DIGITAL SIGNAL PROCESSING

Using the ARM® Cortex®-M4

Donald S. Reay

WILEY

Table of Contents

List of Illustrations

List of Tables

DIGITAL SIGNAL PROCESSING USING THE ARM® CORTEX®-M4

DONALD S. REAY
Heriot-Watt University

WILEY

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

Published simultaneously in Canada

Library of Congress Cataloging-in-Publication Data:

Reay, Donald (Donald S.), author.

Digital signal processing using the ARM Cortex-M4 / Donald Reay.

pages cm

Includes bibliographical references and index.

ISBN 978-1-118-85904-9 (pbk.)

1. Signal processing–Digital techniques. 2. ARM microprocessors. I. Title.

TK5102.9.R4326 2015

621.382′2–dc23

2015024771

To Reiko

Preface

This book continues the series started in 1990 by Rulph Chassaing and Darrell Horning's *Digital Signal Processing with the TMS320C25*, which tracked the development of successive generations of digital signal processors by Texas Instruments. More specifically, each book in the series up until now has complemented a different inexpensive DSP development kit promoted for teaching purposes by the Texas Instruments University Program. 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 an electrical engineering laboratory setting.

It was Rulph Chassaing's belief, and this author continues to believe, 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, finite impulse response (FIR) and infinite impulse response (IIR) filtering, the Fourier transform, and adaptive filtering, have changed little. Every academic year brings another cohort of students wanting to study this material. However, each book has featured a different DSP development kit.

In 2013, Robert Owen suggested to me that hands-on DSP teaching could be implemented using an inexpensive ARM® Cortex-M4® microcontroller. I pointed out that a Texas Instruments C674x processor was very significantly more computationally powerful than an ARM Cortex-M4. But I also went ahead and purchased a Texas Instruments Stellaris LaunchPad. I constructed an audio interface using a Wolfson WM8731 codec and successfully ported the program examples from my previous book to that hardware platform.

This book is aimed at senior undergraduate and postgraduate electrical engineering students who have some knowledge of C programming and linear systems theory, but it is intended, and hoped, that it may serve as 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 for first making me aware of the ARM Cortex-M4; to Khaled Benkrid at the ARM University Program and to the Royal Academy of Engineering for making possible a six-month Industrial Secondment to ARM during which teaching materials for the STM32f01 platform were developed; to Gordon McLeod and Scott Hendry at Wolfson Microelectronics for their help in getting the Wolfson Pi audio card to work with the STM32f01 Discovery; to Sean Hong, Karthik Shivashankar, and Robert Iannello at ARM for all their help; to Joan Teixidor Buixeda for helping to debug the program examples; to Cathy Wicks at the TI University Program and Hieu Duong at CircuitCo for developing the audio booster pack; and to Kari Capone and Brett Kurzman at Wiley for their patience. But above all, I thank Rulph Chassaing for inspiring me to get involved in teaching hands-on DSP.

Donald S. Reay

Edinburgh

2015

Chapter 1
ARM® CORTEX®-M4 Development Systems

1.1 Introduction

Traditionally, real-time digital signal processing (DSP) has been implemented using specialized and relatively expensive hardware, for example, digital signal processors or field-programmable gate arrays (FPGAs). The ARM® Cortex®-M4 processor makes it possible to process audio in real time (for teaching purposes, at least) using significantly less expensive, and simpler, microcontrollers.

The ARM Cortex-M4 is a 32-bit microcontroller. Essentially, it is an ARM Cortex-M3 microcontroller that has been enhanced by the addition of DSP and single instruction multiple data (SIMD) instructions and (optionally) a hardware floating-point unit (FPU). Although its computational power is a fraction of that of a floating-point digital signal processor, for example, the Texas Instruments C674x, it is quite capable of implementing DSP algorithms, for example, FIR and IIR filters and fast Fourier transforms for audio signals in real-time.

A number of semiconductor manufacturers have developed microcontrollers that are based on the ARM Cortex-M4 processor and that incorporate proprietary peripheral interfaces and other IP blocks. Many of these semiconductor manufacturers make available very-low-cost evaluation boards for their ARM Cortex-M4 microcontrollers. Implementing real-time audio frequency example programs on these platforms, rather than on more conventional DSP development kits, constitutes a reduction of an order of magnitude in the hardware cost of implementing hands-on DSP teaching. For the first time, students might realistically be expected to own a hardware platform that is useful not only for general microcontroller/microprocessor programming and interfacing activities but also for implementation of real-time DSP.

1.1.1 Audio Interfaces

At the time that the program examples presented in this book were being developed, there were no commercially available low-cost ARM Cortex-M4 development boards that incorporated high-quality audio input and output. The STMicroelectronics STM32F407 Discovery board features a high-quality audio digital-to-analog converter (DAC) but not a correspondinganalog-to-digital converter (ADC). Many ARM Cortex-M4 devices, including both the STMicroelectronics STM32F407 and the Texas Instruments TM4C123, feature multichannel instrumentation-quality ADCs. But without additional external circuitry, these are not suitable for the applications discussed in this book.

The examples in this book require the addition (to an inexpensive ARM Cortex-M4 development board) of an (inexpensive) audio interface.

In the case of the STMicroelectronics STM32F407 Discovery board and of the Texas Instruments TM4C123 LaunchPad, compatible and inexpensive audio interfaces are provided by the Wolfson Pi audio card and the CircuitCo audio booster pack, respectively. The low-level interfacing details and the precise performance characteristics and extra features of the two audio interfaces are subtly different. However, each facilitates the input and output of high-quality audio signals to and from an ARM Cortex-M4 processor on which DSP algorithms may be implemented.

Almost all of the program examples presented in the subsequent chapters of this book are provided, in only very slightly different form, for both the STM32F407 Discovery and the TM4C123 LaunchPad, on the partner website http://www.wiley.com/go/Reay/ARMcortexM4.

However, in most cases, program examples are described in detail, and program listings are presented, only for one or other hardware platform. Notable exceptions are that, in Chapter 2, low-level i/o mechanisms (implemented slightly differently in the two devices) are described in detail for both hardware platforms and that a handful of example programs use features unique to one or other processor/audio interface.

This book does not describe the internal architecture or features of the ARM Cortex-M4 processor in detail. An excellent text on that subject, including details of its DSP-related capabilities, is *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors* by Yiu [1].

1.1.2 Texas Instruments TM4C123 LaunchPad and STM32F407 Discovery Development Kits

The Texas Instruments and STMicroelectronics ARM Cortex-M4 processor boards used in this book are shown in Figures 1.1 and 1.2. The program examples presented in this book assume the use of the *Keil MDK-ARM* development environment, which is compatible with both development kits. An alternative development environment, Texas Instruments' *Code Composer Studio*, is available for the TM4C123 LaunchPad and the program examples have been tested using this. Versions of the program examples compatible with *Code Composer Studio version 6* are provided on the partner website http://www.wiley.com/go/Reay/ARMcortexM4.

Figure 1.1 Texas Instruments TM4C123 LaunchPad.

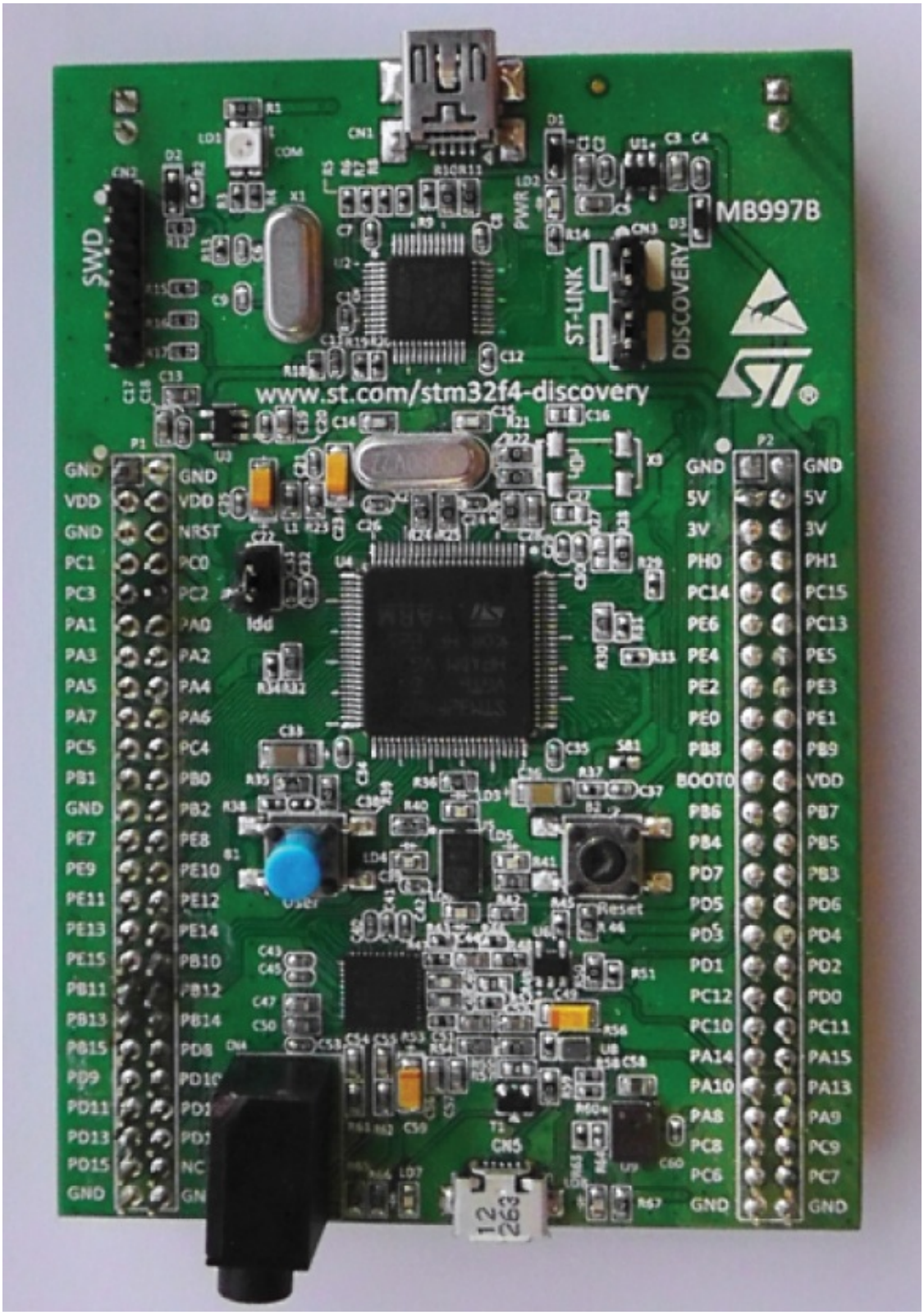


Figure 1.2 STMicroelectronics STM32F407 Discovery.

The CircuitCo audio booster pack (for the TM4C123 LaunchPad) and the Wolfson Pi audio card (for the STM32F407 Discovery) are shown in Figures1.3 and 1.4. The audio booster pack and the launchpad plug together, whereas the Wolfson audio card, which was designed for use with a Raspberry Pi computer, must be connected to the Discovery using a custom ribbon cable (available from distributor Farnell).

