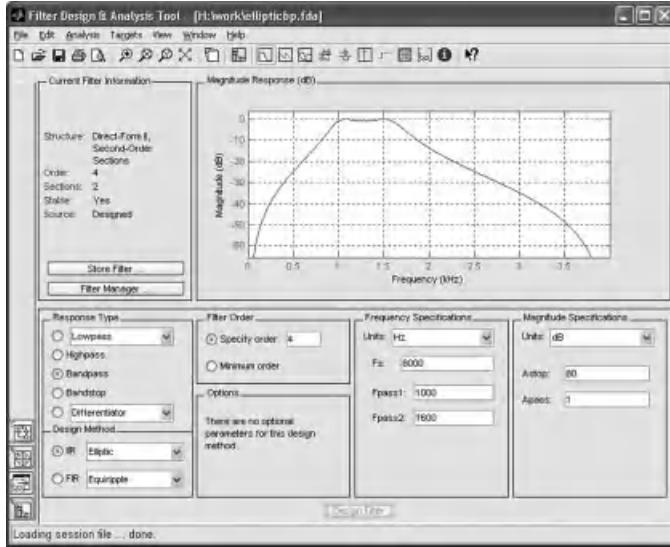


Figure 4.45 fdatool used to design a fourth-order elliptic band-pass filter.

- (1) Design an alternative fourth-order IIR filter using fdatool. Figure 4.45 shows the design of an elliptic band-pass filter.
- (2) Export the filter coefficients from fdatool to the MATLAB workspace as variables SOS and G.
- (3) At the MATLAB command line, type `aic3106_biquad(SOS, G)` and enter a filename, for example, `bandpass_coefficients.h`.

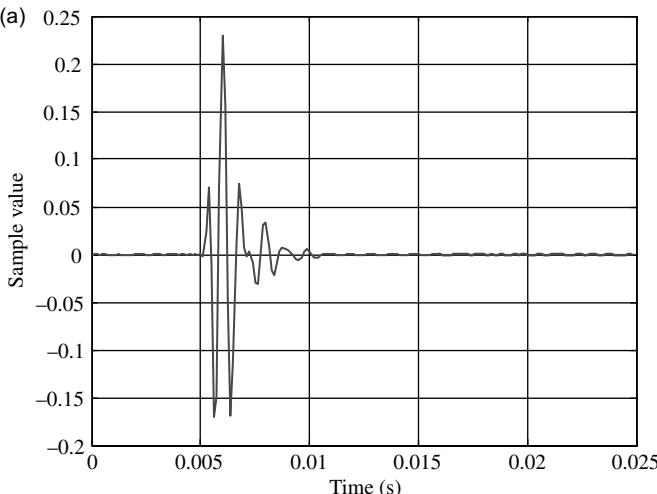
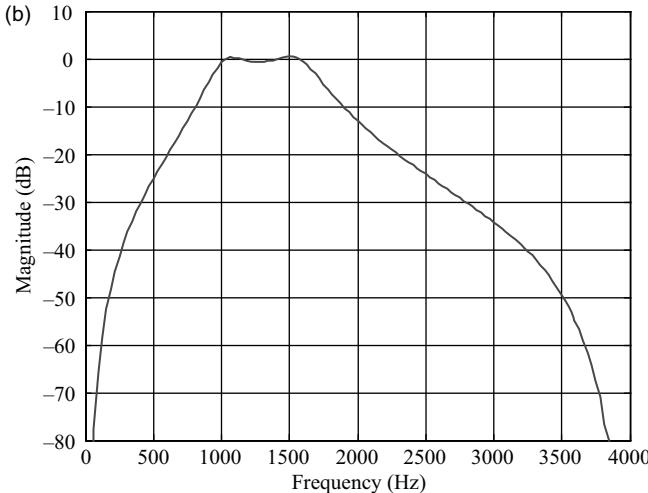


Figure 4.46 Impulse response and magnitude frequency response of signal path through DAC, connecting cable, and ADC shown in Figure 4.40 with AIC3106 biquad filter programmed as an elliptic band-pass filter and enabled, plotted using MATLAB function `L138_logfft()`.

**Figure 4.46 (Continued)**

- (4) Cut and paste the statements contained in file `bandpass_coeffs.h` into program `L138_sysid_biquad_intr.c` (in place of those used in the previous example) immediately following the call to function `L138_initialise_intr()` and immediately preceding the statement `AIC3106_writeRegister(12, 0x0A);`.
- (5) Build and run the program as before.
- (6) Observe the response identified using either *Tools > Graph > FFT Magnitude* in the Code Composer Studio IDE or MATLAB function `L138_logfft()` to plot the values stored in array `w`.

Figure 4.46 shows the identified frequency response for the coefficients contained in file `bandpass_coeffs.h` using MATLAB function `L138_lofft()`.

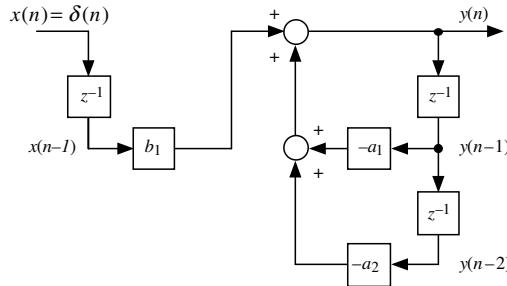
EXAMPLE 4.10: Generation of a Sine Wave Using a Difference Equation (L138_sinegenDE_intr)

In Chapter 3, it was shown that the z -transform of a sinusoidal sequence $y(n) = \sin(n\omega T)$ is given by

$$Y(z) = \frac{z \sin(\omega T)}{z^2 - 2 \cos(\omega T)z + 1}. \quad (4.44)$$

Comparing this with the z -transfer function of the second-order filter of Example 4.1,

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_1 z}{z^2 + a_1 z + a_2}. \quad (4.45)$$

**Figure 4.47** Block diagram representation of (4.47).

It is apparent that by appropriate choice of filter coefficients, we can configure that filter to act as a sine wave generator, that is, to have a sinusoidal impulse response.

Choosing $a_2 = 1.0$ and $a_1 = -2 \cos(\omega T)$, the denominator of the transfer function becomes $z^2 - 2 \cos(\omega T)z + 1$, which corresponds to a pair of complex conjugate poles located *on* the unit circle in the z -plane. The filter can be set oscillating by applying an impulse to its input. Rearranging Equation 4.44 and setting $x(n) = \delta(n)$, ($X(z) = 1.0$), and $b_1 = \sin(\omega T)$,

$$Y(z) = \frac{X(z)b_1 z}{z^2 - 2 \cos(\omega T)z + 1} = \frac{\sin(\omega T)z}{z^2 - 2 \cos(\omega T)z + 1}. \quad (4.46)$$

Equation 4.46 is equivalent to Equation 4.44, implying that the filter impulse response is $y(n) = \sin(n\omega T)$. Equation 4.46 corresponds to the difference equation

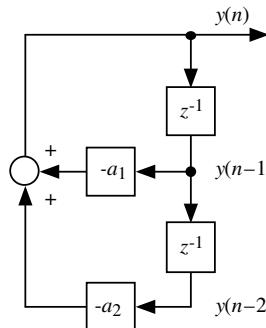
$$y(n) = \sin(\omega T)x(n-1) + 2 \cos(\omega T)y(n-1) - y(n-2), \quad (4.47)$$

which is illustrated in Figure 4.47.

Since the input $x(n) = \delta(n)$ to the filter is nonzero only at sampling instant $n = 0$, for all other n , the difference equation is

$$y(n) = 2 \cos(\omega T)y(n-1) - y(n-2) \quad (4.48)$$

and the sine wave generator may be implemented as shown in Figure 4.48, using no input signal but using nonzero initial values for $y(n-1)$ and $y(n-2)$. The initial values used determine the amplitude of the sinusoidal output.

**Figure 4.48** Block diagram representation of (4.48).

```

// L138_sinegenDE_intr.c
//

#include "L138_aic3106_init.h"
#include <math.h>
#define FREQ 2000
#define SAMPLING_FREQ 8000
#define AMPLITUDE 8000
#define PI 3.14159265358979

float y[3];
float a1;

interrupt void interrupt4(void) // interrupt service routine
{
    y[0] = -(y[1]*a1)-y[2];           // new y(n)
    y[2] = y[1];                     // update y(n-2)
    y[1] = y[0];                     // update y(n-1)
    output_left_sample((int16_t)(y[0])); // output to L DAC
    return;
}

int main(void)
{
    y[1] = 0.0;
    y[2] = AMPLITUDE*sin(2.0*PI*FREQ/SAMPLING_FREQ); // amplitude
    a1 = -2.0*cos(2.0*PI*FREQ/SAMPLING_FREQ);          // frequency

    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);
}

```

Figure 4.49 Program to generate a sine wave using a difference equation (L138_sinegenDE_intr.c).

Since the frequency of oscillation ω is fixed by the choice of $a_1 = -2 \cos(\omega T)$ and $a_2 = 1$, the initial values chosen for $y(n-1)$ and $y(n-2)$ represent two samples of a sinusoid of frequency ω , that are one sampling period or T seconds apart in time, that is,

$$\begin{aligned} y(n-1) &= A \sin(\omega t + \phi), \\ y(n-2) &= A \sin(\omega(t+T) + \phi). \end{aligned}$$

The initial values of $y(n-1)$ and $y(n-2)$ determine the amplitude A of the sine wave generated. A simple solution to the equations, implemented in program L138_sinegenDE_intr.c, is

$$\begin{aligned} y(n-1) &= 0, \\ y(n-2) &= A \sin(\omega T). \end{aligned}$$

Build and run this program as supplied ($\text{FREQ} = 2000$) and verify that the output is a 2000 Hz tone (Figure 4.49). Change the value of the constant FREQ , build and run the program, and verify the generation of a tone of the frequency selected.

**EXAMPLE 4.11: Generation of DTMF Signal Using Difference Equations
(L138_sinegenDTMF_intr)**

```
// L138_sinegenDTMF_intr.c
//

#include "L138_aic3106_init.h"
#include "math.h"
#include "evmomap1138_dip.h"
#define FREQLO 770
#define FREQHI 1336
#define SAMPLING_FREQ 8000
#define AMPLITUDE 6000
#define BUFSIZE 256
#define PI 3.14159265358979

float ylo[3];
float yhi[3];
float allo, alhi;
float out_buffer[BUFSIZE];
int bufindex = 0;
float output;
volatile short DIP1_ON = 0;

interrupt void interrupt4(void) // interrupt service routine
{
    ylo[0] = -(ylo[1]*allo)-ylo[2]; // update y1(n-2)
    ylo[2] = ylo[1]; // update y1(n-1)
    ylo[1] = ylo[0]; // update y1(n-1)
    yhi[0] = -(yhi[1]*alhi)-yhi[2]; // update y1(n-2)
    yhi[2] = yhi[1]; // update y1(n-1)
    yhi[1] = yhi[0]; // update y1(n-1)
    output = (yhi[0]+ylo[0]);
    out_buffer[bufindex++] = output;
    if (bufindex >= BUFSIZE) bufindex = 0;
    if (DIP1_ON) output_left_sample((int16_t)(output));
    else output_left_sample(0); // output to L DAC
    return;
}

int main(void)
{
    int32_t rtn;
    uint8_t dip_val;

    ylo[1] = 0.0;
    ylo[2] = AMPLITUDE*sin(2.0*PI*FREQLO/SAMPLING_FREQ);
    allo = -2.0*cos(2.0*PI*FREQLO/SAMPLING_FREQ);
    yhi[1] = 0.0;
    yhi[2] = AMPLITUDE*sin(2.0*PI*FREQHI/SAMPLING_FREQ);
    alhi = -2.0*cos(2.0*PI*FREQHI/SAMPLING_FREQ);
    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    rtn = DIP_init();
    while(1)
    {
        rtn = DIP_get(0,&dip_val);
        if (dip_val) DIP1_ON = 1; else DIP1_ON = 0;
    }
}
```

Figure 4.50 Program to generate DTMF tone using difference equations (L138_sinegenDTMF_intr.c).

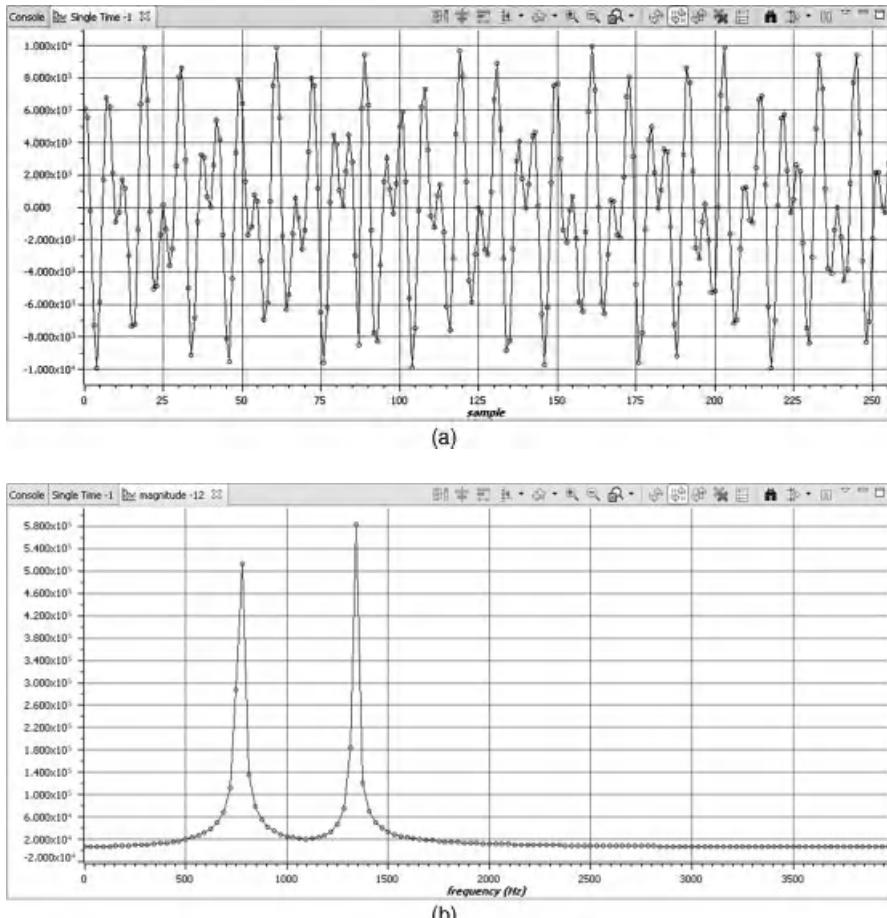


Figure 4.51 (a) Two hundred and fifty-six samples of the waveform generated by program L138_sinegenDTMF_intr.c. (b) FFT magnitude of those samples displayed using Code Composer Studio software.

Program L138_sinegenDTMF_intr.c, listed in Figure 4.50, uses the same difference equation method as program L138_sinegenDE_intr.c to generate two sinusoidal signals of different frequencies, which, added together, form a DTMF tone (see also Example 2.12 which used a lookup table method). The DTMF tone is output via the codec only while DIP switch #1 is ON. The program also incorporates a buffer (array `out_buffer`) that is used to store the 256 most recent output samples. Figure 4.51 shows the contents of that buffer in time and frequency domains, plotted using *Tools > Graph > Single Time* and *Tools > Graph > FFT Magnitude* in the Code Composer Studio IDE, and Figure 4.52 shows the output from the program captured using an oscilloscope.

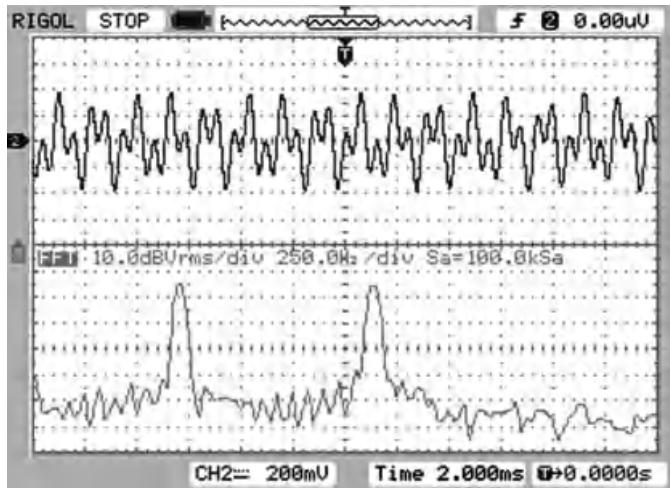


Figure 4.52 Output waveform generated by program L138_sinegenDTMF_intr.c displayed using a *Rigol DS1052E* oscilloscope.

**EXAMPLE 4.12: Generation of a Swept Sinusoid Using a Difference Equation
(L138_sweepDE_intr)**

Figure 4.53 shows a listing of the program L138_sweepDE_intr.c that generates a sinusoidal signal, sweeping in frequency. The program implements the difference equation

$$y(n) = -a_1 y(n-1) - y(n-2),$$

where $a_1 = -2 \cos(\omega T)$ and the initial conditions are $y(n-1) = 0$ and $y(n-2) = \sin(\omega T)$. Example 4.9 illustrated the generation of a sine wave using this difference equation.

Compared to the lookup table method of Example 2.15, making step changes in the frequency of the output signal generated using a difference equation is slightly more problematic. Each time program L138_sweepDE_intr.c changes its output frequency, it reinitializes the stored values of previous output samples $y(n-1)$ and $y(n-2)$. These values determine the amplitude of the sinusoidal output at the new frequency and must be chosen appropriately. Using the existing values, leftover from the generation of a sinusoid at the previous frequency, might cause the amplitude of the output sinusoid to change. In order to avoid discontinuities, or glitches, in the output waveform, a further constraint on the parameters of the program must be observed. Since at each change in frequency the output waveform starts at the same phase in its cycle, it is necessary to ensure that each different frequency segment is output for an integer number of cycles. This can be achieved by making the number of samples output between step changes in frequency equal to the sampling frequency divided by the frequency increment. As listed in Figure 4.53, the frequency increment is 20 Hz and the sampling frequency is 8000 Hz. Hence, the number of samples output at each different frequency is equal to $8000/20 = 400$. Different choices for the values of the constants STEP_FREQ and SWEEP_PERIOD are possible.

Build and run this program. Verify that the output is a swept sinusoidal signal starting at frequency 200 Hz and taking $(\text{SWEEP_PERIOD}/\text{SAMPLING_FREQ})^*(\text{MAX_}$

```

// L138_sweepDE_intr.c
//

#include "L138_aic3106_init.h"

#include <math.h>
#define MIN_FREQ 200
#define MAX_FREQ 3800
#define STEP_FREQ 20
#define SWEEP_PERIOD 400
#define SAMPLING_FREQ 8000
#define AMPLITUDE 10000
#define PI 3.14159265358979

float y[3] = {0.0, 0.0, 0.0};
float a1;
float freq = MIN_FREQ;
short sweep_count = 0;

void coeff_gen(float freq)
{
    a1 = -2.0*cos(2.0*PI*freq/SAMPLING_FREQ);
    y[0] = 0.0;
    y[2] = AMPLITUDE*sin(2.0*PI*freq/SAMPLING_FREQ);
    y[1] = 0.0;
    return;
}

interrupt void interrupt4() // interrupt service routine
{
    sweep_count++;
    if (sweep_count >= SWEEP_PERIOD)
    {
        if (freq >= MAX_FREQ) freq = MIN_FREQ;
        else freq += STEP_FREQ;
        coeff_gen(freq);
        sweep_count = 0;
    }
    y[0] = -(y[1]*a1)-y[2];                                // update y1(n-2)
    y[2] = y[1];                                         // update y1(n-1)
    y[1] = y[0];
    output_left_sample((int16_t)(y[0])); // output to L DAC
    return;
}

int main(void)
{
    coeff_gen(freq);
    L138_initialise_intr(FS_8000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
    while(1);
}

```

Figure 4.53 Program to generate a sweeping sinusoid using a difference equation (L138_sweepDE_intr.c).

FREQ-MIN_FREQ) /STEP_FREQ seconds to increase in frequency to 3800 Hz. Change the values of START_FREQ and STOP_FREQ to 2000 and 3000, respectively. Build the project again, load and run the program, and verify that the frequency sweep is from 2000 to 3000 Hz.

EXAMPLE 4.13: Sine Generation Using a Difference Equation with C Calling an ASM Function (`L138_sinegencasm_intr`)

This example is based on Example 5.8 but uses an assembly language function to generate a sine wave using a difference equation. Program `L138_sinegencasm_intr.c`, listed in Figure 4.54, calls the assembly language function `sinegencasmfunc()`, defined in file

```
// L138_.sinegencasm_intr.c
//

#include "L138_aic3106_init.h"

int16_t y[3] = {0, 15137, 11585};
int16_t a1 = 12540;

interrupt void interrupt4(void) // interrupt service routine
{
    sinegencasmfunc(&y[0], a1); // call ASM function
    output_left_sample(y[0]); // output to L DAC
    return;
}

void main()
{
    L138_initialise_intr(FS_8000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
    while(1);
}
```

Figure 4.54 C source program that calls an ASM function to generate a sine wave using a difference equation (`sinegencasm.c`).

```
;Sinegencasmfunc.asm ASM func to generate sine using DE
;A4 = address of y array, B4 = A

.def _sinegencasmfunc ;ASM function called from C
_sinegencasmfunc:
    LDH    *+A4[0], A5      ;y[n-2]-->A5
    LDH    *+A4[1], A2      ;y[n-1]-->A2
    LDH    *+A4[2], A3      ;y[n]-->A3
    NOP    3                ;NOP due to LDH
    MPY    B4, A2, A8      ;A*y[n-1]
    NOP    1                ;NOP due to MPY
    SHR    A8, 14, A8      ;shift right by 14
    SUB    A8, A5, A8      ;A*y[n-1]-y[n-2]
    STH    A8, *+A4[2]      ;y[n]=A*y[n-1]-y[n-2]
    STH    A2, *+A4[0]      ;y[n-2]=y[n-1]
    STH    A8, *+A4[1]      ;y[n-1] = y[n]
    B     B3                ;return addr to calling routine
    NOP    5                ;delays to to branching
.end
```

Figure 4.55 ASM function called from C to generate a sine wave using a difference equation (`sinegencasmfunc.asm`).

`sinegencasmfunc.asm`. The C source program shows the array `y`, which is initialized to contain the values $y(0)$, $y(1)$, and $y(2)$, and the coefficient $a_1 = -2 \cos(\omega T)$, calculated to generate a 1500 Hz sine wave. The address of the array `y` and the value of the coefficient `a1` are passed to the ASM function using registers A4 and B4, respectively. The values in the array `y` and the coefficient `a1` were scaled by 16,384 to allow a fixed-point implementation. As a result, within the ASM function, A8 initially containing $a_1 y(n - 1)$ is shifted right 14 bits.

Build and run the program and verify that a 1500 Hz sine wave is generated.

REFERENCE

1. *Low-Power Stereo Audio Codec for Portable Audio/Telephony*, SLAS509E, Texas Instruments, Dallas, TX, 2008.

Chapter 5

Fast Fourier Transform

- The fast Fourier transform (FFT) using radix-2 and radix-4
- Decomposition in time and in frequency
- Program examples using C code
- Frame-based processing

5.1 INTRODUCTION

Fourier analysis describes the transformations between time and frequency domain representations of signals. Four different forms of Fourier transformation (the Fourier transform (FT), the Fourier Series (FS), the discrete-time Fourier transform (DTFT) and the discrete Fourier transform (DFT)) are applicable to different classes of signal according to whether, in either domain, they are discrete or continuous and whether they are periodic or aperiodic. The DFT is the form of Fourier analysis applicable to signals that are discrete and periodic in both domains, that is, it transforms a discrete, periodic, time domain sequence into a discrete, periodic, frequency domain representation. A periodic signal may be characterized entirely by just one cycle; if that signal is discrete, then one cycle comprises a finite number of samples. Hence, both forward and inverse DFTs are described by finite summations as opposed to infinite summations or integrals. This is very important in digital signal processing since it means that it is practical to compute the DFT using a digital signal processor or digital hardware.

The fast Fourier transform is a computationally efficient algorithm for computing the DFT. It requires fewer multiplications than a more straightforward programming implementation of the DFT and its relative advantage in this respect increases with the length of the sample sequences involved. The FFT makes use of the periodic nature of twiddle factors and of symmetries in the structure of the DFT expression. Applicable to spectrum analysis and to filtering, the FFT is one of the most commonly used operations in digital signal processing.

Various, slightly different, versions of the FFT can be derived from the DFT and in this chapter the decimation-in-time (DIT) and decimation-in-frequency (DIF) radix-2 and radix-4 versions are described in detail.

These versions of the FFT differ in the exact form of the intermediate computations that make them up. However, ignoring rounding errors, they each produce exactly the same results as the DFT. In this respect, the terms DFT and FFT are interchangeable.

5.2 DEVELOPMENT OF THE FFT ALGORITHM WITH RADIX-2

The N -point complex DFT of a discrete-time signal $x(n)$ is

$$X_N(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}. \quad (5.1)$$

The constants W are referred to as *twiddle constants* or *twiddle factors*, where

$$W_N = e^{-j(2\pi/N)}. \quad (5.2)$$

Computing all N values of $X_N(k)$ involves the evaluation of N^2 terms of the form $x(n)W_N^{kn}$, each of which (apart from the requirement of computing W_N^{kn}) requires a complex multiplication. For larger N , the computational requirements of the DFT can be very great (N^2 complex multiplications).

The FFT algorithm takes advantage of the periodicity

$$W^{k+N} = W^k \quad (5.3)$$

and symmetry

$$W^{k+N/2} = -W^k \quad (5.4)$$

of W_N (where N is even).

Figure 5.1 illustrates the twiddle factors W_N^{kn} for $N = 8$ plotted as vectors in the complex plane. Due to the periodicity of W_N^{kn} , the 64 different combinations of n and k used in evaluation of Equation 5.1 result in only 8 distinct values for W_N^{kn} . The FFT makes use of this small number of precomputed and stored values of W_N^{kn} rather than computing each one as it is required. Furthermore, due to the symmetry of W_N^{kn} , only $N/2 = 4$ distinct numerical values need actually be precomputed and stored.

Another way in which the FFT saves computational effort is by decomposing N -point DFTs into combinations of $N/2$ -point DFTs. In the case of radix-2, that is, where N is an integer power of 2, further decomposition of $N/2$ -point DFTs into combinations of $N/4$ -point DFTs, and so on, can be carried out until 2-point DFTs have been reached. Computation of the 2-point DFT does not involve multiplication since the twiddle factors involved are equal to ± 1 yielding $X_2(0) = x(0) + x(1)$ and $X_2(1) = x(0) - x(1)$.

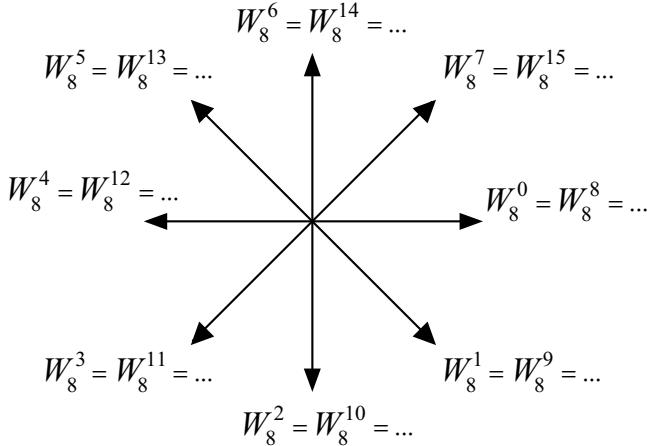


Figure 5.1 Twiddle factors W_N^{kn} for $N = 8$ represented as vectors in the complex plane.

5.3 DECIMATION-IN-FREQUENCY FFT ALGORITHM WITH RADIX-2

Consider the N -point DFT

$$X_N(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \quad (5.5)$$

where

$$W_N = e^{-j(2\pi/N)}, \quad (5.6)$$

and let N be an integer power of 2.

Decompose the length N input sequence $x(n)$ into two length $N/2$ sequences, one containing the first $N/2$ values $x(0), x(1), \dots, x(N/2 - 1)$ and the other containing the remaining $N/2$ values $x(N/2), x(N/2 + 1), \dots, x(N - 1)$. Splitting the DFT summation into two parts

$$\begin{aligned} X_N(k) &= \sum_{n=0}^{(N/2)-1} x(n) W_N^{nk} + \sum_{n=0}^{(N/2)-1} x\left(n + \frac{N}{2}\right) W_N^{(n+(N/2))k} \\ &= \sum_{n=0}^{(N/2)-1} x(n) W_N^{nk} + W_N^{(N/2)k} \sum_{n=0}^{(N/2)-1} x\left(n + \frac{N}{2}\right) W_N^{nk}. \end{aligned} \quad (5.7)$$

This may be viewed as the $N/2$ -point DFT of the first $N/2$ input values added to the $N/2$ -point DFT of the remaining $N/2$ input values, multiplied by $W_N^{(N/2)k}$. The term $W_N^{(N/2)k}$ is placed outside the second summation since it is not a function of n .

Making use of

$$W_N^{(N/2)k} = e^{-j\pi k} = (-1)^k, \quad (5.8)$$

Equation 5.7 becomes

$$X_N(k) = \sum_{n=0}^{(N/2)-1} \left[x(n) + (-1)^k x\left(n + \frac{N}{2}\right) \right] W_N^{nk}. \quad (5.9)$$

Because $(-1)^k = 1$ for even k and $(-1)^k = -1$ for odd k , Equation 5.9 can be split into two parts:

$$\text{For even } k, X_N(k) = \sum_{n=0}^{(N/2)-1} \left[x(n) + x\left(n + \frac{N}{2}\right) \right] W_N^{nk}. \quad (5.10)$$

$$\text{For odd } k, X_N(k) = \sum_{n=0}^{(N/2)-1} \left[x(n) - x\left(n + \frac{N}{2}\right) \right] W_N^{nk}. \quad (5.11)$$

For $k = 0, 1, \dots, ((N/2) - 1)$, that is, for an $N/2$ -point sequence, and making use of

$$W_{\frac{N}{2}}^{kn} = e^{-j((2\pi 2nk)/N)} = W_N^{2nk}, \quad (5.12)$$

$$\begin{aligned} X_N(2k) &= \sum_{n=0}^{(N/2)-1} \left[x(n) + x\left(n + \frac{N}{2}\right) \right] W_N^{2nk} \\ &= \sum_{n=0}^{(N/2)-1} \left[x(n) + x\left(n + \frac{N}{2}\right) \right] W_{(N/2)}^{nk}, \end{aligned} \quad (5.13)$$

and

$$\begin{aligned} X_N(2k+1) &= \sum_{n=0}^{(N/2)-1} \left[x(n) - x\left(n + \frac{N}{2}\right) \right] W_N^{(2k+1)n} \\ &= \sum_{n=0}^{(N/2)-1} \left[x(n) - x\left(n + \frac{N}{2}\right) \right] W_{(N/2)}^{nk} W_N^n. \end{aligned} \quad (5.14)$$

Letting

$$a(n) = x(n) + x\left(n + \frac{N}{2}\right) \quad (5.15)$$

and

$$b(n) = x(n) - x\left(n + \frac{N}{2}\right), \quad (5.16)$$

Equations 5.13 and 5.14 may be written as

$$X_N(2k) = \sum_{n=0}^{(N/2)-1} a(n) W_{(N/2)}^{nk} \quad (5.17)$$

and

$$X_N(2k + 1) = \sum_{n=0}^{(N/2)-1} b(n) W_N^n W_{(N/2)}^{nk}. \quad (5.18)$$

In other words, the even elements of $X_N(k)$ are given by the $N/2$ -point DFT of $a(n)$, where $a(n)$ are combinations of the elements of the N -point input sequence $x(n)$. The odd elements of $X_N(k)$ are given by the $N/2$ -point DFT of $b(n)W_N^n$, where $b(n)$ are combinations of the elements of the N -point input sequence $x(n)$.

This structure is illustrated graphically, for $N = 8$, in Figure 5.2.

Consider next that each of the 4-point DFTs in Figure 5.2 may further be decomposed into two 2-point DFTs (Figure 5.3). Each of these 2-point DFTs may

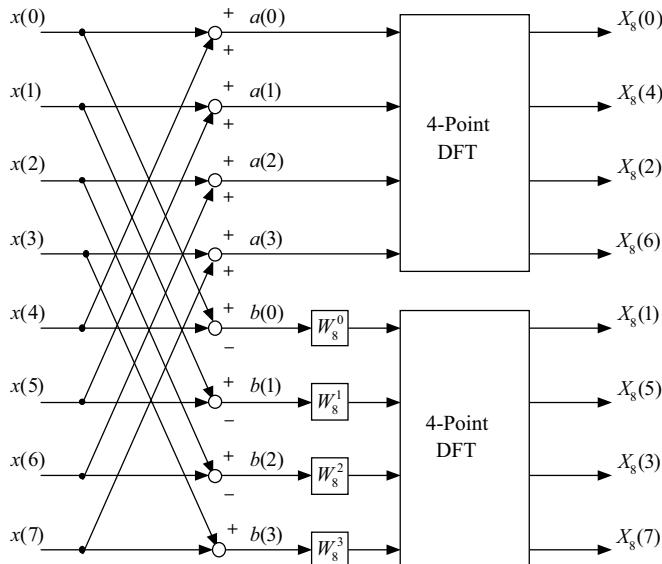


Figure 5.2 Decomposition of 8-point DFT into two 4-point DFTs using decimation-in-frequency.

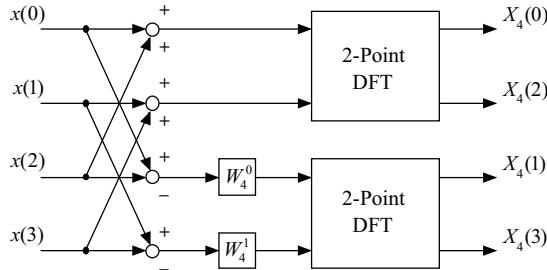


Figure 5.3 Decomposition of 4-point DFT into two 2-point DFTs using decimation-in-frequency.

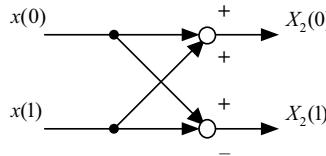


Figure 5.4 2-Point FFT butterfly structure.

further be decomposed into two 1-point DFTs. However, the 1-point DFT of a single value is equal to that value itself and for this reason, the 2-point DFT *butterfly* structure shown in Figure 5.4 is generally treated as the smallest subpart of the FFT structure.

The 8-point DFT has thus been decomposed into a structure (Figure 5.5) that comprises only a small number of multiplications (by factors other than ± 1). The FFT

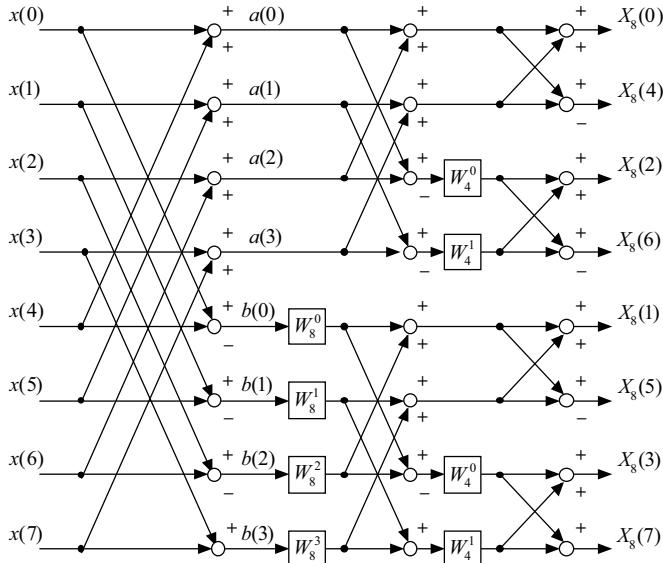


Figure 5.5 Block diagram representation of 8-point FFT using decimation-in-frequency with radix-2.

is not an approximation of the DFT. It yields the same result as the DFT with fewer computations required. This reduction becomes more and more important, and advantageous, with higher order FFTs.

The decimation-in-frequency process may be considered as taking the N -point input sequence $x(n)$ in sequence and reordering the output sequence $X_N(k)$ in pairs, corresponding to the outputs of the final stage of $N/2$ 2-point DFT blocks.

5.4 DECIMATION-IN-TIME FFT ALGORITHM WITH RADIX-2

Whereas the DIF process decomposes the DFT output sequence $X_N(k)$ into a set of shorter subsequences, decimation-in-time is a process that decomposes the DFT input sequence $x(n)$ into a set of shorter subsequences.

Consider again the N -point DFT:

$$X_N(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \quad (5.19)$$

where

$$W_N^{kn} = e^{-j(2\pi/N)}, \quad (5.20)$$

and let N be an integer power of 2.

Decompose the length N input sequence $x(n)$ into two length $N/2$ sequences, one containing the even-indexed values $x[0], x[2], \dots, x[N - 2]$ and the other containing the odd-indexed values $x[1], x[3], \dots, x[N - 1]$. Splitting the DFT summation into two parts,

$$\begin{aligned} X_N(k) &= \sum_{n=0}^{(N/2)-1} x(2n) W_N^{2nk} + \sum_{n=0}^{(N/2)-1} x(2n+1) W_N^{(2n+1)k} \\ &= \sum_{n=0}^{(N/2)-1} x(2n) W_{(N/2)}^{nk} + W_N^k \sum_{n=0}^{(N/2)-1} x(2n+1) W_{(N/2)}^{nk}, \end{aligned} \quad (5.21)$$

since

$$W_N^{2kn} = W_{(N/2)}^{kn}$$

and

$$W_N^{(2n+1)k} = W_N^k W_N^{2nk}.$$

Letting

$$C(k) = \sum_{n=0}^{(N/2)-1} x(2n) W_{(N/2)}^{nk} \quad (5.22)$$

and

$$D(k) = \sum_{n=0}^{(N/2)-1} x(2n+1) W_{(N/2)}^{nk}, \quad (5.23)$$

Equation 5.21 may be written as

$$X_N(k) = C(k) + W_N^k D(k). \quad (5.24)$$

However, $(N/2)$ -point DFTs $C(k)$ and $D(k)$ are defined only for $0 \leq k < (N/2)$, whereas N -point DFT $X_N(k)$ is defined for $0 \leq k < N$.

$C(k)$ and $D(k)$ must be evaluated for $0 \leq k < (N/2)$ and $C(k + (N/2))$ and $D(k + (N/2))$ must be evaluated for $0 \leq k < (N/2)$.

Substituting $(k + (N/2))$ for k in the definition of $C(k)$,

$$C(k + (N/2)) = \sum_{n=0}^{(N/2)-1} x(2n) W_{(N/2)}^{n(k+(N/2))}. \quad (5.25)$$

Since

$$W_{(N/2)}^{n(k+(N/2))} = W_{(N/2)}^{nk} \quad (5.26)$$

and

$$W_N^{n(k+(N/2))} = -W_N^{nk}, \quad (5.27)$$

Equation 5.24 becomes

$$\begin{aligned} X_N(k) &= C(k) + W_N^k D(k), \\ X_N(k + (N/2)) &= C(k) - W_N^k D(k), \end{aligned} \quad (5.28)$$

evaluated for $0 \leq k < (N/2)$.

In other words, the N -point DFT of input sequence $x(n)$ is decomposed into two $(N/2)$ -point DFTs ($C(k)$ and $D(k)$) of the even- and odd-indexed elements of $x(n)$, the results of which are combined in weighted sums to yield N -point output sequence $X_N(k)$. This is illustrated graphically, for $N = 8$, in Figure 5.6.

Consider next that each of the 4-point DFTs in that figure may further be decomposed into two 2-point DFTs (Figure 5.3) and that each of these 2-point DFTs is the 2-point FFT *butterfly* structure shown in Figure 5.4 and used in the decimation-in-frequency approach (Figure 5.7).

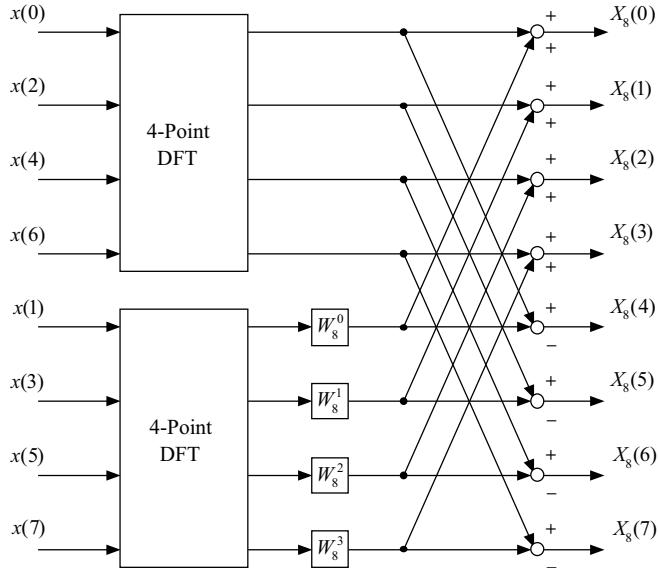


Figure 5.6 Decomposition of 8-point DFT into two 4-point DFTs using decimation-in-time with radix-2.

The 8-point DFT has thus been decomposed into a structure (Figure 5.8) that comprises only a small number of multiplications (by factors other than ± 1).

The decimation-in-time approach may be considered as using pairs of elements from a reordered N -point input sequence $x(n)$ as inputs to $N/2$ 2-point DFT blocks and producing a correctly ordered N -point output sequence $X_N(k)$.

5.4.1 Reordered Sequences in the Radix-2 FFT and Bit-Reversed Addressing

The reordered input sequence $x(n)$ in the case of the DIT approach and the reordered output sequence $X_N(k)$ in the case of DIF can be described with reference to *bit-reversed addressing*.

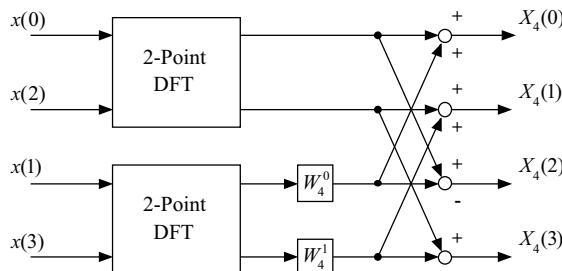


Figure 5.7 Decomposition of 4-point DFT into two 2-point DFTs, as part of 8-point DFT, using decimation-in-time with radix-2.