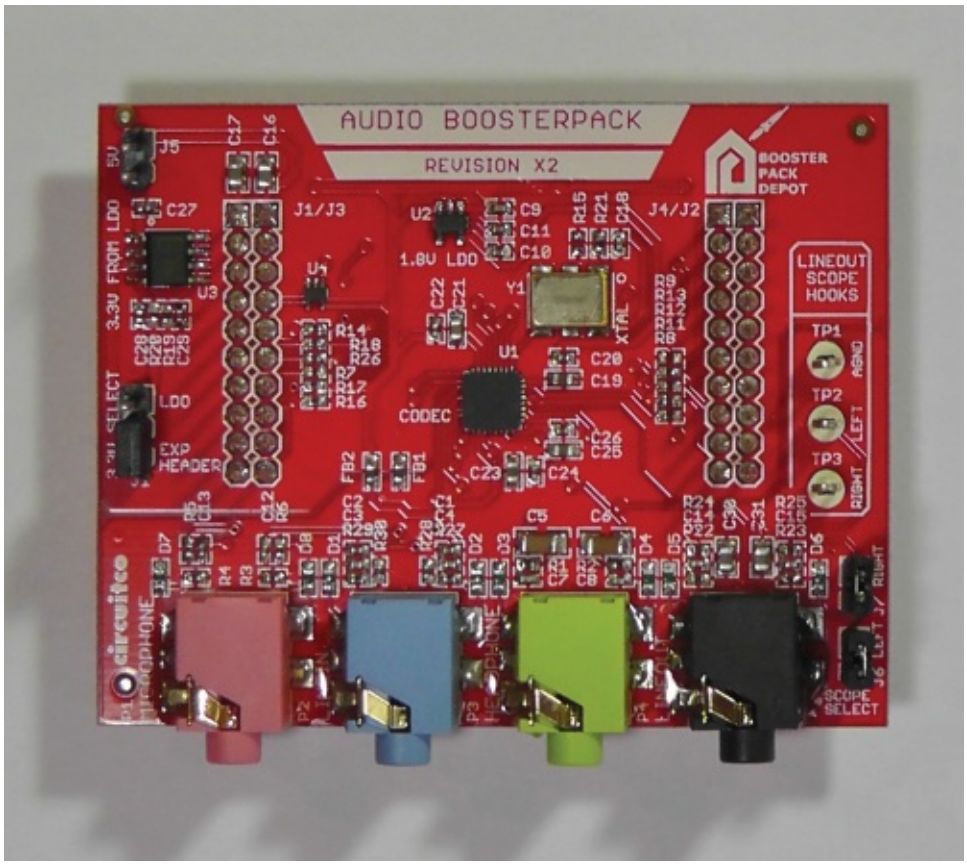


Figure 1.3 AIC3104 audio booster pack.

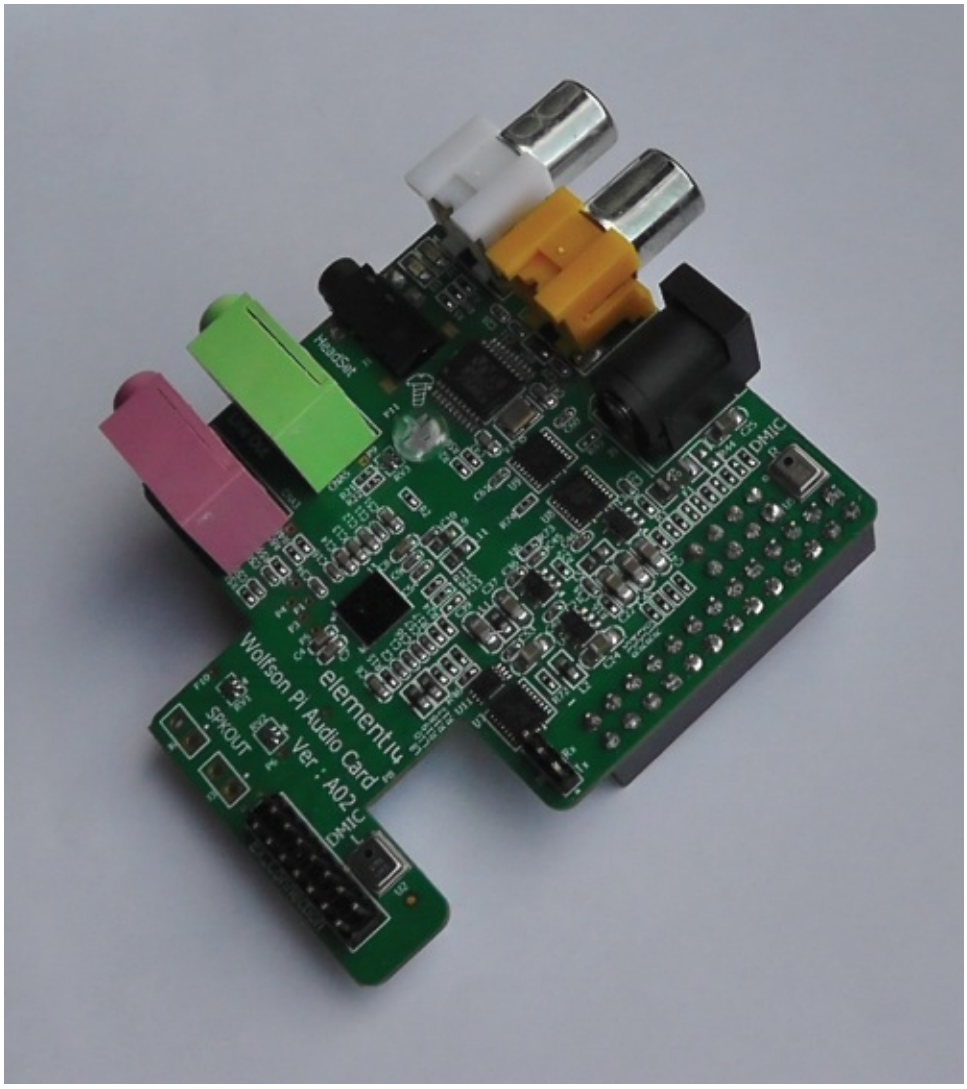


Figure 1.4 Wolfson Pi audio card.

Rather than presenting detailed instructions here that may be obsolete as soon as the next version of *MDK-ARM* is released, the reader is directed to the "getting started" guide at the partner website http://www.wiley.com/go/Reay/ARMcortexM4. and before progressing to the next chapter of this book will need to install *MDK-ARM*, including the "packs" appropriate to the hardware platform being used and including the CMSIS DSP library, download the program examples from the website, and become familiar with how to open a project in *MDK-ARM*, add and remove files from a project, build a project, start and stop a debug session, and run and halt a program running on the ARM Cortex-M4 processor.

Some of the example programs implement DSP algorithms straightforwardly, and with a view to transparency and understandability rather than computational efficiency or elegance. In several cases, ARM's CMSIS DSP library functions are used. These are available for both the STMicroelectronics and Texas Instruments processors as part of the *MDK-ARM* development environment. In appropriate circumstances, these library functions are particularly computationally efficient. This is useful in some of the program examples where the demands of running in real-time approach the limits of what is achievable with the ARM Cortex-M4. One difference between the two devices used in this book is that STM32F407 uses a processor

clock speed of 168 MHz, whereas the TM4C123 clock speed is 84 MHz. As presented in the book, all of the program examples will run in real time on either device. However, if the parameter values used are changed, for example, if the number of coefficients in an FIR filter is increased, it is likely that the limits of the slower device will be reached more readily than those of the faster one.

All of the program examples have been tested using the free, code size-limited, version of *MDK-ARM*. The aim of hands-on DSP teaching, and the intention of this book, is not to teach about the architecture of the ARM Cortex-M4. The device is used because it provides a capable and inexpensive platform. Nor is it the aim of hands-on DSP teaching, or the intention of this book, to teach about the use of *MDK-ARM*. The aim of hands-on DSP teaching is to reinforce DSP theory taught in lectures through the use of illustrative examples involving the real-time processing of audio signals in an electrical engineering laboratory environment. That is to say where test equipment such as oscilloscopes, signal generators, and connecting cables are available.

1.1.3 Hardware and Software Tools

To perform the experiments described in this book, a number of software and hardware resources are required.

1. An ARM Cortex-M4 development board and audio interface. Either aTexas Instruments TM4C123 LaunchPad and a CircuitCo audio booster pack or an STMicroelectronics STM32F407 Discovery board and a Wolfson Microelectronics Pi audio card are suitable hardware platforms.
2. A host PC running an integrated development environment (IDE) and with a spare USB connection. The program examples described in this book were developed and tested using the *Keil MDK-ARM* development environment. However, versions of the program examples for the TM4C123 LaunchPad and project files compatible with Texas Instruments *Code Composer Studio* IDE are provided on the partner website `http://www.wiley.com/go/Reay/ARMcortexM4`.
3. The TM4C123 LaunchPad and the STM32F407 Discovery board use slightly different USB cables to connect to the host PC. The launchpad is supplied with a USB cable, while the STM32F407 Discovery is not.
4. Whereas the audio booster pack and the launchpad plug together, the Wolfson Pi audio card does not plug onto the STM32F407 Discovery board. Connections between the two can be made using a custom ribbon cable, available from distributor Farnell.
5. An oscilloscope, a signal generator, a microphone, headphones, and various connecting cables. Several of these items will be found in almost any electrical engineering laboratory. If you are using the STM32F407 Discovery and Wolfson Pi audio card, then a microphone is unnecessary. The audio card has built-in digital MEMS microphones. The Wolfson Pi audio card is also compatible with combined microphone and headphone headsets (including those supplied with Apple and Samsung smartphones). Stereo 3.5 mm

jack plug to 3.5 mm jack plug cables and stereo 3.5 mm jack plug to (two) RCA (phono) plugs and RCA to BNC adapters are the specific cables required.

6. Project and example program files from the partner website http://www.wiley.com/go/Reay/ARMcortexM4.

Reference

1 .Yiu, J., "*The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*", Third Edition, Elsevier Inc., 2014.

Chapter 2
Analog Input and Output

2.1 Introduction

A basic DSP system, suitable for processing audio frequency signals, comprises a digital signal processor (DSP) and analog interfaces as shown in Figure 2.1. The Texas Instruments TM4C123 LaunchPad and audio booster pack provide such a system, using a TM4C123 ARM® Cortex®-M4 processor and a TLV320AIC3104 (AIC3104) codec [1]. The STMicro STM32F407 Discovery and the Wolfson audio card provide such a system, using an STM32407 ARM® Cortex®-M4 processor and a WM5102 codec [2]. The term codec refers to the coding of analog waveforms as digital signals and the decoding of digital signals as analog waveforms. The AIC3104 and WM5102 codecs perform both the analog-to-digital conversion (ADC) and digital-to-analog conversion (DAC) functions shown in Figure 2.1.

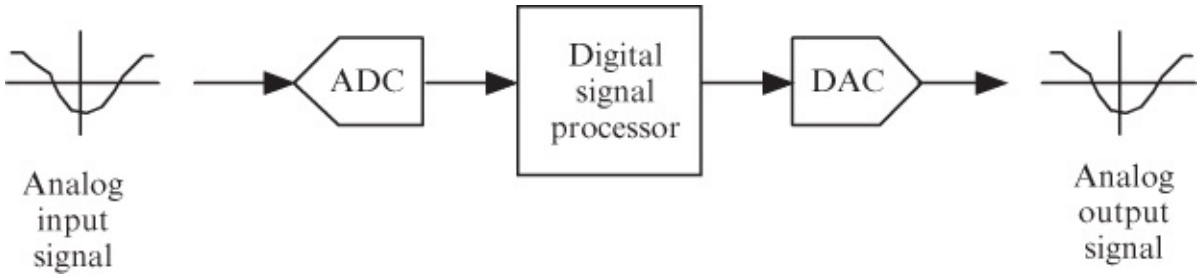


Figure 2.1 Basic digital signal processing system.

Both the AIC3104 and WM5102 codecs communicate with their associated processors (TM4C123 and STM32F407) using I2C bus for control (writing to the codec's control registers) and I2S for (audio) data transfer.

2.1.1 Sampling, Reconstruction, and Aliasing

Within DSPs, signals are represented as sequences of discrete sample values, 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 at frequencies below half the sampling frequency to pass but that attenuates greatly, or stops, frequency components at frequencies greater than or equal to 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 discrete sample values represented in digital hardware into a continuous analogue electrical signal. When viewed purely from a signal processing perspective, a DAC acts as a

reconstruction filter. Although they differ in a number of respects, both the AIC3104 and WM5102 codecs contain both digital and analog antialiasing and reconstruction filters and therefore do not require additional external filters.

2.2 TLV320AIC3104 (AIC3104) Stereo Codec for Audio Input and Output

The audio booster pack makes use of a TLV320AIC3104 (AIC3104) codec for analog input and output (see Figures 2.2 and 2.3). The AIC3104 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 (ADC), or coder, part of the codec converts an analog input signal into a sequence of (16-bit, 24-bit, or 32-bit signed integer) sample values to be processed by the DSP. The digital-to-analog converter (DAC), or decoder, part of the codec reconstructs an analog output signal from a sequence of (16-bit, 24-bit, or 32-bit signed integer)sample values that have been processed by the DSP and written to the DAC.

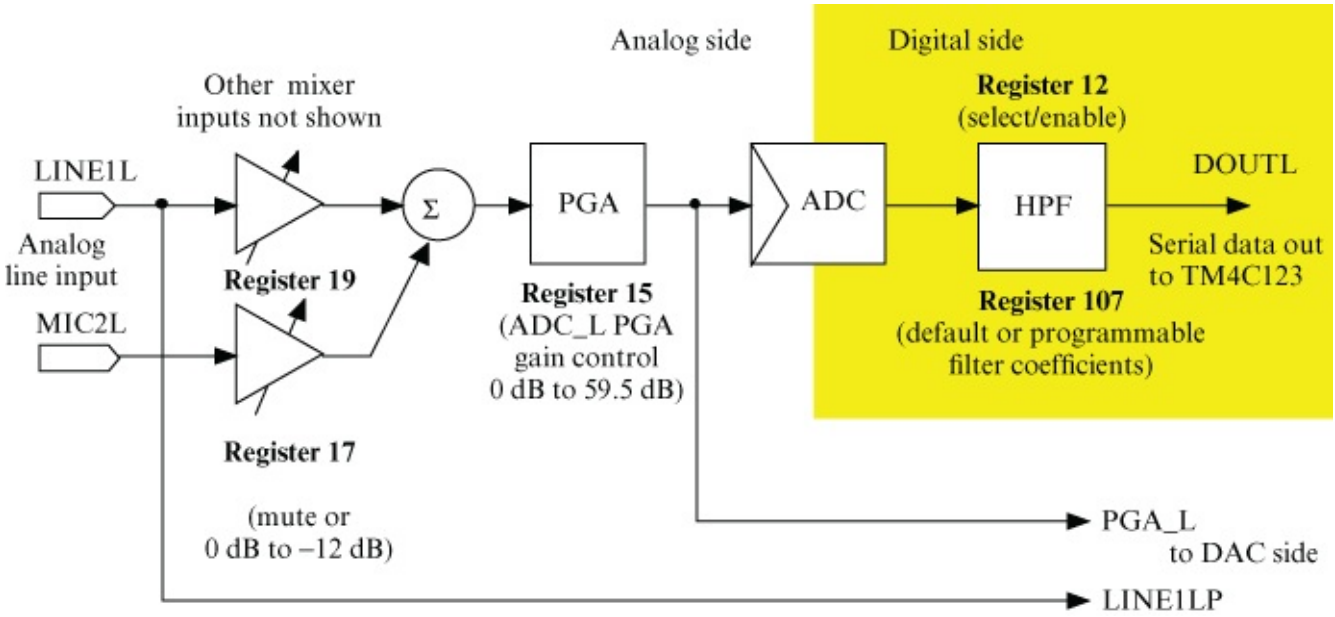


Figure 2.2 Simplified block diagram representation of input side of AIC3104 codec showing selected blocks and signal paths used by the example programs in this book (left channel only).

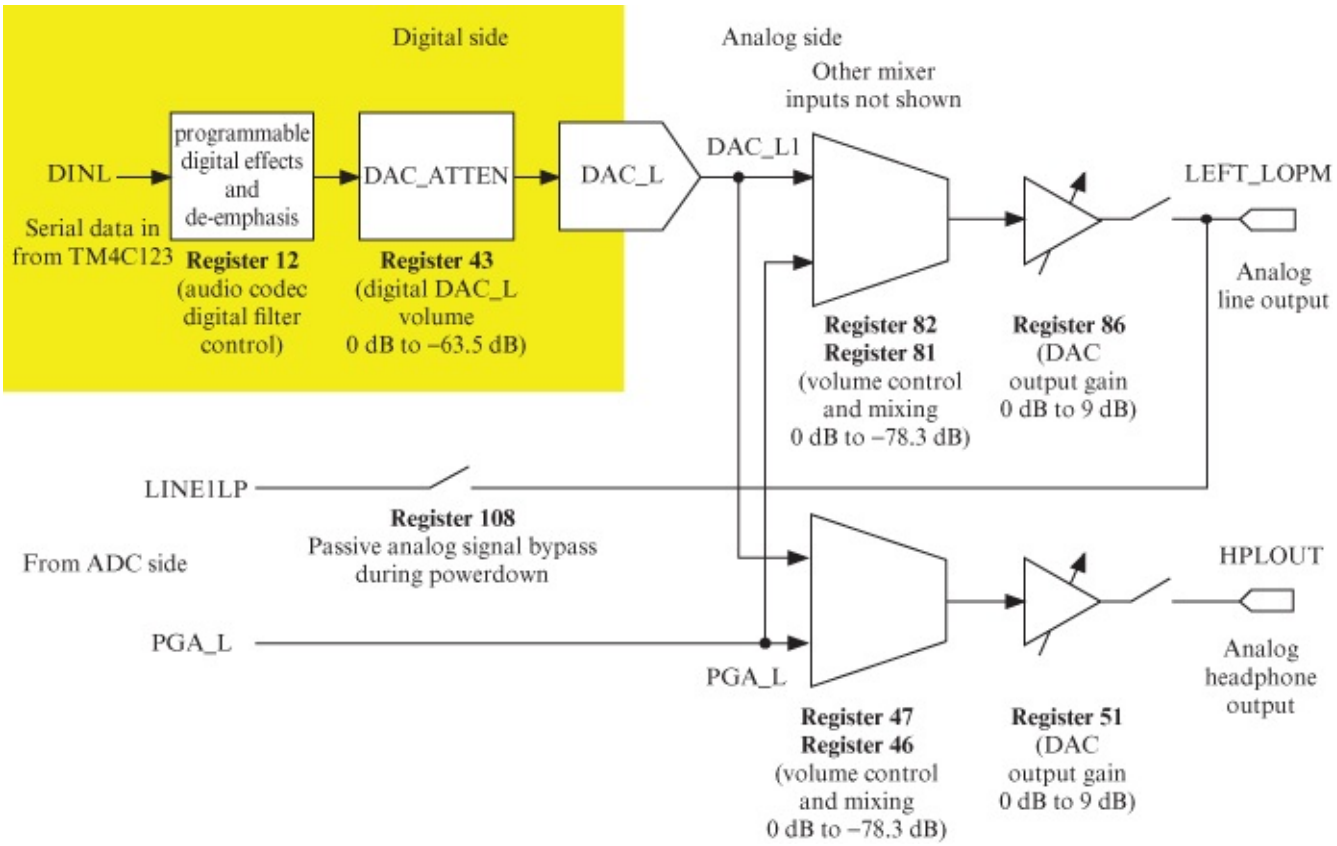
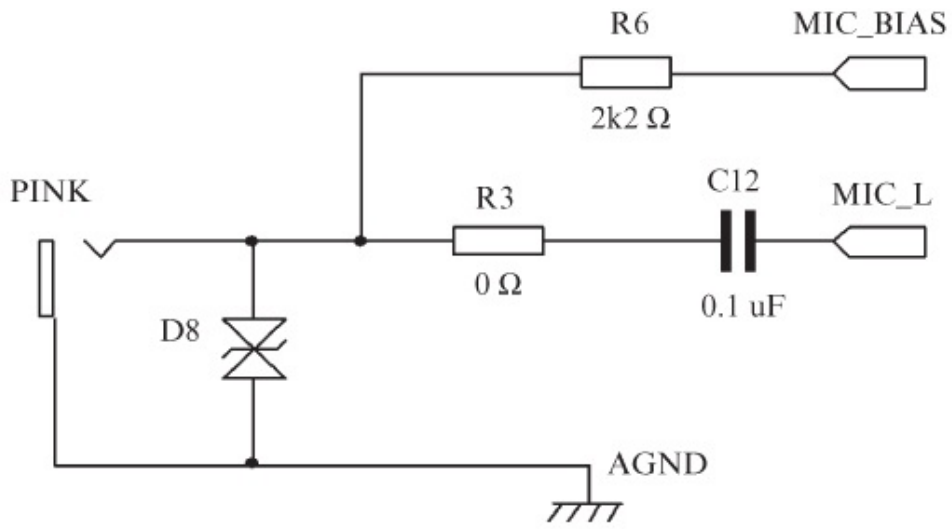


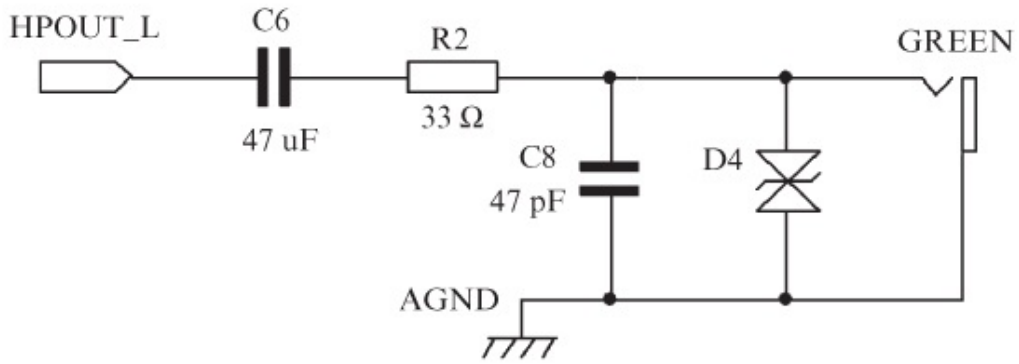
Figure 2.3 Simplified block diagram representation of output side of AIC3104 codec showing selected blocks and signal paths used by the example programs in this book (left channel only).

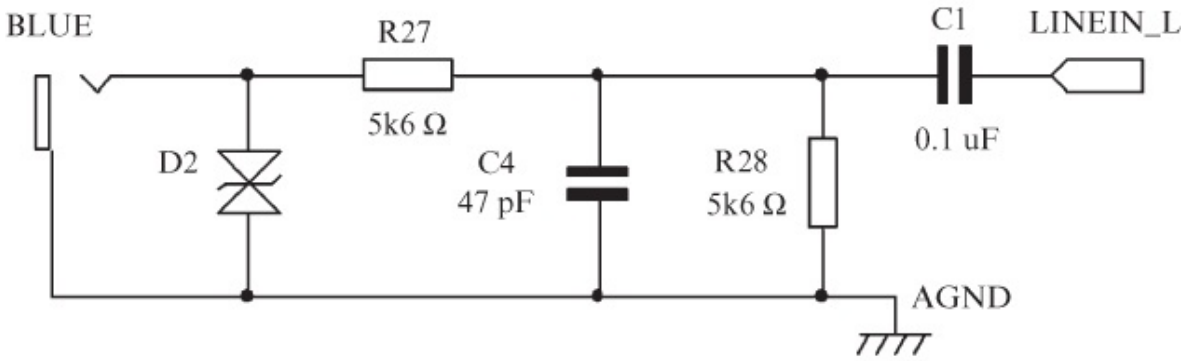
Also contained in the device are several programmable digital filters and gain blocks. The codec is configured using a number of control registers, offering so many options that it is beyond the scope of this text to describe them fully. However, choices of sampling frequency, input connection, and ADC PGA gain are made available in the example programs through the parameters passed to function `tm4c123_aic3104_init()`. In addition, it is possible to write to any of the codec control registers using function `I2CRegWrite()`.

Later in this chapter, examples of enabling some of the internal digital filter blocks by writing to the control registers of the AIC3104 are described. In Chapter 4, the characteristics of the programmable digital filters within the AIC3104 are examined in greater detail.