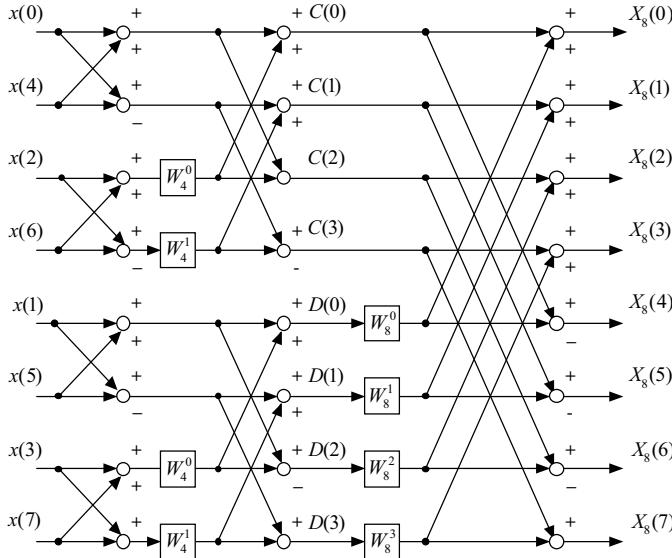


Figure 5.8 Block diagram representation of 8-point FFT using decimation-in-time with radix-2.

Taking as an example the reordered length N sequence $x(n)$ in the case of DIT, the index of each sample may be represented by a $\log_2(N)$ bit binary number (recall that in the foregoing examples, N is an integer power of 2). If the binary representations of the N index values 0 to $(N - 1)$ have the order of their bits reversed, for example, 001 becomes 100, 110 becomes 011, and so on, then the resulting N values represent the indices of the input sequence $x(n)$ in the order that they appear at the input to the FFT structure.

In the case of $N = 8$ (Figure 5.6), the input values $x(n)$ are ordered $x(0), x(4), x(2), x(6), x(1), x(5), x(3), x(7)$. The binary representations of these indices are 000, 100, 010, 110, 001, 101, 011, 111. These are the bit-reversed versions of the sequence 000, 001, 010, 011, 100, 101, 110, 111. The bit-reversed interpretation of the reordering holds for all N (that are integer powers of 2).

5.5 DECIMATION-IN-FREQUENCY FFT ALGORITHM WITH RADIX-4

Radix-4 decomposition of the DFT is possible if N is an integer power of 4. If $N = 4^M$, then the decomposition will comprise M stages. The smallest DFT block that will appear in the structure is a 4-point DFT. Both decimation-in-frequency and decimation-in-time approaches are possible.

In the decimation-in-frequency approach, the input sequence and the N -point DFT are split into four sections such that

$$\begin{aligned} X_N(k) = & \sum_{n=0}^{(N/4)-1} x(n) W_N^{nk} + W_N^{kN/4} \sum_{n=0}^{(N/4)-1} x\left(n + \frac{N}{4}\right) W_N^{nk} \\ & + W_N^{kN/2} \sum_{n=0}^{(N/4)-1} x\left(n + \frac{N}{2}\right) W_N^{nk} + W_N^{3kN/4} \sum_{n=0}^{(N/4)-1} x\left(n + \frac{3N}{4}\right) W_N^{nk}, \end{aligned} \quad (5.29)$$

which may be interpreted as four $(N/4)$ -point DFTs. (Compare this with Equation 5.7 in which an N -point DFT was split into two $(N/2)$ -point DFTs.) Using

$$\begin{aligned} W_N^{kN/4} &= (e^{-j2\pi/N})^{kN/4} = e^{-jk\pi/2} = (-j)^k, \\ W_N^{kN/2} &= e^{-jk\pi} = (-1)^k, \\ W_N^{3kN/4} &= (j)^k, \end{aligned}$$

Equation 5.29 becomes

$$\begin{aligned} X_N(k) = & \sum_{n=0}^{(N/4)-1} \left[x(n) + (-j)^k x\left(n + \frac{N}{4}\right) + (-1)^k x\left(n + \frac{N}{2}\right) \right. \\ & \left. + (j)^k x\left(n + \frac{3N}{4}\right) \right] W_N^{nk}. \end{aligned} \quad (5.30)$$

Letting $W_N^4 = W_{N/4}$,

$$X_N(4k) = \sum_{n=0}^{(N/4)-1} \left[x(n) + x\left(n + \frac{N}{4}\right) + x\left(n + \frac{N}{2}\right) + x\left(n + \frac{3N}{4}\right) \right] W_{N/4}^{nk}, \quad (5.31)$$

$$X_N(4k+1) = \sum_{n=0}^{(N/4)-1} \left[x(n) - jx\left(n + \frac{N}{4}\right) - x\left(n + \frac{N}{2}\right) + jx\left(n + \frac{3N}{4}\right) \right] W_N^n W_{N/4}^{nk}, \quad (5.32)$$

$$X_N(4k+2) = \sum_{n=0}^{(N/4)-1} \left[x(n) - x\left(n + \frac{N}{4}\right) + x\left(n + \frac{N}{2}\right) - x\left(n + \frac{3N}{4}\right) \right] W_N^{2n} W_{N/4}^{nk}, \quad (5.33)$$

$$X_N(4k+3) = \sum_{n=0}^{(N/4)-1} \left[x(n) + jx\left(n + \frac{N}{4}\right) - x\left(n + \frac{N}{2}\right) - jx\left(n + \frac{3N}{4}\right) \right] W_N^{3n} W_{N/4}^{nk}, \quad (5.34)$$

for $k = 0, 1, \dots, ((N/4) - 1)$.

Equations 5.31–5.34 represent four ($N/4$)-point DFTs that in combination yield one N -point DFT.

5.6 INVERSE FAST FOURIER TRANSFORM

The inverse discrete Fourier transform (IDFT) converts a discrete frequency domain sequence $X_N(k)$ into a corresponding discrete-time domain sequence $x(n)$. It is defined as

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X_N(k) W_N^{-kn}. \quad (5.35)$$

Comparing this with the DFT Equation 5.1, we see that the forward FFT algorithms described previously can be used to compute the inverse DFT if the following two changes are made:

- (1) Scale the result by $1/N$.
- (2) Replace twiddle factors W_N^{kn} by their complex conjugates W_N^{-kn} .

Program examples illustrating this technique are presented later in this chapter.

5.7 PROGRAMMING EXAMPLES USING C CODE

EXAMPLE 5.1: DFT of a Sequence of Real Numbers with Output in the CCS Graphical Display Window and in MATLAB (L138_dft)

This example illustrates the complex DFT of an N -point, real-valued sequence, $x(n)$. Program L138_dft.c, listed in Figure 5.9, calculates the complex DFT given by

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{(-j2\pi kn)/N}, \quad k = 0, 1, \dots, N-1. \quad (5.36)$$

Using Euler's relation to represent a complex exponential

$$e^{-j\omega t} = \cos(\omega t) - j \sin(\omega t), \quad (5.37)$$

the real and imaginary parts of $X(k)$ are computed by the program as

$$\text{Re}\{X(k)\} = \sum_{n=0}^{N-1} (\text{Re}\{x(n)\} \cos(2\pi kn/N) + \text{Im}\{x(n)\} \sin(2\pi kn/N)), \quad (5.38)$$

$$\text{Im}\{X(k)\} = \sum_{n=0}^{N-1} (\text{Im}\{x(n)\} \cos(2\pi kn/N) - \text{Re}\{x(n)\} \sin(2\pi kn/N)). \quad (5.39)$$

A structured data type COMPLEX is used by the program to represent the complex valued time and frequency domain values of $X(k)$ and $x(n)$.

```

// L138_dft.c
//
// N-point DFT of sequence read from lookup table
//

#include <stdio.h>
#include <math.h>

#define PI 3.14159265358979
#define N 100
#define TESTFREQ 800.0
#define SAMPLING_FREQ 8000.0

typedef struct
{
    float real;
    float imag;
} COMPLEX;

COMPLEX samples[N];

void dft(COMPLEX *x)
{
    COMPLEX result[N];
    int k,n;

    for (k=0 ; k<N ; k++)
    {
        result[k].real = 0.0;
        result[k].imag = 0.0;

        for (n=0 ; n<N ; n++)
        {
            result[k].real += x[n].real*cos(2*PI*k*n/N)
                            + x[n].imag*sin(2*PI*k*n/N);
            result[k].imag += x[n].imag*cos(2*PI*k*n/N)
                            - x[n].real*sin(2*PI*k*n/N);
        }
    }
    for (k=0 ; k<N ; k++)
    {
        x[k] = result[k];
    }
}

void main()
{
    int n;

    for(n=0 ; n<N ; n++)
    {
        samples[n].real = cos(2*PI*TESTFREQ*n/SAMPLING_FREQ);
        samples[n].imag = 0.0;
    }
    printf("real input data stored in array samples\n");
    dft(samples);           //call DFT function
    printf("done!\n");
    return;
}

```

Figure 5.9 Listing of program L138_dft.c.

Function `dft()` has been written such that it replaces the input samples $x(n)$, stored in array `x` with their frequency domain representation $X(k)$.

As supplied, the time domain sequence $x(n)$ consists of exactly 10 cycles of a real-valued cosine wave (assuming a sampling frequency of 8 kHz, the frequency of the cosine wave is 800 Hz). The DFT of this sequence $X(k)$ is equal to 0 for all k , except $k = 10$ and $k = 90$. These two real values correspond to frequency components at ± 800 Hz. Different time domain input sequences can be used in the program, most readily by changing the value of the constant `TESTFREQ`. Build and load the program `L138_dft`.

To test the program,

- (1) Place a breakpoint at the line

```
dft(samples); // call DFT function
```

in the source file `L138_dft.c`.

- (2) Run the program and it should halt at the breakpoint. The message

```
real input data stored in array samples
```

should appear in the console.

- (3) Select *Tools > Graph > Single Time* and set the *Graph Properties* as shown in Figure 5.10. You should see the real-valued, time domain input sequence as shown in Figure 5.11. Note that you have plotted only the real part of the complex values stored in array `samples`. Setting the *Display Properties* to *Display data as a Connected line with markers* serves to emphasize that the DFT operates on discrete data.

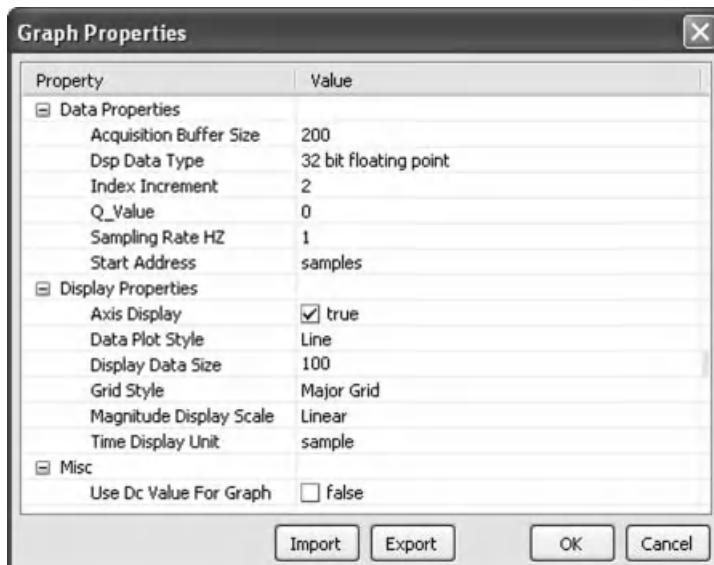


Figure 5.10 *Graph Properties* used to display real part of array `samples` in program `L138_dft.c`.

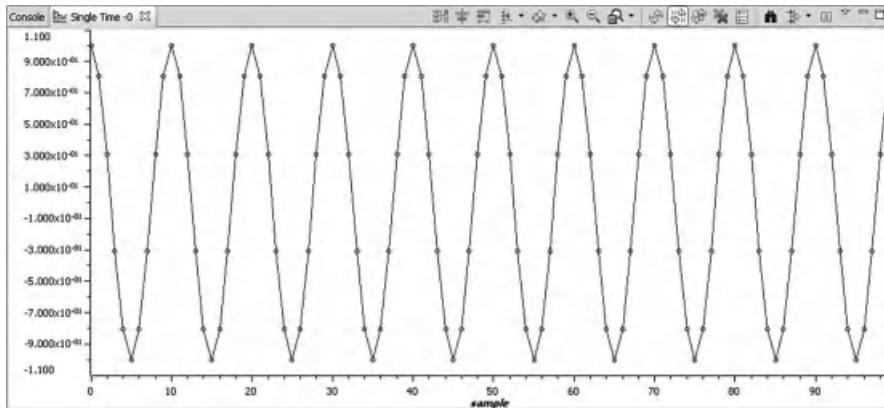


Figure 5.11 Graphical display of real part of array samples produced by program L138_dft.c (TEST_FREQ = 800).

- (4) Run the program again (from the breakpoint) and it should run to completion at which point the contents of array `samples` will be equal to the frequency domain representation $X(k)$ of the input data $x(n)$. The message

done!

should appear in the console.

- (5) The real part of $X(k)$ will now be displayed in the *Graph* window and you should be able to see two distinct spikes corresponding to $k = 10$ and $k = 90$, representing frequency components at ± 800 Hz, as shown in Figure 5.12.

Alternatively, the frequency domain output sequence $X(k)$ may be exported from the Code Composer Studio IDE to a .dat file of type TI Data Format and plotted using MATLAB.

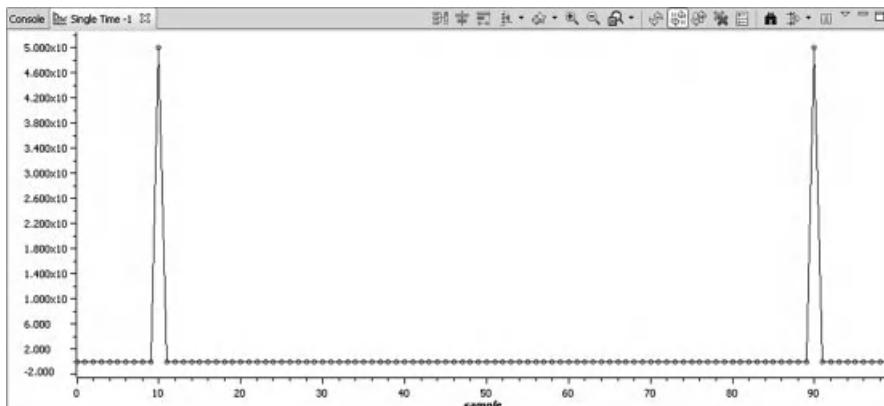


Figure 5.12 Contents of array `samples` transformed into frequency domain by program L138_dft.c (TEST_FREQ = 800). Only the real part of the data is plotted.

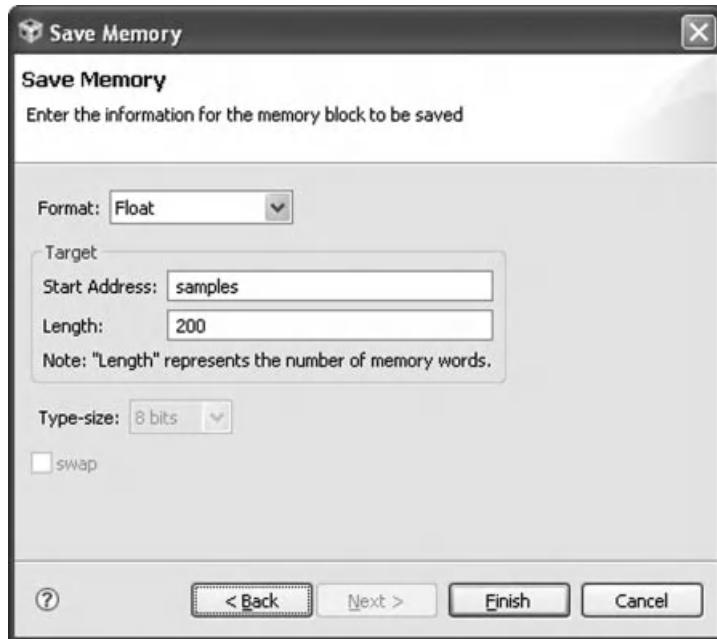


Figure 5.13 Save Memory parameters used to save complex contents of array samples to TI Data Format file.

Select *View > Memory* and then save the contents of array `samples` using the parameters shown in Figure 5.13.

Use the MATLAB function `L138_plot_complex()` in order to see the data as shown in Figure 5.14. Note the very small magnitude of the imaginary part of the data. Theoretically, this should be equal to 0. That it is not is due to rounding errors in the calculation of the DFT.

Change the frequency of the input waveform to 900 Hz (`#define TESTFREQ 900 . 0`) and repeat the procedure listed above. After the program has been run, you should see a number of nonzero values in the frequency domain sequence $X(k)$, as shown in Figure 5.15. This effect is referred to as spectral leakage and is due to the fact that the N -sample time domain sequence stored in array `samples` does not now represent an integer number of cycles of a sinusoid. Correspondingly, the frequency of the sinusoid is not exactly equal to one of the N discrete frequency components, spaced at intervals of $(8000.0/N)$ Hz in the frequency domain representation of $X(k)$.

The nature of the structured data type `COMPLEX` is such that array `samples` comprises $2N$ values of type `float` ordered so that the first floating-point value is the real part of $X(0)$, the second is the imaginary part of $X(0)$, the third is the real part of $X(1)$, and so on. In the Code Composer Studio *Graph* window, the real parts of $X(k)$ may be displayed by setting the *DSP Data Type* to *32 bit floating point*, the *Index Increment* to 2, and the *Start Address* to `samples` (the address of the first value of type `float` in the array `samples`) in the *Graph Properties* window. In order to display the imaginary parts of the sequence $X(k)$, *Start Address* must be set to the address of the second value of type `float` in the array `samples`. The address of array `samples` can be found by moving the cursor over the identifier `samples` in the source file `L138_dft.c`. Its hexadecimal address will appear in a pop-up box, as shown in Figure 5.16.

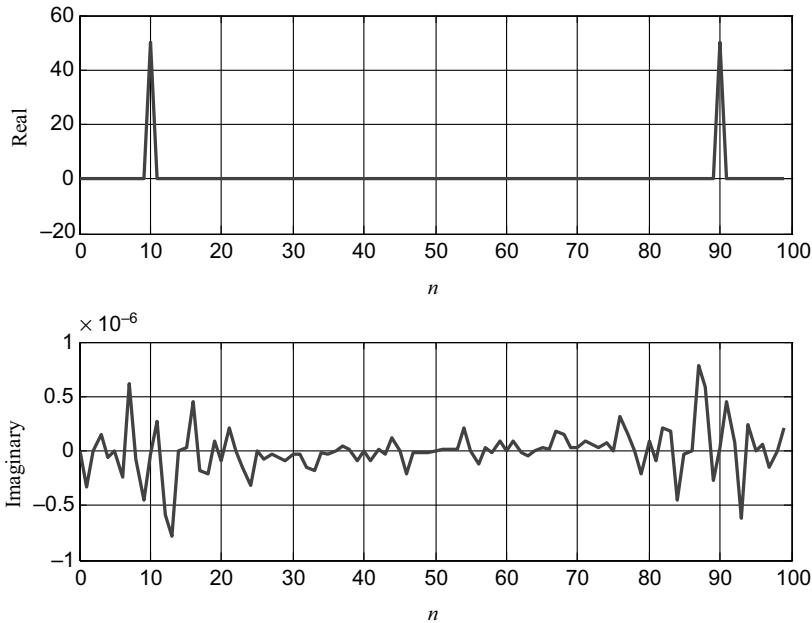


Figure 5.14 Complex contents of array `samples` (`TEST_FREQ = 800`) plotted using MATLAB function `L138_plot_complex()`.

Entering this value in the *Start Address* field of the *Graph Properties* window in place of the identifier `samples` will result in exactly the same graphical display. Adding 4 (the number of bytes used to store one 32-bit floating-point value) to the value in the *Start Address* field of the *Graph Properties* window will result in the imaginary parts of the sequence of complex values being displayed. Alternatively, the address of the array `samples` and the address of the first imaginary element in that array may be determined from the *Memory View* window, as shown in

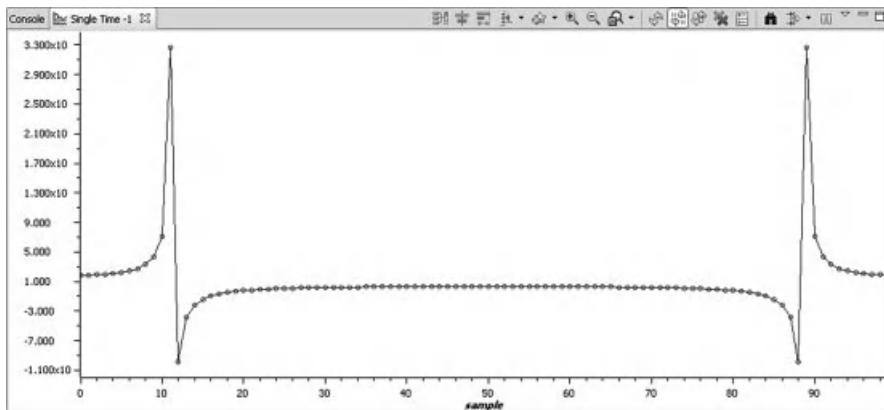
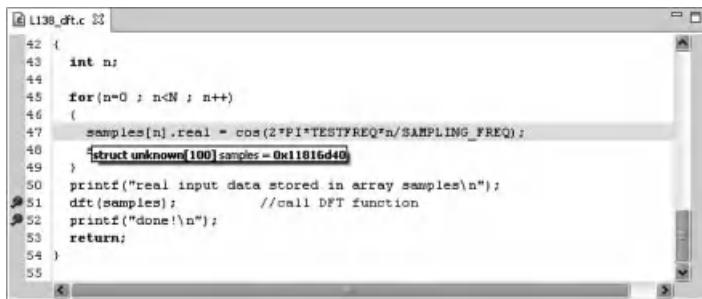


Figure 5.15 Contents of array `samples` transformed into frequency domain by program `L138_dft.c` (`TEST_FREQ = 900`). Only the real part of the data is plotted.



```

42 {
43     int n;
44
45     for(n=0 ; n<N ; n++)
46     {
47         samples[n].real = cos(2*PI*TESTFREQ*n/SAMPLING_FREQ);
48         struct unknown[100] samples = 0x11816D40;
49     }
50     printf("real input data stored in array samples\n");
51     dft(samples);           //call DFT function
52     printf("done!\n");
53     return;
54 }
55

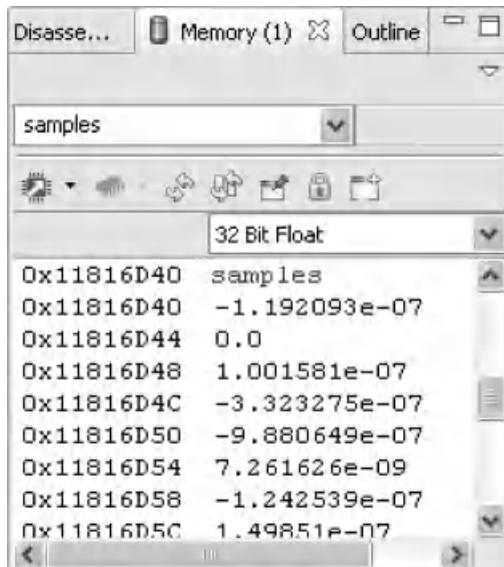
```

Figure 5.16 Pop-up window showing address in memory of array samples.

Figure 5.17. The MATLAB function `L138_plot_complex()` displays both real and imaginary parts of the complex data stored in it reads from a .dat file (Figures 5.18 and 5.19).

Twiddle factors

Whereas the radix-2 FFT is applicable only if N is an integer power of 2. The DFT can be applied to an arbitrary length sequence, for example, $N = 100$, as illustrated by program `L138_dft.c`. In spite of this, the FFT is widely used because of its computational efficiency and, if necessary, input data sequences are zero-padded to a length that is an integer power of 2. Part of the efficiency of the FFT is due to the use of precalculated twiddle factors, stored in a lookup table, rather than the repeated evaluation of `sin()` and `cos()` functions as implemented in function `dft()` in program `L138_dft.c`. The use of precalculated twiddle factors can be applied to the function `dft()` to give significant efficiency improvements to program `L138_dft.c`. As it stands, computationally expensive calls to the math library

**Figure 5.17** Memory window showing address in memory of array samples.

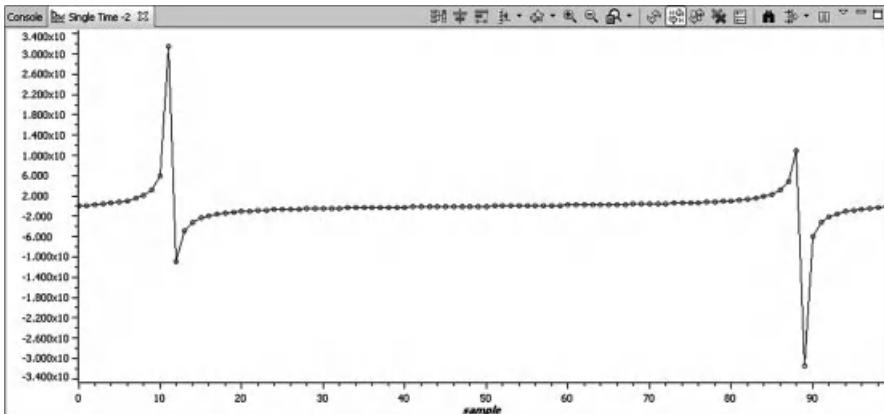


Figure 5.18 Contents of array `samples` transformed into frequency domain by program `L138_dft.c` (`TEST_FREQ = 900`). Only the imaginary part of the data is plotted.

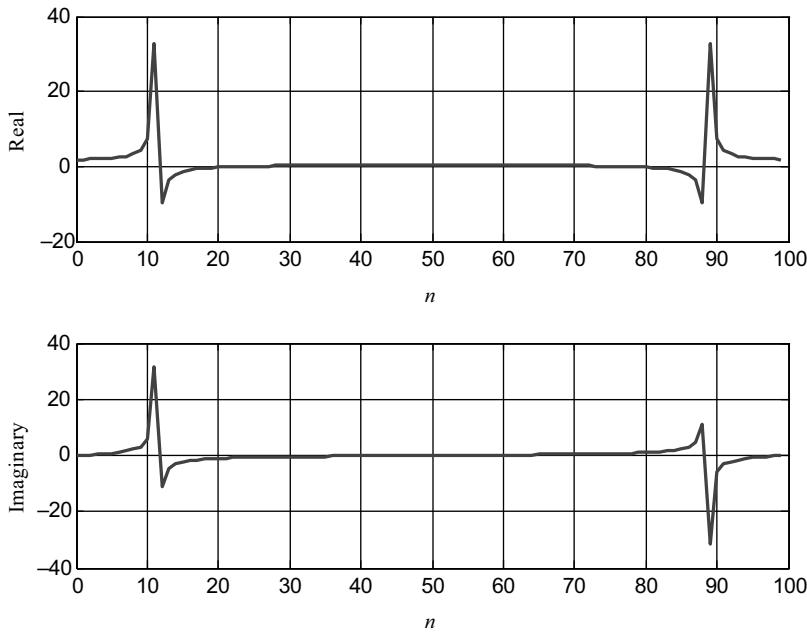


Figure 5.19 Complex contents of array `samples` (`TEST_FREQ = 900`) plotted using MATLAB function `L138_plot_complex()`.

```

// L138_dftw.c
//
// N-point DFT of sequence read from lookup table
// using pre-computed twiddle factors
//

#include <stdio.h>
#include <math.h>

#define PI 3.14159265358979
#define N 100
#define TESTFREQ 800.0
#define SAMPLING_FREQ 8000.0

typedef struct
{
    float real;
    float imag;
} COMPLEX;

COMPLEX samples[N];
COMPLEX twiddle[N];

void dftw(COMPLEX *x, COMPLEX *w)
{
    COMPLEX result[N];
    int k,n;

    for (k=0 ; k<N ; k++)
    {
        result[k].real=0.0;
        result[k].imag = 0.0;

        for (n=0 ; n<N ; n++)
        {
            result[k].real += x[n].real*w[(n*k)%N].real
                - x[n].imag*w[(n*k)%N].imag;
            result[k].imag += x[n].imag*w[(n*k)%N].real
                + x[n].real*w[(n*k)%N].imag;
        }
    }
    for (k=0 ; k<N ; k++)
    {
        x[k] = result[k];
    }
}

void main()
{
    int n;

    for(n=0 ; n<N ; n++)
    {
        twiddle[n].real = cos(2*PI*n/N);
        twiddle[n].imag = -sin(2*PI*n/N);
    }
    for(n=0 ; n<N ; n++)
    {
        samples[n].real = cos(2*PI*TESTFREQ*n/SAMPLING_FREQ);
        samples[n].imag = 0.0;
    }
    printf("real input data stored in array samples\n");
    dftw(samples,twiddle);      //call DFT function
    printf("done!\n");
}

```

Figure 5.20 Listing of program L138_dftw.c.

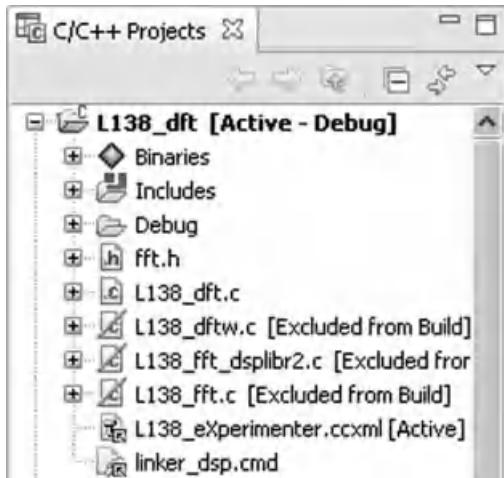


Figure 5.21 *C/C++ Projects* window showing all source files, except `L138_dft.c` excluded from build.

functions `sin()` and `cos()` are made a total of $4N^2$ times in function `dft()`. In program `L138_dftw.c`, listed in Figure 5.20, these function calls are replaced by reading precalculated twiddle factors from array `twiddle`.

The source file `L138_dftw.c` is stored in project folder `L138_dft` and can be substituted for source file `L138_dft.c` in project `L138_dft` by right-clicking on the program names of the two source files in the *C/C++ Projects* window and selecting *Exclude File(s) from Build*. Build and run program `L138_dftw.c` and verify that it gives results similar to those of program `L138_dft.c` (Figure 5.21).

EXAMPLE 5.2: Estimating Execution Times for DFT and FFT Functions
`(L138_dft, L138_dftw, L138_fft, L138_fft_dsplibr2)`

The computational expense of function `dft()` can be illustrated using Code Composer's *Profile Clock* (see Example 1.2). In this example, the functions `dft()` and `dftw()` used in Example 5.1 are compared with both a third function `fft()`, which implements the FFT in C, and a fourth function `DSPF_sp_cfftr2_dit()` from the C674x DSPLIB library.

Edit the lines in programs `L138_dft.c` and `L138_dftw.c` that read

```
#define N 100
      to read
#define N 128
```

so that similar length input sequences will be used by `dft()`, `dftw()`, `fft()`, and `DSPF_sp_cfftr2_dit()` functions. As discussed earlier, the FFT functions are suitable only for N equal to an integer power of 2. Then,

- (1) In the *C/C++ Projects* window in the *C/C++* perspective, ensure that source files `L138_dftw.c`, `L138_fft.c`, and `L138_fft_dsplibr2.c`, but not `L138_dft.c` are excluded from the build.

- (2) Right-click on the project name and select *Build Properties*. Under the *Tool Settings* tab, select *C6000 Compiler > Basic Options* and set the *Optimization Level* to 2. The same compiler optimization level will be used for each of the four programs. Click *Apply* and then *OK*.
- (3) In the *Debug* perspective, select *Project > Build Active Project* in order to build and load the program.
- (4) Set breakpoints at the lines `dft(samples);` and `printf("done!\n")` in program `L138_dft.c`.
- (5) In the *Debug* perspective, select *Target > Clock > Enable*.
- (6) Run the program. It should halt at the first breakpoint and the profile clock should display approximately 170,000 instruction cycles.
- (7) Reset the *Profile Clock* by double-clicking on its icon in the bottom right-hand corner of the Code Composer Studio IDE window.
- (8) Run the program. It should stop at the second breakpoint.

The number of instruction cycles counted by the *Profile Clock* (approximately 21,000,000) gives an indication of the computational expense of executing function `dft()`. On a 375 MHz OMAP-L138, 21,000,000 instruction cycles correspond to an execution time of 56 ms.

Repeat the preceding experiment excluding source file `L138_dft.c`, but including file `L138_dftw.c` in the project. The modified DFT function using twiddle factors `dftw()` uses approximately 44,000 instruction cycles, corresponding to 0.117 ms and representing a decrease in execution time by a factor of more than 450. At a sampling rate of 48 kHz, 0.117 ms corresponds to approximately six sampling periods.

Repeat the experiment using program `L138_fft.c`. This computes the FFT using a function written in C and defined in the file `fft.h`. Function `fft()` takes 22,000 instruction cycles or 0.059 ms (approximately three sampling periods at 48 kHz) to execute. The advantage, in terms of execution time, of the FFT over the DFT should increase with the number of points N used. Repeat this example using different values of N , for example, 1024 or 2048 in order to verify that the relative speed improvement increases.

Finally, repeat the experiment using program `L138_fft_dsplibr2.c`. This computes the FFT using an optimized function supplied as part of the C674x DSPLIB library. Place breakpoints at the lines

```
DSPF_sp_cfftr2_dit(x, w, N); // call FFT function
```

and

```
bit_rev(x, N);
```

5.7.1 Frame- or Block-Based Processing

Rather than processing one sample at a time, the DFT and FFT algorithms process blocks or frames of samples. Consequently, in real-time applications, their use is suited to direct memory access (DMA)-based I/O. Program `L138_loop_edma.c` (introduced in Chapter 2) used DMA-based I/O simply to take samples read from the ADC

```

// L138_fft.c
//
// N-point FFT of sequence read from lookup table
//

#include <stdio.h>
#include <math.h>
#include "fft.h"

#define PI 3.14159265358979
#define N 128
#define TESTFREQ 800.0
#define SAMPLING_FREQ 8000.0

COMPLEX samples[N];
COMPLEX twiddle[N];

void main()
{
    int n;

    for (n=0 ; n<N ; n++) //set up FFT twiddle factors
    {
        twiddle[n].real = cos(PI*n/N);
        twiddle[n].imag = -sin(PI*n/N);
    }

    for(n=0 ; n<N ; n++)
    {
        samples[n].real = cos(2*PI*TESTFREQ*n/SAMPLING_FREQ);
        samples[n].imag = 0.0;
    }
    printf("real input data stored in array samples\n");
    fft(samples,N,twiddle); //call FFT function
    printf("done!\n");
}

```

Figure 5.22 Listing of program L138_fft.c.

and write them to the DAC. DMA was used to transfer blocks of samples read from the ADC into memory and the CPU was interrupted not at every sampling instant, but only at the end of each DMA transfer. In that program, function `process_buffer()`, called once per DMA transfer, simply copied the contents of the input buffer most recently filled with new input samples to the appropriate output buffer, but it could just as well have applied some form of processing to those samples. Program `L138_iir-sos_edma.c` (introduced in Chapter 4) applied an IIR filter to the contents of each input buffer before writing the filter output to the corresponding output buffer. However, the IIR filter algorithm works on a sample-by-sample basis, reading each input sample and computing one corresponding filter output value in turn. It is not absolutely necessary for an entire buffer of input samples to be available at once to the IIR filter algorithm. The FFT, on the other hand, does require that an entire buffer, or block, of input samples is available to be processed at once.

A real-time constraint on function `process_buffer()` is that it must execute in less time than the next DMA transfer takes to complete, that is, N sampling periods.

```

// fft.h complex FFT function taken from Rulph's C31 book
// this file contains definition of complex data structure

struct cmpx           //complex data structure used by FFT
{
    float real;
    float imag;
};

typedef struct cmpx COMPLEX;

void fft(COMPLEX *Y, int M, COMPLEX *w)
{
    COMPLEX templ,temp2;      // temporary storage
    int i,j,k;                // loop counters
    int upper_leg, lower_leg; // upper/lower butterfly leg
    int leg_diff;              // upper/lower leg difference
    int num_stages=0;          // number of FFT stages
    int index, step;           // twiddle factor index and step
    i=1;                      // no. stages = log2(no. points)
    do
    {
        num_stages+=1;
        i=i*2;
    } while (i!=M);

    leg_diff=M/2;             // starting difference
    step=2;                   // twiddle step
    for (i=0;i<num_stages;i++) // for M-point FFT
    {
        index=0;
        for (j=0;j<leg_diff;j++)
        {
            for (upper_leg=j;upper_leg<M;upper_leg+=(2*leg_diff))
            {
                lower_leg=upper_leg+leg_diff;
                templ.real=(Y[upper_leg]).real + (Y[lower_leg]).real;
                templ.imag=(Y[upper_leg]).imag + (Y[lower_leg]).imag;
                temp2.real=(Y[upper_leg]).real - (Y[lower_leg]).real;
                temp2.imag=(Y[upper_leg]).imag - (Y[lower_leg]).imag;
                (Y[lower_leg]).real=temp2.real*(w[index]).real
                                -temp2.imag*(w[index]).imag;
                (Y[lower_leg]).imag=temp2.real*(w[index]).imag
                                +temp2.imag*(w[index]).real;
                (Y[upper_leg]).real=templ.real;
                (Y[upper_leg]).imag=templ.imag;
            }
            index+=step;
        }
        leg_diff=leg_diff/2;
        step*=2;
    }
    j=0;
    for (i=1;i<(M-1);i++) // bit reversal for resequencing data
    {
        k=M/2;
        while (k<=j)
        {
            j=j-k;
            k=k/2;
        }
        j=j+k;
    }
}

```

Figure 5.23 Listing of header file **fft.h**.

```

        if (i < j)
        {
            temp1.real=(Y[j]).real;
            temp1.imag=(Y[j]).imag;
            (Y[j]).real=(Y[i]).real;
            (Y[j]).imag=(Y[i]).imag;
            (Y[i]).real=temp1.real;
            (Y[i]).imag=temp1.imag;
        }
    }
    return;
}

```

Figure 5.23 (Continued)

The next section describes the basic characteristics of DMA and how it is implemented using the OMAP-L138's enhanced direct memory access (EDMA3) controller.

Direct memory access

Direct memory access is a method of transferring data from one area of memory to another without CPU intervention, that is, without using CPU instructions. It can therefore improve the computational efficiency of an application. The areas of memory involved in DMA transfer can include memory-mapped peripherals and the efficiency benefits of DMA can be especially pronounced when servicing memory-mapped peripheral devices that operate more slowly than RAM. In the case of the OMAP-L138 processor, DMA is implemented using the enhanced direct memory access (EDMA3) controller [1]. EDMA3 is named for its enhanced features and because of the three dimensions used to specify the data transfers it implements.

EDMA3 Architecture and Programming The OMAP-L138 EDMA3 controller is a powerful and complicated system. However, for the purposes of the examples in this book, only a small number of its features will be described in detail.

An EDMA3 controller comprises a channel controller and a transfer controller. The EDMA3 channel controller acts as a user interface to the EDMA3 transfer controller which is responsible for reading and writing data from/to memory. Effectively the channel controller keeps track of a number of concurrent DMA transfers, scheduling memory reads and writes to be carried out by the transfer controller. This is summarized in Figure 5.24.

The OMAP-L138 EDMA3 controller contains two channel controllers, EDMA3_CC0 and EDMA_CC1, each capable of handling up to 32 DMA transfers concurrently. However, the program examples in this book use only EDMA3_CC0 and subsequently this will be referred to as the EDMA3 controller.

Specifying a DMA transfer In order to carry out a DMA transfer operation, a number of parameters must be programmed into the EDMA3 controller. At the very least, these must include a source address (from which data will be read), a destination address (to which data will be written), and the number of bytes of data to be

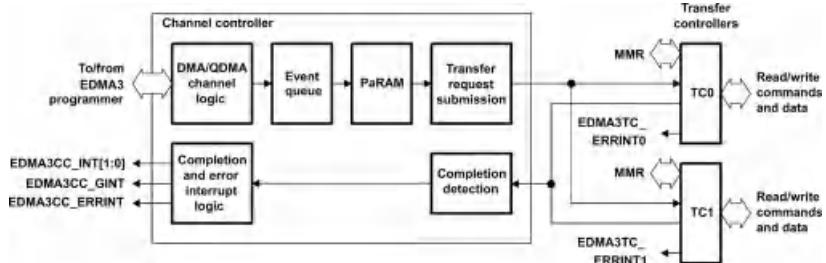


Figure 5.24 Architecture of EDMA3 controller (courtesy of Texas Instruments).

transferred. A basic form of DMA operation might transfer a given number of bytes of data stored contiguously in memory starting at the given source address to another area of memory, of similar size, starting at the given destination address. Using EDMA3, however, a number of variations on that basic operation are possible and these require further parameters to be programmed into the controller.

In addition, it is necessary to consider both what will trigger (cause) a DMA transfer to take place and how its completion will be signaled to (a program running on) the CPU.

EDMA3 data transfers may be initiated by several different means. The two mechanisms used in the examples in this book are event triggering and manual triggering.

Event-triggered transfers Event-triggered transfers are initiated by peripheral devices or external hardware. Each EDMA3 transfer is associated with an EDMA3 channel and each EDMA3 channel is associated with a different synchronization event, as summarized in Figure 5.25 [2]. The real-time examples in this book make use of event #0 McASP0 Receive and event #1 McASP0 Transmit. Synchronization events cause bits in the 32-bit EDMA3 event register (ER) to be set and, in turn, DMA transfers will be triggered if the corresponding bit in the EDMA3 event enable register (EER) is set.