Data is passed to and from the AIC3104 via its I2S serial interface. MIC IN (pink), LINE IN (blue), LINE OUT (green), and HP OUT (black) connections are made available via four 3.5 mm jack sockets on the audio booster pack, and these are connected to the AIC3104 as shown in Figure 2.4. In addition, for reasons explained later in this chapter, jumpers J6 and J7 on the audio booster pack allow connection of first-order low-pass filters and scope hook test points TP2 and TP3 to LINE OUT on the AIC3104.


R6
MIC_BIAS
2k2 Ω
PINK
R3
C12
MIC_L
0 Ω
0.1 uF
D8
AGND


HPOUT_L
C6
R2
GREEN
47 uF
33 Ω
C8
47 pF
D4
AGND


BLUE
R27
C1
LINEIN_L
5k6 Ω
0.1 uF
D2
C4
47 pF
R28
5k6 Ω
AGND

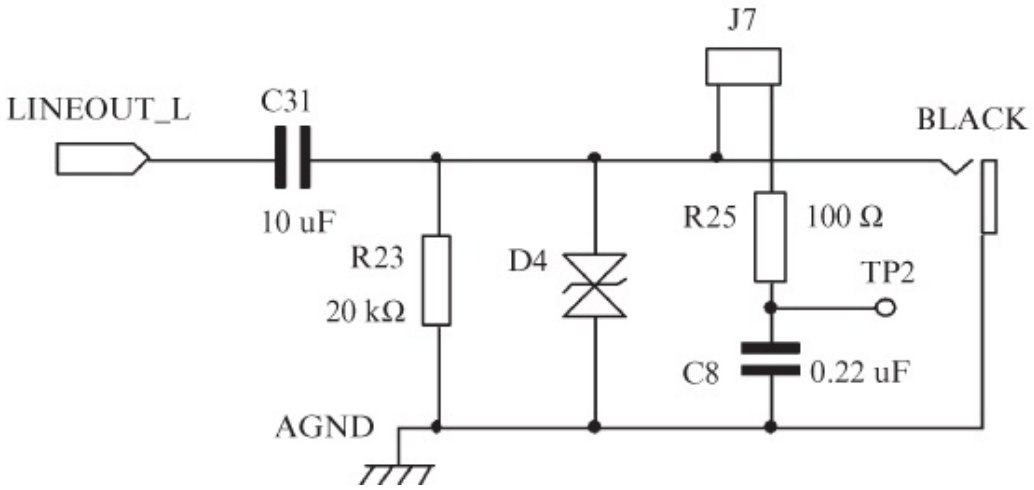

J7
LINEOUT_L
C31
BLACK
10 uF
R25
100 Ω
R23
20 kΩ
D4
TP2
C8
0.22 uF
AGND

Figure 2.4 Analog input and output connections on the AIC3104 audio booster pack.

2.3 WM5102 Audio Hub Codec for Audio Input and Output

The Wolfson audio card makes use of a WM5102 audio hub for analog input and output. The WM5102 features a low-power, high-performance audio codec.

Data is passed to and from the WM5102 via its I2S serial interface, and the device is configured by writing to its control registers via an I2C interface. In addition to a number of configurable filter and gain blocks, the WM5102 codec contains a programmable DSP. However, use of this proprietary DSP is beyond the scope of this book.

LINE IN (pink), LINE OUT (green), and combined MIC IN and HP OUT (black) connections are made available via three 3.5 mm jack sockets on the Wolfson audio card.

2.4 Programming Examples

The following examples illustrate analog input and output using either the TM4C123 LaunchPad and audio booster pack or the STM32F407 Discovery and Wolfson audio card. The program examples are available for either platform, although in most cases, only one platform is mentioned per example. A small number of example programs in this chapter concern programming the internal digital filters in the AIC3104 codec and are therefore applicable only to the Texas Instruments hardware platform. A small number of example programs concern use of the 12-bit DAC built in to the STM32F407 processor and are therefore applicable only to the STMicroelectronics hardware platform.

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 DMA-based i/o in order to implement real-time applications. Many of the concepts and techniques described in this chapter are revisited in subsequent chapters.

2.5 Real-Time Input and Output Using Polling, Interrupts, and Direct Memory Access (DMA)

Three basic forms of real-time i/o are demonstrated in the following examples. Polling- and interrupt-based i/o methods work on a sample-by-sample basis, and processing consists of executing a similar set of program statements at each sampling instant. 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. Processing consists of executing a similar set of program statements after each DMA transfer. 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 the use of the three different i/o mechanisms in order to implement a simple talk-through function. Throughout the rest of this book, use is made primarily of interrupt- and DMA-based methods. Compared to polling- and interrupt-based methods, there is a greater time delay between a signal entering the digital signal processing system and leaving it introduced by the DMA-based method. It is possible to make use of the DMA mechanism with a frame size of just one sample, but this rather defeats the purpose of using DMA-based i/o.

Example 2.1 Basic Input and Output Using Polling (`tm4c123_loop_poll.c`).

Listing 2.1 Program `tm4c123_loop_poll.c`

```
// tm4c123_loop_poll.c
#include “tm4c123_aic3104_init.h”
void SSI_interrupt_routine(void){while(1){}}
int main(void)
{
  AIC3104_data_type sample_data;
  float32_t input_left, input_right;
  tm4c123_aic3104_init(FS_48000_HZ,
                       AIC3104_MIC_IN,
                       IO_METHOD_POLL,
                       PGA_GAIN_6_DB);
  while(1)
  {
    SSIDataGet(SSI1_BASE,&sample_data.bit32);
    input_left = (float32_t)(sample_data.bit16[0]);
    SSIDataGet(SSI0_BASE,&sample_data.bit32);
    input_right = (float32_t)(sample_data.bit16[0]);
    sample_data.bit32 = ((int16_t)(input_left));
    SSIDataPut(SSI1_BASE,sample_data.bit32);
    sample_data.bit32 = ((int16_t)(input_right));
    SSIDataPut(SSI0_BASE,sample_data.bit32);
  }
}
```

The C language source file for program, `tm4c123_loop_poll.c`, which simply copies input samples read from the AIC3104 codec ADC to the AIC3104 codec DAC as output samples, is shown in Listing 2.1. Effectively, the MIC IN input socket is connected straight through to the LINE OUT and HP OUT output sockets on the audio booster pack via the AIC3104 codec and the TM4C123 processor. Function `tm4c123_aic3104_init()`, called by program `tm4c123_loop_poll.c`, is defined in support file `tm4c123_aic3104_init.c`. In this way, the

C source file tm4c123_loop_poll.c is kept as short as possible and potentially distracting low-level detail is hidden. The implementation details of function tm4c123_aic3104_init() and other functions defined in tm4c123_aic3104_init.c need not be studied in detail in order to use the examples presented in this book.

2.5.1 I2S Emulation on the TM4C123

The TM4C123 processor does not feature an I2S interface. Instead, two synchronous serial interface (SSI) interfaces, SSI0 and SSI1, are used to emulate a bidirectional stereo I2S interface and to pass audio data to and from the AIC3104 codec. One SSI interface handles the left channel and the other handles the right channel. Details of the I2S emulation are described in application note SPMA042 [3].

2.5.2 Program Operation

Following the call to function tm4c123_aic3104_init(), program tm4c123_loop_poll.c enters an endless while loop and repeatedly copies left and right channel input sample values into variables input_left and input_right, using function SSIDataGet(), before writing these sample values to the AIC3104 DAC, using function SSIDataPut(). Function SSIDataGet() waits until there is data in the receive FIFO of the specified SSI peripheral, SSI0_BASE or SSI1_BASE, and function SSIDataPut() waits until there is space available in the transmit FIFO of the specified SSI peripheral. In this way, the real-time operationof the program is controlled by the timing of the I2S interface, which, in turn, is determined by the AIC3104 codec (acting as I2S master). Functions SSIDataGet() and SSIDataPut() are defined in the TM4C123 device family pack (DFP) installed as part of the *MDK-ARM* development environment. Although the AIC3104 is configured to use 16-bit sample values, function SSIDataGet() returns a 32-bit value and function SSIDataPut() is passed a 32-bit value.

Function SSI_interrupt_routine() is not used by program tm4c123_loop_poll.c but has been defined here as a trap for unexpected SSI peripheral interrupts.

In this simple example, it is not strictly necessary to convert the 16-bit sample values read from the AIC3104 ADC by function SSIDataGet() into 32-bit floating-point values. However, in subsequent program examples, DSP algorithms are implemented using floating-point arithmetic and input sample values are converted into type float32_t. Processing of the floating-point sample values could be implemented by adding program statements between

```
input_right = (float32_t)(sample_data.bit16[0]);
```

and

```
sample_data.bit32 = ((int16_t)(input_left));
```

2.5.3 Running the Program

Connect a microphone to the (pink) MIC IN socket on the audio booster card and headphones

to the (green) HP OUT socket. Run the program and verify that the input to the microphone can be heard in the headphones.

2.5.4 Changing the Input Connection to LINE IN

Change the program statement

```
tm4c123_aic3104_init(FS_48000_HZ,
                     AIC3104_MIC_IN,
                     IO_METHOD_POLL,
                     PGA_GAIN_6_DB);
```

to read

```
tm4c123_aic3104_init(FS_48000_HZ,
                     AIC3104_LINE_IN,
                     IO_METHOD_POLL,
                     PGA_GAIN_6_DB);
```

Rebuild the project and run the program again using a signal from a sound card, a signal generator, or an MP3 player connected to the (blue) LINE IN socket as input.

2.5.5 Changing the Sampling Frequency

Change the sampling frequency used by passing parameter value `FS_8000_HZ` rather than `FS_48000_HZ` to the codec initialization function, that is, by changing the program statement

```
tm4c123_aic3104_init(FS_48000_HZ,
                     AIC3104_LINE_IN,
                     IO_METHOD_POLL,
                     PGA_GAIN_6_DB);
```

to read

```
tm4c123_aic3104_init(FS_8000_HZ,
                     AIC3104_LINE_IN,
                     IO_METHOD_POLL,
                     PGA_GAIN_6_DB);
```

Rebuild the project and run the program again. Signals passed through the system should sound less bright than previously due to the lower sampling rate and correspondingly reduced system bandwidth.

Valid parameter values (constants) that may be passed to function `tm4c123_AIC3104_init()` are

```
FS_48000_HZ
FS_44100_HZ
FS_32000_HZ
FS_24000_HZ
FS_22050_HZ
FS_16000_HZ
```

```
FS_11025_HZ
FS_8000_HZ
```

which set the sampling rate,

```
IO_METHOD_POLL
IO_METHOD_INTR
IO_METHOD_DMA
```

which set the i/o method,

```
AIC3104_MIC_IN
AIC3104_LINE_IN
```

which set the input connection used, and

```
PGA_GAIN_0_DB
PGA_GAIN_1_DB
PGA_GAIN_2_DB
PGA_GAIN_3_DB
PGA_GAIN_4_DB
PGA_GAIN_5_DB
PGA_GAIN_6_DB
PGA_GAIN_7_DB
PGA_GAIN_8_DB
PGA_GAIN_9_DB
PGA_GAIN_10_DB
PGA_GAIN_11_DB
PGA_GAIN_12_DB
```

which set the gain of the PGA that precedes the ADC (shown in Figure 2.2). Parameter value `PGA_GAIN_6_DB` is used by default in order to compensate for the potential divider circuits between the LINE IN socket on the audio booster pack and LINEIN_L and LINEIN_R on the AIC3104 (as shown in Figure 2.4).

Example 2.2

Basic Input and Output Using Polling (`stm32f4_loop_poll.c`).

Program `stm32f4_loop_poll.c`, shown in Listing 2.4, is functionally equivalent to program `tm4c123_loop_poll.c` but runs on the STM32F407 Discovery. Full duplex I2S communication between the WM5102 codec and the STM32F407 processor is implemented on the STM32F407 using two I2S instances SPI/I2S2 and I2S2_ext. SPI/I2S2 is configured as a receiver and I2S2_ext as a transmitter. The WM5102 codec, which operates in master mode, generates the I2S word and bit clock signals (WCLK and BCLK), and the STM32F407 I2S peripheral operates in slave mode.