Manually triggered transfers Manually triggered transfers are initiated by the CPU writing to the 32-bit EDMA3 event set register (ESR). Manually triggered transfers occur regardless of the bits set in the EER. Each bit of the ESR corresponds to a different one of the 32 EDMA3 channels in the corresponding EDMA3 channel controller. Hence, setting bit 5 in the ESR will manually trigger a transfer on EDMA3 channel #5.

Three-dimensional data transfer The data transferred by EDMA3 are viewed as a three-dimensional block comprising *CCNT* frames of *BCNT* arrays of *ACNT* bytes.

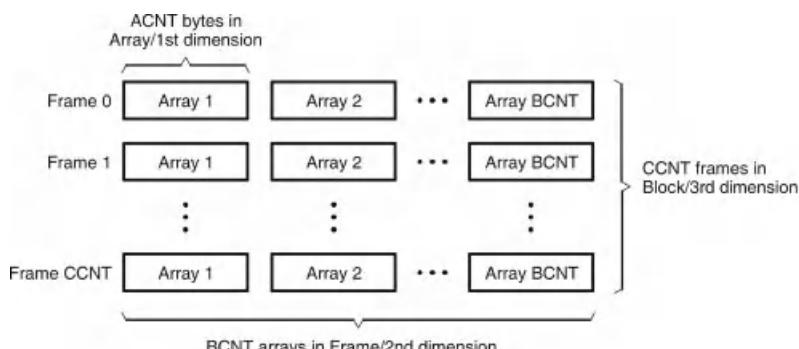
This mechanism allows flexibility in the arrangement of data in memory. It need not be contiguous. Arrays and frames may be distributed, offset at intervals, in memory as illustrated in Figure 5.26. Furthermore, transfers may be configured as either A-synchronized, in which case each event triggers the transfer of a single array of ACNT bytes, or AB-synchronized, in which case each event triggers the transfer of a single frame of BCNT arrays of ACNT bytes.

EDMA3 Channel Controller 0			
Event	Event Name / Source	Event	Event Name / Source
0	McASP0 Receive	16	MMCSDD Receive
1	McASP0 Transmit	17	MMCSDD Transmit
2	McBSP0 Receive	18	SPI1 Receive
3	McBSP0 Transmit	19	SPI1 Transmit
4	McBSP1 Receive	20	PRU_EVTOUT8
5	McBSP1 Transmit	21	PRU_EVTOUT7
6	GPIO Bank 0 Interrupt	22	GPIO Bank 2 Interrupt
7	GPIO Bank 1 Interrupt	23	GPIO Bank 3 Interrupt
8	UART0 Receive	24	I2C0 Receive
9	UART0 Transmit	25	I2C0 Transmit
10	Timer64P0 Event Out 12	26	I2C1 Receive
11	Timer64P0 Event Out 34	27	I2C1 Transmit
12	UART1 Receive	28	GPIO Bank 4 Interrupt
13	UART1 Transmit	29	GPIO Bank 5 Interrupt
14	SPI0 Receive	30	UART2 Receive
15	SPI0 Transmit	31	

EDMA3 Channel Controller 1			
Event	Event Name / Source	Event	Event Name / Source
0	Timer64P2 Compare Event 0	16	GPIO Bank 6 Interrupt
1	Timer64P2 Compare Event 1	17	GPIO Bank 7 Interrupt
2	Timer64P2 Compare Event 2	18	GPIO Bank 8 Interrupt
3	Timer64P2 Compare Event 3	19	Reserved
4	Timer64P2 Compare Event 4	20	Reserved
5	Timer64P2 Compare Event 5	21	Reserved
6	Timer64P2 Compare Event 6	22	Reserved
7	Timer64P2 Compare Event 7	23	Reserved
8	Timer64P3 Compare Event 0	24	Timer64P2 Event Out 12
9	Timer64P3 Compare Event 1	25	Timer64P2 Event Out 34
10	Timer64P3 Compare Event 2	26	Timer64P3 Event Out 12
11	Timer64P3 Compare Event 3	27	Timer64P3 Event Out 34
12	Timer64P3 Compare Event 4	28	MMCSD1 Receive
13	Timer64P3 Compare Event 5	29	MMCSD1 Transmit
14	Timer64P3 Compare Event 6	30	Reserved
15	Timer64P3 Compare Event 7	31	Reserved

Figure 5.25 Synchronization events associated with EDMA3 channels (courtesy of Texas Instruments).

The real-time examples in this book use A-synchronized transfers with ACNT equal to 4, that is, the number of bytes in two 16-bit samples (left and right channels read from or written to the AIC3106 codec), BCNT equal to BUFCOUNT/2 (BUFCOUNT is a constant defined in file L138_AIC3106_init.h), and CCNT equal to 1. In this way, each EDMA3 transfer comprises a block of BUFCOUNT 16-bit

**Figure 5.26** EDMA3 three-dimensional data transfer specification (courtesy of Texas Instruments).

samples comprising two samples (left and right) per sampling period collected over $\text{BUFCOUNT}/2$ sampling periods in total and synchronized to MCASP0 events occurring at the codec sampling rate. Each individual MCASP0 event causes the transfer of two 16-bit samples ($\text{ACNT} = 4$ bytes of data) and $\text{BUFCOUNT}/2$ MCASP0 events are required in order to complete one block transfer. For the real-time input and output used by the example programs in this book, two DMA transfers (one from the ADC to memory and one from memory to the DAC) must take place concurrently.

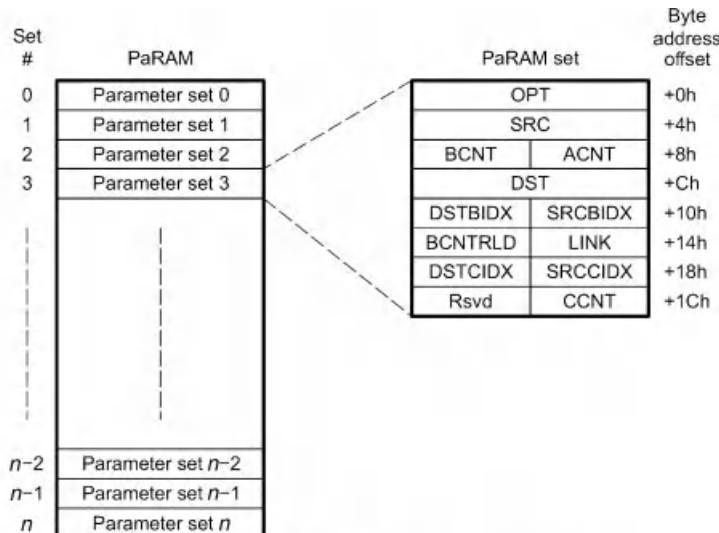
EDMA3 Parameter RAM Each EDMA3 transfer is specified by a set of parameters stored in a 32-byte block of parameter RAM (PaRAM). There are 128 separate blocks of PaRAM per EDMA3 channel controller: one block for each of its 32 EDMA channels and a further 96 blocks whose use will be described later in this section. The format of PaRAM is illustrated in Figure 5.27 [1].

The individual fields of a parameter RAM set are as follows:

Channel options (OPT) This 32-bit word specifies the configuration of an EDMA3 transfer. In the examples in this book, the relevant fields are bit 20 TCINTEN (transfer complete interrupt enable), bits 17-12 TCC (6-bit transfer complete code), and bit 2 SYNCDIM (transfer synchronization dimension).

Parameter RAM set fields SRC and DST specify the initial memory locations involved in the DMA transfer.

Channel Source Address (SRC) This 32-bit word is the starting byte address of the source of the data to be transferred.



Note: n is the number of PaRAM sets supported in the EDMA3CC for a specific device.

Figure 5.27 Format of parameter RAM (courtesy of Texas Instruments).

Channel Destination Address (DST) This 32-bit word is the starting byte address of the destination of the data to be transferred.

Parameter set fields ACNT, BCNT, and CCNT specify the size in bytes of a three-dimensional EDMA3 data transfer.

A Count (ACNT) This is an unsigned 16-bit value representing the number of bytes in an array.

B Count (BCNT) This is an unsigned 16-bit value representing the number of arrays of size ACNT bytes in a frame.

C Count (CCNT) This is an unsigned 16-bit value representing the number of frames of size $ACNT \times BCNT$ bytes in a block (transfer).

Source B Index (SRCBIDX) This is a signed 16-bit byte address offset between the arrays in a source frame. If it is equal to 0, consecutive arrays will be read from exactly the same address, for example, from the ADC.

Destination B Index (DSTBIDX) This is a signed 16-bit byte address offsets between the arrays in a destination frame. If it is equal to 0, then consecutive arrays will be written to exactly the same address, for example, to the DAC.

Link Address (LINK) This is the 16-bit byte address (within PaRAM) of the PaRAM block from which parameters will be copied when the current parameter set is exhausted (transfer completed in our examples). EDMA channel controller 0 PaRAM blocks start at address 0x01C04000 in the OMAP-L138 memory map and are stored contiguously as shown in Figure 5.27.

B Count Reload (BCNTRLD) This is the reload value for the BCNT field when BCNT has decremented to 0 (BCNT is decrement by 1 for each trigger event and array transfer in an A-synchronized transfer).

Source C Index (SRCCIDX) This is the signed 16-bit byte address offset between the frames in a source block.

Destination C Index (DSTCIDX) This is the signed 16-bit byte address offset between the frames in a destination block.

Initiating an EDMA3 transfer In nearly all the real-time example programs in this book, a MCASPO receive or transmit event is used to trigger an EDMA3 transfer (thus synchronizing EDMA3 with MCASPO and the AIC3106 codec). In turn, this means that EDMA3 channels #0 and #1 are used (see Figure 5.25) and that PaRAM sets #0 and #1 must be programmed in order to specify the ADC to memory and memory to DAC transfers. In addition, in order for McASPO receive and transmit events to trigger EDMA3 transfers, it is necessary to set bits 0 and 1 in the EDMA3 event enable register.

EDMA3 Transfer Progress As an EDMA3 transfer progresses, some of the values in the corresponding PaRAM set change. In the case of an A-synchronized transfer, after each triggering event, BCNT is decremented by 1, SRC is incremented by the value in SRCBIDX, and DST is incremented by the value in DSTBIDX. If, for

example, the transfer is from the ADC to memory, then the value of SRC will have been set equal to the MCASP0 DMA port and the value of SRCBIDX will have been set to 0. As each array of data is read from the ADC, no change in the address from which that data are read is necessary.

EDMA3 Transfer Completion When the value of *BCNT* reaches 0, an A-synchronized transfer is complete. If the TCINTEN bit in the OPT field is set, then a CPU system event is caused and bit TCC in the EDMA3 event pending register (EPR) is set. The value of TCC written to the OPT field in a PaRAM set can thus be used to identify which transfer has completed. Whether or not the CPU is interrupted depends on both whether the system event caused by a transfer completion is routed to a CPU interrupt through the interrupt selector, as specified in register INTMUXn, and whether that interrupt has been enabled in the CPU interrupt enable register (IER). In the case of the example programs in this book, the system event caused by an EDMA3 transfer completion is #8 EDMA3_0_CC0_INT1 and this is routed to CPU interrupt INT4 by writing the value 8 to the appropriate bits of INTMUX1.

Parameter linking On completion of an EDMA3 transfer, if the LINK field in the corresponding PaRAM set is not NULL, then the PaRAM set is refreshed by copying into it the contents of the PaRAM set indicated in the LINK field. Typically, the PaRAM set indicated in the LINK field will be one of those not specifically associated with an EDMA3 channel. The linking mechanism makes ping-pong buffering straightforward to implement.

EXAMPLE 5.3: EDMA3 Memory Move (L138_mem_edma)

Program L138_mem_edma.c, listed in Figure 5.28, demonstrates several of the basic features of EDMA3. It uses linked PaRAM sets to move blocks of 32-bit integers from one area of memory to another. Each block transferred comprises BUFCOUNT integer values. Transfers are initiated manually, that is, by program statements as opposed to hardware events. By placing breakpoints in the code, it is possible to follow the progress of each EDMA3 transfer.

Three arrays of 32-bit integers, bufA, bufB, and bufC are declared and initialized as follows. Array bufA is initialized to contain integer values counting down from the value (BUFCOUNT-1) in element bufA[0] to the value 0 in element bufA[(BUFCOUNT-1)]. Arrays bufB and bufC are each initialized to contain integer values counting up from 0 to (BUFCOUNT-1).

Four sets of parameter RAM are programmed. PaRAM set #3 is configured for an A-synchronized transfer of one frame of (BUFCOUNT*sizeof(int32_t)) bytes with source bufA and destination bufB. From the point of view of event triggering, EDMA channel #3 is associated with hardware event McBSP0 Receive. However, in program L138_mem_edma.c, channel #3 transfers are triggered manually by the CPU writing to bit 3 of the event set register. Manually triggered transfers occur regardless of whether the corresponding channel bit in the event enable register is set (this is necessary for event triggering) and therefore bit 3 of the EER is not set by program L138_mem_edma.c.

```

// L138_mem_edma.c
//

#include "types.h"
#define EDMA_3CC_ESR *( unsigned int* )( 0x01C01010 )

#define BUFCOUNT 256

uint32_t bufA[BUFCOUNT],bufB[BUFCOUNT],bufC[BUFCOUNT];
uint32_t *EDMA3_PaRAM_ptr;
uint32_t ACNT, BCNT, CCNT;
int i;

void main( void )
{
    for ( i=0;i<BUFCOUNT;i++ )
        {bufA[i]=BUFCOUNT-1-i; bufB[i]=i; bufC[i]=i; }

    ACNT = BUFCOUNT*sizeof(int32_t);
    BCNT = 1;
    CCNT = 1;

    // initialise PaRAM set #3
    EDMA3_PaRAM_ptr = (unsigned int *) (0x01C04060);

    *EDMA3_PaRAM_ptr++ = 0x00000000;
    *EDMA3_PaRAM_ptr++ = (unsigned int)bufA;
    *EDMA3_PaRAM_ptr++ =(int32_t)
        (((BCNT)<<16) & 0xFFFF0000) | (ACNT & 0x0000FFFF));
    *EDMA3_PaRAM_ptr++ = (unsigned int)bufB;
    *EDMA3_PaRAM_ptr++ =(int32_t)
        (((ACNT)<<16) & 0xFFFF0000) | (ACNT & 0x0000FFFF));
    *EDMA3_PaRAM_ptr++ = 0x00000800;
    *EDMA3_PaRAM_ptr++ = 0x00000000;
    *EDMA3_PaRAM_ptr++ = 0x00000001;

    //initialise PaRAM set #64
    EDMA3_PaRAM_ptr = (unsigned int *) (0x01C04800);

    *EDMA3_PaRAM_ptr++ = 0x00000000;
    *EDMA3_PaRAM_ptr++ = (unsigned int)bufC;
    *EDMA3_PaRAM_ptr++ =(int32_t)
        (((BCNT)<<16) & 0xFFFF0000) | (ACNT & 0x0000FFFF));
    *EDMA3_PaRAM_ptr++ = (unsigned int)bufA;
    *EDMA3_PaRAM_ptr++ =(int32_t)
        (((ACNT)<<16) & 0xFFFF0000) | (ACNT & 0x0000FFFF));
    *EDMA3_PaRAM_ptr++ = 0x00000820;
    *EDMA3_PaRAM_ptr++ = 0x00000000;
    *EDMA3_PaRAM_ptr++ = 0x00000001;

    //initialise PaRAM set #65
    EDMA3_PaRAM_ptr = (unsigned int *) (0x01C04820);

    *EDMA3_PaRAM_ptr++ = 0x00000000;
    *EDMA3_PaRAM_ptr++ = (unsigned int)bufB;
    *EDMA3_PaRAM_ptr++ =(int32_t)
        (((BCNT)<<16) & 0xFFFF0000) | (ACNT & 0x0000FFFF));
    *EDMA3_PaRAM_ptr++ = (unsigned int)bufC;
    *EDMA3_PaRAM_ptr++ =(int32_t)
        (((ACNT)<<16) & 0xFFFF0000) | (ACNT & 0x0000FFFF));
    *EDMA3_PaRAM_ptr++ = 0x00000840;
    *EDMA3_PaRAM_ptr++ = 0x00000000;

```

Figure 5.28 Listing of program L138_mem_edma.c.

```

*EDMA3_PaRAM_ptr++ = 0x00000001;

//initialise PaRAM set #66
EDMA3_PaRAM_ptr = (unsigned int *) (0x01C04840);

*EDMA3_PaRAM_ptr++ = 0x00000000;
*EDMA3_PaRAM_ptr++ = (unsigned int)bufA;
*EDMA3_PaRAM_ptr++ =(int32_t)
    (((BCNT)<<16) & 0xFFFF0000) | (ACNT & 0x0000FFFF));
*EDMA3_PaRAM_ptr++ = (unsigned int)bufB;
*EDMA3_PaRAM_ptr++ =(int32_t)
    (((ACNT)<<16) & 0xFFFF0000) | (ACNT & 0x0000FFFF));
*EDMA3_PaRAM_ptr++ = 0x00000800;
*EDMA3_PaRAM_ptr++ = 0x00000000;
*EDMA3_PaRAM_ptr++ = 0x00000001;

while(1)
    EDMA_3CC_ESR = 0x00000008; // cause EDMA3 event #3
}

```

Figure 5.28 (Continued)

PaRAM set #3 is initialized such that ACNT is equal to BUFCOUNT*sizeof(int32_t), BCNT is equal to 1, and CCNT is equal to 1. SRC is equal to bufA and DST is equal to bufB. DSTBIDX and SRCBIDX are both equal to ACNT. The value of BCNTRLD is not relevant to this transfer. In the OPT field of PaRAM set #3, TCINTEN bit is equal to 0, rendering the transfer complete code (TCC), which is equal to 0, irrelevant to this example, and A-synchronized transfers are selected by setting SYNCIDM bit equal to 0. The LINK field is initialized to indicate PaRAM set #64 and thus once a transfer of BUFCOUNT integers from bufA to bufB has been completed, PaRAM set #3 will be reloaded from PaRAM set #64. The values in PaRAM set #64 are identical to those set initially in PaRAM set #3, except that SRC is equal to bufB, DST is equal to bufC, and the LINK field is set to indicate PaRAM set #65. Once a transfer of (BUFCOUNT*sizeof(int32_t)) bytes from bufB to bufC has been completed, PaRAM set #3 will be reloaded from PaRAM set #65.

The values set in PaRAM set #65 are identical to those set initially in PaRAM set #3, except that SRC is equal to bufC, DST is equal to bufA, and the LINK field is set to indicate PaRAM set #66. Once a transfer of (BUFCOUNT*sizeof(int32_t)) bytes from bufC to bufA has been completed, PaRAM set #3 will be reloaded from PaRAM set #66.

The values set in PaRAM set #66 are identical to those set initially in PaRAM set #3 and hence a cycle of transfers between bufA, bufB, and bufC is established. These PaRAM settings are summarized in Figure 5.29.

Each transfer is initiated by event #3, in this case caused by the statement

```
EDMA_3CC_ESR = 0x00000008; // cause EDMA3 event #3
```

setting bit 3 in the EDMA3 event set register.

Transfer completion will cause the contents of a linked PaRAM set to be copied into PaRAM set #3.

Build and load the program and set a breakpoint at the line

```
EDMA_3CC_ESR = 0x00000008; // cause EDMA3 event #3
```

PaRAM set #3		
OPT	TCINTEN = 0, TCC = 0, SYNCDIM = 0	
SRC	bufA	
BCNT, ACNT	1	BUFCOUNT*4
DST	bufB	
DSTBIDX, SRCBIDX	BUFCOUNT*4	BUFCOUNT*4
N/A, LINK		#64
N/A, N/A		
N/A, CCNT		1

PaRAM set #64		
OPT	TCINTEN = 0, TCC = 0, SYNCDIM = 0	
SRC	bufB	
BCNT, ACNT	1	BUFCOUNT*4
DST	bufC	
DSTBIDX, SRCBIDX	BUFCOUNT*4	BUFCOUNT*4
N/A, LINK		#65
N/A, N/A		
N/A, CCNT		1

PaRAM set #65		
OPT	TCINTEN = 0, TCC = 0, SYNCDIM = 0	
SRC	bufC	
BCNT, ACNT	1	BUFCOUNT*4
DST	bufA	
DSTBIDX, SRCBIDX	BUFCOUNT*4	BUFCOUNT*4
N/A, LINK		#66
N/A, N/A		
N/A, CCNT		1

PaRAM set #66		
OPT	TCINTEN = 1, TCC = 0, SYNCDIM = 0	
SRC	bufA	
BCNT, ACNT	1	BUFCOUNT*4
DST	bufB	
DSTBIDX, SRCBIDX	BUFCOUNT*4	BUFCOUNT*4
N/A, LINK		#64
N/A, N/A		
N/A, CCNT		1

Figure 5.29 Initial PaRAM settings used by program L138_mem_edma.c.

Run the program. It should halt at the breakpoint, at which point the initial contents of the arrays bufA, bufB, and bufC may be viewed by clicking on *Tools > Graph > Single Time* and setting the *Graph Properties* as shown in Figure 5.30. (Viewing the buffer contents in this way relies on the three arrays having been declared such that they use contiguous areas of memory. The correct operation of the program does not rely on this (Figure 5.31).)

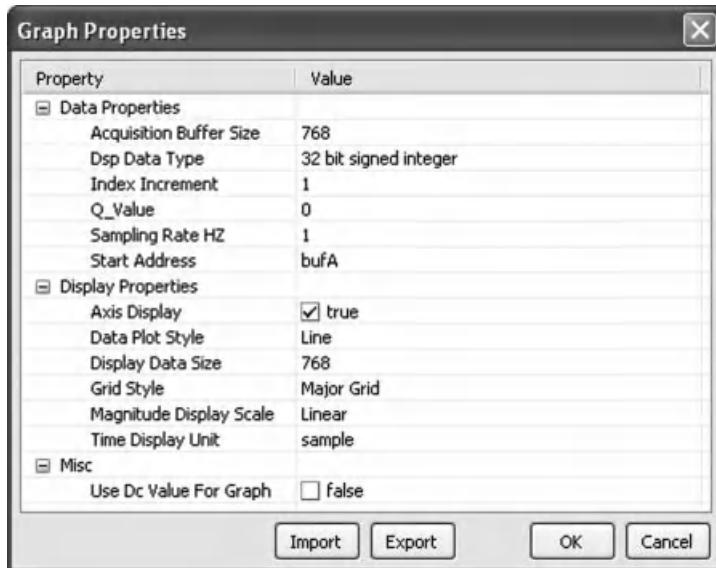


Figure 5.30 *Graph Properties* used to view contents of arrays bufA, bufB, and bufC.

When the program is run again, manual triggering of EDMA3 channel #3 is caused by the program statement

```
EDMA_3CC_ESR = 0x00000008; // cause EDMA3 event #3
```

This will cause the transfer of just one array of data. However, since in this example BCNT and CCNT are both equal to 1, a complete transfer comprises just one array of data.

The program should halt at the same breakpoint again and the result of the transfer (from bufA to bufB) should be reflected in the *Graph* window, as shown in Figure 5.32. By clicking

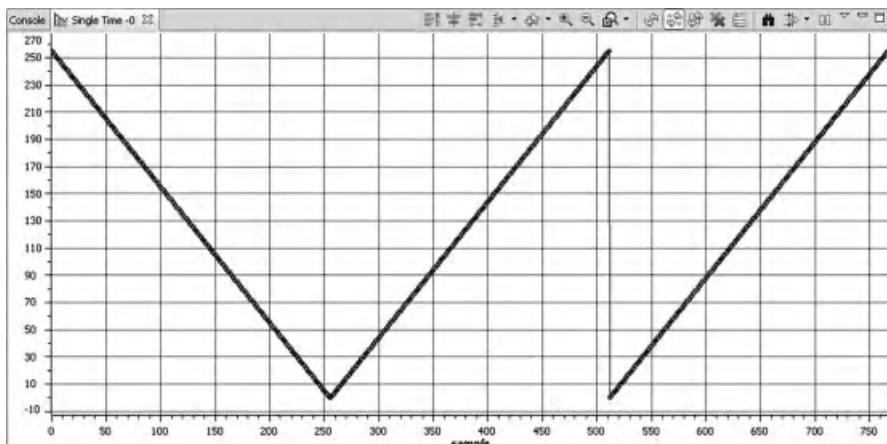


Figure 5.31 Graphical representation of the initial contents of arrays bufA, bufB, and bufC.

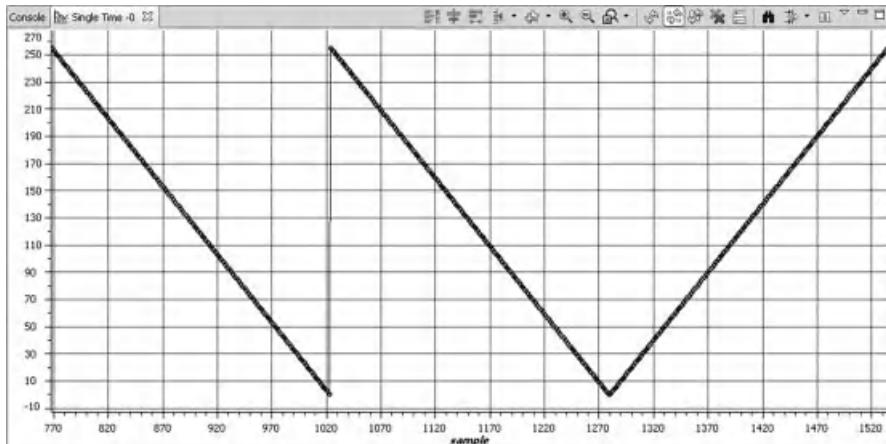


Figure 5.32 Graphical representation of the contents of arrays bufA, bufB, and bufC after completion of one DMA transfer.

repeatedly on the green *Run* button in the *Debug* window toolbar, the user can view the results of successive EDMA3 transfers.

Program L138_mem_edma.c can readily be modified to demonstrate differently configured memory-to-memory DMA transfers. For example, if the values of ACNT and BCNT are changed to BUFCOUNT and `sizeof(int32_t)`, respectively, then the same DMA transfers between buffers will be achieved, but each transfer will require four manual triggering events in order to be completed.

Build and run the modified version of the program and verify that after each triggering event, only one quarter of a buffer of data has been transferred.

EDMA3 Configuration for the real-time Examples in this Book The real-time example programs in this book that use DMA-based I/O implement a ping-pong buffering arrangement. Ping-pong buffering makes use of two sets (ping and pong) of input and output buffers. While the ping input buffer is being filled with new input samples from the ADC and the contents of the ping output buffer are being written to the DAC using DMA, the contents of the pong input buffer are being processed by the CPU and the results stored in the pong output buffer. When the input and output DMA transfers are complete, the roles of the ping and pong buffer sets are swapped. The pong buffers are used for input and output and the contents of the ping input buffer are processed. This arrangement is implemented using the EDMA3 linking mechanism. EDMA channels #0 and #1, associated with MCASP0 Receive and MCASP0 Transmit events, respectively, are used and two more linked PaRAM sets are used for each channel.

Figures 5.33 and 5.34 show the PaRAM settings made in function `EDMA_PaRAM_setup()`, defined in file L138_AIC3106_init.c., and called by function `L138_initialise_edma()`.

Since events McASP0 Receive and McASP0 Transmit are synchronized, only EDMA3 channel #0 transfer completions are configured to cause interrupts. If the ping

PaRAM set #0 <i>McASP0 Receive</i>		
OPT	TCINTEN = 1, TCC = 0, SYNCDIM = 0	
SRC	McASP0 RBUF	
BCNT, ACNT	BUFCOUNT/2	4
DST	pingIN	
DSTBIDX, SRCBIDX	4	0
N/A, LINK		#67
N/A, N/A		
N/A, CCNT		1

PaRAM set #67		
OPT	TCINTEN = 1, TCC = 1, SYNCDIM = 0	
SRC	McASP0 RBUF	
BCNT, ACNT	BUFCOUNT/2	4
DST	pongIN	
DSTBIDX, SRCBIDX	4	0
N/A, LINK		#68
N/A, N/A		
N/A, CCNT		1

PaRAM set #68		
OPT	TCINTEN = 1, TCC = 0, SYNCDIM = 0	
SRC	McASP0 RBUF	
BCNT, ACNT	BUFCOUNT/2	4
DST	pingIN	
DSTBIDX, SRCBIDX	4	0
N/A, LINK		#67
N/A, N/A		
N/A, CCNT		1

Figure 5.33 PaRAM settings for real-time ping-pong buffered input using EDMA3 channel #0.

buffers have just been used for I/O, the corresponding TCC is 0 and bit 0 in the EDMA3 EPR is set on transfer completion. If the pong buffers have just been used for I/O, the corresponding TCC is 1 and bit 1 in the EDMA3 EPR is set on transfer completion.

The system event caused by EDMA3 transfer completion that can be routed to a CPU interrupt is event #8, Channel Controller 0 Shadow Region 1 Transfer Completion Interrupt (EDMA3_0_CC0_INT1 EDMA3_0). In addition to routing this system event to CPU interrupt INT4 by writing the value 8 into the bits of Interrupt Mux Register INTMUX1 and in order to enable interrupts caused by EDMA3 transfer completion, the bits corresponding to TCC must be set in the EDMA3 IER and DMA Region Access Enable 1 (DRAE1) register.

Since $TCC = 0$ and $TCC = 1$ are used (to indicate pingIN and pongIN, respectively), bits 0 and 1 must be set in IER and DRAE1. Program statements

```
EDMA3_IESTR = 0x0003;
```

PaRAM set #1 McASP0 Transmit		
OPT	TCINTEN = 0, TCC = 0, SYNCDIM = 0	
SRC	pingOUT	
BCNT, ACNT	BUFCOUNT/2	4
DST	McASP0 XBUF	
DSTBIDX, SRCBIDX	0	4
N/A, LINK		#64
N/A, N/A		
N/A, CCNT		1

PaRAM set #64		
OPT	TCINTEN = 0, TCC = 1, SYNCDIM = 0	
SRC	pongOUT	
BCNT, ACNT	BUFCOUNT/2	4
DST	McASP0 XBUF	
DSTBIDX, SRCBIDX	0	4
N/A, LINK		#65
N/A, N/A		
N/A, CCNT		1

PaRAM set #65		
OPT	TCINTEN = 0, TCC = 0, SYNCDIM = 0	
SRC	pingOUT	
BCNT, ACNT	BUFCOUNT/2	4
DST	McASP0 XBUF	
DSTBIDX, SRCBIDX	0	4
N/A, LINK		#64
N/A, N/A		
N/A, CCNT		1

Figure 5.34 PaRAM settings for real-time ping-pong buffered output using EDMA3 channel #1.

and

EDMA3_DRAE1 = 0x0003;

achieve this.

EXAMPLE 5.4: DFT of a Signal in Real Time Using a DFT Function with Precalculated Twiddle Factors (L138_dft128_edma)

Program L138_dft128_edma.c, listed in Figure 5.35, combines the DFT function dftw() from program L138_dftw.c and real-time DMA-based I/O in order to implement a basic form of spectrum analyzer.

In spite of its inefficiency compared to that of the FFT, the DFT implemented using function dftw() is capable of running in real time (for $N = 128$ and a sampling frequency of 8 kHz).

```

// L138_dft128_edma.c
//

#include "L138_aic3106_init.h"
#include <math.h>

#define N (BUFCOUNT/2)
#define PI 3.14159265358979

#define TRIGGER 32000

typedef struct
{
    float real;
    float imag;
} COMPLEX;

extern int16_t *pingIN, *pingOUT, *pongIN, *pongOUT;
volatile int buffer_full = 0;
int IPR;
int procBuffer;
COMPLEX twiddle[N];
COMPLEX cbuf[N];
int16_t outbuffer[N];

void dftw(COMPLEX *x, COMPLEX *w)
{
    COMPLEX result[N];
    int k,n;

    for (k=0 ; k<N ; k++)
    {
        result[k].real=0.0;
        result[k].imag = 0.0;

        for (n=0 ; n<N ; n++)
        {
            result[k].real += x[n].real*w[(n*k)%N].real
                            - x[n].imag*w[(n*k)%N].imag;
            result[k].imag += x[n].imag*w[(n*k)%N].real
                            + x[n].real*w[(n*k)%N].imag;
        }
    }
    for (k=0 ; k<N ; k++)
    {
        x[k] = result[k];
    }
}

interrupt void interrupt4(void) // interrupt service routine
{
    switch(EDMA_3CC_IPR)
    {
        case 1:                      // TCC = 0
            procBuffer = PING;       // process ping
            EDMA_3CC_ICR = 0x0001;   // clear EDMA3 IPR bit TCC
            break;
        case 2:                      // TCC = 1
            procBuffer = PONG;      // process pong
            EDMA_3CC_ICR = 0x0002;   // clear EDMA3 IPR bit TCC
    }
}

```

Figure 5.35 Listing of program L138_dft128_edma.c.

```

        break;
    default:                      // may have missed an interrupt
        EDMA_3CC_ICR = 0x0003;    // clear EDMA3 IPR bits 0 and 1
        break;
    }
    EVTCLR0 = 0x00000100;
    buffer_full = 1;             // flag EDMA3 transfer
    return;
}

void process_buffer(void) // should be called from infinite
                           // loop in main() when process
                           // buffer has been filled
{
    int16_t *inBuf, *outBuf;      // pointers to process buffers
    int16_t left_sample, right_sample;
    int i,j;

    if (procBuffer == PING)       // use ping or pong buffers
    {
        inBuf = pingIN;
        outBuf = pingOUT;
    }
    if (procBuffer == PONG)
    {
        inBuf = pongIN;
        outBuf = pongOUT;
    }

    for (i = 0; i < (BUFCOUNT/2) ; i++)
    {
        left_sample = *inBuf++;
        right_sample = *inBuf++;

        cbuf[i].real = (float)left_sample;
        cbuf[i].imag = 0.0;
    }

    dftw(cbuf,twiddle);

    for (i = 0; i < (BUFCOUNT/2) ; i++)
    {
        left_sample = (int16_t)(sqrt(cbuf[i].real*cbuf[i].real
                                      + cbuf[i].imag*cbuf[i].imag)/16.0);
        outbuffer[i]=left_sample;
        if (i==0) left_sample = TRIGGER;

        *outBuf++ = left_sample;
        *outBuf++ = left_sample;
    }
    buffer_full = 0; // indicate that buffer has been processed
    return;
}

int main(void)
{
    int n;

    for(n=0 ; n<N ; n++)
    {
        twiddle[n].real = cos(2*PI*n/N);
}

```

Figure 5.35 (Continued)

```

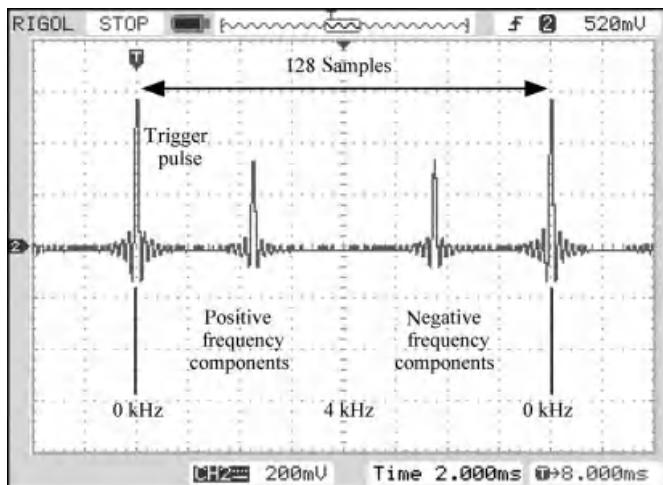
        twiddle[n].imag = -sin(2*PI*n/N);
    }
L138_initialise_edma(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
while(1)
{
    while (!buffer_full);
    process_buffer();
}
}

```

Figure 5.35 (Continued)

The number of samples in a block is set by the constant BUFCOUNT, defined in header file L138_aic3106_init.h. BUFCOUNT sets the number of 16-bit sample values that make up one DMA transfer block. Since the sample data transferred from the AIC3106 codec comprises left- and right-hand channel samples, for the purposes of this and other single channel examples in this chapter, there are BUFCOUNT/2 samples per DMA transfer. Function dftw() makes use of a global constant N to represent the number of samples it processes and hence in program L138_dft128_edma.c, N is set equal to BUFCOUNT/2. In function process_buffer(), local pointers inBuf and outBuf are used to point to the pingIN, pingOUT, pongIN, or pongOUT buffers as determined by reading the EDMA interrupt pending register EDMA_3CC_IPR in the interrupt4() interrupt service routine. The complex DFT of each block of real-valued input samples is computed using function dftw() and magnitude of the frequency domain representation is written to the output buffer. These N values are written to the DAC by the next DMA transfer (and to array outbuffer) and may be viewed using an oscilloscope.

Build and run the program. Use a signal generator connected to LINE IN on the experimenter to input a sinusoidal signal with a peak-to-peak magnitude of approximately 200 mV and connect an oscilloscope to LINE OUT. Vary the frequency of the input signal between 100 and 3500 Hz. Figure 5.36 shows an example of what you should see on the oscilloscope screen. In this case, BUFCOUNT was equal to 256. The two smaller pulses correspond to the magnitude

**Figure 5.36** Output signal from program L138_dft128_edma.c viewed on oscilloscope.

of the frequency content of the input signal computed using the DFT. The larger pulses correspond to impulses added to the output signal every 128 samples, replacing the magnitude of sample $X(0)$, for the purpose of triggering the oscilloscope.

The data in the output buffer is ordered such that the first value corresponds to a frequency of 0 Hz. The next 64 values correspond to frequencies 62.5 Hz (fs/N) to 4 kHz ($fs/2$) inclusive in steps of 62.5 Hz. The following 63 values correspond to frequencies of -3937.5 to -62.5 Hz inclusive, in steps of 62.5 Hz.

Increase the frequency of the input signal and you should see the two smaller pulses move towards a point halfway between the larger trigger pulses. As the frequency of the input signal approaches 4 kHz, the magnitude of the two smaller pulses should diminish, ideally reaching zero at a frequency of 4 kHz. In fact, a slight degree of aliasing should be evident as the input signal frequency is increased past 4 kHz and the magnitude of the smaller pulses diminishes because the magnitude frequency response of the AIC3106 DAC reconstruction filter is only 3 dB down at half the sampling frequency.

If the frequency of the sinusoidal input signal is equal to 1750 Hz, then the magnitude of the DFT of a frame of 128 input samples should be zero except at two points, corresponding to frequencies of ± 1750 Hz. Each block of data output via the DAC will contain one other nonzero value; the trigger pulse inserted at $X(0)$. The three impulses contained in each frame of samples appear on the oscilloscope as three pulses, each with the form of the impulse response of the DAC reconstruction filter. Compare the shapes of the pulses shown in Figure 5.36 with that shown in Figure 2.45.

Figure 5.37 shows the output signal corresponding to a 1750 Hz input signal in more detail.

Change the frequency of the input signal to 1781 Hz and you should see an output waveform similar to that shown in Figure 5.38. As the frequency of the sinusoidal input signal is changed, the shape as well as the position (relative to the trigger pulses) of the smaller pulses will change. The precise shape of the pulses is due to the characteristics of the reconstruction



Figure 5.37 Detail of output signal from program `L138_dft128_edma.c` for input sinusoid frequency of 1750 Hz.



Figure 5.38 Detail of output signal from program L138_dft128_edma.c for input sinusoid frequency of 1781 Hz.

filter in the AIC3106 codec, as discussed in Chapter 2. The fact that the pulse shape changes with input signal frequency in this example is due to the phenomenon of spectral leakage.

Figure 5.39 shows the DFT magnitude (output) data corresponding to the oscilloscope trace of Figure 5.37. The trigger pulse at the start of the block of data causes the impulse response of the reconstruction filter to appear on the oscilloscope. It can be deduced from Figure 5.39 that the frequency of the sinusoidal input signal was exactly equal to 1750 Hz, corresponding to $28f_0$, where $f_0 = 62.5$ Hz is the fundamental frequency associated with a block of 128 samples at a sampling rate of 8 kHz. The solitary nonzero frequency domain value produces an output pulse shape similar to that of the trigger pulse in Figure 5.37.

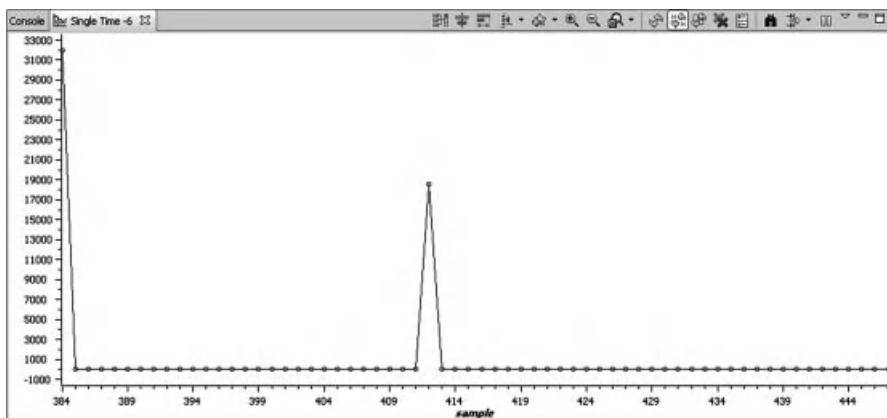


Figure 5.39 Detail of 128-point DFT data calculated in program L138_dft128_edma.c for input sinusoid frequency of 1750 Hz.

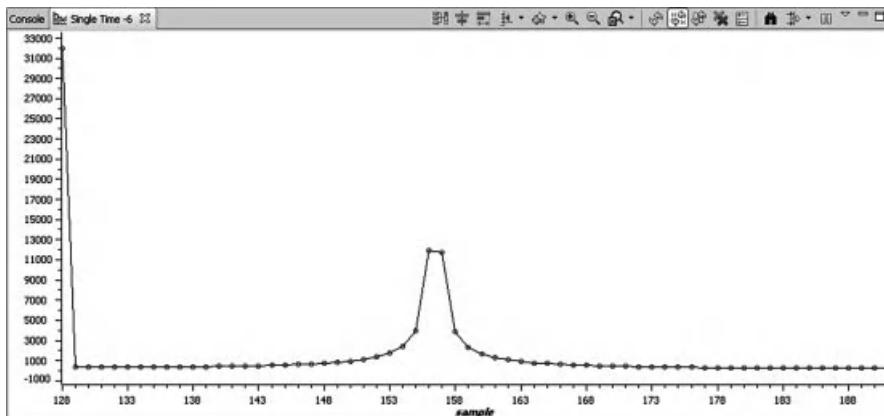


Figure 5.40 Detail of 128-point DFT data calculated in program L138_dft128_edma.c for input sinusoid frequency of 1781 Hz.