Following the call to function `stm32f4_wm5102_init()`, program `stm32f4_loop_poll.c` enters an endless while loop.

Status flag SPI_I2S_FLAG_RXNE in SPI/I2S2 is repeatedly tested using function SPI_I2S_GetFlagStatus() until it is set, indicating that the SPI/I2S2 receive buffer is not empty. Then, status flag I2S_FLAG_CHSIDE is tested. This indicates whether the data received corresponds to the left or the right channel.

If the value of I2S_FLAG_CHSIDE indicates that a left channel sample value has been received, function SPI_I2S_ReceiveData() is used to read that sample from SPI/I2S2 into the int16_tvariable input_left, and after waiting for status flag SPI_I2S_FLAG_TXE to be set, the value of variable input_left is written to I2S2_ext.

If the value of I2S_FLAG_CHSIDE indicates that a right channel sample value has been received, function SPI_I2S_ReceiveData() is used to read that sample from SPI/I2S2 into the int16_t variable input_right, and after waiting for status flag SPI_I2S_FLAG_TXE to be set, the value of variable input_right is written to I2S2_ext.

The endless while loop then returns to testing status flag SPI_I2S_FLAG_RXNE in SPI/I2S2. In this way, the real-time operation of the program is controlled by the timing of the I2S interface, which, in turn, is determined by the WM5102 codec (acting as I2S master).

In this simple talk-through example, it is not strictly necessary to test whether received samples correspond to the left or right channel. However, the program has been written so that processing of the signals on either or both channels could easily be added between, for example, program statements.

```
left_in_sample = SPI_I2S_ReceiveData(I2Sx);
```

and

```
while(SPI_I2S_GetFlagStatus(I2Sxext,
```

Listing 2.2 Program `stm32f4_loop_poll.c`

```
// stm32f4_loop_poll.c
#include "stm32f4_wm5102_init.h"
int main(void)
{
  int16_t left_out_sample, right_out_sample;
  int16_t left_in_sample, right_in_sample;
  stm32_wm5102_init(FS_48000_HZ,
             WM5102_DMIC_IN,
             IO_METHOD_POLL);
  while(1)
  {
    while(SPI_I2S_GetFlagStatus(I2Sx,
          SPI_I2S_FLAG_RXNE ) != SET){}
    if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ⌋ SET)
    {
      left_in_sample = SPI_I2S_ReceiveData(I2Sx);
      left_out_sample = left_in_sample;
      while(SPI_I2S_GetFlagStatus(I2Sxext,
            SPI_I2S_FLAG_TXE ) != SET){}
      SPI_I2S_SendData(I2Sxext, left_out_sample);
    }
    else
    {
      right_in_sample = SPI_I2S_ReceiveData(I2Sx);
      right_out_sample = right_in_sample;
      while(SPI_I2S_GetFlagStatus(I2Sxext,
            SPI_I2S_FLAG_TXE ) != SET){}
      SPI_I2S_SendData(I2Sxext, right_out_sample);
    }
  }
}
```

Valid parameter values (constants) that may be passed to function `stm32_wm5102_init()` are

```
FS_48000_HZ
FS_44100_HZ
FS_32000_HZ
FS_24000_HZ
FS_22050_HZ
FS_16000_HZ
FS_11025_HZ
FS_8000_HZ
```

which set the sampling rate,

```
IO_METHOD_POLL
IO_METHOD_INTR
IO_METHOD_DMA
```

which set the i/o method, and

```
WM5102_MIC_IN
WM5102_DMIC_IN
WM5102_LINE_IN
```

which set the input connection used.

2.5.6 Using the Digital MEMS Microphone on the Wolfson Audio Card

Unlike the audio booster pack for the TM4C123 LaunchPad, which has separate MIC IN and HP OUT sockets, the Wolfson audio card has a single HEADSET socket (shown in Figure 2.5) that may be used for a combined microphone and earphone headset, conventional stereo headphones, an electret microphone that uses a four-pole (TRRS) 3.5 mm jack plug. In addition, the Wolfson audio card features high-quality stereo digital MEMS microphones that may be selected by passing parameter value `WM5102_DMIC_IN` to function `stm32_wm5102_init()`.

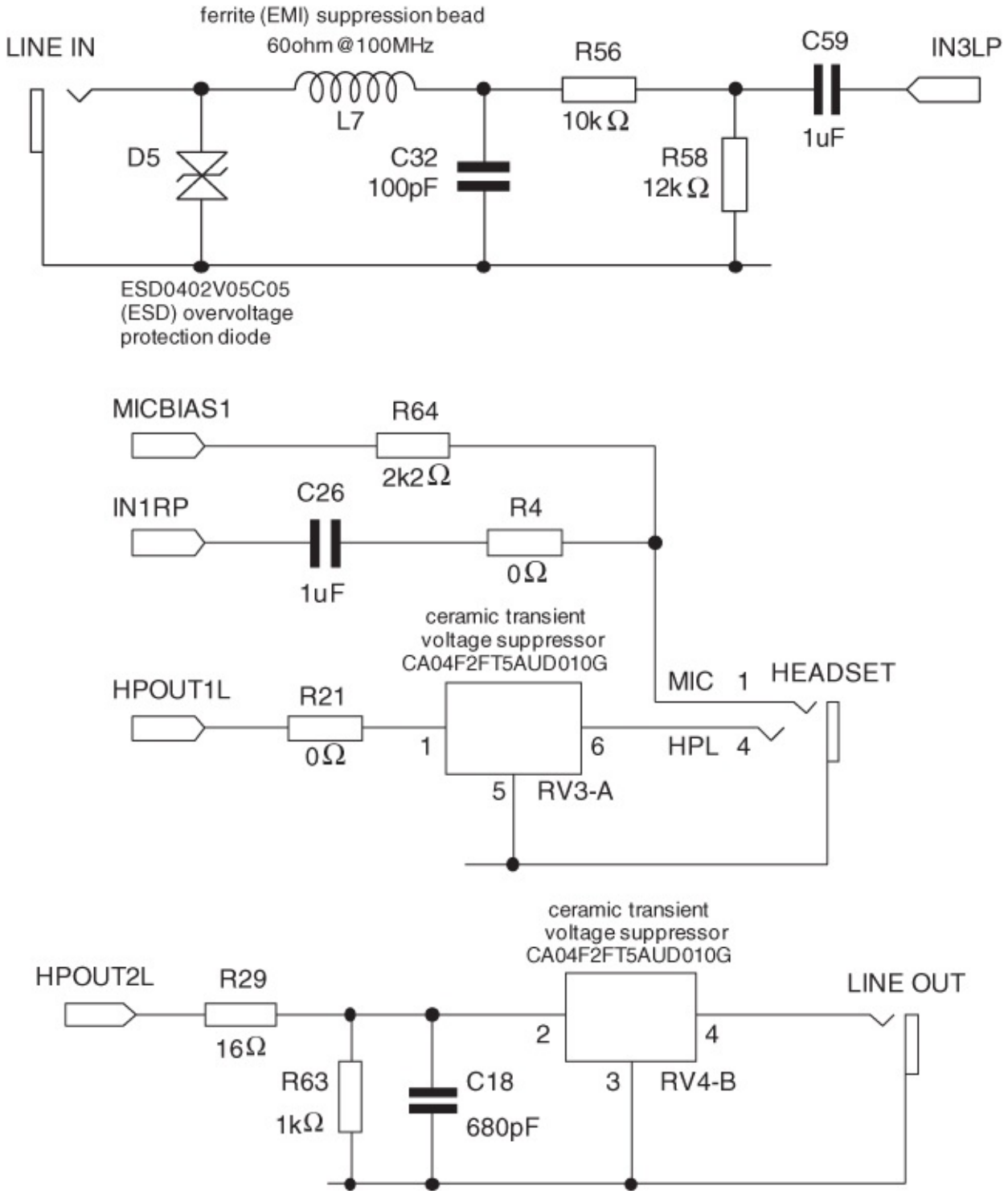


Figure 2.5 Analog input and output connections on the Wolfson audio card.

2.5.7 Running the Program

As provided, program `stm32f4_loop_poll.c` accepts input from the digital MEMS microphones on the audio card and routes this to the (green) LINE OUT and (black) HEADSET sockets. If you want to use a sound card, signal generator, or MP3 player to supply a line-level input signal to the (pink) LINE IN socket, change program statement.

```
stm32f4_wm5102_init(FS_48000_HZ,
```

```
                    WM5102_DMIC_IN,
                    IO_METHOD_INTR);
```

to read

```
stm32f4_wm5102_init(FS_48000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
```

Polling-based i/o is not computationally efficient and is used in very few of the example programs in this book.

Example 2.3

Basic Input and Output Using Interrupts (`tm4c123_loop_intr.c`).

Viewed in terms of analog input and output signals, program `tm4c123_loop_intr.c`, shown in Listing 2.6, is functionally equivalent to program `tm4c123_loop_poll.c` but uses interrupt-based i/o.

Listing 2.3 Program `tm4c123_loop_intr.c`

```
// tm4c123_loop_intr.c
#include “tm4c123_aic3104_init.h”
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  float32_t input_left, input_right;
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  input_left = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  input_right = (float32_t)(sample_data.bit16[0]);
  sample_data.bit32 = ((int16_t)(input_left));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  sample_data.bit32 = ((int16_t)(input_right));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
}
int main(void)
{
  tm4c123_aic3104_init(FS_48000_HZ,
                       AIC3104_MIC_IN,
                       IO_METHOD_INTR,
                PGA_GAIN_6_DB);
  while(1){}
}
```

This simple program is important because many of the example programs in this book use interrupt-based i/o and are structured similarly. Instead of simply copying the sequence of sample values read from the ADC to the DAC, 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 program statements.

```
input_right = (float32_t)(sample_dat.bit16[0]);
```

and

```
sample_dat.bit16[0] = (int16_t)(input_left);
```

For this reason, it is worth taking time to ensure that you understand how program `tm4c123_loop_intr.c` works.

Strictly speaking, it is not good practice to carry out digital signal processing operations within a hardware interrupt service routine, for example, function `SSI_interrupt_routine()`. However, in the program examples in this book, there are no other tasks being carried out by the processor, and, in most cases, the algorithms being demonstrated have been placed within the interrupt service routine function(s).

In function `main()`, parameter value `IO_METHOD_INTR` is passed to initialization function `tm4c123_aic3104_init()`. This selects the use, by the program, of interrupt-based i/o.

```
tm4c123_aic3104_init(FS_48000_HZ,
                     AIC3104_MIC_IN,
                     IO_METHOD_INTR,
                     PGA_GAIN_6_DB);
```

Following initialization, function `main()` enters an endless, and empty, while loop, effectively doing nothing but waiting for interrupts.

SSI0 receive FIFO interrupts (`SSI_RXFF`), which occur at the sampling rate (48 kHz), are handled by interrupt service routine function `SSI_interrupt_routine()`. In this function, left- and right-channel 16-bit sample values are read from the SSI1 and SSI0 receive FIFOs, respectively. These are converted into `float32_t` values `input_left` and `input_right`. Strictly speaking, type conversions are unnecessary in this simple talk-through program since the values of `input_left` and `input_right` are subsequently converted back to type `int16_t` and written to SSI1 and SSI0 transmit FIFOs, respectively.

2.5.8 Running the Program

As provided, program `tm4c123_loop_intr.c` accepts input from the (pink) MIC IN socket on the audio booster card and routes this to both the (black) LINE OUT and (green) HP OUT sockets. If you want to use a sound card, signal generator or MP3 player to supply a line-level input signal, change program statement.

```
tm4c123_aic3104_init(FS_48000_HZ,
                     AIC3104_MIC_IN,
                     IO_METHOD_INTR,
                     PGA_GAIN_6_DB);
```

to read

```
tm4c123_aic3104_init(FS_48000_HZ,
                     AIC3104_LINE_IN,
                     IO_METHOD_INTR,
                     PGA_GAIN_6_DB);
```

GPIO pin PE2 is set high at the start, and reset low near the end, of function `SSI_interrupt_routine()`. This signal is accessible via the J3 connector on the TM4C123 LaunchPad.