In contrast, it may be deduced from Figure 5.40 that the frequency of the sinusoidal input that produced the DFT magnitude data and hence the oscilloscope trace of Figure 5.38 was in between $28f_0$ and $29f_0$, that is, between 1750 and 1812.5 Hz. Figure 5.40 illustrates spectral leakage and Figure 5.38 shows the result of the data shown in Figure 5.40, regarded as time domain samples, filtered by the reconstruction filter in the AIC3106 codec. The shape of the small pulse in Figure 5.38 is different to that of the trigger pulse.

Modifying the program to reduce spectral leakage

One method of reducing spectral leakage is to multiply the frames of input samples by a window function prior to computing the DFT.

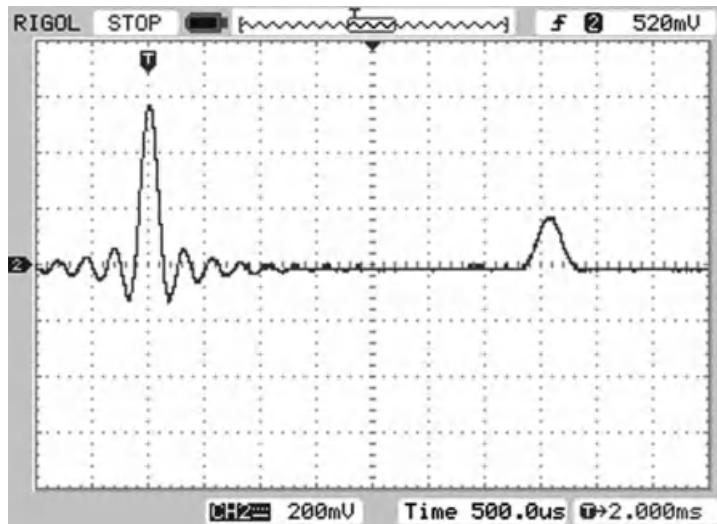


Figure 5.41 Detail of output signal from program L138_dft128_edma.c modified to apply a Hamming window to blocks of input samples for input sinusoid frequency of 1781 Hz.

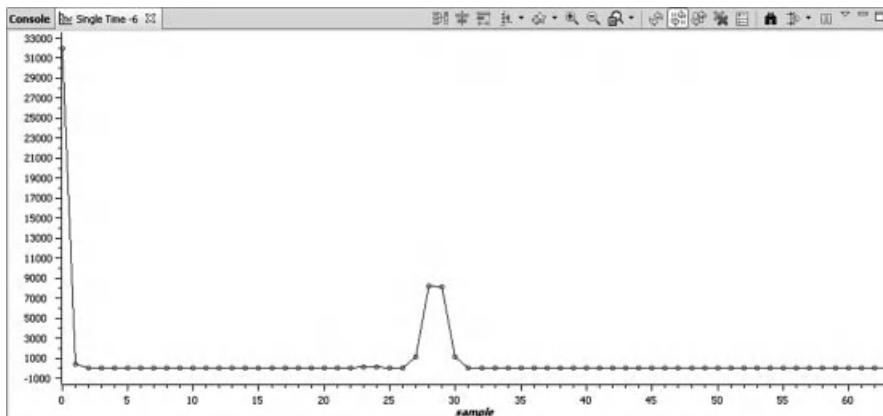


Figure 5.42 Detail of 128-point DFT data calculated in program L138_dft128_edma.c for input sinusoid frequency of 1781 Hz.

Add the line

```
#include "hamm128.h"

to program L138_dft128_edma.c and alter the line that reads
```

```
cbuf[i].real = (float)left_sample;
```

to read

```
cbuf[i].real = (float)left_sample*hamming[i];
```

File hamm128.h contains the declaration of an array hamming initialized to contain values representing a 128-point Hamming window. In order to run this program successfully, the value of the constant BUFCOUNT, set in file L138_aic3106init.h, must be set equal to 256.

In the *Debug* perspective, select *Project > Rebuild Active Project* and then run the program. Figure 5.41 shows the shape of the small pulse you can expect to see on the oscilloscope, regardless of the frequency of the sinusoidal input signal, and Figure 5.42 shows the corresponding DFT magnitude data. The spectral leakage evident in Figure 5.42 is less than that in Figure 5.40.

**EXAMPLE 5.5: FFT of a Real-Time Input Signal Using an FFT Function in C
(L138_fft128_edma.c)**

Program L138_fft128_edma.c, listed in Figure 5.43, implements a 128-point FFT in real time using an external input signal. It calls an FFT function *fft()* written in C. This function is defined in the file *fft.h*. The function was written originally for use with the C31 DSK and is described in Ref. [3]. Program L138_fft128_edma.c is similar to program L138_dft128_edma.c in all respects other than its use of *fft()* in place of the less computationally efficient *dftw()* and the slightly different computation of twiddle factors.

Build and run this program. Repeat the experiments carried out in Example 5.4 and verify that the results obtained are similar.

```

// L138_fft128_edma.c
//

#include "L138_aic3106_init.h"
#include <math.h>
#include "fft.h"

#define N (BUFCOUNT/2)
#define PI 3.14159265358979

#define TRIGGER 32000

extern int16_t *pingIN, *pingOUT, *pongIN, *pongOUT;
volatile int buffer_full = 0;
int procBuffer;
COMPLEX twiddle[N];
COMPLEX cbuf[N];
int16_t outbuffer[128];

interrupt void interrupt4(void) // interrupt service routine
{
    switch(EDMA_3CC_IPR)
    {
        case 1:                      // TCC = 0
            procBuffer = PING;       // process ping
            EDMA_3CC_ICR = 0x0001;   // clear EDMA3 IPR bit TCC
            break;
        case 2:                      // TCC = 1
            procBuffer = PONG;      // process pong
            EDMA_3CC_ICR = 0x0002;   // clear EDMA3 IPR bit TCC
            break;
        default:                     // may have missed an interrupt
            EDMA_3CC_ICR = 0x0003;   // clear EDMA3 IPR bits 0 and 1
            break;
    }
    EVTCLR0 = 0x00000100;
    buffer_full = 1;              // flag EDMA3 transfer
    return;
}

void process_buffer(void) // should be called from infinite
                         // loop in main() when process
                         // buffer has been filled
{
    int16_t *inBuf, *outBuf;      // pointers to process buffers
    int16_t left_sample, right_sample;
    int i,j;

    if (procBuffer == PING)      // use ping or pong buffers
    {
        inBuf = pingIN;
        outBuf = pingOUT;
    }
    if (procBuffer == PONG)
    {
        inBuf = pongIN;
        outBuf = pongOUT;
    }

    for (i = 0; i < (BUFCOUNT/2) ; i++)
    {

```

Figure 5.43 Listing of program L138_fft128_edma.c.

```

left_sample = *inBuf++;
right_sample = *inBuf++;

cbuf[i].real = (float)left_sample;
cbuf[i].imag = 0.0;
}

fft(cbuf,N,twiddle);

for (i = 0; i < (BUFCOUNT/2) ; i++)
{
    left_sample = (int16_t)(sqrt(cbuf[i].real*cbuf[i].real
                                + cbuf[i].imag*cbuf[i].imag)/16.0);
    outbuffer[i]=left_sample;
    if (i==0) left_sample = TRIGGER;

    *outBuf++ = left_sample;
    *outBuf++ = left_sample;
}
buffer_full = 0; // indicate that buffer has been processed
return;
}

int main(void)
{
    int n;

    for (n=0 ; n< N ; n++)
    {
        twiddle[n].real = cos(PI*n/N);
        twiddle[n].imag = -sin(PI*n/N);
    }
    L138_initialise_edma(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1)
    {
        while (!buffer_full);
        process_buffer();
    }
}
}

```

Figure 5.43 (Continued)

EXAMPLE 5.6: FFT of Real-Time Input Using TI's C-Callable Optimized Radix-2 FFT Function (L138_fft128_dsplibr2_edma)

Program L138_fft128_dsplibr2_edma.c uses DSPLIB function DSPF_sp_cfftr2_dit() in order to calculate the FFT of a block of samples and uses function gen_w_r2() in order to compute twiddle factors. Other than that it is similar to the previous two examples and should produce similar results.

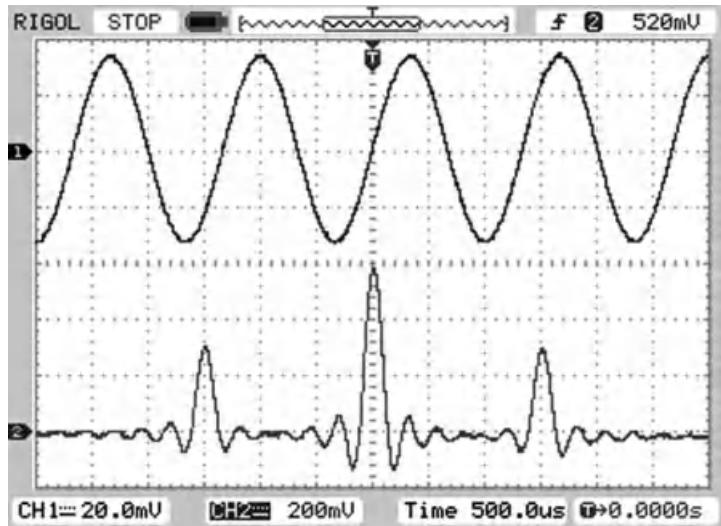


Figure 5.44 Output signal generated by program L138_fft_sinetable_edma.c with BUFCOUNT equal to 64 and FREQ equal to 8, displayed using a Rigol DS1052E oscilloscope.

EXAMPLE 5.7: FFT of a Sinusoidal Signal from a Table Using TI’s C-Callable Optimized DSPLIB FFT Function (L138_fft_sinetable_edma)

This example adapts the previous one to read input from an array initialized to contain one cycle of a sinusoid. Thus, no signal source is required in order to view output signals similar to those in the previous examples. The input signal, read from array sine_table, is output on the right channel of the codec, allowing both input and output signals to be viewed together on an oscilloscope, as shown in Figure 5.44.

The frequency of the input signal may be varied by changing the value of the constant FREQ. Suitable values for this are integers in the range 0 to BUFCOUNT/4 (Figure 5.45).

5.7.2 Fast Convolution

Fast convolution is a technique whereby two time domain sequences of samples are convolved not by direct implementation of the convolution sum but by multiplying together their frequency domain representations. Transformation between time and frequency domains may be implemented efficiently using the fast Fourier transform. An important application of fast convolution is the implementation of FIR filters in which blocks of input samples are convolved with the filter coefficients to produce blocks of filter output samples. The steps involved in fast convolution are as follows:

```

// L138_fastconv_demo.c
//

#include <stdio.h>
#include <math.h>
#include "lp55.cof"
#include "fft.h"

#define PI 3.14159265358979
#define BUFCOUNT 128

float pingIN[BUFCOUNT/2], pingOUT[BUFCOUNT/2];
COMPLEX prodbuf[BUFCOUNT],coeffs[BUFCOUNT],twiddle[BUFCOUNT];
float prodbuf_mag[BUFCOUNT],coeffs_mag[BUFCOUNT];
float overlap[BUFCOUNT/2];
float a,b;
int wt = 0;

int main(void)
{
    int i;

    // set up twiddle factors
    for (i=0 ; i< BUFCOUNT ; i++)
    {
        twiddle[i].real = cos(PI*i/BUFCOUNT);
        twiddle[i].imag = -sin(PI*i/BUFCOUNT);
    }
    // zero overlap array
    for(i=0 ; i<(BUFCOUNT/2) ; i++)
    {
        overlap[i] = 0.0;
    }
    // set up filter coefficients
    for(i=0 ; i<(BUFCOUNT) ; i++)
    { coeffs[i].real = 0.0; coeffs[i].imag = 0.0; }
    for(i=0 ; i<N ; i++) coeffs[i].real = h[i];
    printf("filter coeffs stored in complex array coeffs\n");
    printf("\n"); // place breakpoint here
    // transform zero-padded filter coeffs into frequency domain
    fft(coeffs,BUFCOUNT,twiddle);
    // compute frequency domain magnitude of coefficients
    for (i=0 ; i<BUFCOUNT ; i++)
        coeffs_mag[i] = sqrt(coeffs[i].real*coeffs[i].real
                            + coeffs[i].imag*coeffs[i].imag);
    printf("filter coeffs transformed into frequency domain\n");
    printf("\n"); // place breakpoint here
    while(1)
    {
        // fill array pingIN with new input samples
        for(i=0 ; i<(BUFCOUNT/2) ; i++)
        {
            pingIN[i]=(float)(sin(2*PI*wt/50)+0.25*sin(2*PI*wt/3.3));
            wt++;
        }
        printf("pingIN filled with BUFCOUNT/2 sample values\n");
        printf("\n"); // place breakpoint here
        // zero contents of complex array prodbuf
        for(i=0 ; i<BUFCOUNT ; i++)
        {
            prodbuf[i].real = 0.0;

```

Figure 5.45 Listing of program L138_fastconv_demo.c.

```

    prodbuf[i].imag = 0.0;
}
// copy new input samples into real part of half of prodbuf
for(i=0 ; i<(BUFCOUNT/2) ; i++)
{
    prodbuf[i].real = pingIN[i];
}
printf("prodbuf filled with BUFCOUNT/2 sample values\n");
printf("\n"); // place breakpoint here
// transform zero-padded input samples into frequency domain
fft(prodbuf,BUFCOUNT,twiddle);
for (i=0 ; i<BUFCOUNT ; i++)
// compute frequency domain magnitude of prodbuf contents
    prodbuf_mag[i] = sqrt(prodbuf[i].real*prodbuf[i].real
                           + prodbuf[i].imag*prodbuf[i].imag);
    printf("process buffer transformed\n");
    printf("\n"); // place breakpoint here
// multiply frequency domain representations of input
// and filter coefficients
for (i=0 ; i<BUFCOUNT ; i++)
{
    a = prodbuf[i].real;
    b = prodbuf[i].imag;
    prodbuf[i].real = coeffs[i].real*a - coeffs[i].imag*b;
    prodbuf[i].imag = -(coeffs[i].real*b + coeffs[i].imag*a);
}
// compute frequency domain magnitude of prodbuf contents
for (i=0 ; i<BUFCOUNT ; i++)
    prodbuf_mag[i] = sqrt(prodbuf[i].real*prodbuf[i].real
                           + prodbuf[i].imag*prodbuf[i].imag);
    printf("multiplied\n");
    printf("\n"); // place breakpoint here
// compute inverse FFT
fft(prodbuf,BUFCOUNT,twiddle);
for (i=0 ; i<BUFCOUNT ; i++)
{
    prodbuf[i].real /= BUFCOUNT;
    prodbuf[i].imag /= BUFCOUNT;
}
printf("inverse transformed\n");
printf("\n"); // place breakpoint here
// fill output and overlap buffers
for (i = 0; i < (BUFCOUNT/2) ; i++)
{
    pingOUT[i] = prodbuf[i].real + overlap[i];
    overlap[i] = prodbuf[i+BUFCOUNT/2].real;
}
printf("pingOUT and overlap buffers updated\n");
printf("\n"); // place breakpoint here
}
}

```

Figure 5.45 (Continued)

- (1) Transform a block of input samples into the frequency domain using the FFT.
- (2) Multiply the frequency domain representation of the input signal by the frequency domain representation of the filter coefficients.

- (3) Transform the result back into the time domain using the inverse FFT.

The filter coefficients need to be transformed into the frequency domain only once.

A number of considerations must be taken into account for this technique to be implemented in practice:

- (1) The radix-2 FFT is applicable only to blocks of samples where N is an integer power of 2.
- (2) In order to multiply them together, the frequency domain representations of the input signal and of the filter coefficients must be the same length.
- (3) The result of linearly convolving two sample sequences of lengths N and M is a sequence of length $(N + M - 1)$. If an input sequence is split into blocks of N samples, the result of convolving each block with a block of filter coefficients will be more than N output samples long. In other words, the output due to one block of N input samples extends beyond the corresponding block of N output samples and into the next. These blocks cannot simply be concatenated in order to construct a longer output sequence, but must be overlapped and added.

These considerations are addressed by the following:

- (1) Making N an integer power of 2.
- (2) Processing length $N/2$ blocks of input samples and zero padding both these samples and the filter coefficients used to length N before using an N -point FFT. This requires that the number of filter coefficients is less than $N/2$.
- (3) Overlapping and adding the length N blocks of output samples obtained using an N -point inverse FFT of the result of multiplying frequency domain representations of input samples and filter coefficients.

Fast convolution is implemented by programs `L138_fastconv_demo.c` and `L138_fastconv_edma.c`.

EXAMPLE 5.8: Demonstration of Fast Convolution (`L138_fastconv_demo`)

Program `L138_fastconv_demo.c` does not run in real time, but illustrates the use of fast convolution by allowing the user to halt execution using breakpoints and to examine the contents of various buffers (arrays) within the program at different steps in the process.

A number of `printf()` statements report the progress of the algorithm. Build and load the program and add breakpoints at each line that reads

```
printf("\n"); // place breakpoint here
```

Run the program. The message

```
filter coefficients stored in complex array coeffs
```

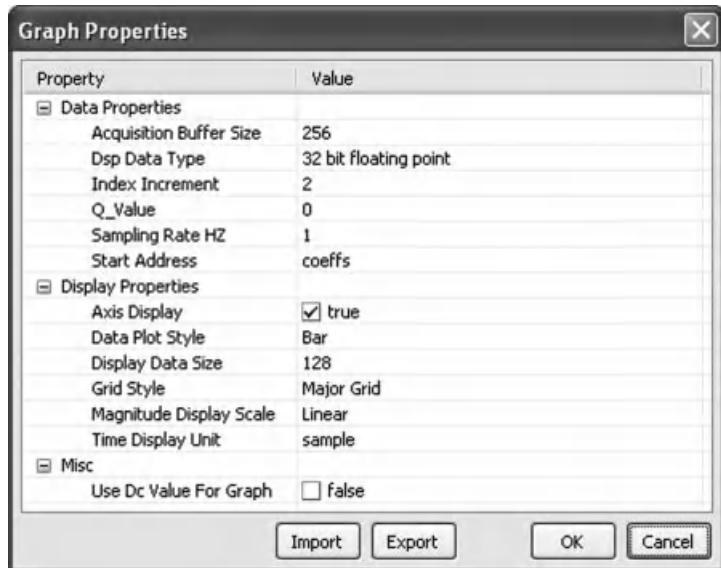


Figure 5.46 *Graph Properties* used to display the real part of the zero-padded filter coefficients used in program L138_fastconv_demo.c.

should appear in the console and the program should halt at the first breakpoint. Select *Tools > Graph > Single Time* and set the *Graph Properties* as shown in Figure 5.46. This should result in a graph of the filter coefficients as shown in Figure 5.47.

This shows the real part of an array of $\text{BUFCOUNT}/2 = 128$ complex floating-point values. Filter coefficients read by the program from file 1p55.coф have been copied into the real parts of the first 55 elements of the array. The imaginary parts of the corresponding elements have been set to zero as have the real and imaginary parts of the remaining 73 elements of the array.

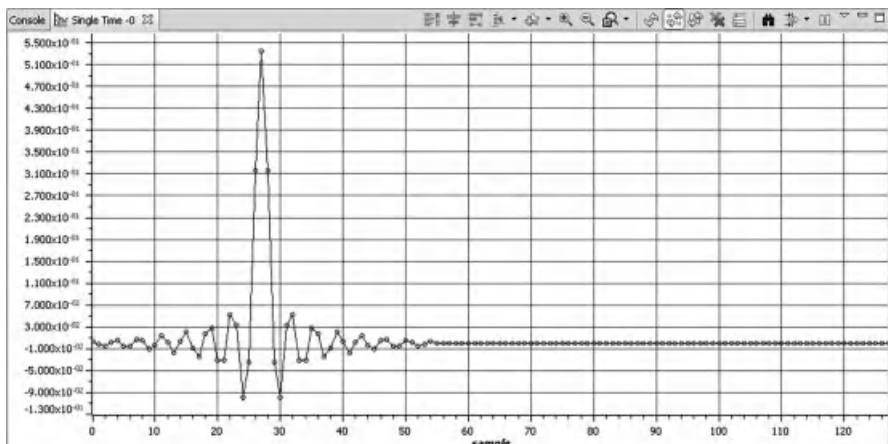


Figure 5.47 Graphical display of the real part of the zero-padded filter coefficients used in program L138_fastconv_demo.c.

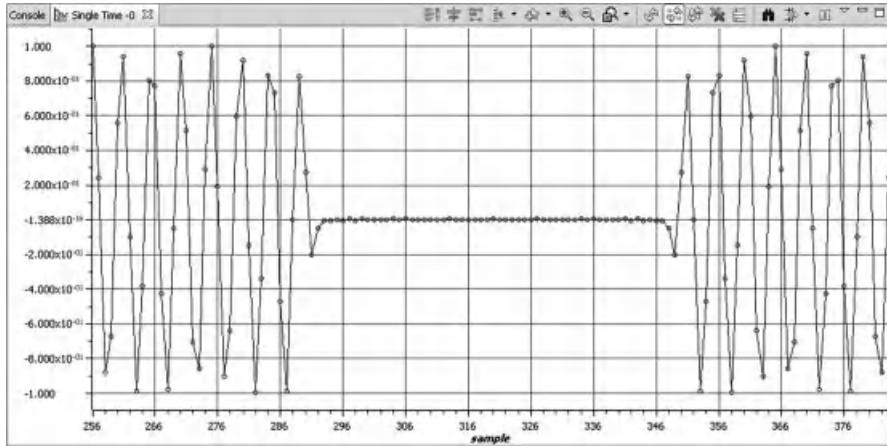


Figure 5.48 Graphical display of the real part of the frequency domain representation of the zero-padded filter coefficients used in program L138_fastconv_demo.c and stored in array coeffs.

Run the program again. The message

```
filter coefficients transformed into frequency domain
```

should appear in the console and the program should halt at the next breakpoint.

The program has applied a 128-point FFT to the complex contents of array coeffs and placed the result in the same array, replacing the time domain filter coefficients with their frequency domain representation. The graph should now appear as shown in Figure 5.48.

The graph shows the real part of a complex frequency domain representation of the filter coefficients. The magnitude of the complex frequency domain representation of the filter coefficients may be displayed by selecting *Tools > Graph > Single Time* and setting *Graph Properties* as shown in Figure 5.49. From the resulting graph, as shown in Figure 5.50, the low pass characteristic of the filter is more readily apparent than in Figure 5.48. The left-hand half of the graph shows the magnitude frequency response of the filter over the positive frequency range 0–4 kHz and the right-hand half shows the magnitude frequency response of the filter over the negative frequency range –4 kHz to 0.

Having computed and stored the frequency domain representation of the filter coefficients, the program enters a loop. Click on the *Run* toolbar button again and the message

```
input buffer pingIN filled with BUFCOUNT/2 new sample values
```

should appear and the program should halt.

Using the *Graph Properties* shown in Figure 5.51, you can now inspect the BUFCOUNT / 2 = 64 real-valued input samples placed in array pingIN (Figure 5.52). These sample values have been calculated according to the expression

$$x(i) = \sin\left(\frac{2\pi i}{50}\right) + 0.25 \sin\left(\frac{2\pi i}{3.3}\right), \quad i = 0, 1, \dots, 63,$$

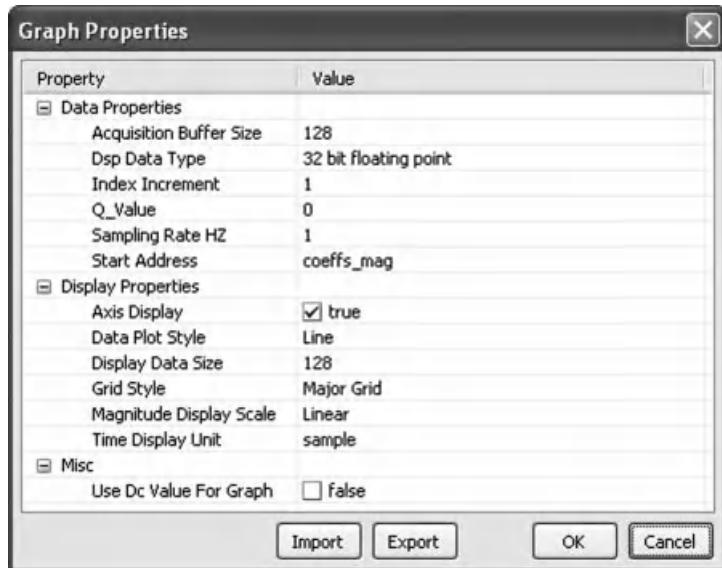


Figure 5.49 *Graph Properties* used to plot the magnitude of the complex values stored in array `coeffs`.

and have been chosen specifically to illustrate the low pass characteristic of the FIR filter implemented. The lower frequency component contained in the input samples should pass through, and the higher frequency component should be stopped by, the filter.

Next, these real-valued input samples are zero padded to length `BUFCOUNT = 128` and copied into the real part of array `procbuf`.

Run the program to the next breakpoint and use the *Graph Properties* shown in Figure 5.53 to display the real part of the contents of array `procbuf` (Figure 5.54).

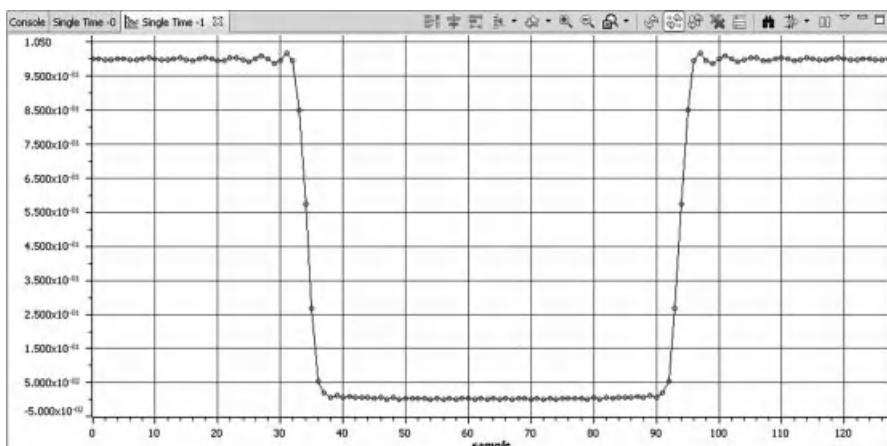


Figure 5.50 The magnitude of the complex values stored in array `coeffs`.

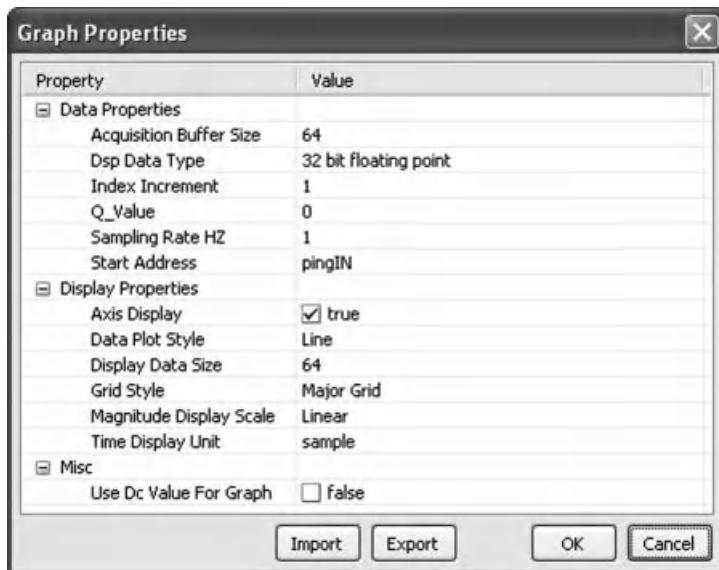


Figure 5.51 *Graph Properties* used to plot the real-valued contents of input buffer pingIN.

A 128-point FFT is then applied to the contents of array prodbuf. Run the program to the next breakpoint, select *Tools > Graph > Single Time*, and use the *Graph Properties* shown in Figure 5.55 to display the magnitude of the complex frequency domain representation of the zero-padded input sample sequence.

In Figure 5.56, the two different frequency components present in the input signal are readily apparent. Next, the complex frequency domain representation of the zero-padded input samples is multiplied by the complex frequency domain representation of the filter coefficients.

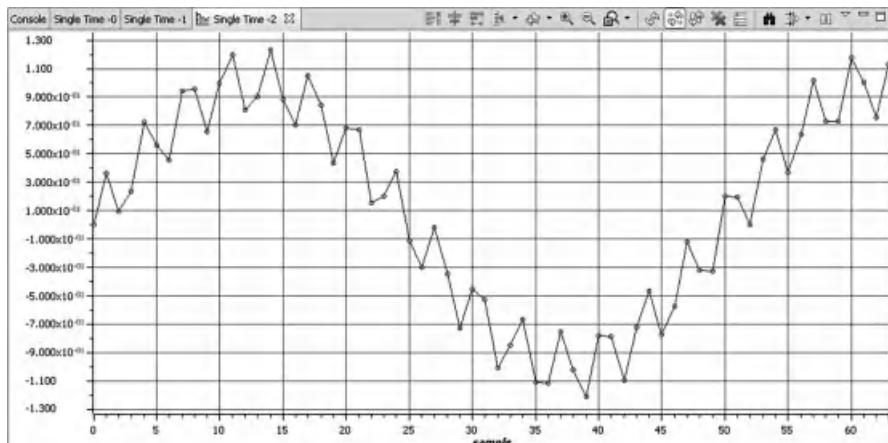


Figure 5.52 The real-valued contents of input buffer pingIN.

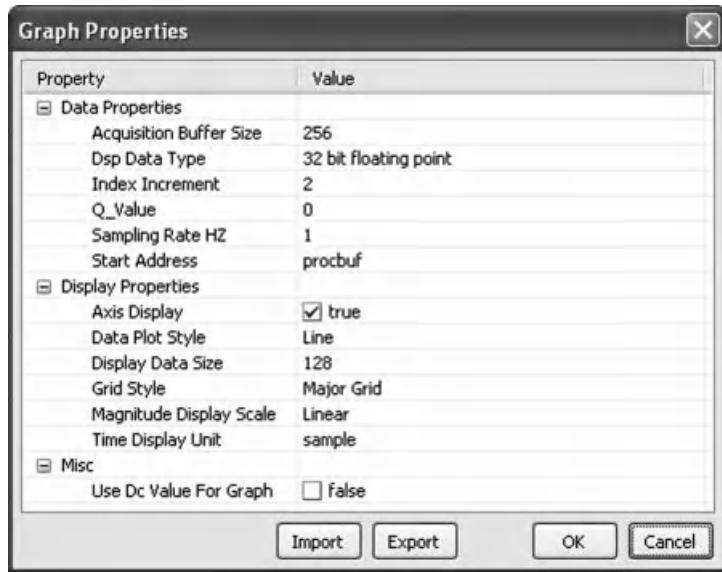


Figure 5.53 Graph Properties used to plot the real part of the contents of array prodbuf.

Run the program to the next breakpoint. The message

multiplied

should appear in the console and the contents of array prodbuf should have changed such that their magnitude is that shown in Figure 5.57.

The program then applies the inverse FFT to the contents of array prodbuf, replacing the complex frequency domain representation of the filter output with its time domain representation. Run the program to the next breakpoint.

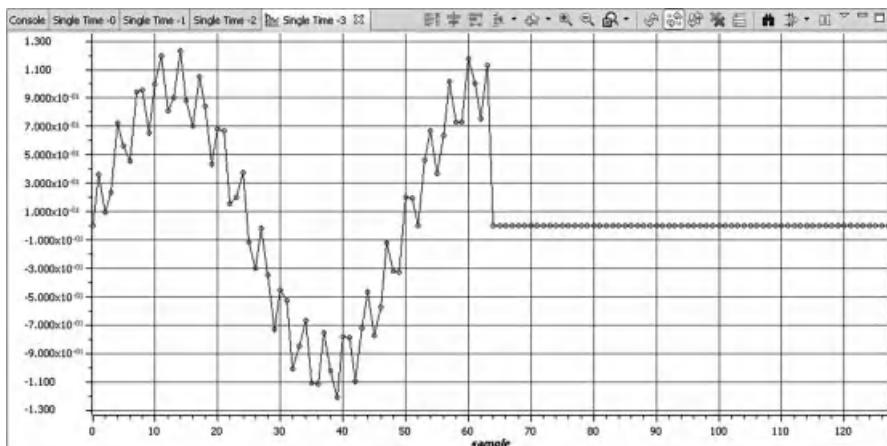


Figure 5.54 The real part of the contents of array prodbuf, showing zero-padded input samples copied from array pingIN.

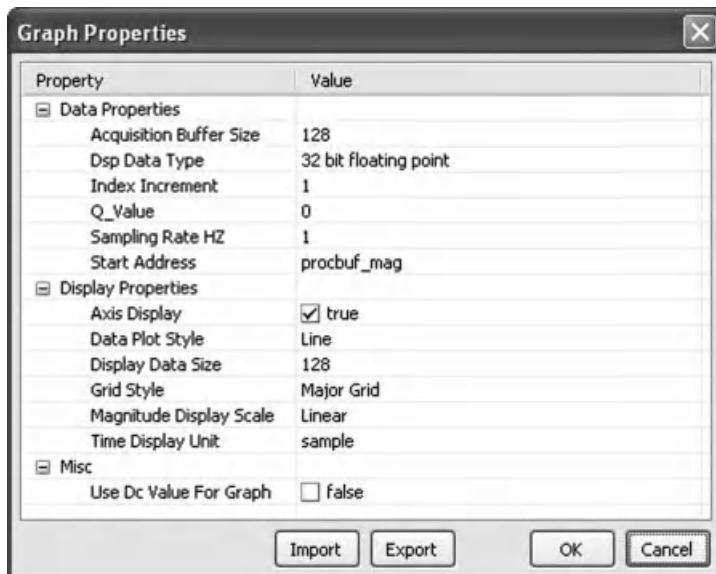


Figure 5.55 *Graph Properties* used to plot the magnitude of the contents of array prodbuf after transformation into the frequency domain.

The message

inverse transformed

should appear on the console.

The real part of the inverse-transformed contents of the array is shown in Figure 5.58.

The sum of the first half of this result (64 samples) and the contents of array overlap is then written to the output buffer pingOUT pointed to by outBuf and the second half of the

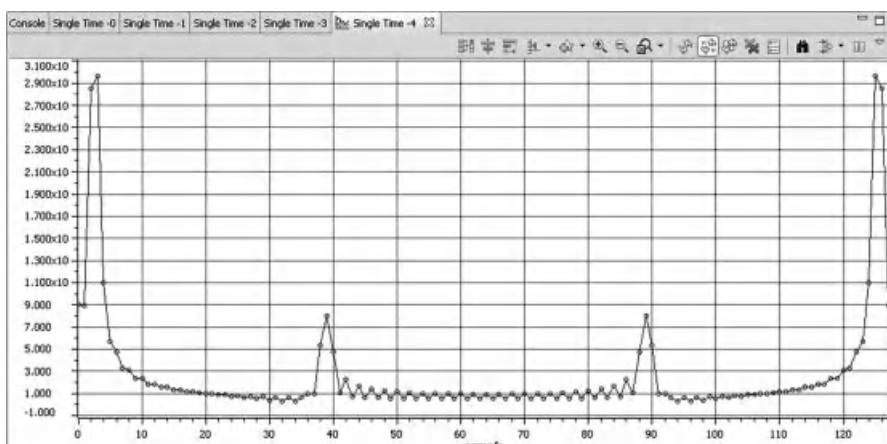


Figure 5.56 The magnitude of the contents of array prodbuf after transformation into the frequency domain.

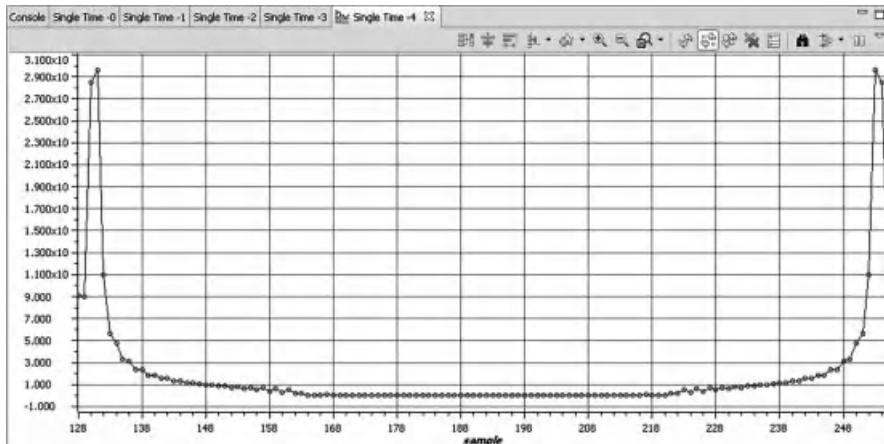


Figure 5.57 The magnitude of the contents of array prodbuf after multiplication by the frequency domain representation of the filter coefficients.

result (64 samples) is saved in array overlap. Run the program to the next breakpoint. The message

pingOUT and overlap buffers updated

should appear on the console. The contents of array pingOUT at this point are shown in Figure 5.60. Evident in this is a delay (in samples) equal to half of the number of filter coefficients used, and a start-up transient applied to the input samples shown in Figure 5.52. The process is then repeated.

- (1) Sixty-four new, real-valued input samples are copied into pingIN.
- (2) These samples are zero padded and copied into complex array prodbuf.

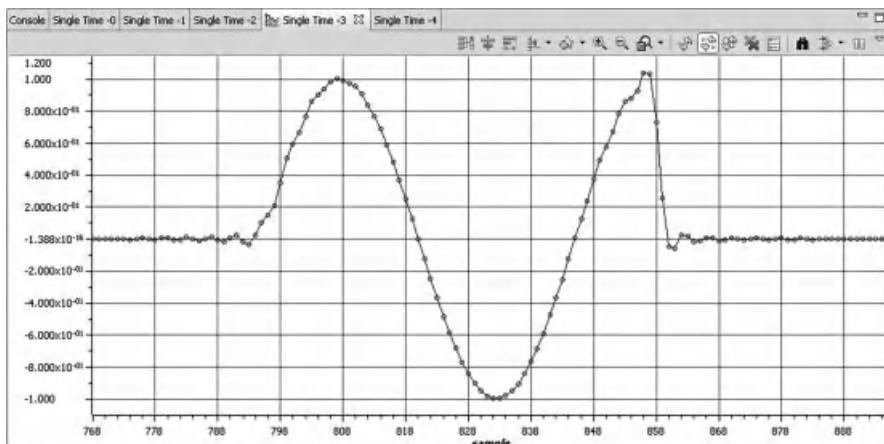


Figure 5.58 The real part of the inverse-transformed contents of the array prodbuf.

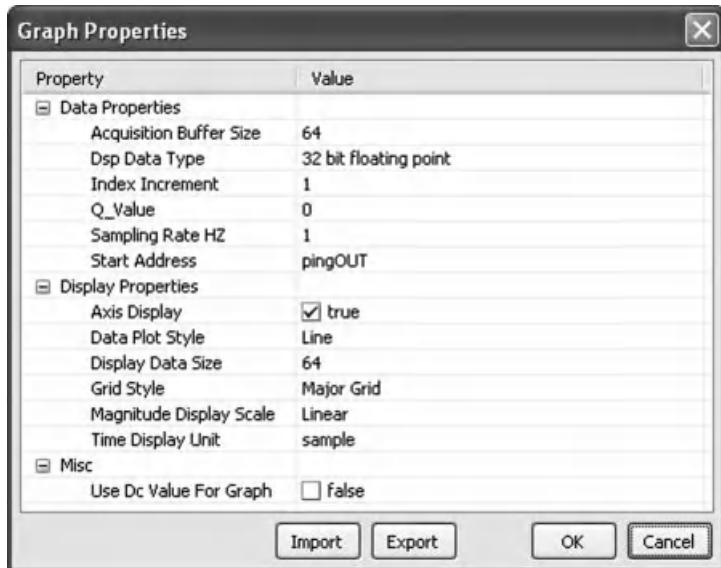


Figure 5.59 Graph Properties used to plot the contents of output array pingOUT.

- (3) The contents of array prodbuf are transformed into the frequency domain.
- (4) The contents of array prodbuf are multiplied by the frequency domain representation of the filter, stored in array coeffs.
- (5) The result in array prodbuf is transformed back into the time domain.

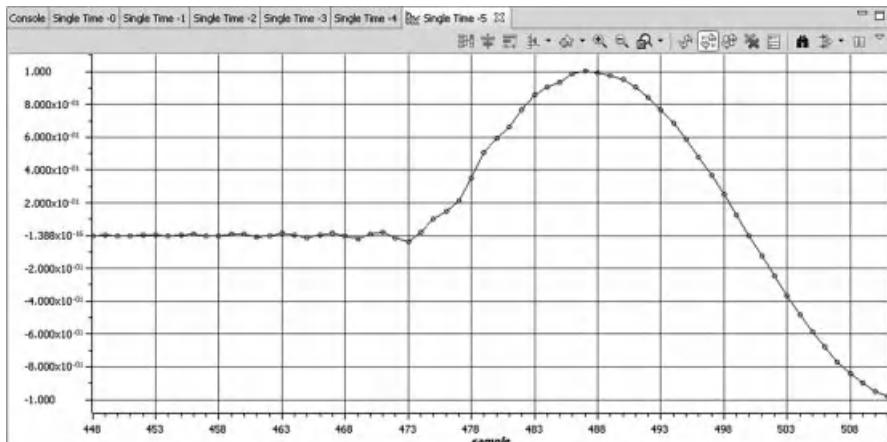


Figure 5.60 Contents of output array pingOUT showing a delay and start up transient.