Example 2.4

Basic Input and Output Using Interrupts (`stm32f4_loop_intr.c`).

Program `stm32f4_loop_intr.c` is shown in Listing 2.8. In terms of analog input and output

signals, it is functionally equivalent to program `tm4c123_loop_intr.c` but is written for the STM32F407 Discovery and Wolfson audio card. Because the WM5012 and AIC3104 codecs differ and because, unlike the TM4C123 processor, the STM32F407 features an I2S interface, there are subtle differences between programs `stm32f4_loop_intr.c` and `tm4c123_loop_intr.c`.

Listing 2.4 Program `stm32f4_loop_intr.c`

```
// stm32f4_loop_intr.c
#include "stm32f4_wm5102_init.h"
void SPI2_IRQHandler()
{
  int16_t left_out_sample, right_out_sample;
  int16_t left_in_sample, right_in_sample;
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ) SET)
  {
    left_in_sample = SPI_I2S_ReceiveData(I2Sx);
    left_out_sample = left_in_sample;
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
  }
  else
  {
    right_in_sample = SPI_I2S_ReceiveData(I2Sx);
    right_out_sample = right_in_sample;
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
  GPIO_ToggleBits(GPIOD, GPIO_Pin_15);
 }
int main(void)
{
  stm32_wm5102_init(FS_48000_HZ,
                    WM5102_DMIC_IN,
                    IO_METHOD_INTR);
  while(1){}
}
```

Following a call to initialization function `stm32_wm5102_init()`, function `main()` enters an endless, and empty, while loop, effectively doing nothing but waiting for interrupts.

The I2S peripheral in the STM32F407 processor is configured for full-duplex slave mode, using PCM standard audio protocol, with 16-bit data packed into 32-bit frames. Receive buffer not empty (RXNE) interrupts are generated by I2S instance SPI/I2S2 when data is received from the WM5102 codec. These interrupts, which occur at the sampling rate (48 kHz) for *both* left- and right-channel samples, are handled by function `SPI2_IRQHandler()`. This tests the status flag `CHSIDE` in order to determine whether a left- or right-channel sample has been

received. If a left-channel sample has been received (`CHSIDE ⟩ 1`), then that value is read from SPI/I2S2 into the variable `input_left` by function `SPI_I2S_ReceiveData(I2Sx)`. It is then copied to variable `output_left` and written to I2S2_ext using function `SPI_I2S_SendData()`. Finally, GPIO pin PD15 is toggled. This pin is accessible via J2 on the STM32F407 Discovery and may be monitored using an oscilloscope. On the STM32F407 Discovery, GPIO pin PD15 drives a blue LED, and, hence, its duty cycle is discernible from the apparent brightness of the blue LED. In this program example, since interrupts occur twice per sampling period (once each for left and right channels), GPIO pin PD15 should output a 48-kHz square wave and the blue LED should emit light with medium intensity.

Example 2.5

Basic Input and Output Using DMA (`tm4c123_loop_dma.c`).

In terms of analog input and output signals, program `tm4c123_loop_dma.c`, shown in Listing 2.10, is functionally equivalent to the preceding program examples but makes use of direct memory access (DMA). 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, and is therefore more computationally efficient than polling- or interrupt-based i/o methods.

Listing 2.5 Program `tm4c123_loop_dma.c`

```
// tm4c123_loop_dma.c
#include "tm4c123_aic3104_init.h"
extern int16_t LpingIN[BUFSIZE], LpingOUT[BUFSIZE];
extern int16_t LpongIN[BUFSIZE], LpongOUT[BUFSIZE];
extern int16_t RpingIN[BUFSIZE], RpingOUT[BUFSIZE];
extern int16_t RpongIN[BUFSIZE], RpongOUT[BUFSIZE];
extern int16_t Lprocbuffer, Rprocbuffer;
extern volatile int16_t LTxcomplete, LRxcomplete;
extern volatile int16_t RTxcomplete, RRxcomplete;
void Lprocess_buffer(void)
{
  int16_t *inBuf, *outBuf;
  int16_t i;
  if (Lprocbuffer ⟩ PING)
  { inBuf = LpingIN; outBuf = LpingOUT; }
  if (Lprocbuffer ⟩ PONG)
  { inBuf = LpongIN; outBuf = LpongOUT; }
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    *outBuf++ = *inBuf++;
  }
  LTxcomplete = 0;
  LRxcomplete = 0;
```

```
  return;
}
void Rprocess_buffer(void)
{
  int16_t *inBuf, *outBuf;
  int16_t i;
  if (Rprocbuffer ⌋ PING)
  { inBuf = RpingIN; outBuf = RpingOUT; }
  if (Rprocbuffer ⌋ PONG)
  { inBuf = RpongIN; outBuf = RpongOUT; }
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    *outBuf++ = *inBuf++;
  }
  RTxcomplete = 0;
  RRxcomplete = 0;
  return;
}
void SSI_interrupt_routine(void){while(1){}}
int main(void)
{
  tm4c123_aic3104_init(FS_48000_HZ,
                       AIC3104_LINE_IN,
                        IO_METHOD_DMA,
                       PGA_GAIN_6_DB);
  while(1)
  {
    while((!RTxcomplete)|(!RRxcomplete));
    Rprocess_buffer();
    while((!LTxcomplete)|(!LRxcomplete));
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
    Lprocess_buffer();
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
 }
}}
```

2.5.9 DMA in the TM4C123 Processor

The TM4C123 DMA controller has 32 channels, each of which may be assigned to one of up to five different peripherals (each peripheral device in the TM4C123 processor is associated with a specific DMA controller channel). Each channel can operate in basic, ping-pong, or scatter-gather mode. DMA transfers are made up of up to 1024 × 8-, 16-, or 32-bit elements. An I2S interface is emulated on the TM4C123 using two separate SSI peripherals (SSI0 and SSI1), SSI1 for the left channel and SSI0 for the right channel. The SSI0 peripheral is set up to trigger on the positive edge of the I2S frame clock, WCLK (generated by the AIC3104 codec), and the SSI1 peripheral is set up to trigger on the negative edge of the I2S frame clock (strictly speaking, the positive edge of the inverted I2S frame clock). The AIC3104 codec on the audio booster pack acts as I2S master and supplies frame and bit clock signals (WCLK and BCLK).

DMA channel control is implemented using a table (aligned on a 1024-byte boundary in memory) of control structures. Associated with each channel are two control structures

(primary (PRI) and alternative (ALT)) and both of these are used in ping-pong mode.

Each control structure includes a source address pointer (SRC), a destination address pointer (DST), and a control word specifying SRC and DST data element sizes, SRC and DST address increments, the total number of elements to transfer, and the transfer mode (basic, ping-pong, or scatter-gather). The SRC and DST address pointers in the control table must be initialized before a DMA transfer starts, and according to the SRC and DST address increments, they are updated as the transfer progresses.

In ping-pong mode, as soon as one DMA transfer (PRI or ALT) has been completed, another transfer (ALT or PRI) on the same channel starts. When a DMA transfer is completed, an interrupt may be generated and this signals an opportunity, in ping-pong mode, not only to process the block of input data most recently transferred, but also to reinitialize the currently inactive (PRI or ALT) control structure.

In the example programs in this book, I2S protocol bidirectional stereo audio data transfer makes use of four unidirectional DMA channels, associated with SSI0TX, SSI0RX, SSI1TX, and SSI1RX.

Transfers specified by the SSI0TX primary (PRI) control structure (from array `RpingOUT` to SSI0 transmit FIFO) alternate with those specified by the SSI0TX alternative (ALT) control structure (from array `RpongOUT` to SSI0 transmit FIFO). Each time either transfer is completed, an interrupt is generated and handled by function `SSI0IntHandler()`. Following completion of an SSI0TX PRI DMA transfer (from array `RpingOUT` to SSI0), the SSI0 interrupt service routine must reinitialize the SSI0TX PRI control structure (principally by resetting SRC and DST addresses). At that point in time, the SSI0TX ALT DMA transfer (from array `RpongOUT` to SSI0) should be in progress.

SSI0RX DMA transfers (PRI from SSI0 receive FIFO to array `RpingIN` and ALT from SSI0 receive FIFO to array `RLpongIN`) take place in parallel with the SSI0TX transfers, and completion of either of these transfers generates interrupts also handled by `SSI0IntHandler()`.

Interrupt service routine `SSI0IntHandler()` must therefore determine which of four possible different DMA transfers has completed and reinitialize the corresponding control structure. This is summarized in Table 2.1. Function `SSI0IntHandler()` is shown in Listing 2.11. Its functions are

1. Determine which one of four possible DMA transfers has completed (and generated an SSI0IRQ interrupt).
2. Reinitialize the corresponding control structure resetting the SRC or DST (memory) address pointers.
3. Set the value of flag `RprocBuffer` to either `PING` or `PONG`.
4. If the completed transfer was from memory to the SSI0 transmit FIFO, set flag `RTxcomplete`.

5. If the completed transfer was from the SSI0 receive FIFO to memory, set flag `RRxcomplete`.

Table 2.1 Summary of DMA Control Structures Used and Flags Set in Interrupt Service Routines `SSI0IntHandler()` and `SSI1IntHandler()` in Program `tm4c123_loop_dma.c`

Channel	Option	SRC	DST	Flags Set
SSI0RX	PRI	SSI0RX	`RpingIN`	`RRxcomplete`
				`Rprocbuffer = PING`
SSI0RX	ALT	SSI0RX	`RpongIN`	`RRxcomplete`
				`Rprocbuffer = PONG`
SSI0TX	PRI	`RpingOUT`	SSI0TX	`RTxcomplete`
				`Rprocbuffer = PING`
SSI0TX	ALT	`RpongOUT`	SSI0TX	`RTxcomplete`
				`Rprocbuffer = PONG`
SSI1RX	PRI	SSI1RX	`LpingIN`	`LRxcomplete`
				`Lprocbuffer = PING`
SSI1RX	ALT	SSI1RX	`LpongIN`	`LRxcomplete`
				`Lprocbuffer = PONG`
SSI1TX	PRI	`LpingOUT`	SSI1TX	`LTxcomplete`
				`Lprocbuffer = PING`
SSI1TX	ALT	`LpongOUT`	SSI1TX	`LTxcomplete`
				`Lprocbuffer = PONG`

In parallel with these DMA transfers, a corresponding set of DMA transfers take place between memory and the SSI1 peripheral. These deal with left channel audio data.

In function `main()`, an endless `while()` loop waits until both `RTxcomplete` and `RRxcomplete` are set before calling function `RprocessBuffer()` This function uses the value of `RprocBuffer` to decide whether to process the contents of array `RpingIN` or `RpongIN`. It also resets flags `RTxcomplete` and `RRxcomplete`.

Function `main()` then waits until both `LTxcomplete` and `LRxcomplete` are set before calling function `LprocessBuffer()`.

Interrupts generated on completion of transfers are handled by the interrupt service routines associated with the peripheral involved, that is, either `SSI0IntHandler()` or `SSI1IntHandler()`, and these are defined in file `tm4c123_aic3104_init.c`.

Each DMA transfer is of `BUFSIZE` 16-bit sample values, corresponding to `BUFSIZE sampling` instants. The value of the constant `BUFSIZE` is defined in header file `tm4c123_aic3104_init.h`. Function `SSI_interrupt_routine()` is not used by program `tm4c123_loop_dma.c` but has been defined here as a trap for unexpected SSI peripheral

interrupts.