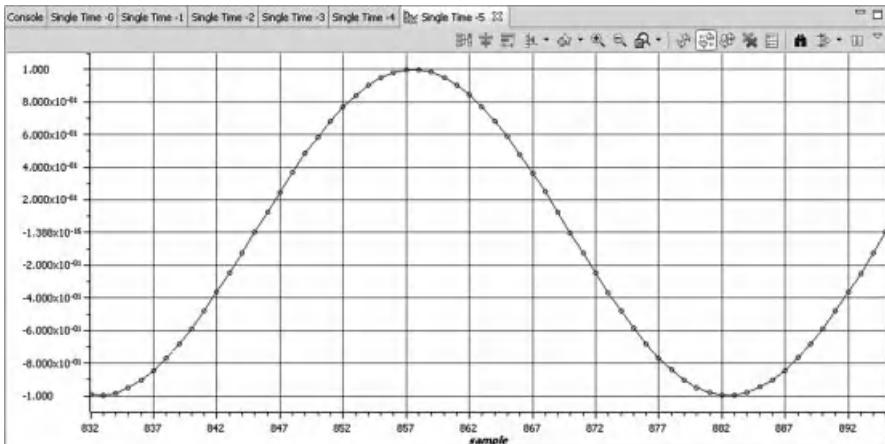


Figure 5.61 Contents of output array pingOUT following the second iteration of the fast convolution algorithm.

- (6) The first half (64 samples) of the time domain result is added to the contents of array overlap and copied into array pingOUT and the second half (64 samples) is copied into array overlap.

The contents of array pingOUT, following the second iteration of the algorithm, is shown in Figure 5.61. After an initial delay and transient evident in the first block of output samples shown in Figure 5.60, the output is a smooth sinusoid.

EXAMPLE 5.9: Real-Time Fast Convolution (L138_fastconv_edma)

Program L138_fastconv_edma.c implements exactly the same fast convolution algorithm illustrated by program L138_fastconv_demo.c, but does so in real time and uses DMA-based I/O in order to read consecutive blocks of BUFCOUNT/2 input samples from the ADC and to write consecutive blocks of BUFCOUNT/2 output samples to the DAC (Figure 5.62).

Build and run the program and verify that it implements a low-pass filter. Functionally, the program is equivalent to program L138_fir_intr.c, described in Chapter 3, but introduces an additional delay of BUFCOUNT sampling instants. The program has been written so that, just as in the case of program L138_fir_intr.c, different FIR filter coefficient (.cof) files can be used simply by changing the line that reads

```
#include "lp55.cof"
```

Note that the maximum possible value of N (the number of filter coefficients, defined in the .cof file) is BUFCOUNT/2. For longer FIR filter impulse responses, the value of BUFCOUNT must be increased by changing the line in file L138_aic3106_init.h that reads

```
#define BUFCOUNT 256.
```

```

// L138_fastconv_edma.c
//
//
// BUFCOUNT is set in L138_aic3106_init.h - usually 256
// this sets the number of samples per edma frame to BUFCOUNT/2
// we will take edma frames of samples and zero pad them to
// BUFCOUNT and apply BUFCOUNT point FFT
//
// in this program, N is the number of filter coefficients

#include "L138_aic3106_init.h"
#include <math.h>
#include "lp55.cof"
#include "fft.h"

#define PI 3.14159265358979

extern int16_t *pingIN, *pingOUT, *pongIN, *pongOUT;
volatile int buffer_full = 0;
int procBuffer;

COMPLEX prodbuf[BUFCOUNT],coeffs[BUFCOUNT],twiddle[BUFCOUNT];
float overlap[BUFCOUNT/2];
float a,b;

interrupt void interrupt4(void) // interrupt service routine
{
    switch(EDMA_3CC_IPR)
    {
        case 1:                      // TCC = 0
            procBuffer = PING;       // process ping
            EDMA_3CC_ICR = 0x0001;   // clear EDMA3 IPR bit TCC
            break;
        case 2:                      // TCC = 1
            procBuffer = PONG;      // process pong
            EDMA_3CC_ICR = 0x0002;   // clear EDMA3 IPR bit TCC
            break;
        default:                     // may have missed an interrupt
            EDMA_3CC_ICR = 0x0003;   // clear EDMA3 IPR bits 0 and 1
            break;
    }
    EVTCLR0 = 0x00000100;
    buffer_full = 1;              // flag EDMA3 transfer
    return;
}

void process_buffer(void) // should be called from infinite
                         // loop in main() when process
                         // buffer has been filled
{
    int16_t *inBuf, *outBuf;      // pointers to process buffers
    int i;

    if (procBuffer == PING)      // use ping or pong buffers
    {
        inBuf = pingIN;
        outBuf = pingOUT;
    }
    if (procBuffer == PONG)
    {
        inBuf = pongIN;

```

Figure 5.62 Listing of program L138_fastconv_edma.c.

```

    outBuf = pongOUT;
}

for (i = 0; i < (BUFCOUNT) ; i++) // initialise prodbuf
{
    prodbuf[i].real = 0.0;
    prodbuf[i].imag = 0.0;
}
for (i = 0; i < (BUFCOUNT/2) ; i++) // fill half of prodbuf
{
    prodbuf[i].real = (float)(*inBuf++);
    inBuf++;
}
fft(prodbuf,BUFCOUNT,twiddle);

for (i=0 ; i<BUFCOUNT ; i++) // filter in frequency domain
{
    a = prodbuf[i].real;
    b = prodbuf[i].imag;
    prodbuf[i].real = coeffs[i].real*a - coeffs[i].imag*b;
    prodbuf[i].imag = -(coeffs[i].real*b + coeffs[i].imag*a);
}
fft(prodbuf,BUFCOUNT,twiddle);

for (i=0 ; i<BUFCOUNT ; i++)
{
    prodbuf[i].real /= BUFCOUNT;
}

for (i = 0; i < (BUFCOUNT/2) ; i++)
{
    *outBuf++ = (int16_t)(prodbuf[i].real + overlap[i]);
    *outBuf++;
    overlap[i] = prodbuf[i+BUFCOUNT/2].real;
}
buffer_full = 0; // flag buffer processed
return;
}

int main(void)
{
    int i;

    for (i=0 ; i< BUFCOUNT ; i++) // set up twiddle factors
    {
        twiddle[i].real = cos(PI*i/BUFCOUNT);
        twiddle[i].imag = -sin(PI*i/BUFCOUNT);
    }
L138_initialise_edma(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
for(i=0 ; i<(BUFCOUNT) ; i++)
{ coeffs[i].real = 0.0; coeffs[i].imag = 0.0;}
for(i=0 ; i<N ; i++) coeffs[i].real = h[i];
fft(coeffs,BUFCOUNT,twiddle);

while(1)
{
    while (!buffer_full);
    process_buffer();
}
}

```

Figure 5.62 (Continued)

```

// L138_graphicEQ_DSPLIB_edma.c
//

#include "L138_aic3106_init.h"
#include <math.h>
#include "dsplib674x.h"
#include "GraphicEQcoeff.h"
#include "evmomapl138_dip.h"

extern int16_t *pingIN, *pingOUT, *pongIN, *pongOUT;
volatile int buffer_full = 0;
int procBuffer;

short NUMCOEFFS = sizeof(lpcoeff)/sizeof(float);

void gen_w_r2(float* w, int n);
void bit_rev(float* x, int n);

float xL[2*BUFCOUNT];
float xR[2*BUFCOUNT];
float w[2*BUFCOUNT];
float coeffs[2*BUFCOUNT];
float bass[2*BUFCOUNT], mid[2*BUFCOUNT], treble[2*BUFCOUNT];
float overlapL[BUFCOUNT/2];
float overlapR[BUFCOUNT/2];
float a,b;
float bass_gain = 0.0;
float mid_gain = 0.0;
float treble_gain = 1.0;

#pragma DATA_ALIGN(xL, 8);
#pragma DATA_ALIGN(xR, 8);
#pragma DATA_ALIGN(w, 8);
#pragma DATA_ALIGN(coeffs, 8);

interrupt void interrupt4(void) // interrupt service routine
{
    switch(EDMA_3CC_IPR)
    {
        case 1:                      // TCC = 0
            procBuffer = PING;       // process ping
            EDMA_3CC_ICR = 0x0001;   // clear EDMA3 IPR bit TCC
            break;
        case 2:                      // TCC = 1
            procBuffer = PONG;      // process pong
            EDMA_3CC_ICR = 0x0002;   // clear EDMA3 IPR bit TCC
            break;
        default:                     // may have missed an interrupt
            EDMA_3CC_ICR = 0x0003;   // clear EDMA3 IPR bits 0 and 1
            break;
    }
    EVTCLR0 = 0x000000100;
    buffer_full = 1;              // flag EDMA3 transfer
    return;
}

void process_buffer(void) // should be called from infinite
// loop in main() when process

```

Figure 5.63 Listing of program L138_graphicEQ_DSPLIB_edma.c.

```

        // buffer has been filled
{
    int16_t *inBuf, *outBuf;           // pointers to process buffers
    int i;

    if (procBuffer == PING)          // use ping or pong buffers
    {
        inBuf = pingIN;
        outBuf = pingOUT;
    }
    if (procBuffer == PONG)
    {
        inBuf = pongIN;
        outBuf = pongOUT;
    }
    for (i=0; i<(BUFCOUNT*2) ; i++)
    {
        xL[i] = 0.0;
        xR[i] = 0.0;
    }
    for (i=0; i<(BUFCOUNT/2) ; i++)
    {
        xL[i*2] = (float)(*inBuf++);
        xR[i*2] = (float)(*inBuf++);
    }

    DSPF_sp_cfftr2_dit (xL, w, BUFCOUNT);
    bit_rev(xL, BUFCOUNT);
    DSPF_sp_cfftr2_dit (xR, w, BUFCOUNT);
    bit_rev(xR, BUFCOUNT);

    for (i=0 ; i<BUFCOUNT ; i++)
        coeffs[i] = bass[i]*bass_gain
                    + mid[i]*mid_gain
                    + treble[i]*treble_gain;
    for (i=0 ; i<BUFCOUNT ; i++) // filter L in frequency domain
    {
        a = xL[i*2]; // real part of result
        b = xL[i*2+1]; // imag part of result
        xL[i*2] = coeffs[i*2]*a - coeffs[i*2+1]*b;
        xL[i*2 + 1] = -(coeffs[i*2]*b + coeffs[i*2+1]*a);
    }
    for (i=0 ; i<BUFCOUNT ; i++) // filter R in frequency domain
    {
        a = xR[i*2]; // real part of result
        b = xR[i*2+1]; // imag part of result
        xR[i*2] = coeffs[i*2]*a - coeffs[i*2+1]*b;
        xR[i*2 + 1] = -(coeffs[i*2]*b + coeffs[i*2+1]*a);
    }
    DSPF_sp_cfftr2_dit (xL, w, BUFCOUNT);
    bit_rev(xL, BUFCOUNT);
    for (i=0 ; i<BUFCOUNT ; i++) xL[i*2] /= BUFCOUNT;
    DSPF_sp_cfftr2_dit (xR, w, BUFCOUNT);
    bit_rev(xR, BUFCOUNT);
    for (i=0 ; i<BUFCOUNT ; i++) xR[i*2] /= BUFCOUNT;

    for (i = 0; i < (BUFCOUNT/2) ; i++)
    {
        *outBuf++ = (int16_t)(xL[i*2] + overlapL[i]);
        *outBuf++ = (int16_t)(xR[i*2] + overlapR[i]);
        overlapL[i] = xL[(i*2)+BUFCOUNT];
    }
}

```

Figure 5.63 (Continued)

```

    overlapR[i] = xR[(i*2)+BUFCOUNT];
}
buffer_full = 0; // indicate that buffer has been processed
return;
}

void main(void)
{
    int i;
    uint32_t rtn;
    uint8_t DIP_value;

    L138_initialise_edma(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    gen_w_r2(w,BUFCOUNT);
    bit_rev(w,BUFCOUNT>>1);

    rtn = DIP_init();
    if (rtn != ERR_NO_ERROR)
    {
        printf("error initializing dip switches!\r\n");
        return (rtn);
    }

    for(i=0 ; i<(BUFCOUNT*2) ; i++)
    {
        coeffs[i] = 0.0;
        bass[i] = 0.0;
        mid[i] = 0.0 ;
        treble[i] = 0.0;
    }
    for(i=0 ; i<(BUFCOUNT/2) ; i++)
    {
        overlapL[i] = 0.0;
        overlapR[i] = 0.0;
    }
    for(i=0 ; i<NUMCOEFFS ; i++)
    {
        bass[i*2] = lpcoeff[i];
        mid[i*2] = bpcoeff[i];
        treble[i*2] = hpcoeff[i];
    }
    DSPF_sp_cfftr2_dit(bass, w, BUFCOUNT);
    bit_rev(bass, BUFCOUNT);
    DSPF_sp_cfftr2_dit(mid, w, BUFCOUNT);
    bit_rev(mid, BUFCOUNT);
    DSPF_sp_cfftr2_dit(treble, w, BUFCOUNT);
    bit_rev(treble, BUFCOUNT);

    while(1)
    {
        while (!buffer_full);
        process_buffer();
        rtn = DIP_get(0, &DIP_value);
        if(DIP_value==0) bass_gain=1.0; else bass_gain=0.0;
        rtn = DIP_get(1, &DIP_value);
        if(DIP_value==0) mid_gain=1.0; else mid_gain=0.0;
        rtn = DIP_get(2, &DIP_value);
        if(DIP_value==0) treble_gain=1.0; else treble_gain=0.0;
    }
}

```

Figure 5.63 (Continued)

```

// generate real and imaginary twiddle table of size n/2
void gen_w_r2(float* w, int n)
{
    int i;
    float pi = 4.0 * atan(1.0);
    float e = pi * 2.0 / n;

    for(i = 0; i < (n >> 1); i++)
    {
        w[2 * i] = cos(i * e);
        w[2 * i + 1] = sin(i * e);
    }
}

// bit reverse coefficients
void bit_rev(float* x, int n)
{
    int i, j, k;
    float rtemp, itemp;

    j = 0;

    for(i=1; i < (n-1); i++)
    {
        k = n >> 1;
        while(k <= j)
        {
            j -= k;
            k >>= 1;
        }
        j += k;
        if(i < j)
        {
            rtemp = x[j*2];
            x[j*2] = x[i*2];
            x[i*2] = rtemp;
            itemp = x[j*2+1];
            x[j*2+1] = x[i*2+1];
            x[i*2+1] = itemp;
        }
    }
}

```

Figure 5.63 (Continued)**EXAMPLE 5.10: Graphic Equalizer (L138_graphicEQ_DSPLIB_edma)**

Figure 5.63 shows a listing of the program L138_graphicEQ_DSPLIB_edma.c, which implements a three-band graphic equalizer. TI's floating-point complex radix-2 FFT library function DSPF_sp_cfftr2_dit() is used again in this project (see also Examples 5.6 and 5.7).

The coefficient file graphicEQcoeff.h contains three sets of coefficients: low pass at 1.3 kHz, band pass between 1.3 and 2.6 kHz, and high pass at 2.6 kHz, designed using the MATLAB function fir1(). The three sets of filter coefficients are transformed into the

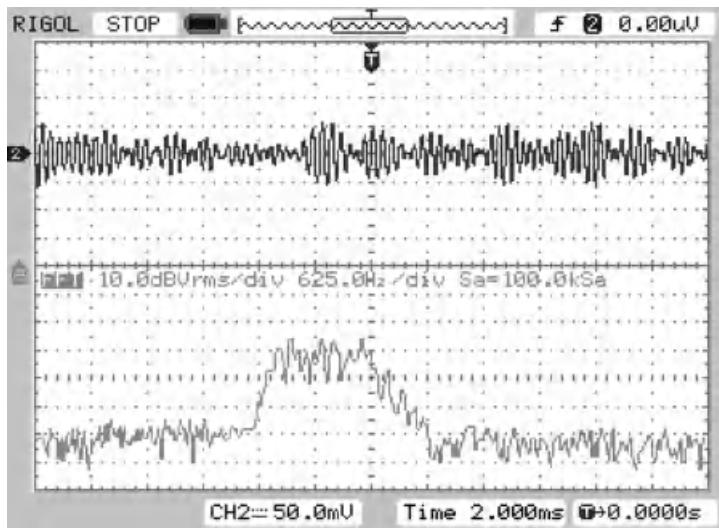


Figure 5.64 Output from program L138_graphicEQ_DSPLIB_edma.c, using pseudorandom noise as input, with DIP switch #1 ON, DIP switch #2 ON, and DIP switch #3 OFF.

frequency domain and linear, weighted sums of their frequency domain representations are used in the fast convolution process. A similar overlap-add scheme to that used in Examples 5.8 and 5.9 is employed.

Build and run this program using the test file `left_noise.wav`, played using *Goldwave*, as an input signal and observe the frequency content of the output signal using

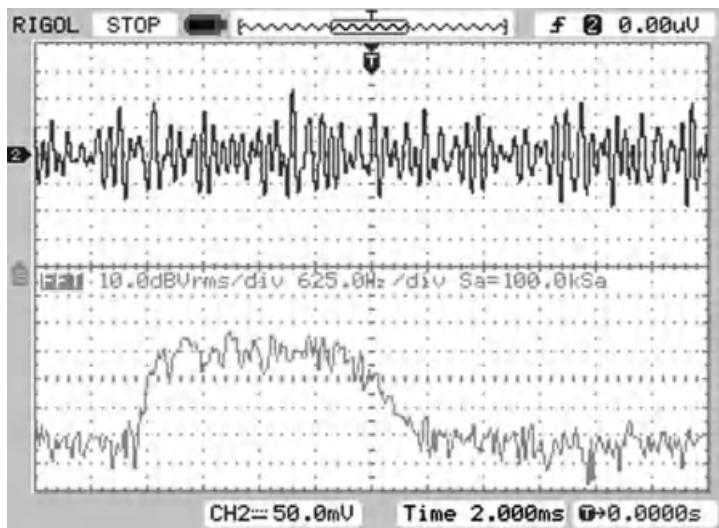


Figure 5.65 Output from program L138_graphicEQ_DSPLIB_edma.c, using pseudorandom noise as input, with DIP switch #1 ON, DIP switch #2 OFF, and DIP switch #3 OFF.

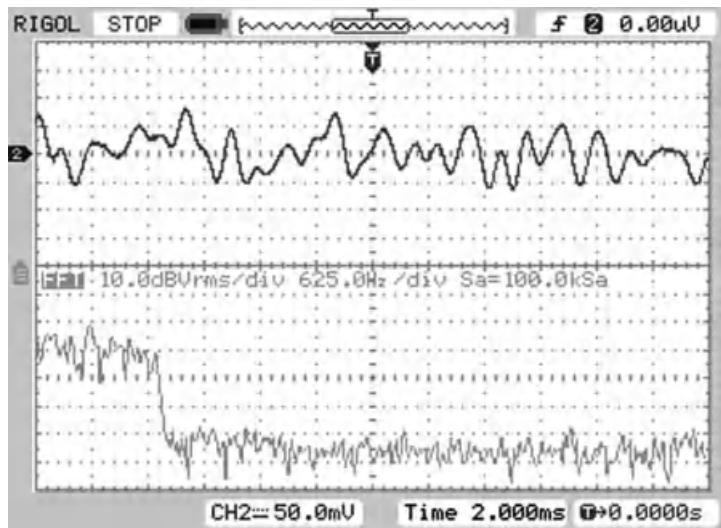


Figure 5.66 Output from program L138_graphicEQ_DSPLIB_edma.c, using pseudorandom noise as input, with DIP switch #1 OFF, DIP switch #2 ON, and DIP switch #3 ON.

an oscilloscope or spectrum analyzer. DIP switches #1, #2, and #3 control whether the gains in the three frequency bands are set equal to 0.0 or 1.0. These settings may be changed while the program is running. Figures 5.64–5.66 show the output signals for three different DIP switch settings, using wideband noise as an input signal.

REFERENCES

1. *TMS320C6748/46/42 and OMAP-L138 Processor Enhanced Direct Memory Access (EDMA3) Controller User's Guide*, SPRUGP9B, Texas Instruments, Dallas, TX, 2010.
2. *OMAP-L138 C6-Integra™ DSP+ARM® Processor*, SPRS586C, Texas Instruments, Dallas, TX, 2011.
3. R. Chassaing, *Digital Signal Processing Laboratory Experiments Using C and the TMS320C31 DSK*, John Wiley & Sons, Inc., New York, NY, 1999.

Chapter 6

Adaptive Filters

- Adaptive filter configurations
- Linear adaptive filters
- The least mean squares (LMS) algorithm
- Programming examples using C

6.1 INTRODUCTION

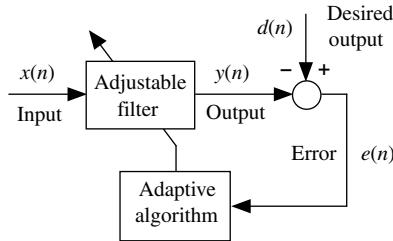
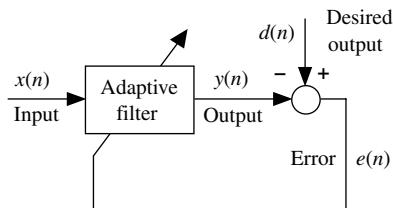
Adaptive filters are used in situations where the characteristics or statistical properties of the signals involved are either unknown or time-varying. Typically, a nonadaptive FIR or IIR filter is designed with reference to particular signal characteristics. But if the signal characteristics encountered by such a filter are not those for which it was specifically designed, then its performance may be suboptimal. The coefficients of an adaptive filter are adjusted in such a way that its performance according to some measure improves with time and approaches optimum performance. Thus, an adaptive filter can be very useful either when there is uncertainty about the characteristics of a signal or when these characteristics are time-varying.

Adaptive systems have the potential to outperform nonadaptive systems. However, they are, by definition, nonlinear and more difficult to analyze than linear, time-invariant systems. This chapter is concerned with linear adaptive systems, that is, systems that, when adaptation is inhibited, have linear characteristics. More specifically, the filters considered here are adaptive FIR filters.

At the heart of the adaptive systems considered in this chapter is the structure shown in block diagram form in Figure 6.1.

Its component parts are an adjustable filter, a mechanism for performance measurement (in this case, a comparator to measure the instantaneous error $e(n)$ between adaptive filter output $y(n)$ and desired output $d(n)$) and an adaptation mechanism or algorithm. In subsequent figures, the adaptation mechanism is incorporated into the adjustable filter block, as shown in Figure 6.2.

It is conventional to refer to the coefficients of an adaptive FIR filter as weights, and the filter coefficients of the adaptive filters in the program examples in this chapter

**Figure 6.1** Basic adaptive filter structure.**Figure 6.2** Simplified block diagram of basic adaptive filter structure.

are stored in arrays using the identifiers w or `weights` rather than h as used for FIR filters in Chapter 3. The weights of the adaptive FIR filter are adjusted so as to minimize the mean squared value $\xi(n)$ of the error $e(n)$. The mean squared error $\xi(n)$ is defined as the expected value, or hypothetical mean, of the square of the error, that is,

$$\xi(n) = E[e^2(n)] \quad (6.1)$$

This quantity may also be interpreted as representing the variance, or power, of the error signal.

6.2 ADAPTIVE FILTER CONFIGURATIONS

Four basic configurations into which the adaptive filter structure of Figure 6.2 may be incorporated are commonly used. The differences between the configurations concern the derivation of the desired output signal $d(n)$.

Each configuration may be explained assuming that the adaptation mechanism will adjust the filter weights so as to minimize the mean squared value $\xi(n)$ of the error signal $e(n)$, but without the need to understand *how* the adaptation mechanism works.

6.2.1 Adaptive Prediction

In this configuration (Figure 6.3), a delayed version of the desired signal $x(n) = s(n - k)$ is input to the adaptive filter which predicts the current value of the

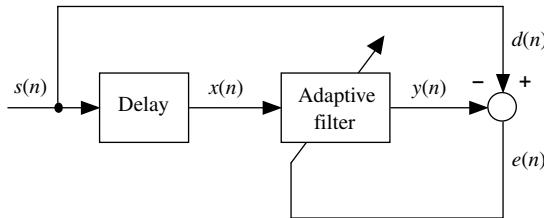


Figure 6.3 Basic adaptive filter structure configured for prediction.

desired signal $s(n)$. In doing this, the filter learns something about the characteristics of the signal $s(n)$ and of the process that generated it. Adaptive prediction is used widely in signal encoding and noise reduction.

6.2.2 System Identification or Direct Modeling

In this configuration, (Figure 6.4) broadband noise $s(n)$ is input to both the adaptive filter and an unknown plant or system. If adaptation is successful and the mean squared error is minimized (to zero), then it follows that the outputs of both systems (in response to the same input signal) are similar and that the characteristics of the systems are equivalent. The adaptive filter has identified the unknown plant by taking on its characteristics. This configuration was introduced in Chapter 2 as a means of identifying or measuring the characteristics of the AIC3106 codec and was used again in some of the examples in Chapters 3 and 4. A common application of this configuration is echo cancellation in communication systems (Figure 6.4).

6.2.3 Noise Cancellation

This configuration differs from the previous two in that while the mean squared error is minimized, it is not, even in the ideal case, minimized to zero and it is the error signal $e(n)$ rather than the adaptive filter output $y(n)$ that is the principal signal of interest. Consider the system illustrated in Figure 6.5. A primary sensor is positioned so as to pick up signal s . However, this signal is corrupted by uncorrelated additive noise n_0 , that is, the primary sensor picks up signal $(s + n_0)$. A second reference sensor

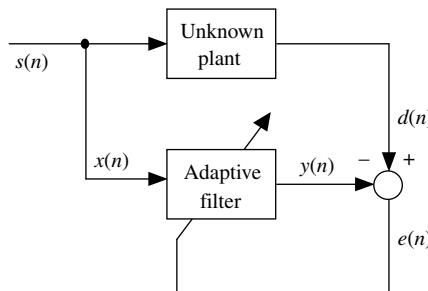


Figure 6.4 Basic adaptive filter structure configured for system identification.

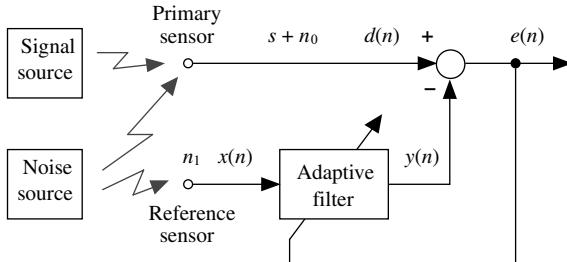


Figure 6.5 Basic adaptive filter structure configured for noise cancellation.

is positioned so as to pick up noise from the same source as n_0 , but without picking up signal s . This noise signal is represented in Figure 6.5 as n_1 . Since they originate from the same source, it may be assumed that noise signals n_0 and n_1 are strongly correlated. Here it is also assumed that neither noise signal is correlated with signal s . In practice, the reference sensor may pick up signal s to some degree and there may be some correlation between signal and noise signals, leading to a reduction in the performance of the noise cancellation system.

The noise cancellation system aims to subtract the additive noise n_0 from the primary sensor output ($s + n_0$). The role of the adaptive filter is therefore to estimate, or derive, n_0 from n_1 and intuitively (since the two signals originate from the same source) this appears feasible.

An alternative representation of the situation described above, regarding the signals detected by the two sensors and the correlation between n_0 and n_1 , is shown in Figure 6.6. Here it is emphasized that n_0 and n_1 have taken different paths from the same noise source to the primary and reference sensors, respectively. Note the similarity between this and the system identification configuration of Figure 6.4. The mean squared error $\xi(n)$ may be minimized if the adaptive filter is adjusted to have similar characteristics to the block shown between signals n_1 and n_0 . In effect, the adaptive filter learns the difference in the paths between the noise source and the primary and reference sensors, represented in Figure 6.6 by the block labeled $H(z)$. The minimized error signal will, in this idealized situation, be equal to the signal s , that is, a noise-free signal.

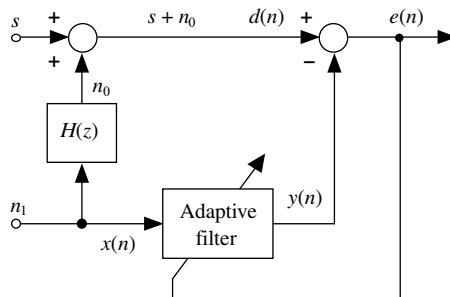


Figure 6.6 Alternative representation of basic adaptive filter structure configured for noise cancellation emphasizing the difference $H(z)$ in paths from a single noise source to primary and reference sensors.

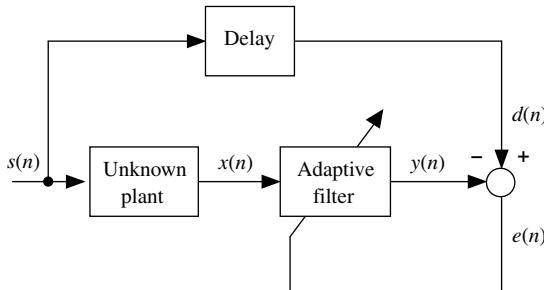


Figure 6.7 Basic adaptive filter structure configured for equalization.

6.2.4 Equalization

In this configuration, (Figure 6.7) the adaptive filter is used to recover a delayed version of signal $s(n)$ from signal $x(n)$ (formed by passing $s(n)$ through an unknown plant or filter). The delay is included to allow for the propagation of signals through the plant and adaptive filter. After successful adaptation, the adaptive filter takes on the inverse characteristics of the unknown filter, although there are limitations on the nature of the unknown plant for this to be achievable. Commonly, the unknown plant is a communication channel and $s(n)$ is the signal being transmitted through that channel. It is logical at this point to ask why, if a delayed but unfiltered version of signal $s(n)$ is available for use as the desired signal $d(n)$ at the receiver, it is necessary to attempt to derive a delayed but unfiltered version of $s(n)$ from signal $x(n)$. In general, a delayed version of $s(n)$ is not available at the receiver; but for the purposes of adaptation over short periods of time, it is effectively made available by transmitting a predetermined sequence $s(n)$ and using a copy of this stored at the receiver as the desired signal $d(n)$. Often a pseudorandom, broadband signal is used for this purpose.

6.3 PERFORMANCE FUNCTION

Consider the block diagram representation of an FIR filter introduced in Chapter 3 (Figure 6.8).

In the following equations, the filter weights and the input samples stored in the FIR filter delay line at the n th sampling instant are represented as vectors $\mathbf{w}(n)$ and $\mathbf{x}(n)$, respectively, where

$$\mathbf{w}(n) = [w_0(n), w_1(n), \dots, w_{N-1}(n)]^T \quad (6.2)$$

and

$$\mathbf{x}(n) = [x(n), x(n-1), \dots, x(n-N+1)]^T. \quad (6.3)$$

Hence, using vector notation, the filter output at the n th sample instant is given by

$$y(n) = \mathbf{w}^T(n)\mathbf{x}(n) = \mathbf{x}^T(n)\mathbf{w}(n). \quad (6.4)$$

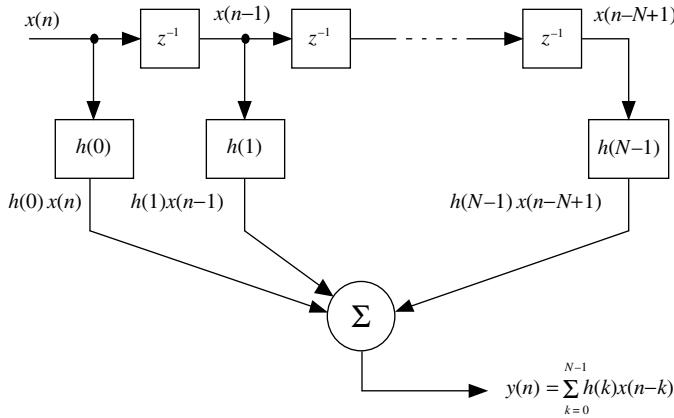


Figure 6.8 Block diagram representation of FIR filter.

Instantaneous error is given by

$$\begin{aligned} e(n) &= d(n) - y(n) \\ &= d(n) - \mathbf{x}^T(n)\mathbf{w}(n), \end{aligned} \quad (6.5)$$

and the instantaneous squared error is given by

$$e^2(n) = d^2(n) - 2d(n)\mathbf{x}^T(n)\mathbf{w}(n) + \mathbf{w}^T(n)\mathbf{x}(n)\mathbf{x}^T(n)\mathbf{w}(n). \quad (6.6)$$

Mean squared error (expected value of squared error) is therefore given by

$$\begin{aligned} \xi(n) &= E[e^2(n)] \\ &= E[d^2(n) - 2d(n)\mathbf{x}^T(n)\mathbf{w}(n) + \mathbf{w}^T(n)\mathbf{x}(n)\mathbf{x}^T(n)\mathbf{w}(n)]. \end{aligned} \quad (6.7)$$

The expected value of a sum of variables is equal to the sum of the expected values of those variables. However, the expected value of a product of variables is the product of the expected values of the variables only if those variables are statistically independent. Signals $d(n)$ and $x(n)$ are generally not statistically independent. If the signals $d(n)$ and $x(n)$ are statistically time invariant, the expected values of the products $d(n)\mathbf{x}(n)$ and $\mathbf{x}(n)\mathbf{x}^T(n)$ are constants and hence

$$\begin{aligned} \xi(n) &= E[d^2(n)] + \mathbf{w}^T(n)E[\mathbf{x}(n)\mathbf{x}^T(n)]\mathbf{w}(n) - 2E[d(n)\mathbf{x}^T(n)]\mathbf{w}(n) \\ &= E[d^2(n)] + \mathbf{w}^T(n)\mathbf{R}\mathbf{w}(n) - 2\mathbf{p}^T\mathbf{w}(n). \end{aligned} \quad (6.8)$$

where the vector of cross-correlation between input and desired output \mathbf{p} is defined as

$$\mathbf{p} = E[d(n)\mathbf{x}(n)] \quad (6.9)$$

and the input autocorrelation matrix \mathbf{R} is defined as

$$\mathbf{R} = E[\mathbf{x}(n)\mathbf{x}^T(n)]. \quad (6.10)$$

The performance function, or surface, $\xi(n)$ is a quadratic function of $\mathbf{w}(n)$ and as such is referred to in the following equations as $\xi(\mathbf{w})$. Since it is a quadratic function of $\mathbf{w}(n)$, it has one global minimum corresponding to $\mathbf{w}(n) = \mathbf{w}_{\text{opt}}$. The optimum value of the weights, \mathbf{w}_{opt} , may be found by equating the gradient of the performance surface to zero, that is,

$$\begin{aligned} \frac{\partial \xi(\mathbf{w})}{\partial \mathbf{w}} &= \frac{\partial}{\partial \mathbf{w}} [E[d^2(n)] + \mathbf{w}^T \mathbf{R} \mathbf{w} - 2\mathbf{p}^T \mathbf{w}] \\ &= 2\mathbf{R}\mathbf{w} - 2\mathbf{p} \end{aligned} \quad (6.11)$$

and hence, in terms of the statistical quantities just described,

$$2\mathbf{R}\mathbf{w}_{\text{opt}} - 2\mathbf{p} = 0, \quad (6.12)$$

$$\mathbf{w}_{\text{opt}} = \mathbf{R}^{-1}\mathbf{p}. \quad (6.13)$$

In a practical, real-time application, solving Equation 6.13 may not be possible either because signal statistics are unavailable or simply because of the computational effort involved in inverting the input autocorrelation matrix \mathbf{R} .

6.3.1 Visualizing the Performance Function

If there is just one weight in the adaptive filter, then the performance function will be a parabolic curve, as shown in Figure 6.9. If there are two weights, the performance function will be a three-dimensional surface, a paraboloid; and if there are more than two weights, then the performance function will be a hypersurface that is difficult to visualize. In each case, the role of the adaptation mechanism is to adjust the filter coefficients to those values that correspond to a minimum in the performance function.

6.4 SEARCHING FOR THE MINIMUM

An alternative to solving Equation 6.13 by matrix inversion is to search the performance function for \mathbf{w}_{opt} , starting with an arbitrary set of weight values and adjusting these at each sampling instant.

One way of doing this is to use the steepest descent algorithm. At each iteration (of the algorithm) in a discrete-time implementation, the coefficients are adjusted in

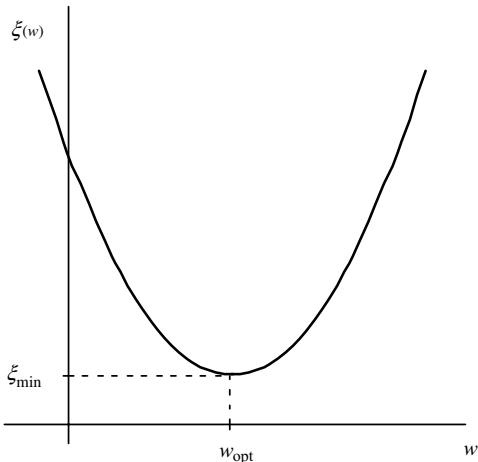


Figure 6.9 Performance function for single weight case.

the direction of the negative gradient of the performance function and by an amount proportional to the magnitude of the gradient, that is,

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \beta \frac{\partial \xi(\mathbf{w})}{\partial \mathbf{w}} \Big|_{\mathbf{w}=\mathbf{w}(n)}, \quad (6.14)$$

where $\mathbf{w}(n)$ represents the value of the weights at the n th iteration and β is an arbitrary positive constant that determines the rate of adaptation. If β is too large, then instability may ensue. If the statistics of the signals $x(n)$ and $d(n)$ are known, then it is possible to set a quantitative upper limit on the value of β , but in practice, it is usual to set β equal to a very low value.

The steepest descent algorithm described by Equation 6.14 is illustrated for the case of a single weight in Figure 6.10.

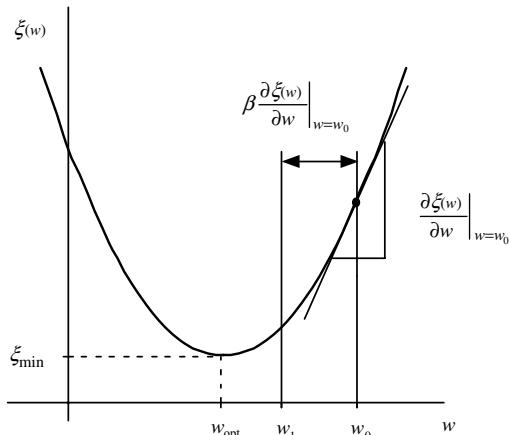


Figure 6.10 Steepest descent algorithm for single weight case.

6.5 LEAST MEAN SQUARES ALGORITHM

The steepest descent algorithm requires an estimate of the gradient $\nabla(n)$ of the performance surface $\xi(\mathbf{w})$ at each step. But since this depends on the statistics of the signals involved, it may be computationally expensive to obtain. The least mean squares algorithm uses instantaneous squared error $e^2(n)$ as an estimate of mean squared error $E[e^2(n)]$ and yields an estimated gradient:

$$\hat{\nabla}(n) = \begin{bmatrix} \frac{\partial \hat{\xi}(n)}{\partial w_0(n)} \\ \frac{\partial \hat{\xi}(n)}{\partial w_1(n)} \\ \vdots \\ \frac{\partial \hat{\xi}(n)}{\partial w_{N-1}(n)} \end{bmatrix} = \begin{bmatrix} \frac{\partial e^2(n)}{\partial w_0(n)} \\ \frac{\partial e^2(n)}{\partial w_1(n)} \\ \vdots \\ \frac{\partial e^2(n)}{\partial w_{N-1}(n)} \end{bmatrix} \quad (6.15)$$

or

$$\hat{\nabla}(n) = \frac{\partial \hat{\xi}(n)}{\partial \mathbf{w}(n)} = \frac{\partial e^2(n)}{\partial \mathbf{w}(n)}. \quad (6.16)$$

Equation 6.6 gave an expression for instantaneous squared error:

$$e^2(n) = d^2(n) - 2d(n)\mathbf{x}^T(n)\mathbf{w}(n) + \mathbf{w}^T(n)\mathbf{x}(n)\mathbf{x}^T(n)\mathbf{w}(n).$$

Differentiating this with respect to $\mathbf{w}(n)$,

$$\begin{aligned} \frac{\partial e^2(n)}{\partial \mathbf{w}(n)} &= 2\mathbf{x}(n)\mathbf{x}^T(n)\mathbf{w}(n) - 2d(n)\mathbf{x}(n) \\ &= 2\mathbf{x}(n)(\mathbf{x}^T(n)\mathbf{w}(n) - d(n)) \\ &= -2e(n)\mathbf{x}(n). \end{aligned} \quad (6.17)$$

Hence, the steepest descent algorithm, using this gradient estimate, is

$$\begin{aligned} \mathbf{w}(n+1) &= \mathbf{w}(n) - \beta \hat{\nabla}(n) \\ &= \mathbf{w}(n) + 2\beta e(n)\mathbf{x}(n). \end{aligned} \quad (6.18)$$

This is the LMS algorithm. Gradient estimate $\hat{\nabla}(n)$ is imperfect and therefore the LMS adaptive process may be noisy. This is a further motivation for choosing a conservatively low value for β .

The LMS algorithm is well established, computationally inexpensive, and therefore widely used. Other methods of adaptation include recursive least squares, which is more computationally expensive but converges faster, and normalized LMS, which takes explicit account of signal power.

Given that in practice the choice of value for β is somewhat arbitrary, a number of simpler fixed step size variations are practicable, although somewhat counterintuitively these variants may be computationally more expensive to implement using a digital signal processor with single-cycle multiplication capability than the straightforward LMS algorithm.

6.5.1 LMS Variants

For the *sign-error* LMS algorithm, Equation 6.18 becomes

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \beta \operatorname{sgn}(e(n))\mathbf{x}(n), \quad (6.19)$$

where $\operatorname{sgn}()$ is the signum function:

$$\operatorname{sgn}(u) = \begin{cases} 1, & \text{if } u \geq 0, \\ -1, & \text{if } u < 0. \end{cases} \quad (6.20)$$

For the *sign-data* LMS algorithm, Equation 6.18 becomes

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \beta \operatorname{sgn}(\mathbf{x}(n))e(n). \quad (6.21)$$

For the *sign-sign* LMS algorithm, Equation 6.18 becomes

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \beta \operatorname{sgn}(\mathbf{x}(n)) \operatorname{sgn}(e(n)), \quad (6.22)$$

which reduces to

$$\mathbf{w}(n+1) = \begin{cases} \mathbf{w}(n) + \beta, & \text{if } \operatorname{sgn}(e(n)) = \operatorname{sgn}(\mathbf{x}(n)), \\ \mathbf{w}(n) - \beta, & \text{otherwise,} \end{cases} \quad (6.23)$$

and which involves no multiplications.

6.6 PROGRAMMING EXAMPLES

The following program examples illustrate adaptive filtering using the LMS algorithm applied to an FIR filter.

EXAMPLE 6.1: Adaptive Filter Using C Code ([L138_adaptc](#))

This example implements the LMS algorithm as a C program. It illustrates the following steps for the adaptation process using the adaptive structure shown in Figure 6.2:

- (1) Obtain new samples of the desired signal d and the reference input to the adaptive filter x .

- (2) Calculate the adaptive FIR filter's output y , applying Equation 6.4.
- (3) Calculate the error signal e by applying Equation 6.5.
- (4) Update each filter coefficient (weight) w by applying the LMS algorithm (6.18).
- (5) Shift the contents of the adaptive filter delay line, containing previous input samples, by one.
- (6) Repeat the adaptive process for the next sampling instant.

Figure 6.11 shows a listing of the program L138_adaptc.c, which implements the LMS algorithm for the adaptive filter structure of Figure 6.2. The desired signal is chosen to be $2 \cos(2\pi f/F_s)$ and the input to the adaptive filter is chosen to be $\sin(2\pi f/F_s)$, where signal frequency f is equal to 1 kHz and sampling frequency F_s is equal to 8 kHz. The adaptation rate β , filter order N , and number of samples processed in the program are equal to 0.01, 21, and 60, respectively.

```
// L138_adaptc.c - non real-time adaptation demonstration

#include <stdio.h>
#include <math.h>
#define beta 0.01           //convergence rate
#define N 21                //order of filter
#define NS 60               //number of samples
#define Fs 8000              //sampling frequency
#define pi 3.1415926
#define DESIRED 2*cos(2*pi*t*1000/Fs) //desired signal
#define NOISE sin(2*pi*t*1000/Fs)    //noise signal

float desired[NS], y_out[NS], error[NS];

void main()
{
    long i, t;
    float d, y, e;
    float w[N+1] = {0.0};
    float x[N+1] = {0.0};

    for (t = 0; t < NS; t++)          //start adaptive algorithm
    {
        x[0] = NOISE;                //new noise sample
        d = DESIRED;                 //desired signal
        y = 0;                       //zero filter output
        for (i = 0; i <= N; i++)
            y += (w[i] * x[i]);      //calculate filter output
        e = d - y;                  //calculate error signal
        for (i = N; i >= 0; i--)
        {
            w[i] = w[i] + (beta*e*x[i]); //update filter coeffs
            if (i != 0) x[i] = x[i-1];   //update delay line
        }
        desired[t] = d;
        y_out[t] = y;
        error[t] = e;
    }
    printf("done!\n");
}
```

Figure 6.11 Adaptive filter program L138_adaptc.c.

The overall output is the adaptive filter's output y , which converges to the desired cosine signal d .

The project is supplied in folder `c:\eXperimenter\L138_chapter6\L138_adaptc`. Because the program does not use any real-time input or output, it is not necessary for the project to use files `L138_aic3106_init.c` or `vectors_intr.asm`.

Figure 6.12 shows plots of the desired output `desired`, adaptive filter output `y_out`, and error `error`, plotted using *Tools > Graph > Single Time* in the Code Composer Studio IDE. Within 60 sampling instants, the filter output converges to the desired cosine signal. Change the adaptation or convergence rate `beta` to 0.02 and verify a faster rate of adaptation and convergence.

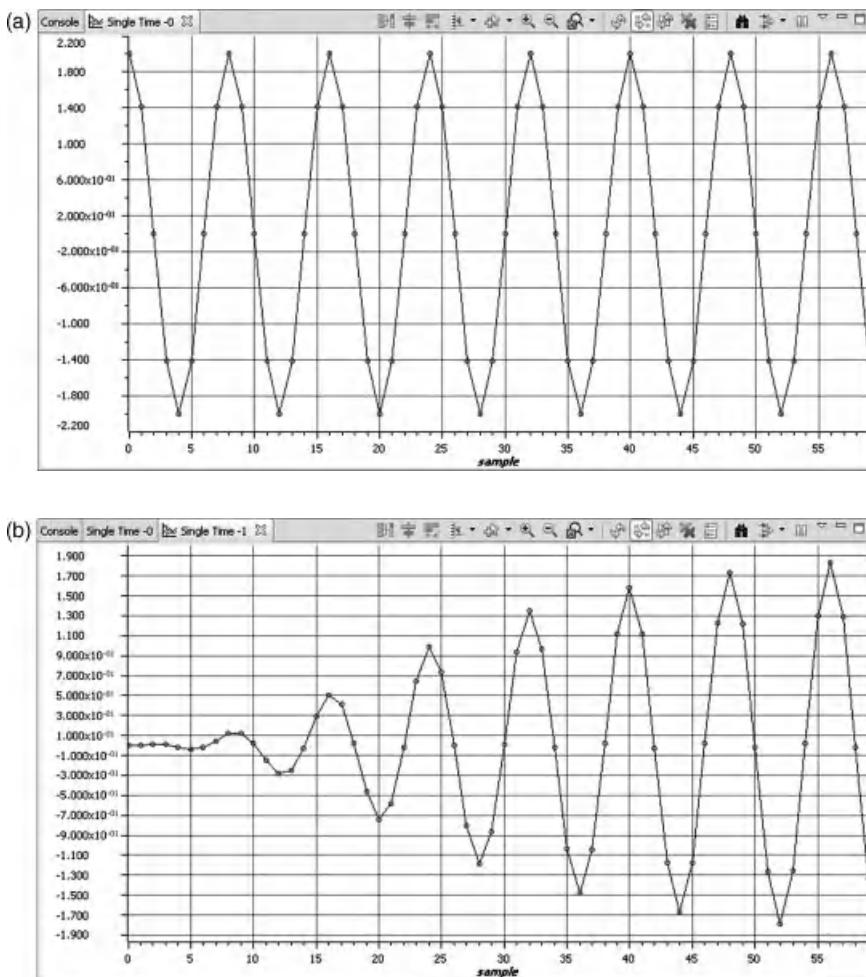


Figure 6.12 Plots of (a) desired output, (b) adaptive filter output, and (c) error using program `L138_adaptc.c`.

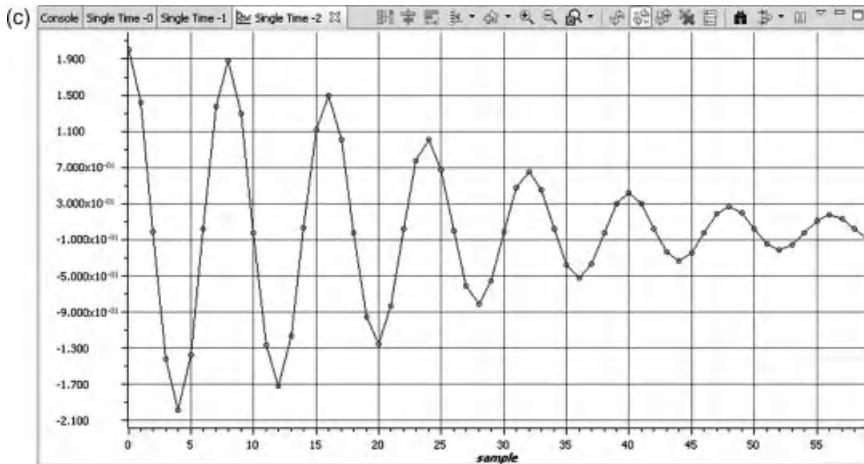


Figure 6.12 (Continued)

EXAMPLE 6.2: Adaptive Filter for Sinusoidal Noise Cancellation (L138_adaptnoise_intr)

This example illustrates the use of the LMS algorithm to cancel an undesired sinusoidal noise signal. Figure 6.13 shows a listing of program L138_adaptnoise_intr.c, which implements an adaptive FIR filter using the structure shown in Figure 6.5.

A desired sine wave `signal` of frequency `SIGNAL_FREQ` (2500 Hz), with an added (undesired) sine wave `signoise`, of frequency `NOISE_FREQ` (1200 Hz) forms one of two inputs to the noise cancellation structure and represents the signal plus noise from the primary sensor in Figure 6.5. A cosine wave `refnoise`, with a frequency of `NOISE_FREQ` (1200 Hz), represents the reference noise signal in Figure 6.5 and is the input to an N-coefficient adaptive FIR filter. The signal `refnoise` is strongly correlated with the signal `signoise`, but not with the desired signal.

At each sampling instant, the output of the adaptive FIR filter is calculated, its N weights are updated, and the contents of the delay line `x` are shifted. The `error` signal is the overall desired output of the adaptive structure. It comprises the desired signal and additive noise from the primary sensor (`signal + signoise`) from which the adaptive filter output `yn` has been subtracted.

The input signals used in this example are generated within the program and both the input signal `signal + signoise` and the output signal `error` are output via the AIC3106 codec on right and left channels, respectively (Figure 6.14).

Build and run the program and verify the following output result. The undesired 1200 Hz sinusoidal component of the output signal `error` is gradually reduced (canceled), while the desired 2500 Hz signal remains. A faster rate of adaptation can be observed by using a larger value of `beta`. However, if `beta` is too large, the adaptation process may become unstable.

Figure 6.15 shows the output from the program displayed using *Goldwave*. The waterfall plot on the left-hand side shows the magnitude of the 1200 Hz noise component in the error signal decreasing as adaptation takes place.

```

// L138_adaptnoise_intr.c
//

#include "L138_aic3106_init.h"
#include "math.h"

#define SAMPLING_FREQ 8000
#define PI 3.14159265358979
#define NOISE_FREQ 1200.0
#define SIGNAL_FREQ 2500.0
#define NOISE_AMPLITUDE 8000.0
#define SIGNAL_AMPLITUDE 8000.0
#define beta 1E-13           // adaptive learning rate
#define N 30                // number of weights

float w[N];                  // adaptive filter weights
float x[N];                  // adaptive filter delay line
short buffercount=0;

float theta_increment_noise;
float theta_noise = 0.0;

float theta_increment_signal;
float theta_signal = 0.0;

AIC31_data_type codec_data;

interrupt void interrupt4(void) // interrupt service routine
{
    short i;
    float yn, error, signal, signoise, refnoise;

    theta_increment_noise = 2*PI*NOISE_FREQ/SAMPLING_FREQ;
    theta_noise += theta_increment_noise;
    if (theta_noise > 2*PI) theta_noise -= 2*PI;

    theta_increment_signal = 2*PI*SIGNAL_FREQ/SAMPLING_FREQ;
    theta_signal += theta_increment_signal;
    if (theta_signal > 2*PI) theta_signal -= 2*PI;

    refnoise = (NOISE_AMPLITUDE*cos(theta_noise));
    signoise = (NOISE_AMPLITUDE*sin(theta_noise));
    signal = (SIGNAL_AMPLITUDE*sin(theta_signal));

    x[0] = refnoise;          // reference input to adaptive filter
    yn = 0;                   // compute adaptive filter output
    for (i = 0; i < N; i++)
        yn += (w[i] * x[i]);

    error = signal + signoise - yn; // compute error

    for (i = N-1; i >= 0; i--) // update weights and delay line
    {
        w[i] = w[i] + beta*error*x[i];
        x[i] = x[i-1];
    }

    codec_data.channel[LEFT]= ((uint16_t)(error));
    codec_data.channel[RIGHT]= ((uint16_t)(signal + signoise));
    output_sample(codec_data.uint);
    return;
}

```

Figure 6.13 Adaptive FIR filter program for sinusoidal noise cancellation L138_adaptnoise_intr.c.

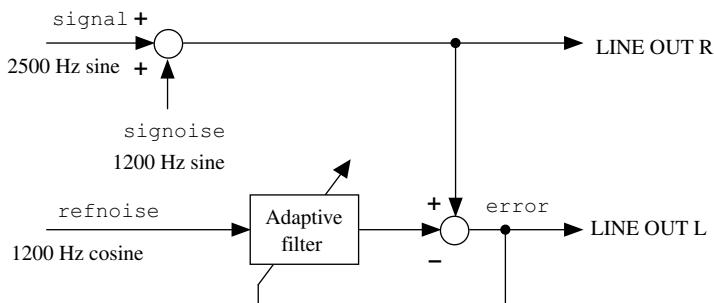
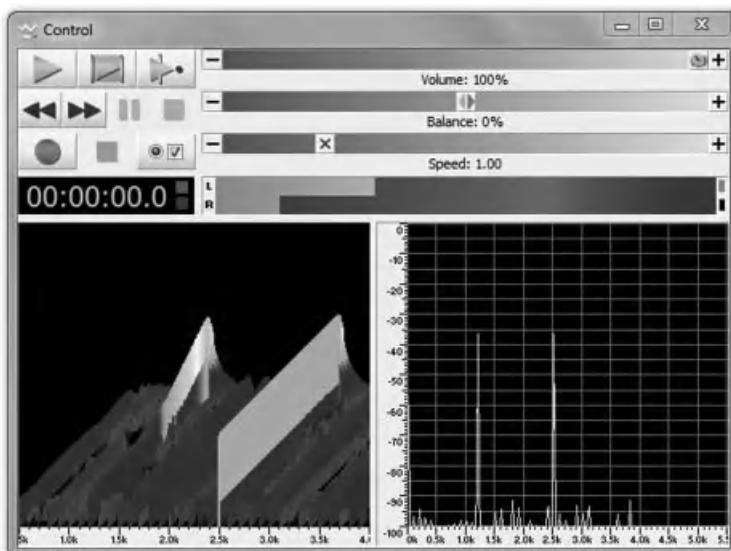
```

}

int main()
{
    int16_t i;

    for (i=0 ; i<N ; i++) // initialise weights and delay line
    {
        w[i] = 0.0;
        x[i] = 0.0;
    }
    L138_initialise_intr(FS_8000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
    while(1);
}

```

Figure 6.13 (Continued)**Figure 6.14** Block diagram representation of program L138_adaptnoise_intr.c.**Figure 6.15** Output from program L138_adaptnoise_intr.c. The left channel shows the output signal `error` as adaptation takes place, and the right channel shows the noisy input signal `signal + signoise`.

**EXAMPLE 6.3: Adaptive FIR Filter for Noise Cancellation Using External Inputs
(L138_adaptnoise_2IN_iir_intr)**

This example extends the previous one to cancel an undesired noise signal using external inputs. Figure 6.16 shows the source program L138_adaptnoise_2IN_iir_intr.c

```
// L138_adaptnoise_2IN_iir_intr.c
//  
  
#include "L138_aic3106_init.h"  
#include "bilinear.cof"  
  
#define beta 1E-11           // learning rate  
#define N 128                // number of adaptive filter weights  
  
AIC31_data_type codec_data;  
  
float weights[N];           // adaptive filter weights  
float x[N];                // adaptive filter delay line  
  
float w[NUM_SECTIONS][2];  
  
interrupt void interrupt4(void) // interrupt service routine  
{  
    short i;  
    float input, refnoise, signal, signoise, wn, yn, error;  
    int section;  
  
    codec_data.uint = input_sample();  
    refnoise =(codec_data.channel[LEFT]); // reference sensor  
    signal = (codec_data.channel[RIGHT]); // primary sensor  
  
    input = refnoise;  
    for (section=0 ; section<NUM_SECTIONS ; section++)  
    {  
        wn = input - a[section][1]*w[section][0]  
            - a[section][2]*w[section][1];  
        yn = b[section][0]*wn + b[section][1]*w[section][0]  
            + b[section][2]*w[section][1];  
        w[section][1] = w[section][0];  
        w[section][0] = wn;  
        input = yn;  
    }  
    signoise = yn + signal;  
  
    yn=0.0;  
    x[0] = refnoise;  
    for (i = 0; i < N; i++) // compute adaptive filter output  
        yn += (weights[i] * x[i]);  
  
    error = (signoise) - yn; // compute error  
    for (i = N-1; i >= 0; i--) // update weights and delay line  
    {  
        weights[i] = weights[i] + beta*error*x[i];  
        x[i] = x[i-1];  
    }  
    codec_data.channel[LEFT]= ((uint16_t)(error));  
    codec_data.channel[RIGHT]= ((uint16_t)(error));  
    output_sample(codec_data.uint);  
  
    return;  
}  
  
int main()  
{  
    int i;
```

Figure 6.16 Program L138_adaptnoise2IN_iir_intr.c.

```

for (i= 0; i < N; i++) // initialise weights and delay line
{
    weights[i] = 0;
    x[i] = 0;
}
for (i= 0; i < NUM_SECTIONS; i++)
{
    w[i][0] = 0.0;
    w[i][1] = 0.0;
}
L138_initialise_intr(FS_8000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
while(1);
}

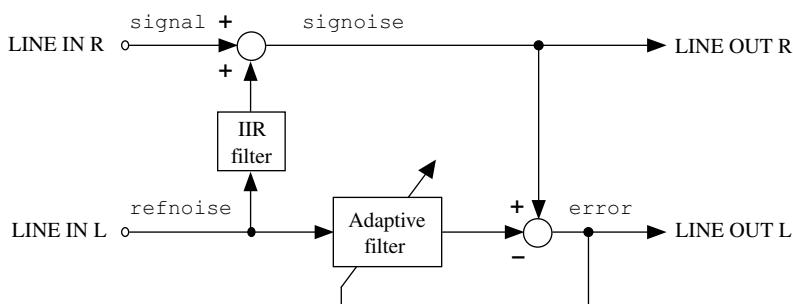
```

Figure 6.16 (Continued)

that requires two external inputs, a desired signal and a reference noise signal to be input to left and right channels, respectively. A stereo 3.5 mm jack plug to dual RCA jack plug cable is useful for implementing this example using two different signal sources. Alternatively, a test input signal is provided to accompany this book in the form of file `speechnoise.wav`. This may be played through a sound card and input to the eXperimenter via a stereo 3.5 mm jack plug to 3.5 mm jack plug cable. `speechnoise.wav` comprises pseudorandom noise on the left channel and speech on the right channel.

Figure 6.17 shows the program in block diagram form. Within the program, a primary noise signal correlated with the reference noise signal input on the left channel is formed by passing the reference noise through an IIR filter. The primary noise signal is added to the desired signal (speech) input on the right channel.

Build and run the program and test it using file `speechnoise.wav`. As adaptation takes place, the output on the left channel of LINE OUT should gradually change from speech plus noise to speech only. You may need to adjust the volume at which you play the file `speechnoise.wav`. If the input signals are too quiet, then the adaptation may be very slow.

**Figure 6.17** Block diagram representation of program `L138_adaptnoise2IN_iir_intr.c`.

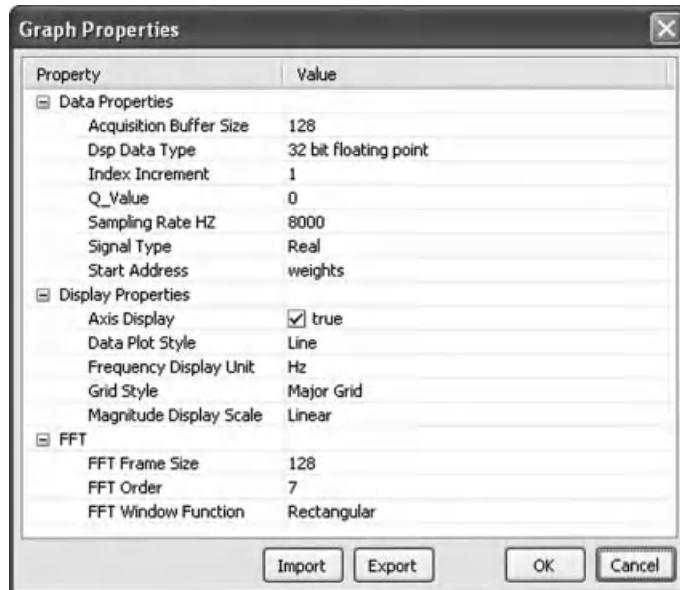


Figure 6.18 Graph Properties used to view IIR filter magnitude frequency response identified using program L138_adaptnoise2IN_iir_intr.c.

Using *View > Graph > Single Time* and the *Graph Properties* shown in Figure 6.18, you should be able to see the magnitude frequency response of the IIR filter, as identified by the adaptive filter.

EXAMPLE 6.4: Adaptive FIR Filter for System Identification of a Fixed FIR Filter as an Unknown System (L138_adaptIDFIR_intr)

Figure 6.19 shows a listing of the program L138_adaptIDFIR_intr, which uses an adaptive FIR filter to identify an unknown system. A block diagram of the system implemented in this example is shown in Figure 6.20.

The unknown system to be identified is a 55-coefficient FIR band-pass filter centered at 2000 Hz. The coefficients of this fixed FIR filter are read from the file bp55.coef, previously used in Example 3.7. A 60-coefficient adaptive FIR filter is used to identify the fixed (unknown) FIR band-pass filter.

A pseudorandom binary noise sequence, generated within the program, is input to both the fixed (unknown) and the adaptive FIR filters and an error signal formed from their outputs. The adaptation process seeks to minimize the variance of that error signal. It is important to use wideband noise as an input signal in order to identify the characteristics of the unknown system over the entire frequency range from zero to half the sampling frequency.

```

// L138_adaptIDFIR_intr.c
//

#include "L138_aic3106_init.h"
#include "bp55.cof"
#include "evmomapl138_dip.h"

#define ADAPT_OUT 0
#define FIXED_OUT 1
float beta = 5E-13;
#define WLENGTH 60

float w[WLENGTH+1] ;
int dly_adapt[WLENGTH+1];      // adaptive filter delay line
int dly_fix[N+1];            // fixed filter delay line
volatile short out_type = ADAPT_OUT;
AIC31_data_type codec_data;

interrupt void interrupt4(void) // interrupt service routine
{
    int i;
    int fir_out = 0;           // fixed filter output
    int adaptfir_out = 0;      // adaptive filter output
    float E;                  // error

    dly_fix[0] = prbs();       // input noise to fixed filter
    dly_adapt[0]=dly_fix[0];   // and to adaptive filter
    for (i = N-1; i >= 0; i--) // compute fixed filter output
    {
        fir_out +=(h[i]*dly_fix[i]);
        dly_fix[i+1] = dly_fix[i];
    }
    for (i = 0; i < WLENGTH; i++) // compute adaptive filter
        adaptfir_out +=(w[i]*dly_adapt[i]);

    E = fir_out - adaptfir_out; // compute error
    for (i = WLENGTH-1; i >= 0; i--) // update filter weights
    {
        w[i] = w[i]+(beta*E*dly_adapt[i]);
        dly_adapt[i+1] = dly_adapt[i];
    }
    if(out_type == ADAPT_OUT)
        codec_data.channel[RIGHT]=((uint16_t)(adaptfir_out));
    else
        codec_data.channel[RIGHT]=((uint16_t)(fir_out));
    codec_data.channel[LEFT]= ((uint16_t)(E));
    output_sample(codec_data.uint);
    return;
}

int main()
{
    int i;
    uint32_t rtn;
    uint8_t DIP_value;

    for (i=0; i<WLENGTH; i++) // initialise adaptive filter
    {
        w[i] = 0.0;
        dly_adapt[i] = 0.0;
    }
}

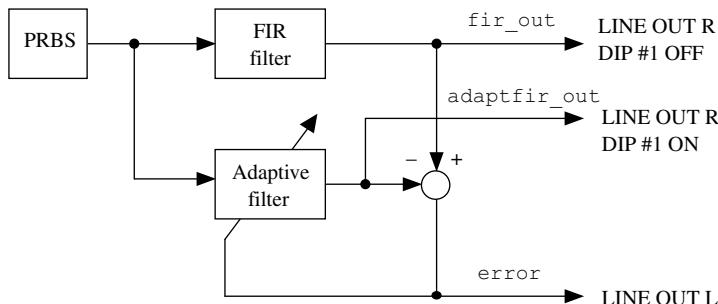
```

Figure 6.19 Program L138_adaptIDFIR_intr.c.

```

for (i = 0; i < N; i++) // initialise fixed filter
    dly_fix[i] = 0.0;
L138_initialise_intr(FS_8000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
rtn = DIP_init();
if (rtn != ERR_NO_ERROR)
{
    printf("error initializing dip switches!\r\n");
    return (rtn);
}
while(1)
{
    rtn = DIP_get(0, &DIP_value);
    if(DIP_value == 0) out_type = ADAPT_OUT;
    else out_type = FIXED_OUT;
}
}
}

```

Figure 6.19 (Continued)**Figure 6.20** Block diagram representation of the system identification scheme implemented by program L138_adaptIDFIR_intr.c.

An extra memory location is used in each of the two input sample delay lines (`dly_fix` and `dly_adapt`). These extra memory locations are used when shifting the samples in the delay lines.

Build, load, and run the program. DIP switch #1 may be used to select either `fir_out` (the output from the fixed (unknown) FIR filter) or `adaptfir_out` (the output from the adaptive FIR filter) as the signal written to the right channel of LINE OUT on the eXperimenter. `E` (the error signal) is always written to the left channel of LINE OUT. Verify that the output of the adaptive FIR filter (`adaptfir_out`) converges to bandlimited noise similar in frequency content to the output of the fixed FIR filter (`fir_out`) and that the variance of the error signal (`error`) gradually diminishes as adaptation takes place.

Edit the program to include the coefficient file `bs55.cof` (in place of `bp55.cof`) which implements a 55-coefficient FIR band-stop filter centered at 2 kHz.

Rebuild and run the program and verify that the output of the adaptive FIR filter (with DIP switch #1 ON) is almost identical to that of the FIR band-stop filter (with DIP switch #1 OFF).

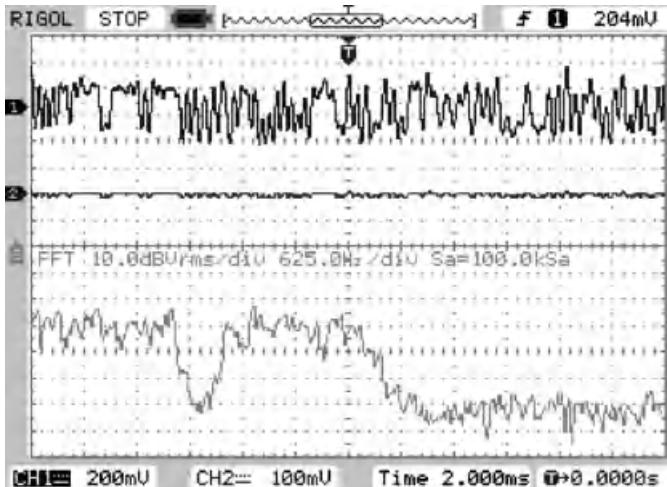


Figure 6.21 Output from program L138_adaptIDFIR_intr.c using coefficient file bs55.coef, viewed using Rigol DS1052E oscilloscope.

Figure 6.21 shows the output of the program while adaptation takes place. The upper time domain trace shows the output of the adaptive FIR filter, the lower trace shows the error signal, and the magnitude of the FFT of the output of the adaptive FIR filter is shown below these. Increase (decrease) the value of beta by a factor of 10 to observe a faster (slower) rate of convergence. Change the number of weights (coefficients) from 60 to 40 and verify a slight degradation in the identification process.

EXAMPLE 6.5: Adaptive FIR for System ID of a Fixed FIR as an Unknown System with Weights of an Adaptive Filter Initialized as a FIR Band-Pass (L138_adaptIDFIR_init_intr)

In this example, program L138_adaptIDFIR_intr.c has been modified slightly to create program L138_adaptIDFIR_init_intr.c. This program initializes the weights w of the adaptive FIR filter using the coefficients of an FIR band-pass filter centered at 3 kHz rather than initializing the weights to zero. Both sets of filter coefficients (adaptive and fixed) are read from file L138_adaptIDFIR_init_coeffs.h.

Build, load, and run the program. Initially, the frequency content of the output of the adaptive FIR filter is centered at 3 kHz. Then, gradually, as the adaptive filter identifies the fixed (unknown) FIR band-pass filter, its output changes to bandlimited noise centered at frequency 2 kHz.

The adaptation process is illustrated in Figure 6.22 which shows the frequency content of the output of the adaptive filter at different stages in the adaptation process.

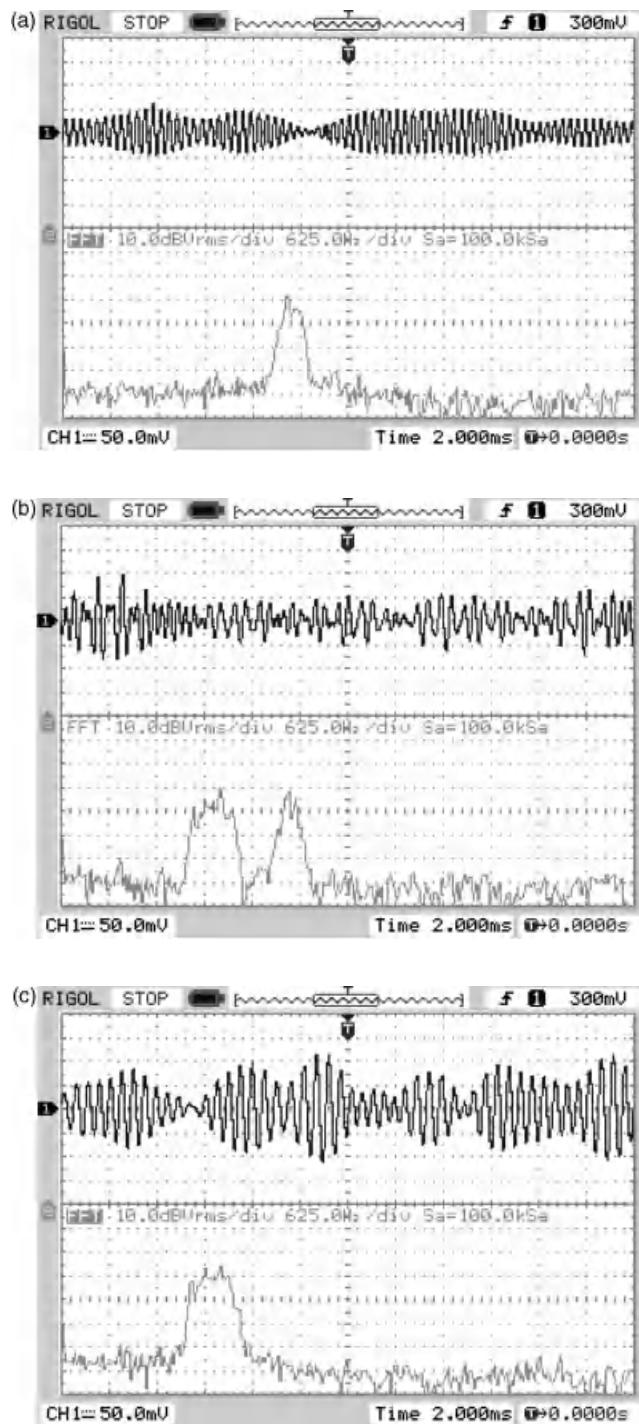


Figure 6.22 Output from adaptive filter in program L138_adaptIDFIR_init_intr.c.

**EXAMPLE 6.6: Adaptive FIR for System ID of Fixed IIR as an Unknown System
(L138_iirssosadapt_intr.c)**

An adaptive FIR filter can be used to identify the characteristics not only of other FIR filters but also of IIR filters (provided that the substantial part of the IIR filter impulse response is shorter than that possible using the adaptive FIR filter). Program L138_iirssosadapt_intr.c combines parts of programs L138_iirssos_intr.c (Example 4.1) and L138_adaptIDFIR_intr.c in order to illustrate this.

The IIR filter coefficients used are those of a fourth-order low-pass elliptic filter (see Example 4.4) and are read from file `elliptic.cof`.

Build and run the program and verify that the adaptive filter converges to a state in which the frequency content of its output matches that of the (unknown) IIR filter. Listening to the decaying error signal output on the left channel of LINE OUT gives an indication of the progress of the adaptation (Figure 6.23). Figures 6.24 and 6.25 show the output of the adaptive filter (displayed using the FFT function of a *Rigol DS1052E* oscilloscope) and the magnitude FFT of the coefficients of the adaptive FIR filter (weights) saved to a file and displayed using MATLAB function `logfft()`. The result of the adaptive system identification procedure is similar in form to that obtained by recording the impulse response of an elliptic low-pass filter in Example 4.5.

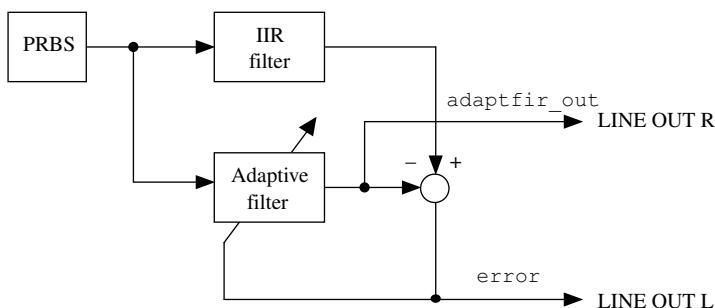


Figure 6.23 Block diagram representation of program L138_iirssosadapt.c.



Figure 6.24 Output from adaptive filter implemented by program L138_iirssosadapt.c, viewed using a *Rigol DS1052E* oscilloscope.

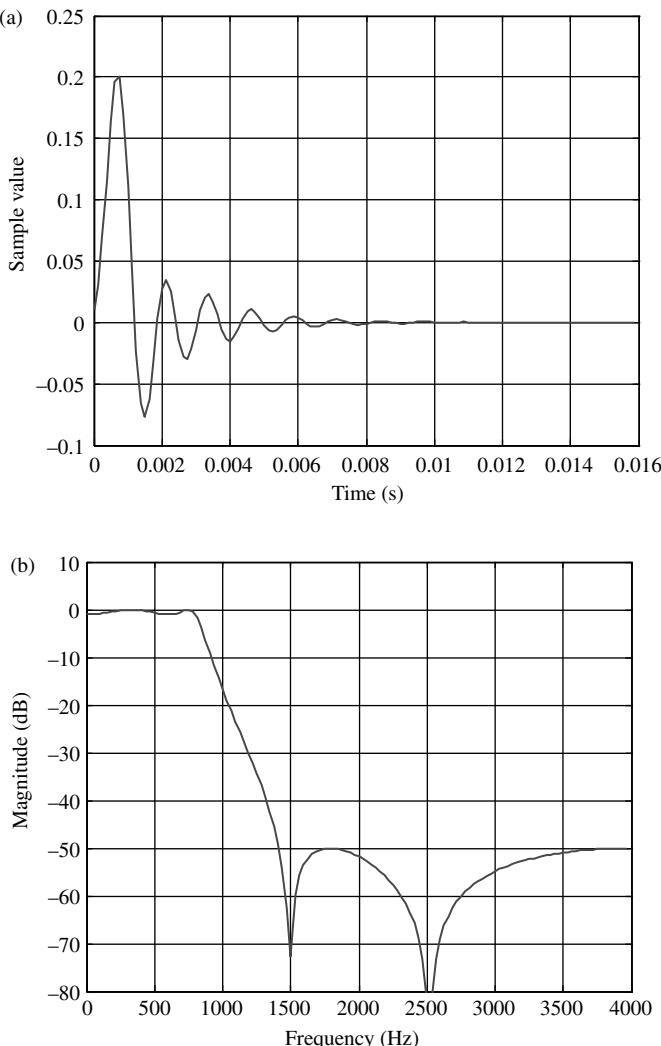


Figure 6.25 Adaptive filter coefficients from program L138_iirssosadapt.c plotted using MATLAB function L138_logfft().

EXAMPLE 6.7: Adaptive FIR Filter for System Identification of System External to the eXperimenter (L138_sysid_intr)

Program L138_sysid_intr.c, introduced in Chapter 2 (Figure 6.26), extends the previous examples to allow the identification of a system external to the eXperimenter board, connected between the LINE OUT and LINE IN sockets. In Example 3.3, program L138_sysid_intr.c was used to identify the characteristics of a moving average filter

implemented using a second eXperimenter. Other filter programs, for example, L138_fir_-intr.c or L138_iirsos_intr.c could just as well be run in place of average.c on the second eXperimenter.

Alternatively, a purely analog system or a filter implemented using different DSP hardware could be connected between LINE OUT and LINE IN and its characteristics identified.

```
// L138_sysid_intr.c
//

#include "L138_aic3106_init.h"

#define beta 10E-12           // learning rate
#define WLENGTH 256

float w[WLENGTH+1];          // adaptive filter weights
float dly_adapt[WLENGTH+1];   // adaptive filter delay line

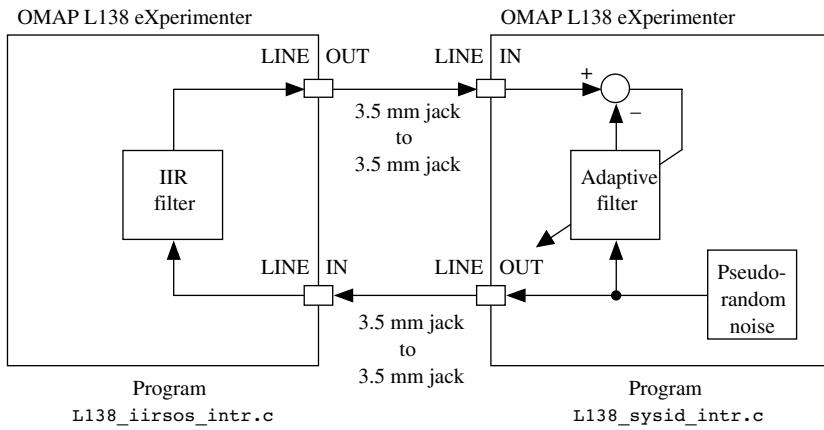
interrupt void interrupt4(void) // interrupt service routine
{
    int16_t left_sample;
    int i;
    float fir_out, E, adaptfir_out;

    left_sample = input_left_sample(); // read from L ADC
    fir_out = (float)(left_sample);
    dly_adapt[0] = prbs();           // apply noise to input
    left_sample = (int16_t)(dly_adapt[0]); // of adaptive filter
    output_left_sample(left_sample); // and to L DAC
    adaptfir_out = 0.0;
    for (i = 0; i < WLENGTH; i++)
        adaptfir_out +=(w[i]*dly_adapt[i]); // compute filter output
    E = fir_out - adaptfir_out;       // and error signal
    for (i = WLENGTH-1; i >= 0; i--)
    {
        w[i] = w[i]+(beta*E*dly_adapt[i]); // update weights
        dly_adapt[i+1] = dly_adapt[i];      // update delay line
    }
    return;
}

int main(void)
{
    int i;

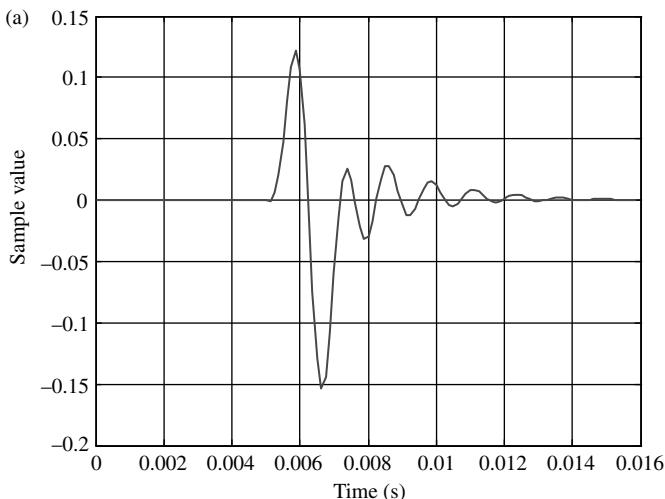
    for (i = 0; i <= WLENGTH; i++)
    {
        w[i] = 0.0;           // initialise adaptive filter weights
        dly_adapt[i] = 0.0;   // and delay line
    }
    L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
    while(1);
}
```

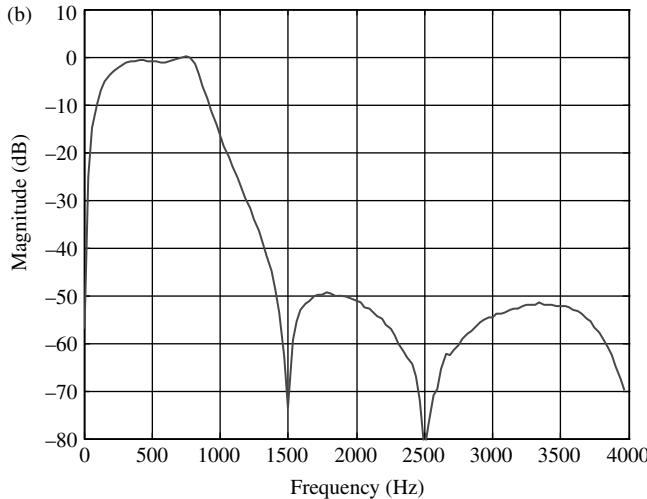
Figure 6.26 Listing of program L138_sysid_intr.c.

**Figure 6.27** Connection diagram for Example 6.7.

Connect two eXperimenters as shown in Figure 6.27. Load and run program L138_iirssos_intr.c, including coefficient file elliptic.cof on the first. Run program L138_sysid_intr.c on the second. Halt the program after a few seconds and select first *Tools > Graph > Single Time* and then *Tools > Graph > FFT Magnitude* in order to examine the coefficients of the adaptive filter. Figure 6.28 shows typical results.

A number of features of the plots shown in Figure 6.28 are worthy of note. Compare the magnitude frequency response in Figure 6.28b with that in Figure 6.25b. The characteristics of

**Figure 6.28** Adaptive filter coefficients from program L138_sysid_intr.c in Example 6.7, plotted using MATLAB function L138_logfft().

**Figure 6.28 (Continued)**

the codec reconstruction and antialiasing filters and of the AC-coupling between codecs and jack sockets on both boards are included in the signal path identified by program L138_sysid_intr.c in this example and are apparent in the roll-off of the magnitude frequency response at frequencies above 3800 Hz and below 100 Hz. There is no roll-off in Figure 6.25b. Compare the impulse response in Figure 6.28b with that in Figure 6.25b. Apart from its slightly different form, corresponding to the roll-off in its magnitude frequency response, there is a delay of approximately 5 ms.