Listing 2.6 Function `SSI0IntHandler()`, defined in file `tm4c123_aic3104_init.c`

```
void SSI0IntHandler(void)
{
  unsigned long ulModeTXPRI,ulModeTXALT;
  unsigned long ulModeRXPRI,ulModeRXALT;
  ulModeTXPRI = ROM_uDMAChannelModeGet(UDMA_CHANNEL_SSI0TX |
                                       UDMA_PRI_SELECT);
  ulModeTXALT = ROM_uDMAChannelModeGet(UDMA_CHANNEL_SSI0TX |
                                       UDMA_ALT_SELECT);
  ulModeRXPRI = ROM_uDMAChannelModeGet(UDMA_CHANNEL_SSI0RX |
                                       UDMA_PRI_SELECT);
  ulModeRXALT = ROM_uDMAChannelModeGet(UDMA_CHANNEL_SSI0RX |
                                       UDMA_ALT_SELECT);
  if(ulModeTXPRI ｜ UDMA_MODE_STOP)
  {
    Rprocbuffer = PING;
    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_SSI0TX |
                               UDMA_PRI_SELECT,
                               UDMA_MODE_PINGPONG,
                               RpingOUT,
                               (void *)(SSI0_BASE + 0x008),
                               BUFSIZE);
    RTxcomplete = 1;
  {
  if(ulModeTXALT ｜ UDMA_MODE_STOP)
  {
    Rprocbuffer = PONG;
    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_SSI0TX |
                               UDMA_ALT_SELECT,
                               UDMA_MODE_PINGPONG,
                               RpongOUT,
                               (void *)(SSI0_BASE + 0x008),
                               BUFSIZE);
    RTxcomplete = 1;
  }
  if(ulModeRXPRI ｜ UDMA_MODE_STOP)
  {
    Rprocbuffer = PING;
    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_SSI0RX |
                               UDMA_PRI_SELECT,
                               UDMA_MODE_PINGPONG,
                               (void *)(SSI0_BASE + 0x008),
                               RpingIN,
                               BUFSIZE);
    RRxcomplete = 1;
  }
  if(ulModeRXALT ｜ UDMA_MODE_STOP)
  {
```

```
    Rprocbuffer = PONG;
    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_SSI0RX |
                               UDMA_ALT_SELECT,
                               UDMA_MODE_PINGPONG,
                               (void *)(SSI0_BASE + 0x008),
                               RpongIN,
                               BUFSIZE);
    RRxcomplete = 1;
  }
}
```

2.5.10 Running the Program

Build and run program `tm4c123_loop_dma.c` and verify its operation. As supplied, the program uses the (blue) LINE IN connection for input. If you wish to use a microphone connected to the (pink) MIC IN connection as an input device, you will have to change the parameters passed to function `tm4c123_aic3104_init()` to

```
tm4c123_aic3104_init(FS_48000_HZ,
                     AIC3104_MIC_IN,
                     IO_METHOD_DMA
                     PGA_GAIN_6_DB);
```

2.5.11 Monitoring Program Execution

GPIO pin PE2 is set just before, and reset just after, the call to function `Lprocessbuffer`. Hence, the signal on that pin is a rectangular pulse, the duration of which indicates the time taken to execute function `Lprocessbuffer`. In this example, the duration of the pulse is very short since no significant processing takes place in the function. The rectangular pulse is repeated with a period equal to the time between consecutive DMA transfers, that is, `BUFSIZE` sampling periods. An example of the pulse output on GPIO pin PE2 by program `tm4c123_loop_dma.c` is shown in Figure 2.6. In this case, the value of the constant `BUFSIZE` is 256, the sampling rate is 48 kHz, and hence, the time between consecutive pulses is equal to $256/48,000 = 5.33$ ms.

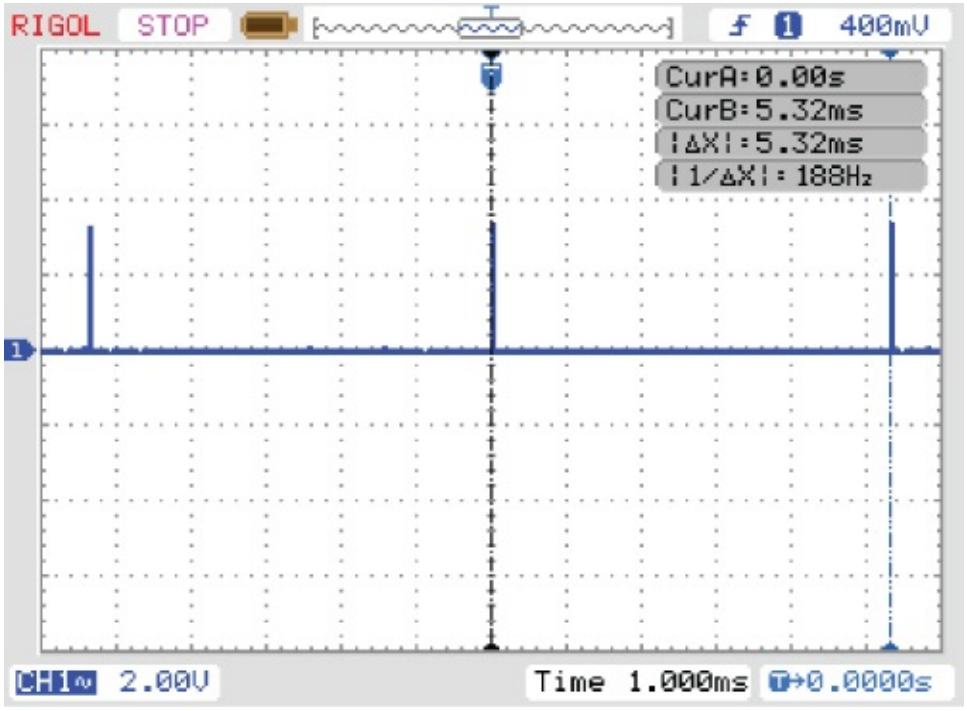


Figure 2.6 Pulse output on GPIO pin PE2 by program `tm4c123_loop_dma.c`.

2.5.12 Measuring the Delay Introduced by DMA-Based I/O

The extra delay between analog input and output, of `2*BUFSIZE` sampling periods, introduced by DMA-based i/o using ping-pong buffering as implemented on the TM4C123, may be measured using a signal generator and oscilloscope.

Connect the signal generator output to both the left channel of the (blue) LINE IN socket on the audio booster pack and one channel of the oscilloscope, and connect an oscilloscope probe from another channel on the oscilloscope to the left channel scope hook (TP2) on the audio booster pack. Scope selector jumper J6 should be fitted. Figure 2.7 shows a delay of approximately 11.7 ms introduced by the program to a rectangular pulse of duration 1.0 ms. In contrast, program `tm4c123_loop_intr.c` introduces a delay of approximately 1.0 ms, as shown in Figure 2.8. The additional delay introduced by the use of DMA-based i/o is equal to approximately 10.7 ms. At a sampling rate of 48 kHz, ping-pong mode DMA transfers of `BUFSIZE = 256` samples correspond to a delay of 2 * 256/48,000 = 10.67 ms. The value of constant `BUFSIZE` is defined in file `tm4c123_aic3104_init.h`.

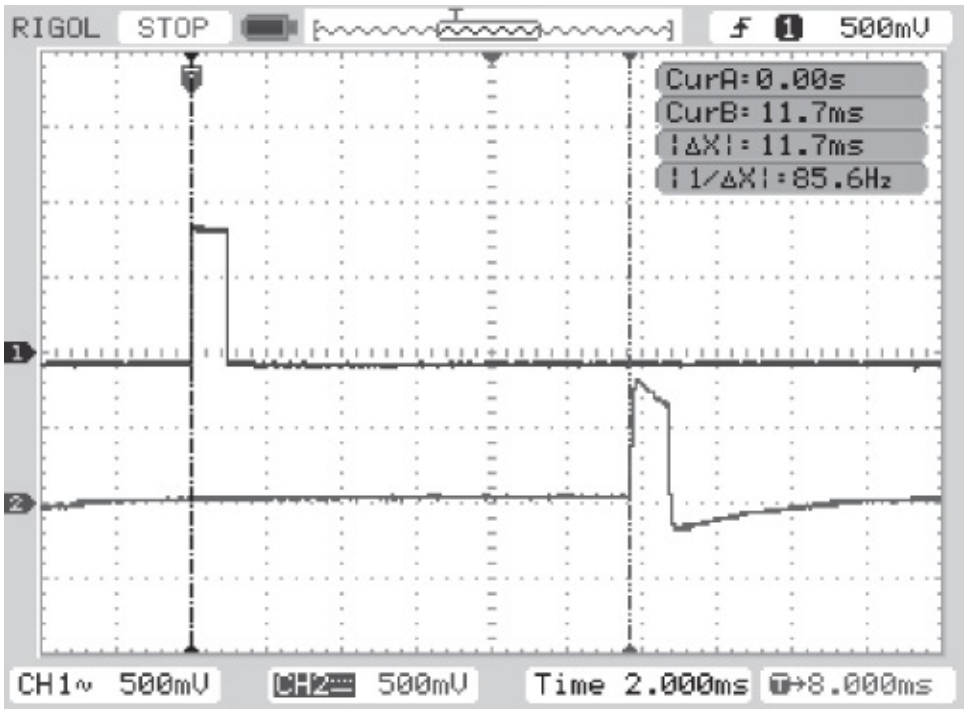


Figure 2.7 Delay introduced by use of DMA-based i/o in program `tm4c123_loop_dma.c`. Upper trace shows rectangular pulse of duration 1 ms applied to LINE IN, lower trace shows output from LINE OUT. `BUFSIZE = 256`, sampling rate 48 kHz.

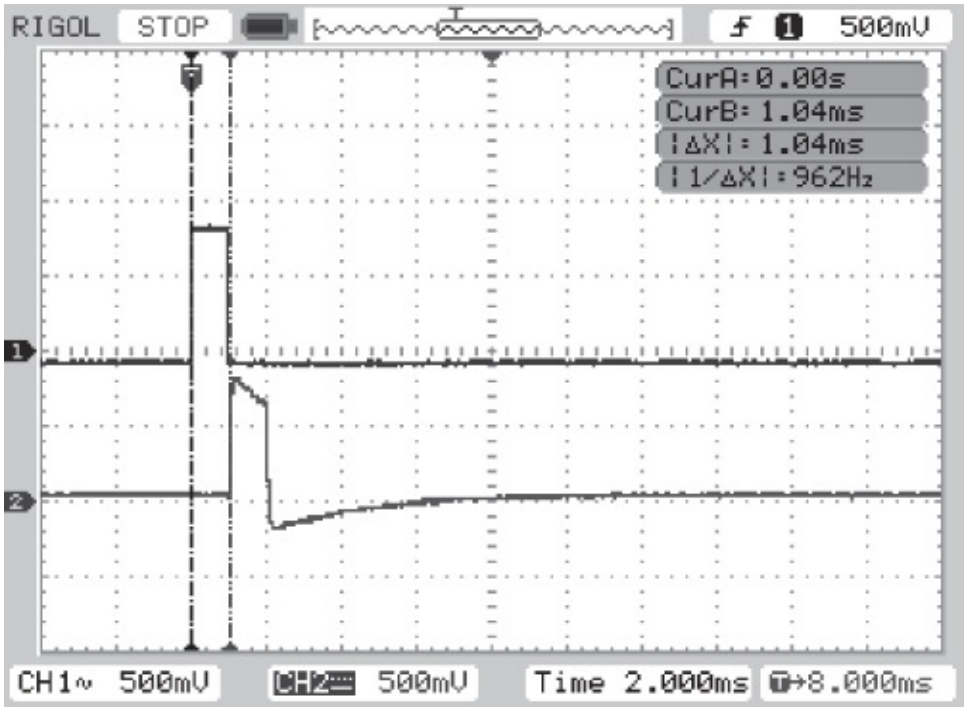


Figure 2.8 Delay introduced by use of interrupt-based i/o in program `tm4c123_loop_intr.c`. Upper trace shows rectangular pulse of duration 1 ms applied to LINE IN, lower trace shows output from LINE OUT. Sampling rate 48 kHz.

Example 2.6

Basic Input and Output Using DMA (`stm32f4_loop_dma.c`).

ARM Cortex-M4 processors from different manufacturers implement DMA slightly differently (although the underlying principles are similar). Program `stm32f4_loop_dma.c`, shown in Listing 2.13, is functionally equivalent to program `tm4c123_loop_dma.c` but differs slightly because of the differences between the DMA peripherals in the TM4C123 and STM32F407 processors.