EXAMPLE 6.8: Adaptive FIR Filter for System Identification of System External to the eXperimenter Using DSPLIB Function DSPF_sp_fir_gen() (L138_sysid_DSPLIB_edma)

Functionally, program L138_sysid_DSPLIB_edma.c is quite similar to program L138_sysid_intr.c. Both programs use an adaptive FIR filter to identify the impulse response of a system connected between LINE OUT and LINE IN connections to the eXperimenter board. Program L138_sysid_DSPLIB_edma.c makes use of an adaptive filtering routine supplied as part of the c674x DSPLIB library. Function DSPF_sp_fir_gen() is more computationally efficient than the C-coded adaptive filter used in program L138_sysid_intr.c and hence program L138_sysid_DSPLIB_edma.c can run at higher sampling rates. Because function DSPF_sp_fir_gen() processes blocks of input data rather than just one sample at a time, it is appropriate to use DMA-based rather than interrupt-based I/O in this example. However, DMA-based I/O introduces an extra delay into the signal

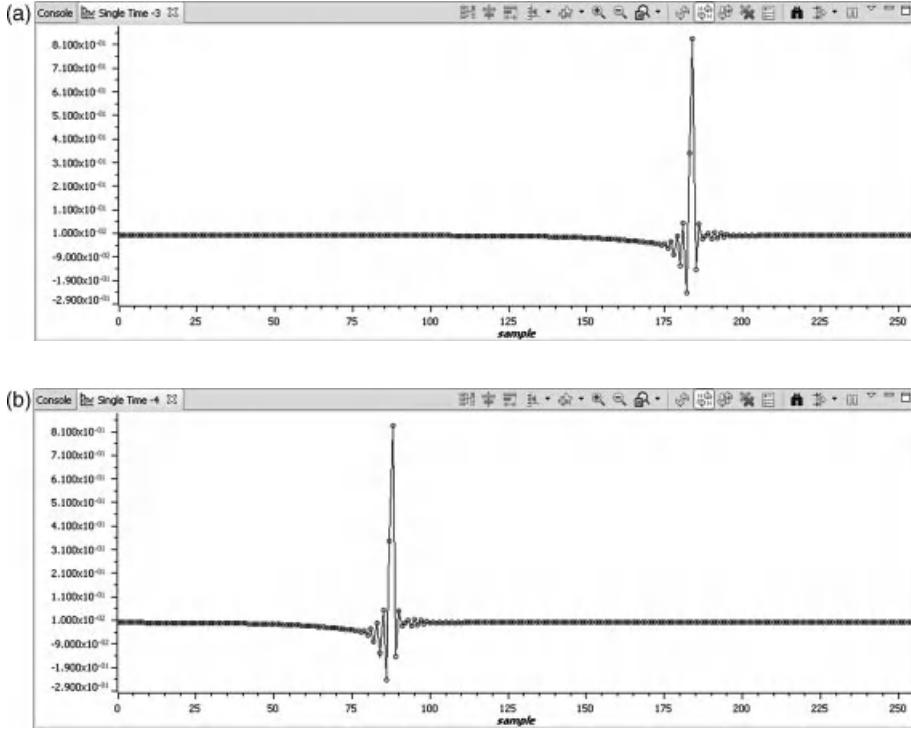


Figure 6.29 Two examples of the adaptive filter weights after running program L138_sysid_DSPLIB_edma.c. (a) with constant BUFCOUNT equal to 32 and (b) with constant BUFCOUNT equal to 128.

path identified and this is evident in the examples of successfully adapted weights shown in Figure 6.29. The 256 adaptive filter weights used by function DSPF_sp_fir_gen() in program L138_sysid_DSPLIB_edma.c are stored in reverse order and therefore the delays in Figure 6.29 can be measured from the right-hand ends of the graphs.

Chapter 7

DSP/BIOS and Platform Support Package

- DSP/BIOS™ real-time operating system
- DSP/BIOS™ software kernel foundation platform support package (PSP)

The examples in this chapter illustrate the use of the DSP/BIOS real-time operating system and the platform support package.

DSP/BIOS provides real-time scheduling, analysis, and data transfer capabilities for an application running on a Texas Instruments DSP. The platform support package provides a further level of abstraction and an API for real-time applications.

7.1 INTRODUCTION TO DSP/BIOS

The following examples introduce some of the real-time scheduling features of DSP/BIOS. Example programs from previous chapters are adapted to run as DSP/BIOS applications. The XDS100 emulator built into the eXperimenter board does not support the real-time analysis (RTA) and real-time data transfer (RTDX) features of DSP/BIOS [1].

7.1.1 DSP/BIOS Threads

At the heart of DSP/BIOS is a preemptive real-time scheduler. This determines which one of a number of different threads is executed by the DSP at any given time. For the purposes of this chapter, DSP/BIOS threads are considered to be sections of program code executed by the DSP. The scheduler determines which thread to execute at a particular time according to its type and associated priority. There are several different types of thread that can be used in a DSP/BIOS application.

- *Hardware Interrupts* (HWIs) have the highest priority in a DSP/BIOS application. Their execution is triggered by interrupts from on-chip peripherals or external devices and they always run to completion. Hardware interrupt threads are not preempted by any other threads and for that reason, HWIs should be used for the most time-critical activities within an application, and the code associated with them, that is, interrupt service routines, should ideally be kept as short as possible.
- *Software Interrupts* (SWIs) are triggered (posted) from within a program. The priority of a SWI thread can be set to one of a number of different levels. SWI threads run to completion unless preempted by a higher priority SWI or by a HWI. The code associated with a SWI is effectively an interrupt service routine, but unlike a HWI interrupt service routine, it can be preempted by higher priority threads. Typically, SWIs are posted by program statements within HWI threads. That way, more time-consuming interrupt processing can be carried out in a SWI thread without blocking or disabling other HWIs.
- *Periodic Functions* (PRDs) are a special type of SWI thread triggered by a dedicated hardware timer. PRD threads may be preempted by higher priority SWI threads and by HWI threads.
- *Tasks* (TSKs) are used for less time-critical activities. They run to completion, but may be preempted by HWIs, SWIs, PRDs, and higher priority TSKs. Tasks can be created dynamically within a DSP/BIOS application, in which case they will be executed starting at the time at which they are created. Otherwise, TSKs will start execution at the start of the DSP/BIOS application. However, TSKs cannot be created from within HWIs or SWIs. A number of different priority levels can be set for a TSK.
- *Idle Functions* (IDLs) are executed repeatedly as part of the lowest priority thread in a DSP/BIOS application. The idle loop runs continuously, but is preempted by HWIs, SWIs, PRDs, and TSKs. If no functions are configured explicitly as IDL threads, then a default idle loop will be entered by DSP/BIOS when no other threads are being executed.

To facilitate synchronization of and communication between tasks, DSP/BIOS provides a *Semaphore* mechanism (SEM). Semaphore objects may be created, destroyed, posted, and waited on using DSP/BIOS functions.

7.1.2 DSP/BIOS Configuration Tool

The DSP/BIOS Configuration Tool is used to create and set the properties of the DSP/BIOS threads that make up an application. It is a visual editor for textual configuration (.tcf) files. In addition, it generates assembly language, header, and linker command files and adds these to a project automatically. A configuration file is central to a DSP/BIOS application since it defines all of the threads used and their parameter settings. In order to create a DSP/BIOS application, program source files, defining the functions used in the threads, must be added to a project. Figure 7.1 shows

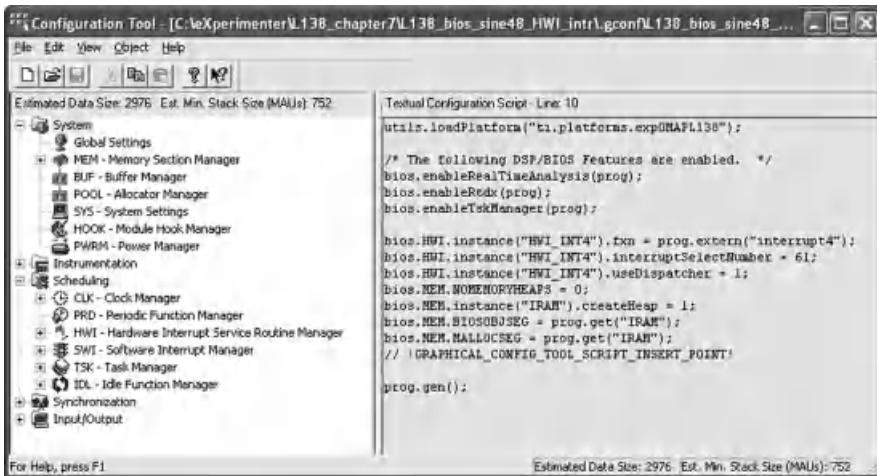


Figure 7.1 A DSP/BIOS Configuration Tool window.

an example of a Configuration Tool window. On the right-hand side of the window is a listing of the .tcf file, the contents of which are altered using the tool.

7.1.3 DSP/BIOS Start-Up Sequence

At the start of the execution of a DSP/BIOS application, the following sequence of steps is followed:

- (1) Initialize the DSP hardware. Initially, all interrupts are disabled.
- (2) Initialize the DSP/BIOS modules. The Interrupt Selector Registers (INTMUXn) and Interrupt Service Table (IST) are set up and the Non-maskable Interrupt (NMI) is enabled by setting bit 2 in the Interrupt Enable Register (IER). Maskable interrupts INT4–INT15 are not enabled.
- (3) Call the function `main()`, defined in a user-supplied source file included in the project. At this point, interrupts are still disabled and application-specific initialization functions (defined in user-supplied source files) can be called. If required, maskable interrupts may be enabled by setting the appropriate bits in the Interrupt Enable Register. DSP/BIOS function `HWI_enable()`, which enables CPU interrupts by setting the Global Interrupt Enable (GIE) bit in the Control Status Register (CSR) should not be called and neither should the GIE bit of the CSR be set by any other means. Function `main()` runs to completion.
- (4) Start DSP/BIOS. At this point, hardware and software interrupts are enabled, the clock that is used by PRD threads is started, and TSKs are enabled. Execution of the highest priority TSK will start and all TSKs will eventually be run to completion.

- (5) When all TSKs have run to completion or are blocked, and when no HWI, SWI, or PRD thread is being executed, the DSP/BIOS idle loop is entered. The idle loop runs forever, but its execution may be preempted by hardware or software interrupts, or by higher priority tasks. The idle loop may comprise the default idle loop or there may be a number of functions configured as IDL threads. IDL threads are run repeatedly in round-robin fashion.

More detailed information about DSP/BIOS can be found in the TMS320 DSP/BIOS v5.41 User's Guide [2].

The following examples illustrate the use of some of the different types of DSP/BIOS threads.

7.1.4 Hardware Interrupts

Many of the example programs in previous chapters made use of interrupts generated by McASP0 in order to perform in real time. The following example illustrates the use of these hardware interrupts in a DSP/BIOS application.

**EXAMPLE 7.1: Sine Generation Using DSP/BIOS Hardware Interrupts HWIs
(L138_bios_sine48_intr_HWI)**

This example illustrates the steps involved in the creation of a simple DSP/BIOS application, based on a previous example. The source file L138_bios_sine48_intr_HWI.c, listed in Figure 7.2, is stored in folder L138_bios_sine48_intr_HWI in workspace L138_chapter7, but no project file is supplied in that folder. The project will be created from scratch.

In this example, program L138_sine48_intr.c has been modified to run as a DSP/BIOS application. That program, listed in Figure 2.30, is quite straightforward. After calling the initialization function L138_initialise_intr(), function main() enters an endless while loop. The interrupt service routine interrupt4() is assigned to interrupt INT4 by means of the interrupt service table set up in file vectors_intr.asm.

The differences between the source file used in Chapter 2 and the file L138_bios_sine48_HWI_intr.c are as follows:

- (1) The while loop in function main() has been removed. There is no need to supply an explicit idle loop since in a DSP/BIOS application, after function main() has completed execution, the application will enter the default DSP/BIOS idle loop.
- (2) The interrupt keyword has been removed from the definition of function interrupt4(). The interrupt keyword specifies that a function is treated as an interrupt function and the compiler saves and restores registers before and after execution based on the rules for interrupt functions [3]. This is handled by DSP/BIOS when a function is specified as a HWI thread and the DSP/BIOS Dispatcher is used.
- (3) Initialization function L138_bios_initialise_intr() defined in file L138_bios_aic3106_init.c is used in place of function L138_initialise_intr(). The differences between these functions that are relevant to this

```

// L138_bios_sine48_intr_HWI.c
//

#include "L138_bios_aic3106_init.h"

#define LOOPLENGTH 48

static int16_t sinetable[LOOPLENGTH] = {0, 1305, 2588, 3827,
    5000, 6088, 7071, 7934, 8660, 9239, 9659, 9914, 10000,
    9914, 9659, 9239, 8660, 7934, 7071, 6088, 5000, 3827,
    2588, 1305, 0, -1305, -2588, -3827, -5000, -6088, -7071,
    -7934, -8660, -9239, -9659, -9914, -10000, -9914, -9659,
    -9239, -8660, -7934, -7071, -6088, -5000, -3827, -2588,
    -1305};

static int sine_ptr = 0; // pointer into lookup table

void interrupt4(void) // interrupt service routine
{
    uint16_t left_sample;

    left_sample = sinetable[sine_ptr];
    sine_ptr = (sine_ptr+1)%LOOPLENGTH;
    output_left_sample(left_sample);

    return;
}

int main(void)
{
    L138_bios_initialise_intr(FS_48000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
}

```

Figure 7.2 Listing of program L138_bios_sine48_HWI_intr.c.

example are that the address of the interrupt service table (`vectors`) is not written to the Interrupt Service Routine Table Pointer (ISTP) and the system event number (#61, McASPO Receive) is not written to the Interrupt Selector Register (INTMUX) because all interrupt configuration is handled by DSP/BIOS as specified in a `.tcf` file. It is still necessary, however, in function `L138_bios_initialise_intr()` to enable interrupt INT4 by setting bit 4 of the Interrupt Enable Register.

(4) Headerfile `L138_bios_aic3106_init.h` replaces `L138_aic3106_init.h`.

The file `vectors_intr.asm` is not required in this example since the mapping of interrupts to interrupt service routines is defined in a textual configuration file and the files it generates.

The steps involved in setting up a new DSP/BIOS project for this example are as follows:

- (1)** Launch the Code Composer Studio IDE, selecting workspace `c:\eXperimenter\`L138_chapter7`. A number of projects associated with other examples in this chapter will appear in the Project View window, but for this particular example, a new project will be created. Create a new project by selecting *File > New > CCS Project* and entering the *Project Name* `L138_bios_sine48_intr_HWI`, as shown in Figure 7.3. Set the *Project Type* as `C6000` and check the *Debug* and *Release* configuration boxes as shown in Figure 7.4. Do not select any interproject dependencies. Select the *Project Settings* as shown in Figure 7.5 and click *Next* rather than *Finish*. Select the *Empty Example* template under *DSP/BIOS v5.xx Examples*, as

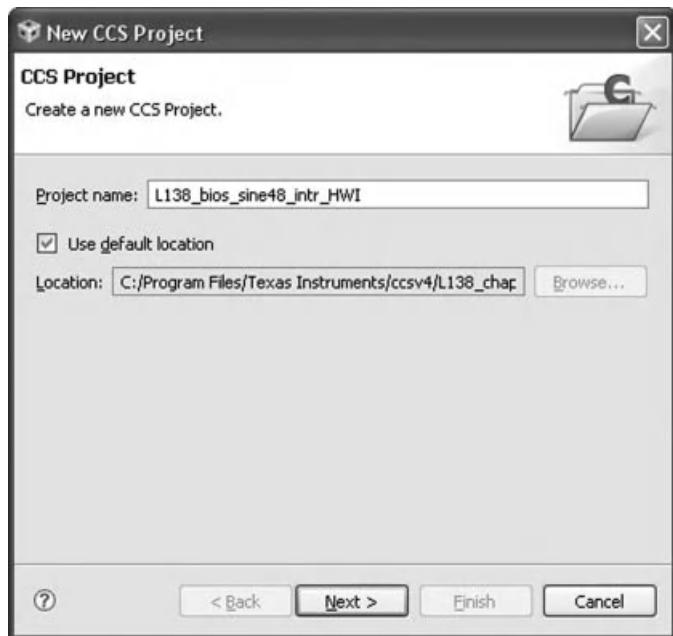


Figure 7.3 Create a new project for program L138_bios_sine48_intr_HWI.c.

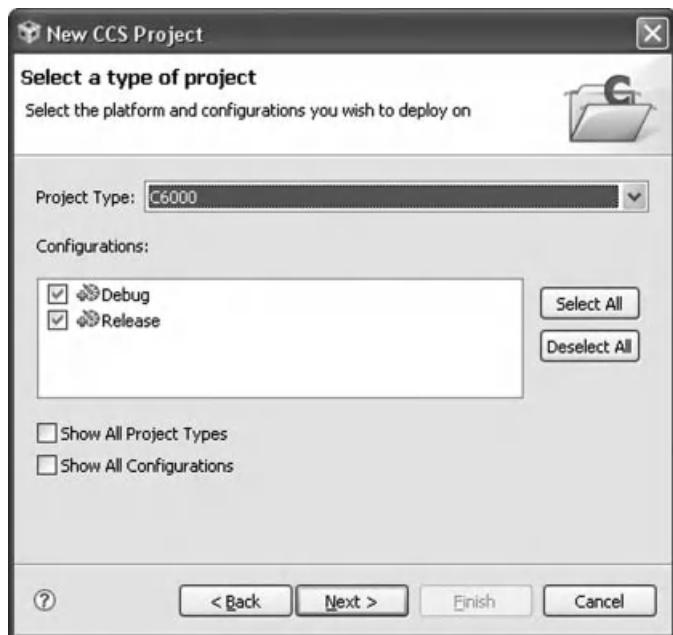


Figure 7.4 Project Type settings for program L138_bios_sine48_intr_HWI.c.

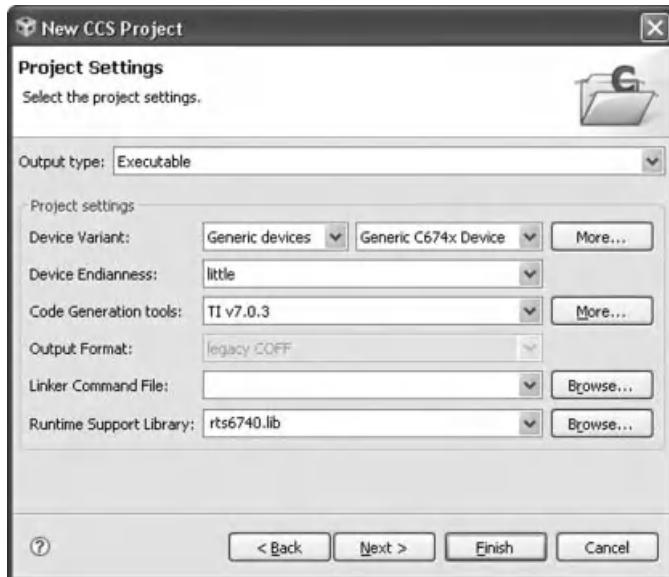


Figure 7.5 Project Settings for program L138_bios_sine48_intr_HWI.c.

shown in Figure 7.6, and then click *Finish*. You should now see a new project, named L138_bios_sine48_intr_HWI, in the *C/C++ Project View* window, and it should contain an *Includes* branch and the source file L138_bios_sine48_intr_HWI.c. The icon to the left of the project name, in the *Project View* window, will have the letters RTSC (Real Time Software Components) rather than the letter C above it. If the project is not active, then right-click on it and *Set as Active Project* (Figure 7.7).

- (2) Select *Project > Link Files to Active Project* to add links to the following files located in folder L138_support. L138_bios_aic3106_init.c, L138_bios_aic3106_init.h, and L138_eXperimenter.ccxml. Linking target configuration file L138_eXperimenter.ccxml to the project is an alternative to specifying the copy of that file stored in folder c:\Documents and Settings \USERID\user\Targetconfigurations as a default. Do not add links to file vectors_intr.asm or to file linker_dsp.cmd. The interrupt service table will be configured by DSP/BIOS according to settings in the textual configuration file. The allocation to memory of sections produced by the compiler, and specification of stack and heap sizes, will also be handled by DSP/BIOS.
- (3) Create a new textual configuration file for the project. In the *C/C++* perspective, select *File > New > DSP/BIOS v5.x Configuration File* and choose L138_bios_sine24_intr_HWI.tcf as a filename (this should be the default choice), as shown in Figure 7.8. Click *Next*. As a platform, choose ti.platforms.expOMAPL138 (Figure 7.9). Click *Next*. Leave the *DSP/BIOS Configuration* as shown in Figure 7.10 and click *Finish*. At this point, a *Configuration Tool* window should open.

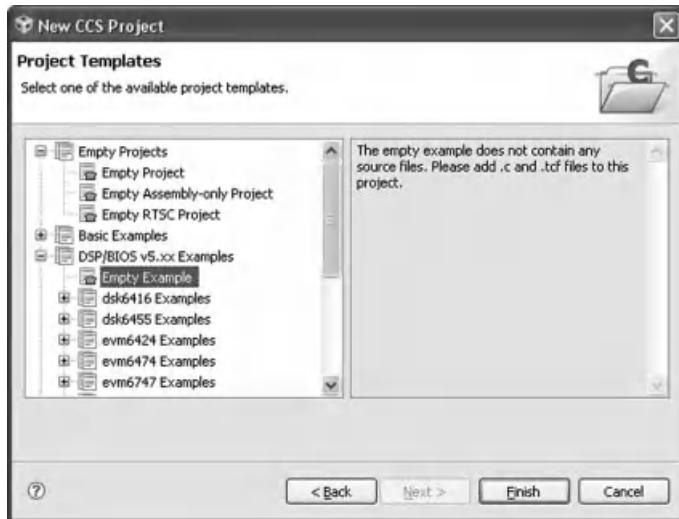


Figure 7.6 Select Project Template *DSP/BIOS v5.xx Examples > Empty Example*.

- (4) Expand *Scheduling* in the left-hand pane of the *Configuration Tool* window and right-click on *HWI – Hardware Interrupt Service Routine Function Manager > HWI_INT4*. Select *Properties* and, in the *HWI_INT4 Properties* pop-up window, enter *_interrupt4* as the *function*, as shown in Figure 7.11. This will associate the function *interrupt4()*, defined in the source file *L138_bios_sine48_intr_HWI.c*, with INT4. Note the underscore prefixing the function name. By convention, this identifies it as a C function and DSP/BIOS will save and restore context appropriately. Set the *interrupt selection number* to 61. This will route event

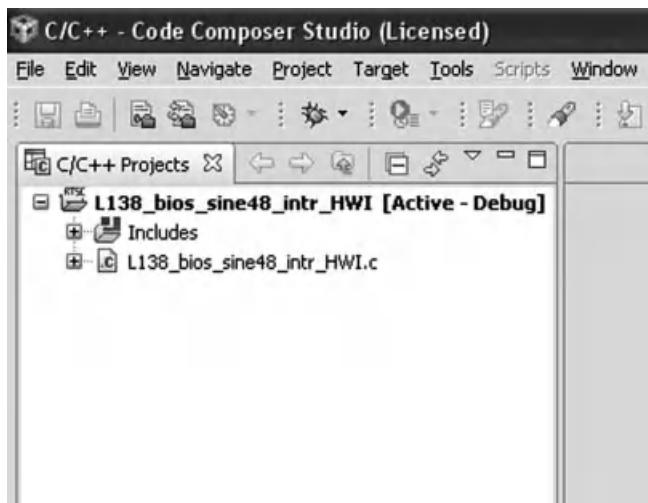


Figure 7.7 Project View window showing newly created project *L138_bios_sine48_intr_HWI*.

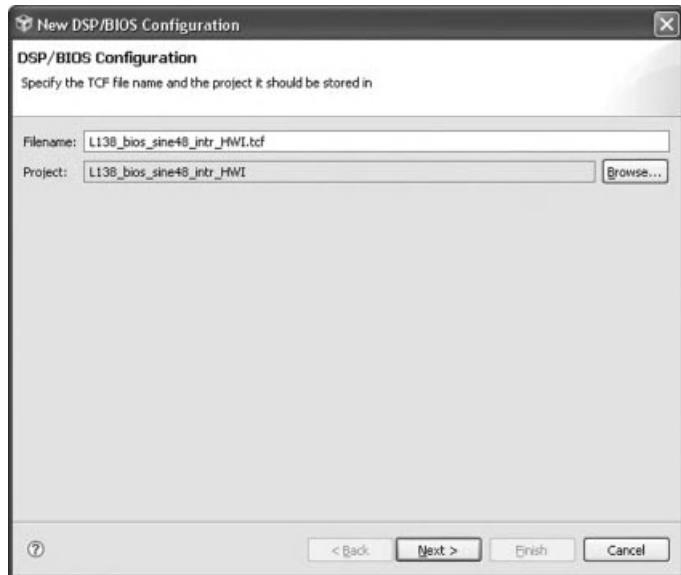


Figure 7.8 Create a textual configuration file for project L138_bios_sine48_intr_HWI.

#61 (McASP0 TX/RX) to CPU interrupt INT4. In previous examples, this routing was set up in function L138_initialise_intr() by writing to the Interrupt Selector Register (INTMUX), but in this example it is handled by DSP/BIOS. Check Use Dispatcher under the *Dispatcher* tab, as shown in Figure 7.12, and click *OK*.

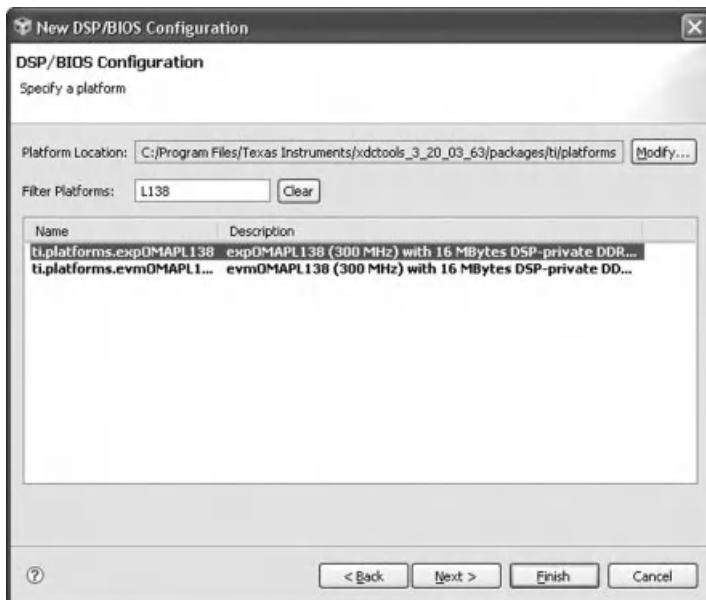


Figure 7.9 Select platform ti.platforms.expOMAPL138 for project L138_bios_sine48_intr_HWI.

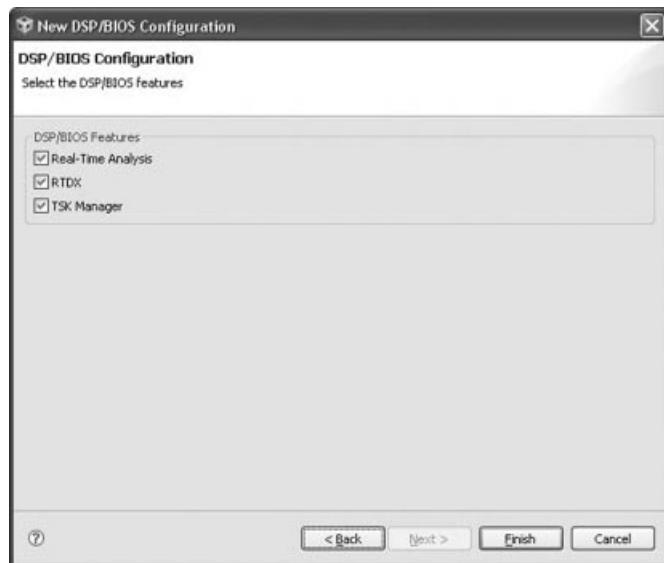


Figure 7.10 DSP/BIOS configuration settings for project L138_bios_sine48_intr_HWI.

- (5) Create dynamic memory heaps. This is a necessary step for DSP/BIOS applications. In the *Configuration Tool* window, right-click on *MEM – Memory Section Manager* (under *System*) and select *Properties*. Uncheck *No Dynamic Memory Heaps* and a warning message, as shown in Figure 7.13, will appear in a pop-up window. Click *OK*

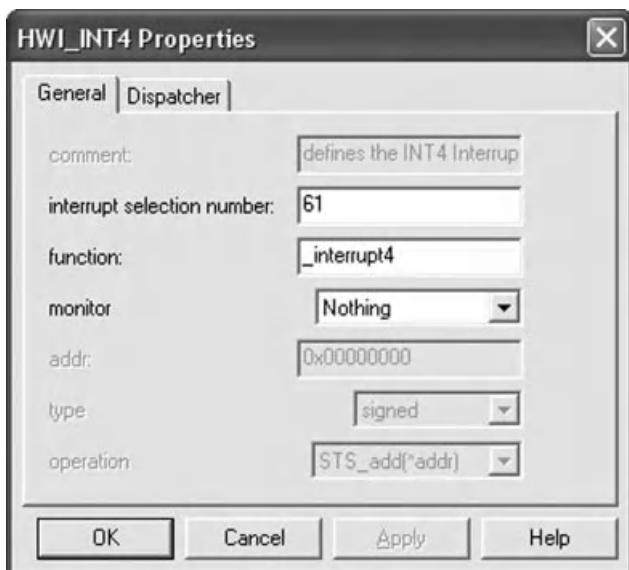


Figure 7.11 Configure CPU interrupt INT4 to be triggered by system event #61 and to use function `interrupt4()` as its interrupt service routine.

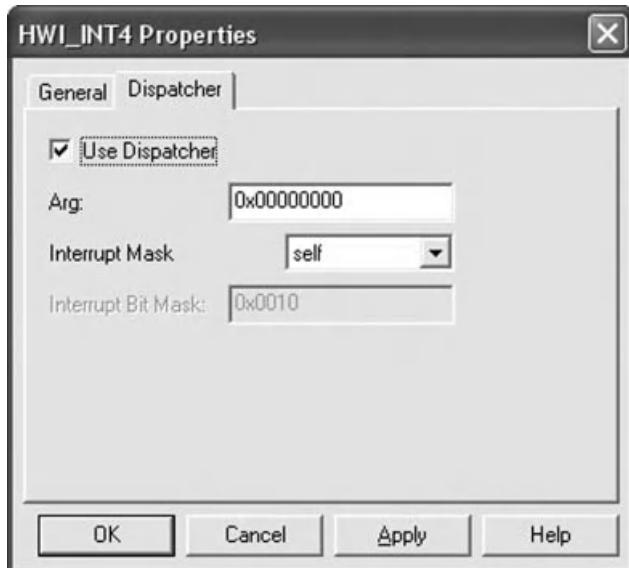


Figure 7.12 Under the *Dispatcher* tab, check *Use Dispatcher*.

in that window. You are just about to reconfigure heaps under individual memory segments. In the Configuration Tool, right-click on *IRAM* (under *MEM – Memory Section Manager*) and select *Properties*. Check *create a heap in this memory* and accept the default properties shown in Figure 7.14 by clicking *OK*. Bring up the *MEM – Memory Section Manager Properties* window again and set both *Segment For DSP/BIOS Objects* and *Segment For malloc()/free()* to *IRAM*, as shown in Figure 7.15. Click *OK*.

- (6) The contents of the textual configuration file *L138_bios_sine48_intr_HWI.tcf* created using the steps above are displayed at the right-hand side of the *Configuration Tool* window which should now look as shown in Figure 7.16. File *L138_bios_sine48_intr_HWI.tcf* is listed in Figure 7.17.
- (7) Close the *Configuration Tool* window, saving the changes to file *L138_bios_sine48_HWI_intr.tcf*.
- (8) Right-click on the project name *L138_bios_sine48_intr_HWI* in the *C/C++ Project View* window and select *Build Properties*. You must tell the compiler



Figure 7.13 Warning message displayed after unchecking *No Dynamic Memory Heaps*.

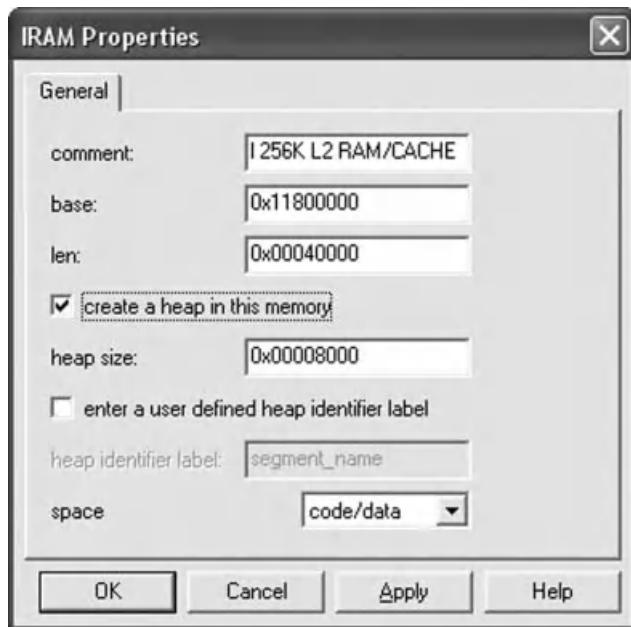


Figure 7.14 Accept these default properties when creating dynamic heaps.



Figure 7.15 Allocate dynamic heaps to IRAM memory segments.

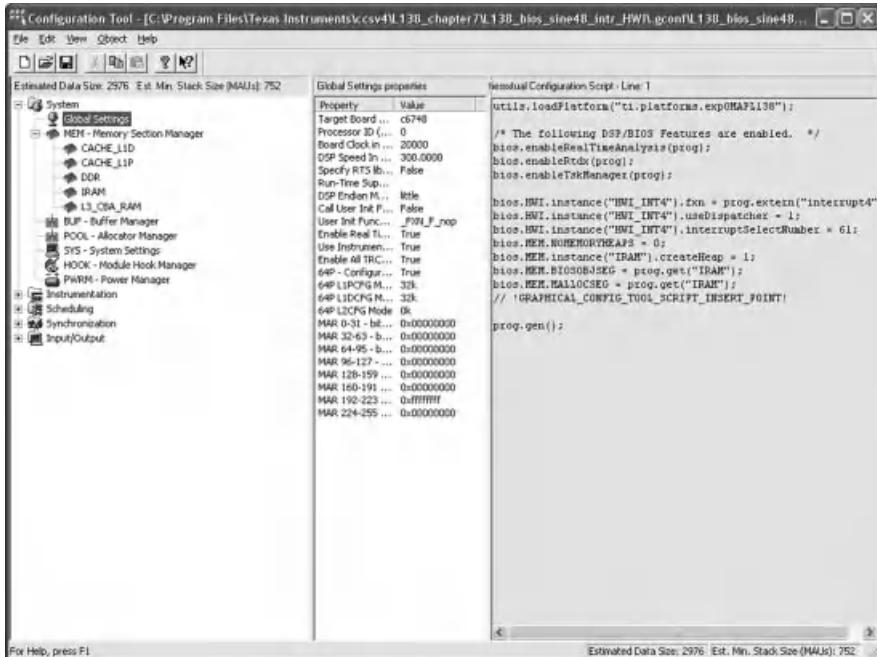


Figure 7.16 Configuration Tool window as it should appear following steps 1–6 above.

where to find certain header and library files. Click on *C6000 Compiler > Include Options* and then on the *Add* button next to *Add dir to #include search path (-include_path, -I)* and add the paths to the required folders. Click on the *File system* button in the window that pops up and browse for the folder containing the OMAP-L138 board support library (BSL) header files, *c:/omapl138/bsl/inc*. Click *OK*. Repeat this procedure in order to add the folder *c:/eXperimenter/L138_support* that contains the header file *L138_bios_aic3106_init.h*.

```
utils.loadPlatform("ti.platforms.expOMAPL138");

/* The following DSP/BIOS Features are enabled. */
bios.enableRealTimeAnalysis(prog);
bios.enableRtdx(prog);
bios.enableTskManager(prog);

bios.HWI.instance("HWI_INT4").fxn = prog.extern("interrupt4");
bios.HWI.instance("HWI_INT4").interruptSelectNumber = 61;
bios.HWI.instance("HWI_INT4").useDispatcher = 1;
bios.MEM.NOMEMORYHEAPS = 0;
bios.MEM.instance("IRAM").createHeap = 1;
bios.MEM.BIOSOBJSEG = prog.get("IRAM");
bios.MEM.MALLOCSEG = prog.get("IRAM");
// !GRAPHICAL_CONFIG_TOOL_SCRIPT_INSERT_POINT!

prog.gen();
```

Figure 7.17 Listing of textual configuration file L138_bios_sine48_intr_HWI.tcf.

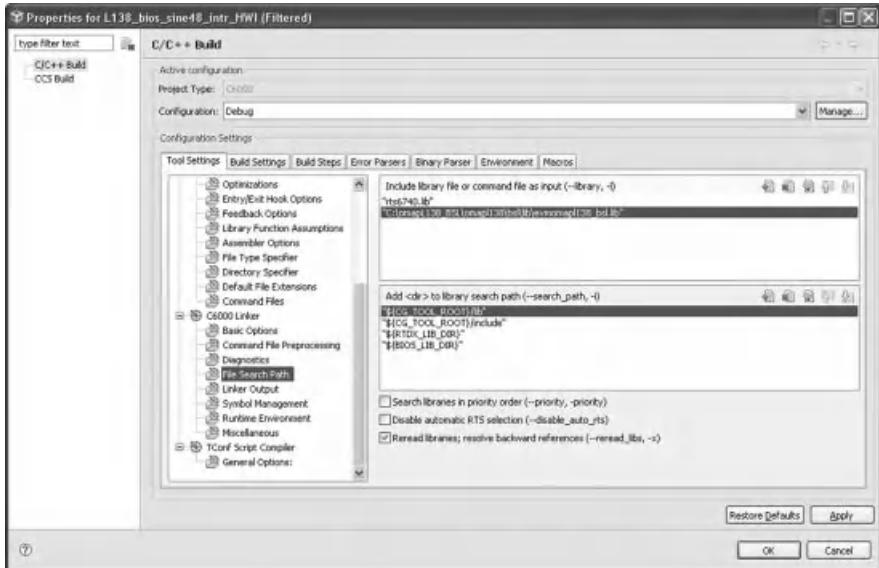


Figure 7.18 Build Properties window as it should appear following addition of path names for header and library files.

Next, click *C6000 Linker > File Search Path* and Add the board support library file `c:\omapl138\bsl\lib\evmomapl138_bsl.lib`. After this step, the *Build Properties* window should appear, as shown in Figure 7.18. Click *Apply* and then *OK*.

- (9) Click on *Debug Active Project* to build the project and then run the program from the debugger. Verify that a 1 kHz sine wave is generated on the left channel of LINE OUT.

7.1.5 Software Interrupts

Because HWI threads have the highest priority in DSP/BIOS, always run to completion, and are not preempted by any other threads, it is advisable to minimize the amount of processing performed by a HWI function. In general, it is recommended that HWI threads deal only with time-critical data transfers and post software interrupts that trigger SWI threads to perform lower priority processing tasks. This technique is illustrated in the following example.

EXAMPLE 7.2: FIR Filter Using DSP/BIOS Hardware Interrupts HWIs and Software Interrupts SWIs (L138_bios_firprn_intr_SWI)

In this example, the FIR filter program `L138_firprn_intr.c`, introduced in Chapter 4, is adapted for use with DSP/BIOS. Whereas in program `L138_firprn_intr.c` the filter output was computed within the hardware interrupt service routine `interrupt4()`, in program `L138_bios_firprn_intr_SWI.c` that computation is performed by a separate function `fir_isr()`. Function `interrupt4()`, executed as HWI object `HWI_INT4`, simply reads a

new input sample from function `prbs()`, writes a filter output sample to the DAC, and posts a software interrupt in order to trigger execution of SWI thread `SWI_fir_isr`. This thread comprises function `fir_isr()` in which a new filter output sample is calculated.

Further modifications made to program `L138_firprn_intr.c` are as follows:

- (1) The program statement `while(1);` in function `main()` has been deleted. After function `main()` has completed execution, the application will enter the default DSP/BIOS idle loop.
- (2) The keyword `interrupt` preceding `void c_int11()` has been deleted.
- (3) The lines `#include <std.h>` and `#include <swi.h>` have been added. These header files contain function prototypes and structure definitions used by SWI threads.
- (4) The line `extern far SWI_Obj SWI_fir_isr;` has been added. This enables the SWI thread `SWI_fir_isr`, added to the DSP/BIOS application, to be used by functions defined in file `L138_bios_firprn_intr_SWI.c` (Figure 7.19).

```
// L138_bios_firprn_intr_SWI.c
//

#include <std.h>
#include <swi.h>

#include "L138_bios_aic3106_init.h"

#include "lp33.coф"           //filter coefficient file
float yn;                   //filter output
float x[N];                 //filter delay line

extern far SWI_Obj SWI_fir_isr;

void interrupt4(void)
{
    x[0] = (float)(prbs());      //get new input into delay line
    output_left_sample((short)(yn)); //output to codec
    SWI_post(&SWI_fir_isr);     // post software interrupt
    return;
}

void fir_isr(void)
{
    short i;

    yn = 0.0;                  //initialise filter output
    for (i=0 ; i<N ; i++)      //calculate filter output
        yn += h[i]*x[i];
    for (i=(N-1) ; i>0 ; i--)  //shuffle delay line contents
        x[i] = x[i-1];
    return;
}

int main(void)
{
    L138_bios_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
}
```

Figure 7.19 Listing of program `L138_bios_firprn_intr_SWI.c`.

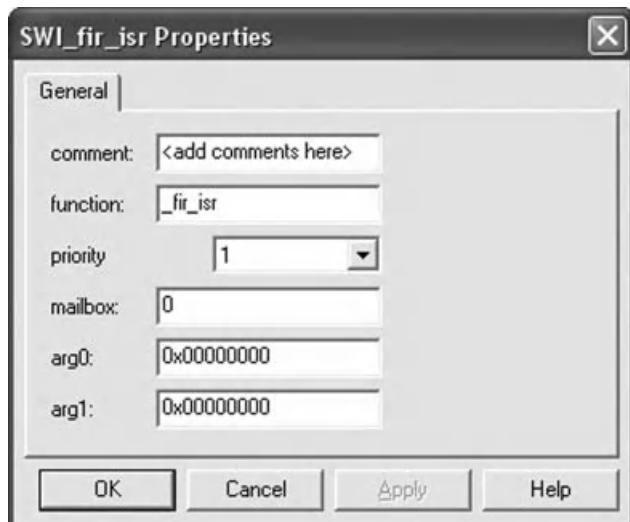


Figure 7.20 Properties of SWI thread SWI_fir_isr.

The program is stored in project folder L138_bios_firprn_intr_SWI along with other files required to build a DSP/BIOS application. Open L138_bios_firprn_intr_SWI.tcf using the Configuration Tool, by double-clicking on it in the *Project View* window and verify that the HWI_INT4 thread *function* has been set to _interrupt4, *interrupt selection number* to 61, and that *Use Dispatcher* has been checked. Verify also that a SWI thread named SWI_fir_isr with its function set to _fir_isr has been configured, as shown in Figure 7.20.

Build, load, and run the program. Low-pass filtered noise should be output via the left channel of LINE OUT.

Also supplied in project folder L138_bios_firprn_intr_SWI, but excluded from build, is program L138_bios_fir_intr_SWI.c that may be used as an alternative to L138_bios_firprn_intr_SWI.c and an input signal supplied using, for example, a signal generator.

7.1.6 Tasks and Idle Functions

EXAMPLE 7.3: System Identification Using DMA-Based I/O and Semaphore to Signal Transfer Complete (L138_bios_sysid_edma_TSK.c)

This example introduces the use of task threads (TSK) and semaphores (SEM). Functionally, program L138_bios_sysid_edma_TSK.c is similar to programs L138_sysid_intr.c and L138_sysid_dsplib_edma.c described in Chapter 6.

Unlike Examples 7.1 and 7.2, this example uses DMA-based rather than interrupt-based I/O. As far as DSP/BIOS configuration is concerned, hardware interrupts are generated by the

EDMA3 channel controller on transfer completion and so just as in Examples 7.1 and 7.2, a HWI object is required. As described in Chapter 2, EDMA3 transfer completion is system event #8 (EDMA3_0_CCO_INT1) and this can be routed to CPU interrupt INT4 using the interrupt selection number parameter of the HWI thread configured using the Configuration Tool. The interrupt service routine `interrupt4()` defined in file `L138_sysid_edma_TSK.c` is similar to the interrupt service routines in other DMA-based I/O example program insofar as it sets the value of the variable `procBuffer` according to which set of I/O buffers was used for the most recent EDMA3 transfer. However, instead of signaling the completion of a transfer by setting the value of variable `buffer_full` equal to 1, it signals using a DSP/BIOS binary semaphore, `SEM_buffer_full` (Figure 7.21).

The program statement

```
SEM_postBinary(SEM_buffer_full);
in source file L138_sysid_edma_TSK.c replaces program statement
buffer_full = 1;
in source file L138_sysid_dsplib_edma.c.
```

In previous DMA-based I/O example programs, function `main()` enters an endless loop

```
while(1)
{
    while (!buffer_full); // wait for DMA transfer complete
    process_buffer();    // process contents of buffer
}
```

but in program `L138_bios_sysid_edma_TSK.c`, function `main()` runs to completion and DSP/BIOS then executes a TSK thread comprising function `endless_task()` that contains the endless loop

```
while(1)
{
    status = SEM_pendBinary(SEM_buffer_full, SYS_FOREVER);
    process_buffer();
}
```

Function `SEM_pendBinary()` waits (forever, if necessary) for function `SEM_postBinary()` to signal using semaphore `SEM_buffer_full`. Function `process_buffer()` contains the implementation of an adaptive filter configured for system identification.

All the files required for this example are stored in project folder `L138_bios_sysid_TSK_edma`. Connect LINE OUT to LINE IN on the eXperimenter board using a 3.5 mm jack to 3.5 mm jack cable. Set project `L138_bios_sysid_edma_TSK` as the active project in workspace `L138_chapter7` and click *Debug Active Project* to build and load the program and then run it. After a few seconds, halt the program and in the *Debug* perspective, select *Tools > Graph > Single Time* and use the *Graph Properties* shown in Figure 7.22 in order to plot the adaptive filter weights. Figure 7.23 shows the adaptive filter weights after running the program with two different values of the constant `BUFCOUNT` (defined in file `L138_bios_aic3106init.h`). Comparing these two graphs, it is apparent that a delay of `BUFCOUNT` sampling periods is introduced by DMA-based I/O. Compare these graphs also with those in Figure 6.28, generated using program `L138_sysid_dsplib_edma.c`.

```

// L138_bios_sysid_edma_TSK.c
//

#include <std.h>
#include <sem.h>
#include "L138_bios_aic3106_init.h"

extern int16_t *pingIN, *pingOUT, *pongIN, *pongOUT;
SEM_Handle SEM_buffer_full;
int procBuffer;

#define beta 1E-12           // learning rate
#define WLENGTH 256

float w[WLENGTH+1];          // adaptive filter weights
float x[WLENGTH+1];          // delay line

void interrupt4(void)         // interrupt service routine
{
    switch(EDMA_3CC_IPR)
    {
        case 1:             // TCC = 0
            procBuffer = PING; // process ping
            EDMA_3CC_ICR = 0x0001; // clear EDMA3 IPR bit TCC
            break;
        case 2:             // TCC = 1
            procBuffer = PONG; // process pong
            EDMA_3CC_ICR = 0x0002; // clear EDMA3 IPR bit TCC
            break;
        default:            // may have missed an interrupt
            EDMA_3CC_ICR = 0x0003; // clear EDMA3 IPR bits 0 and 1
            break;
    }
    EVTCLR0 = 0x00000100;
    SEM_postBinary(SEM_buffer_full);
    return;
}

void process_buffer(void)
{
    int16_t *inBuf, *outBuf;      // pointers to process buffers
    int16_t left_sample, right_sample;
    int i,j;
    float d,y,e;

    if (procBuffer == PING)
    {
        inBuf = pingIN;
        outBuf = pingOUT;
    }
    if (procBuffer == PONG)
    {
        inBuf = pongIN;
        outBuf = pongOUT;
    }
    for (i = 0; i < (BUFCOUNT/2) ; i++)
    {
        left_sample = *inBuf++;
        right_sample = *inBuf++;
        d = (float)(left_sample);

```

Figure 7.21 Listing of program L138_bios_sysid_edma_TSK.c.

```

x[0] = prbs();
left_sample = (int16_t)(x[0]);

*outBuf++ = left_sample;
*outBuf++ = right_sample;

y = 0.0;
for (j = 0; j < WLENGTH; j++)
    y +=(w[j]*x[j]);           //compute filter output
    e = d - y;                // error signal

for (j = WLENGTH-1; j >= 0; j--)
{
    w[j] = w[j]+(beta*e*x[j]); // update weights
    x[j+1] = x[j];            // update delay line
}
}

void endless_task(void)
{
    int status;

    while(1)
    {
        status = SEM_pendBinary(SEM_buffer_full, SYS_FOREVER);
        process_buffer();
    }
}

int main(void)
{
    int i;

    for (i=0; i <= WLENGTH; i++)
    {
        w[i] = 0.0;      // initialise adaptive filter weights
        x[i] = 0.0;      // initialise adaptive filter delay line
    }
    SEM_buffer_full = SEM_create(0, NULL);
    L138_bios_initialise_edma(FS_16000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
}
}

```

Figure 7.21 (Continued)

EXAMPLE 7.4: System Identification Using DMA-Based I/O and Semaphore to Signal Transfer Complete (L138_bios_sysid_edma_IDL)

This example is closely related to the previous one, but instead of using a while loop within a TSK thread, it relies on the repeated execution in DSP/BIOS of a function configured as an IDL thread. In function `endless_idle()`, function `SEM_pendBinary()` waits for function `SEM_postBinary()` to signal using semaphore `SEM_buffer_full` before calling function `process_buffer()`. Function `endless_idle()` is similar to function `endless_task()` in Example 7.3, except that it does not contain a while loop and will

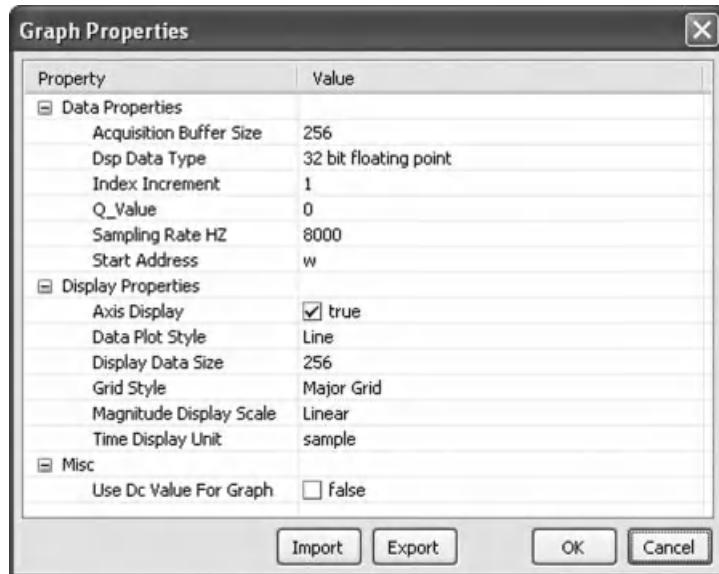
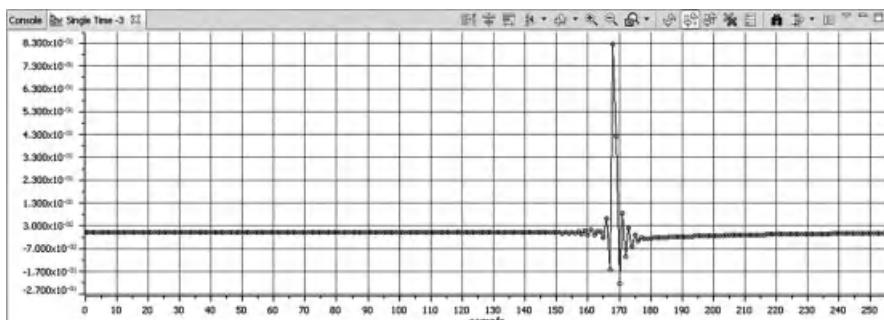
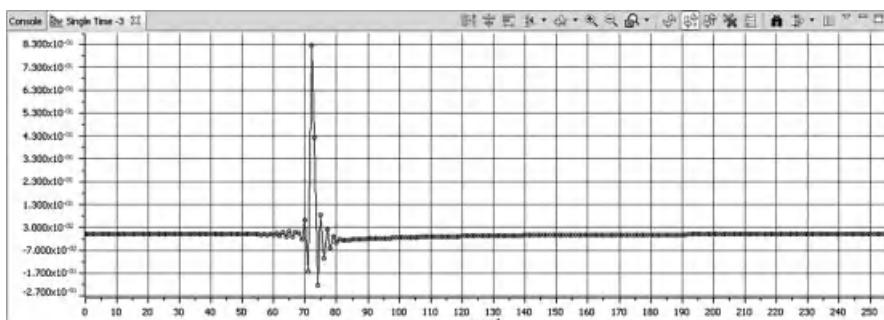


Figure 7.22 Graph Properties used to plot adaptive filter weights used by program L138_bios_sysid_edma_TSK.c.



(a)



(b)

Figure 7.23 Impulse responses identified using program L138_bios_sysid_edma_TSK.c.
(a) BUFCOUNT = 128. (b) BUFCOUNT = 32.

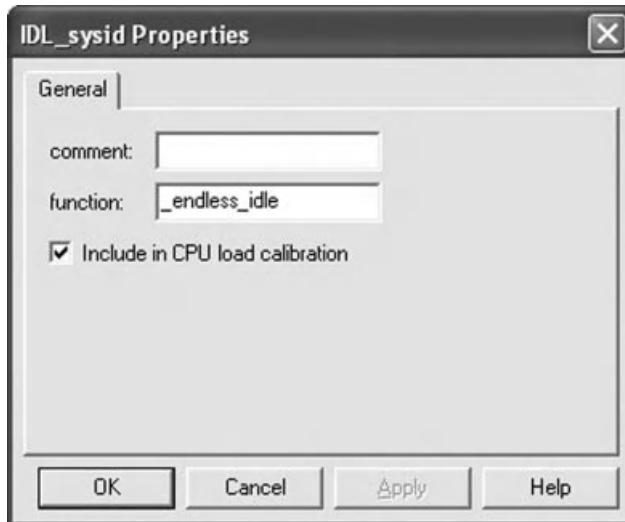


Figure 7.24 Properties of IDL object `IDL_sysid`.

return after calling function `process_buffer()` just once. The DSP/BIOS scheduler executes IDL threads repeatedly, when not preempted by HWI, SWI, PRD, or TSK threads, and hence the buffer processing operation will be executed each time semaphore `SEM_buffer_full` is signaled. Open the configuration file `L138_bios_sysid_edma_IDL.tcf` using the Configuration Tool and verify that there are no TSK objects defined, but that there is an IDL object `IDL_sysid` configured, as shown in Figure 7.24.

Build, load, and run the program and verify performance similar to that of program `L138_bios_sysid_edma_TSK.c`.

7.1.7 Periodic Functions

Periodic threads (PRDs) are similar to hardware interrupt threads except that they are triggered exclusively by interrupts from (internal) clocks.

EXAMPLE 7.5: **Blinking of LEDs at Different Rates Using DSP/BIOS PRDs
(`L138_bios_LED_PRD`)**

Example program `L138_bios_LED_PRD.c`, listed in Figure 7.25, illustrates the use of PRD threads.

Function `main()` performs some necessary eXperimenter board initializations and then returns, passing control to the default DSP/BIOS idle loop. Two PRD objects `PRD_LED0` and `PRD_LED1` have been configured using the *Configuration Tool*, as shown in Figure 7.26. These are scheduled to execute functions `ledToggle()` and `ledToggle1()`, defined in file `L138_bios_LED_PRD.c`, at intervals of 2000 and 200 PRD clock ticks, respectively.

The Board Support Library routine `USTIMER_init()` supplied with the eXperimenter and used in most of the examples in this book uses both timers `TIM0` and `TIM1` and as such is

```

// L138_bios_LED_PRD.c
//

#include "stdio.h"
#include "stdlib.h"
#include "types.h"
#include "evmomapl138.h"
#include "evmomapl138_gpio.h"
#include "evmomapl138_i2c.h"
#include "evmomapl138_led.h"
#include <std.h>
#include <clk.h>

void main(void)
{
    I2C_init(I2C0, I2C_CLK_400K);      // init I2C channel
    LED_init();                      // init LED and DIP BSL
    return;                          // return to BIOS scheduler
}

//-----
// USTIMER_delay()
//
// LogicPD BSL fxn - re-written for a few BSL.c files that
// need it. The original USTIMER_init() is not used because
// it is NOT BIOS compatible and took both timers so that
// BIOS PRDs would not work. This is a workaround.
//
// If you need a "delay" in this app, call this routine with
// the number of usec's of delay you need. It is approximate
// - not exact. value for time<300 is perfect for 1us. We
// padded it some.
//-----

void USTIMER_delay(uint32_t usec)
{
    volatile int32_t i, start, time, current;
    for (i=0; i<usec; i++)
    {
        start = CLK_gettime();
        time = 0;
        while (time < 350)
        {
            current = CLK_gettime();
            time = current - start;
        }
    }
}

void ledToggle(void)
{
    LED_toggle(0);                  // toggle LED_0 on OMAP-L138 EVM
}

void ledToggle1(void)
{
    LED_toggle(1);                 // toggle LED_1 on OMAP-L138 EVM
}

```

Figure 7.25 Listing of program L138_bios_LED_PRD.c.

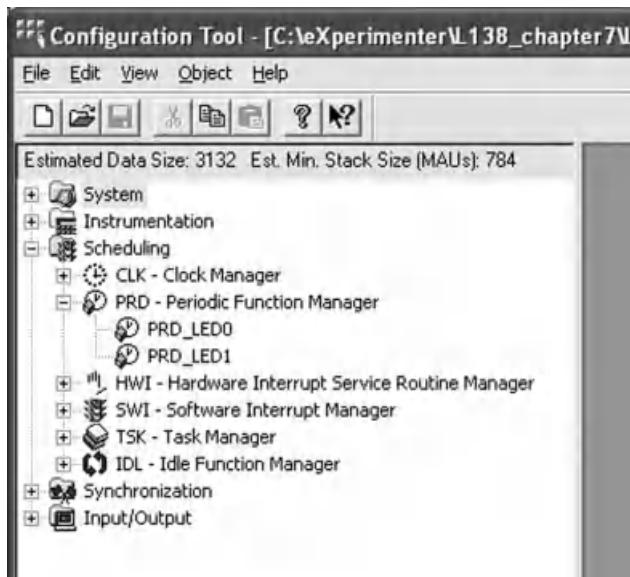


Figure 7.26 PRD objects PRD_LED0 and PRD_LED1 as they appear in the Configuration Tool window.

incompatible with the DSP/BIOS PRD mechanism. This example uses an alternative timer routine `USTIMER_delay()` provided by a Texas Instruments TTO DSP/BIOS workshop [4].

Build and run the program. You should see the two user LEDs on the eXperimenter board switching on and off at frequencies of approximately 5 and 0.5 Hz. You can change the rates at which the LEDs flash by changing the period (ticks) parameter in the properties of PRD objects `PRD_LED0` or `PRD_LED1`, as shown in Figure 7.27.

7.1.8 Real-Time Analysis with DSP/BIOS

Real-time analysis (RTA) and real-time data exchange (RTDX) are not supported by the XDS100 emulator built into the OMAP-L138 eXperimenter board.

7.2 DSP/BIOS PLATFORM SUPPORT PACKAGE

The PSP provides a further level of abstraction than DSP/BIOS and an API for real-time applications.

In order to run the example programs described in this section, the PSP must be installed. The example programs were developed and tested using version 01.30.01, downloaded and installed according to the instructions at http://processors.wiki.ti.com/index.php/GSG_C6748:_Installing_the_SDK_Software.

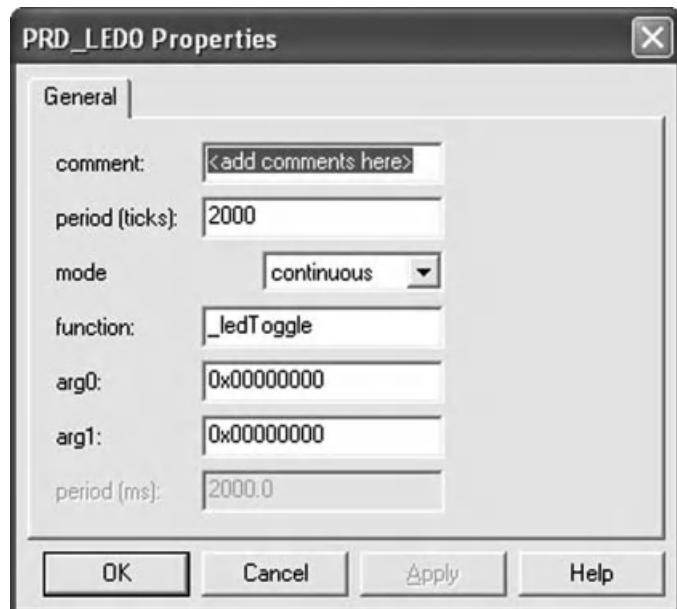


Figure 7.27 Properties of PRD object PRD_LED0.

The following PSP examples are all based on the example program `audio-Sample`, provided with the PSP, which reads an input signal from the AIC3106 ADC and writes it to the AIC3106 DAC. Almost all the real-time example programs in this book can be adapted to fit into the framework provided by the `audio-Sample` example by modifying the source file `audioSample_io.c`. The program uses DMA-based I/O.

In the example programs described here, some, or all, of the following additions and modifications to source file `audioSample_io.c` have been made.

- (1) The example program filename has been added in a comment at the beginning of the source file.
- (2) Some additional header files have been included. In all cases, the file `L138_psp_utils.h` provided with this book in folder `L138_support` is included.
- (3) Some global variables, specific to the example, are declared.
- (4) The value of the constant `BUFSIZE` and the sampling rate may, or may not, have been changed.
- (5) Within function `Audio_echo_Task()`, some additional local variables are declared. `pBuf`, a pointer into the SIO input and output buffers, used by the program is always declared. Other local variable declarations are specific to particular examples.

- (6) Function `L138_AIC3106_writeReg()`, defined in `L138_psp_utils.c`, is used to alter the configuration of the AIC3106 codec and override the parameter settings made during initialization of the codec by the PSP. To be consistent with the other example programs in this book, automatic gain control (AGC) is disabled and ADC gain and DAC attenuation are set to 0 dB. Also, the DAC settings on L and R channels are made independent. In some of the examples, further AIC3106 parameters are altered.
- (7) The processing algorithm specific to an example is added in function `Audio_echo_Task()` after the SIO input buffer `rcv` has been reclaimed from `inStream` and before its contents are copied to the SIO output buffer `xmt`, ready to be issued to `outStream`.

Replace folder `c:\C6748_dsp_1_00_00_11\pspdrivers_01_30_01\packages\ti\pspiom\examples\evmOMAPL138\audio` with folder `c:\experimenter\audio`.

Launch the Code Composer Studio IDE, selecting workspace `L138_chapter7`, and select *Project > Import Existing CCS Project* in order to import the `audioSample` project.

The example programs in this section are all run from the same project folder, different examples being chosen by excluding source files from the project build.

The *Project View* window in the *C/C++* perspective in the Code Composer Studio IDE should look as shown in Figure 7.28.

EXAMPLE 7.6: Basic Input and Output (`L138_psp_loop.c`)

Program `L138_psp_loop.c` makes a bare minimum of changes to program `audio_sample_io.c`. It uses function `L138_AIC3106_regWrite()`, defined in file `L138_psp_utils.c`, in order to configure the AIC3106 codec in the same way as in the other example programs in this book. Specifically, the ADC automatic gain control is disabled and the ADC gain and DAC attenuation are both set to 0 dB. Select *Target > Debug Active Project* having first ensured that only source files `audioSample_main.c`, `L138_psp_loop.c`, and `L138_psp_utils.c` are not *Excluded from Build* in project `audioSample`, as shown in Figure 7.28. Run the program and use a signal generator to verify that as the amplitude of the input signal is varied between approximately 100 and 400 mV, the amplitude of the output signal varies accordingly.

The characteristics of the original program `audioSample_io.c` can be restored in program `L138_psp_loop.c` by commenting out the program statements in which the AIC3106 parameters are altered, that is, by commenting out the following program statements:

```
L138_AIC3106_writeReg(outStream, 0, 0x00); // select page 0
L138_AIC3106_writeReg(outStream, 41, 0x00); // DAC o/p switch
L138_AIC3106_writeReg(outStream, 43, 0x00); // L DAC volume
L138_AIC3106_writeReg(outStream, 44, 0x00); // R DAC volume
L138_AIC3106_writeReg(outStream, 26, 0x00); // L AGC control
L138_AIC3106_writeReg(outStream, 29, 0x00); // R AGC control
```

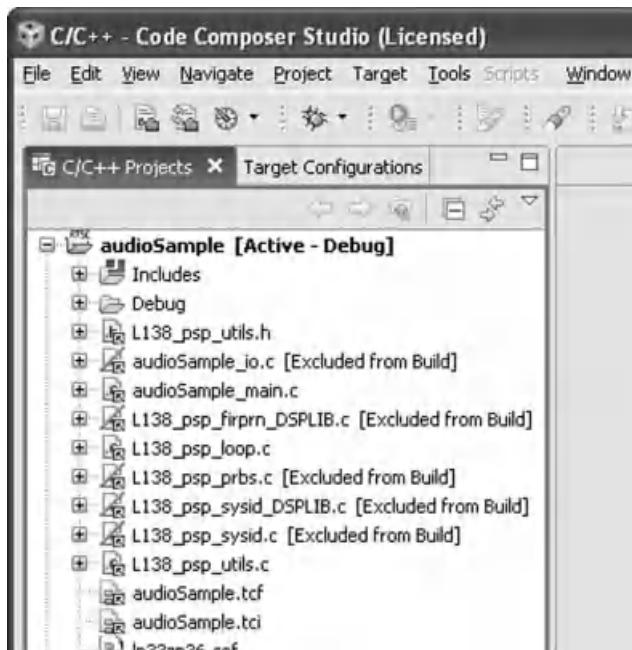


Figure 7.28 Project View window after importing project audioSample.

```
L138_AIC3106_writeReg(outStream, 12, 0x00); // digital filter control
L138_AIC3106_writeReg(outStream, 15, 0x00); // L PGA gain
L138_AIC3106_writeReg(outStream, 16, 0x00); // R PGA gain
```

Rebuild, load, and run program `L138_psp_loop.c` after commenting out these program statements, and verify that due to AGC, the amplitude of the output signal remains constant at approximately 100 mV for a range of different input signal amplitudes.

EXAMPLE 7.7: Generation of Pseudorandom Noise (`L138_psp_prbs.c`)

This example program is the functional equivalent of example program `L138_prbs_intr.c` described in Chapter 2.

The following differences exist between file `L138_prbs_intr.c` and the original version of `audioSample_io.c`:

- (1) The filename `L138_psp_prbs.c` has been appended as a comment at the top of the source file.
- (2) The header file `L138_psp_utils.h` has been included and this allows the functions `prbs()` and `L138_aic3106_regWrite()` to be used.
- (3) The sampling rate has been changed to 8 kHz.
- (4) The local variables `*pBuf`, `left_sample`, `right_sample`, `i`, and `AIC31_data` have been declared in function `Audio_echo_task()`.

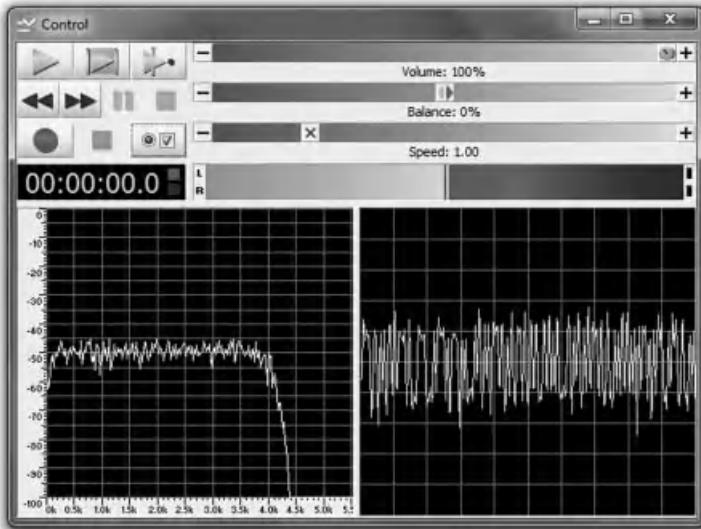


Figure 7.29 Output signal generated using program L138_psp_prbs.c.

- (5) As in program L138_psp_loop.c, some of the codec parameters are set and AGC is disabled.
- (6) After reclaiming the `rcv` buffer from `inStream` and before copying its contents to the `xmt` buffer, it is filled with values from a pseudorandom binary sequence generated using function `prbs()`. The same sequence is written to both left and right channels.

Figure 7.29 shows the signal output by the program displayed using *Goldwave*. Compare this with Figure 2.47.

In order to verify correct operation of function of L138_AIC3106_regWrite(), make the following changes to program L138_psp_prbs.c:

```
L138_AIC3106_regWrite(outStream, 12, 0x01);
L138_AIC3106_regWrite(outStream, 43, 0x18);
```

These will enable de-emphasis on the right channel and set DAC attenuation to -12 dB on the left channel. The resultant output signals viewed using *Goldwave* should be as shown in Figure 7.30.

**EXAMPLE 7.8: FIR Filter Using Pseudorandom Noise as Input
(L138_psp_firprn_DSPLIB)**

This program is the functional equivalent of program L138_firprn_dsplib_edma.c., described in Chapter 3. FIR filter coefficients are read from a separate file, and the FIR filter is implemented using DSPLIB function `DSPF_sp_fir_gen()`, which, since it processes

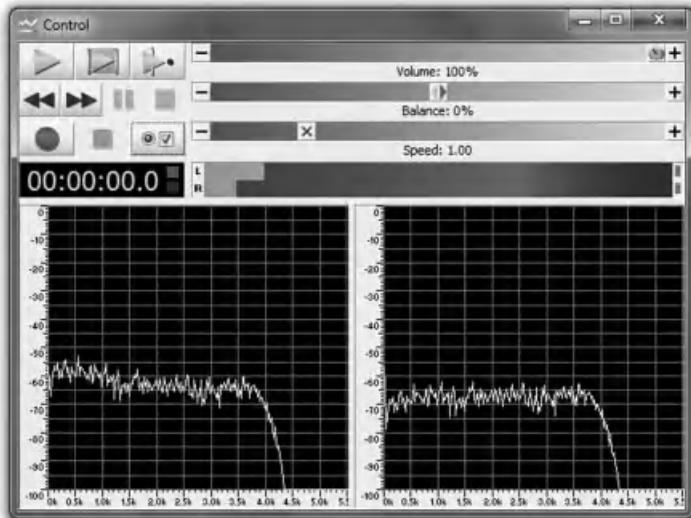


Figure 7.30 Output signal generated using modified version of program L138_psp_prbs.c.

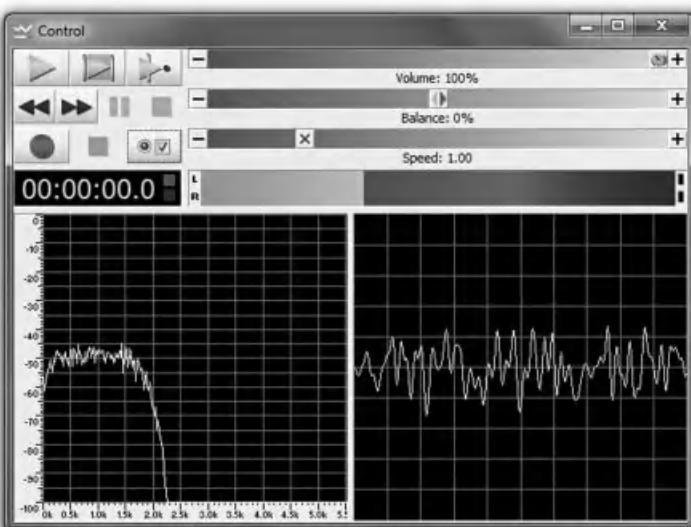


Figure 7.31 Output signal generated using program L138_psp_firprn.c.

frames of samples, is appropriate for the DMA-based I/O in the `audioSample` example. Among other changes, this program requires the lines

```
#include "dsplib674x.h"
```

and

```
#include "lp33zp36.cof"
```

to be added to the original version of `audioSample_io.c`. File `lp33zp36.cof` contains 36 filter coefficients representing a 33-coefficient low-pass filter design zero padded to length 36. Function `DSPF_sp_fir_gen()` requires that the number of coefficients used is an integer multiple of 4. Figure 7.31 shows the filtered noise output by the program.

Nearly all the real-time example programs described in the preceding chapters can be adapted to work with PSP by making straightforward modifications to the `audioSample` example. A number of examples are provided in folder `c:\eXperimenter\audio`.

REFERENCES

1. <http://processors.wiki.ti.com/index.php/XDS100>.
2. *TMS320 DSP/BIOS v5.41 User's Guide*, SPRU423H, Texas Instruments, Dallas, TX, 2009.
3. *TMS320C6000 Optimizing Compiler v 6.1User's Guide*, SPRU187O, Texas Instruments, Dallas, TX, 2008.
4. *C6000 Embedded Design Workshop Using BIOS*, Texas Instruments, Dallas, TX, 2010.

Index

- Adaptive filters, 279
 - for channel equalization, 283
 - for noise cancellation, 281
 - for prediction, 280
 - for sinusoidal noise cancellation, 291
 - for system ID of FIR filter, 296
 - for system ID of IIR filter, 301
 - for system ID of moving average filter, 131
- AIC3106 codec, 1, 39
 - ADC gain, 44
 - DAC attenuation, 44
 - de-emphasis, 78
 - format of data, to and from, 43
 - identification of bandwidth of, 78, 89
 - impulse response of, 77
 - programmable digital effects
 - filter, 83, 96
 - sampling frequency, 46
 - Step response, 85
- Aliasing, 39, 73, 89
 - in impulse invariance method, 178
- Amplitude modulation (AM), 98
- Analog-to-digital converter (ADC), 38
- Antialiasing filter, 39, 88
- ARM processor, 1
- Bilinear transformation (BLT), 167
 - design procedure using, 169
 - frequency warping in, 169, 183
- Bit reversed addressing, 220
- Blackman window function, 115
- Board support library (BSL), 5
- Boot mode, 8
- Breakpoints, 28
- Butterfly structure, 217
- C6748 processor, 1
- Cascade IIR filter structure, 164
- Code Composer Studio (CCS), 5, 6
 - Build Properties, 20
 - Debug window, 7
 - installation and support, 7
 - launching, 8
 - Memory window, 25
 - perspectives, 7
 - Project View window, 7
 - Watch window, 28
- Codec, 38
- Compiler Optimization Level, 31
- Control status register (CSR), 49
- Convolution, 104, 258
- Decimation-in-frequency (DIF) algorithm, 214
- Decimation-in-time (DIT) algorithm, 218
- Difference equations, 111
 - DTMF tone generation using, 206
 - sine generation using, 203
 - swept sinusoid generation using, 208
- Digital-to-analog converter (DAC), 38
- DIP switch
 - Boot mode configuration, 8
 - User, 71
- Direct form I IIR filter structure, 160
- Direct form II IIR filter structure, 161
- Direct form II transpose IIR filter structure, 162
- Direct memory access (DMA), 41, 50, 152, 236
- Discrete Fourier transform (DFT), 212
 - of real-number sequence, 223
 - of real-time signal, 248

- Discrete-time Fourier Transform (DTFT),
112, 212
- DSP/BIOS
configuration tool, 308
hardware interrupt threads (HWI), 308, 310
idle function threads (IDL), 308, 322
periodic threads (PRD), 308, 327
software interrupt threads (SWI), 308, 320
task threads (TSK), 308, 322
threads, 307
- DTMF generation
using difference equations, 206
using lookup tables, 73
- EDMA3, 50, 236
Architecture, 236
Events, 48
Event-triggered transfers, 237
Manually-triggered transfers, 237
Parameter linking, 241
Parameter RAM (PaRAM), 239
- Fast convolution, 258
real-time, 270
demonstration, 261
- Fast Fourier transform (FFT), 212
bit reversed addressing, 220
butterfly structure, 217
decimation-in-frequency algorithm
for, 214
decimation-in-time algorithm for, 218
radix-2, 213
radix-4, 221
of real-time input, 255
of a real-time input signal using an FFT
function in C, 255
of a real-time input signal using DSPLIB
radix-2 function, 257
of a sinusoidal signal, 258
- `fdatool` filter design and analysis
tool, 136, 185
- Finite impulse response (FIR)
filters, 103
with internally generated pseudorandom
noise,
window design method, 113
- Fourier series (FS), 212
- Fourier transform (FT), 212
- Frame-based processing, 233
- Frequency inversion, scrambling by, 150
Frequency warping, 169, 183
- Goldwave*, 5
- Graphic equalizer, 276
- Graph Plotting
in CCS, 22
in MATLAB, 24
- Hamming window function, 114, 254
- Hanning window function, 114
- HWI (DSP/BIOS Hardware interrupt thread),
308, 310
- IDL (DSP/BIOS Idle thread), 308, 322
- Impulse invariance method, 166, 171
- Infinite impulse response (IIR) filters, 159
second order sections, 172
- I/O
DMA-based, 50
interrupt-based, 46
polling-based, 42
- Interrupt clear register (ICR), 49
- Interrupt enable register (IER), 49
- Interrupt selector (IS), 48
- Interrupts, 48
- Interrupt service fetch packet (IFP), 49
- Interrupt service table pointer (ISTP), 49
- Inverse fast Fourier transform (IFFT), 223
- JTAG, 7
- Kaiser window function, 115
- L138_adaptc** project, 288
L138_adaptIDFIR_init_intr
project, 299
L138_adaptIDFIR_intr project, 296
L138_adaptnoise_intr project, 291
L138_adaptnoise2IN_intr
project, 294
L138_aic3106_init.c file, 18, 31
L138_aic3106_init.h file, 18, 32
L138_aliasing_intr project, 89
L138_am_poll project, 98
L138_average_intr project, 123
L138_average_prn_intr project, 126
L138_bios_firprn_intr_SWI
project, 320

- L138_bios_LED_PRD** project, 327
- L138_bios_sine48_intr_HWI** project, 310
- L138_bios_sysid_edma_IDL** project, 325
- L138_bios_sysid_edma_TSK** project, 322
- L138_delay_intr** project, 53
- L138_dft** project, 223
- L138_dft128_edma** project, 248
- L138_dimpulse_intr** project, 77
- L138_dotp4** project, 26
- L138_echo_intr** project, 55
- L138_fastconv_demo** project, 261
- L138_fastconv_edma** project, 270
- L138_fft_sinetable_edma** project, 258
- L138_fft128_dsplibr2_edma** project, 257
- L138_fft128_edma** project, 255
- L138_fir_dsplib_edma** project, 153
- L138_fir_edma** project, 152
- L138_fir_intr** project, 134
- L138_fir3lp_intr** project, 144
- L138_fir4types_intr** project, 147
- L138_FIRcasm_intr** project, 155
- L138_firprn_intr** project, 138
- L138_firprnbuf_intr** project, 141
- L138_flanger_intr** project, 56
- L138_graphicEQ_DSPLIB_edma** project, 276
- L138_iir_intr** project, 195
- L138_iirsos_DSPLIB_edma** project, 192
- L138_iirsos_intr** project, 172
- L138_iirsosadapt_intr** project, 301
- L138_iirsosdelta_intr** project, 177
- L138_iirsosprn_intr** project, 175
- L138_iirsostr_intr** project, 175
- L138_loop_buf_intr** project, 62
- L138_loop_edma** project, 50
- L138_loop_intr** project, 46
- L138_loop_poll** project, 42
- L138_mem_edma** project, 241
- L138_notch2_intr** project, 148
- L138_prandom_intr** project, 85
- L138_prbs_intr** project, 78
- L138_ramp_intr** project, 97
- L138_record_poll** project, 101
- L138_scrambler_intr** project, 150
- L138_sine_DIP_intr** project, 70
- L138_sine_intr** project, 69
- L138_sine48_buf_intr** project, 14
- L138_sine48_intr** project, 65
- L138_sine48_loop_intr** project, 88
- L138_sineDTMF_intr** project, 73
- L138_sinegen_casm_intr** project, 210
- L138_sinegenDE_intr** project, 203
- L138_sinegenDTMF_intr** project, 206
- L138_squarewave_intr** project, 75
- L138_sweep_poll** project, 70
- L138_sweepDE_intr**, 208
- L138_sysid_average_intr** project, 131
- L138_sysid_biquad_intr** project, 196
- L138_sysid_DSPLIB_edma** project, 305
- L138_sysid_intr** project, 89, 128, 302
- Least mean squares (LMS) algorithm, 287
 - sign-data algorithm, 288
 - sign-error algorithm, 288
 - sign-sign algorithm, 288
- LED, 6
- linker_dsp.cmd linker command file, 18
- Lookup table
 - DTMF generation with, 73
 - impulse generation with, 77
 - sine wave generation with, 14, 65
 - square-wave generation with, 75
 - swept sine wave generation with, 71
- Memory data, viewing and saving, 25
- Moving average filter, 123
- Multichannel audio serial port (McASP), 38
- Noise cancellation, 281
- Nonmaskable interrupt (NMI), 48
- Notch filters, to recover corrupted input voice, 148
- Overlap-add, 261
- Parallel form IIR filter structure, 165
- Parks-Miller algorithm, 85
- Performance function, 283
- Ping-pong buffers, 50

- Platform support package (PSP), 5, 329
- pragma directive, 102
- PRBS, 78
- PRD (DSP/BIOS Periodic thread), 308, 327
- Profile clock, 31
- Project view window 16
- Pseudorandom noise, 85
 - as input to FIR filter, 138
 - as input to IIR filter, 175
 - as input to moving average filter, 126
- Radix-2 decimation-in-frequency FFT
 - algorithm, 214
- Radix-2 decimation-in-time FFT
 - algorithm, 218
- Radix-2 fast Fourier transform, 213
- Radix-4 fast Fourier transform, 221
- Reconstruction filter, 39, 75
- sine wave generation
 - using difference equation, 203
 - using lookup table, 14, 65
 - using `sin()` function call, 69
 - with DIP switch control, 70
- Sinusoidal noise cancellation, adaptive filter
 - for, 291
- Spectral leakage, 253
- Square wave generation, 75
- Steepest descent algorithm, 285
- Support files, 31
 - `L138_aic3106_init.c`, 18, 31
 - `L138_aic3106_init.h`, 18, 32
 - `linker_dsp.cmd`, 18, 34
 - `vectors_intr.asm`, 18, 32
 - `vectors_poll.asm`, 32
- SWI (DSP/BIOS Software interrupt thread), 308, 320
- system identification, 281
 - of codec antialiasing and reconstruction filters, 89
 - of FIR filter, 296
 - of IIR filter, 301
 - of moving average filter, 131
- Target configuration file, 10
- TSK (DSP/BIOS Task thread), 308, 322
- Threads in DSP/BIOS, 307
- Twiddle factors, 213, 229
- `vectors_intr.asm` file, 18, 32
- `vectors_poll.asm` file, 32
- Very-long-instruction-word (VLIW)
 - architecture, 6
- Voice recording, using external memory, 101
- Voice scrambling, using filtering and modulation, 150
- Watch window, monitoring, 28
- Window functions, 114
 - Blackman, 115
 - Kaiser, 115
 - Hamming, 114
 - Hanning, 114
 - rectangular, 114
- XDS100, 7
- z -transform (ZT), 105
- zero padding, 261