Listing 2.7 Program `stm32f4_loop_dma.c`

```
// stm32f4_loop_dma.c
#include “stm32f4_wm5102_init.h”
extern uint16_t pingIN[BUFSIZE], pingOUT[BUFSIZE];
extern uint16_t pongIN[BUFSIZE], pongOUT[BUFSIZE];
int rx_proc_buffer, tx_proc_buffer;
volatile int RX_buffer_full = 0;
volatile int TX_buffer_empty = 0;
void DMA1_Stream3_IRQHandler()
{
  if(DMA_GetITStatus(DMA1_Stream3,DMA_IT_TCIF3))
  {
    DMA_ClearITPendingBit(DMA1_Stream3,DMA_IT_TCIF3);
    if(DMA_GetCurrentMemoryTarget(DMA1_Stream3))
      rx_proc_buffer = PING;
    else
      rx_proc_buffer = PONG;
    RX_buffer_full = 1;
  }
}
void DMA1_Stream4_IRQHandler()
{
  if(DMA_GetITStatus(DMA1_Stream4,DMA_IT_TCIF4))
  {
    DMA_ClearITPendingBit(DMA1_Stream4,DMA_IT_TCIF4);
    if(DMA_GetCurrentMemoryTarget(DMA1_Stream4))
      tx_proc_buffer = PING;
    else
      tx_proc_buffer = PONG;
    TX_buffer_empty = 1;
  }
}
void process_buffer()
{
  uint16_t *rxbuf, *txbuf;
  int16_t i;
  if (rx_proc_buffer ⟩ PING)
```

```
    rxbuf = pingIN;
  else
    rxbuf = pongIN;
  if (tx_proc_buffer ⌋ PING)
    txbuf = pingOUT;
  else
    txbuf = pongOUT;
  for (i=0 ; i<(BUFSIZE/2) ; i++)
  {
    *txbuf++ = *rxbuf++;
    *txbuf++ = *rxbuf++;
  }
  TX_buffer_empty = 0;
  RX_buffer_full = 0;
}
int main(void)
{
  stm32_wm5102_init(FS_48000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_DMA);
  while(1)
  {
    while (!(RX_buffer_full && TX_buffer_empty)){}
    GPIO_SetBits(GPIOD, GPIO_Pin_15);
    process_buffer();
    GPIO_ResetBits(GPIOD, GPIO_Pin_15);
  }
}
```

2.5.13 DMA in the STM32F407 Processor

DMA in the STM32F407 processor is organized into unidirectional streams. Two DMA controllers have eight streams each, and streams are subdivided into eight channels. Individual channels are associated with specific peripheral devices. Bidirectional I2S on the STM32F407 is implemented using the SPI/I2S2 and I2Sext peripheral devices. Program `stm32f4_loop_dma.c` makes use of the inbuilt ping-pong mode of buffering possible on the STM32F407. DMA-based i/o is selected by passing parameter value `IO_METHOD_DMA` to function `stm32_wm5102_init()`.

In function `stm32_wm5102_init()`, stream 3 channel #0 is configured to make DMA transfers between the I2S peripheral and input buffers (arrays) in memory (alternately `pingIN` and `pongIN`). It generates an interrupt when a transfer of `BUFSIZE` 16-bit samples has completed. (Those 16-bit samples correspond alternately to L and R audio channels and so a transfer of `BUFSIZE` samples corresponds to `BUFSIZE/2` sampling instants.)

Stream 4 channel #2 is configured to make DMA transfers between output buffers in memory (alternately `pingOUT` and `pongOUT`) and the I2S peripheral. It too generates an interrupt when a transfer of `BUFSIZE` 16-bit samples has completed. (Those 16-bit samples correspond alternately to L and R audio channels and so a transfer of `BUFSIZE` samples corresponds to `BUFSIZE/2` sampling instants.) Two separate interrupt service routines are used; one for each

of the aforementioned DMA processes. The actions carried out in these routines are simply to assign to variables `rx_proc_buffer` and `tx_proc_buffer` the values `PING` or `PONG` and to set flags `RX_buffer_full` and `TX_buffer_empty`. These variables are used in function `process_buffer()`.

Switching between buffers `pingIN`, `pongIN`, `pingOUT`, and `pongOUT` is handled automatically by the STM32F4's DMA mechanism. If, for example, `rx_proc_buffer` is equal to `PING`, this indicates that the most recently completed stream 3 DMA transfer has filled buffer `pingIN` and this data is available to be processed. If `tx_proc_buffer` is equal to `PING`, this indicates that the most recently completed stream 4 DMA transfer has written the contents of buffer `pingOUT` to the I2S peripheral and this buffer is available to be filled with new data.

Function `main()` simply waits until both `RX_buffer_full` and `TX_buffer_empty` flags are set, that is, until both DMA transfers have completed, before calling function `process_buffer()`.

In program `stm32f4_loop_dma.c`, function `process_buffer()` simply copies the contents of the most recently filled input buffer (`pingIN` or `pongIN`) to the most recently emptied output buffer (`pingOUT` or `pongOUT`), according to the values of variables `rx_proc_buffer` and `tx_proc_buffer`.

Frame-based processing may be carried out in function `process_buffer()` using the contents of the most recently filled input buffer as input and writing output sample values to the most recently emptied output buffer. DMA transfers will complete, and function `proc_buffer()` will be called, every `BUFSIZE/2` sampling instants and therefore any processing must be completed within `BUFSIZE/(2*fs)` seconds (or, more strictly speaking, before the next DMA transfer completion).

2.5.14 Running the Program

Run program `stm32f4_loop_dma.c` and verify its operation using a signal source and oscilloscope or headphones. As supplied, the program reads input from the (green) LINE IN socket on the audio card and outputs to the (pink) LINE OUT and (black) HEADSET connections.

2.5.15 Measuring the Delay Introduced by DMA-Based I/O

The extra delay between analog input and output, of `BUFSIZE` sampling periods, introduced by DMA-based i/o as implemented on the STM32F407, may be measured using a signal generator and oscilloscope.

Connect the signal generator output to both the left channel of the (green) LINE IN socket on the Wolfson audio card and one channel of the oscilloscope, and connect another channel on the oscilloscope to the left channel of the (pink) LINE OUT socket on the Wolfson audio card. Figure 2.9 shows a delay of approximately 5.9 ms introduced by the program to a rectangular pulse of duration 1.0 ms. In contrast, program `stm32f4_loop_intr.c` introduces a delay of approximately 560 µs, as shown in Figure 2.10. The additional delay introduced by the use of

DMA-based i/o is equal to approximately 5.3 ms. At a sampling rate of 48 kHz, ping-pong mode DMA transfers of `BUFSIZE = 256` samples (128 samples per channel) correspond to a delay of 256/48,000 = 5.33 ms. The value of constant `BUFSIZE` is defined in file `stm32f4_aic3104_init.h`.

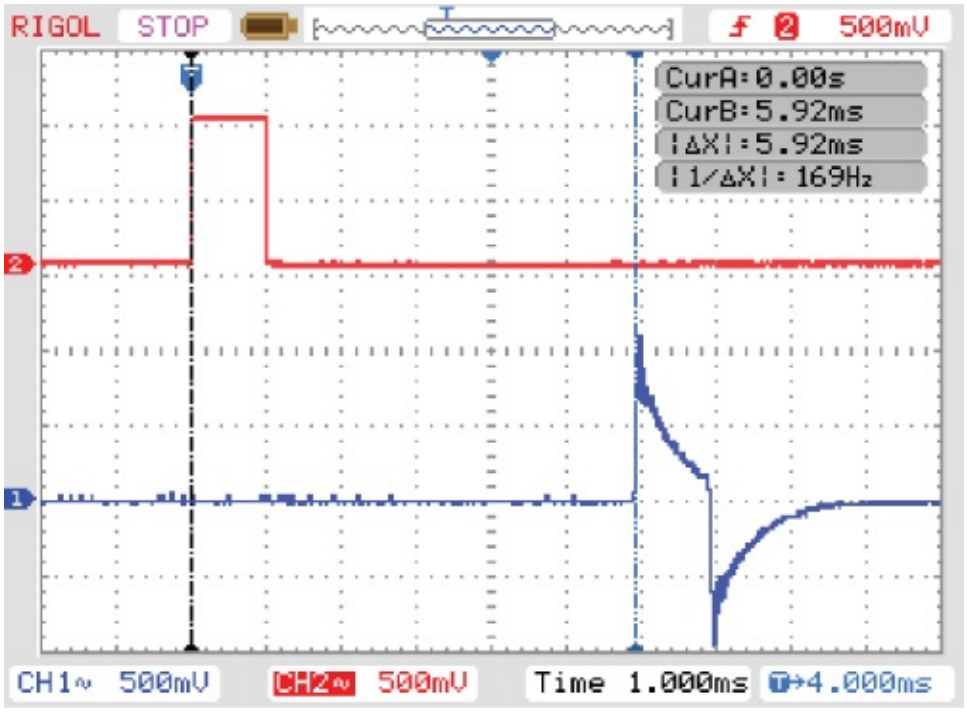


Figure 2.9 Delay introduced by use of DMA-based i/o in program `stm32f4_loop_dma.c`. Upper trace shows rectangular pulse of duration 1 ms applied to LINE IN, lower trace shows output from LINE OUT. `BUFSIZE = 256`, sampling rate 48 kHz.

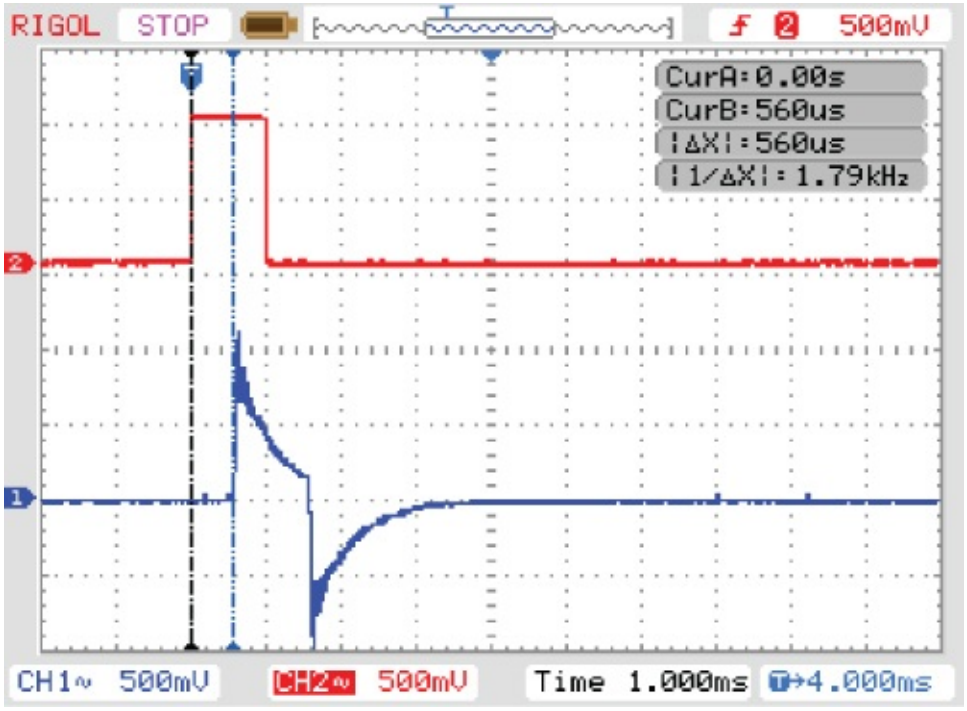


Figure 2.10 Delay introduced by use of interrupt-based i/o in program `stm32f4_loop_intr.c`. Upper trace shows rectangular pulse of duration 1 ms applied to LINE IN, lower trace shows output from LINE OUT. Sampling rate 48 kHz.

Example 2.7

Modifying Program `tm4c123_loop_intr.c` to create a delay (`tm4c123_delay_intr.c`).

Some simple, yet striking, effects can be achieved simply by delaying the sample values as they pass from input to output. Program `tm4c123_delay_intr.c`, shown in Listing 2.15, demonstrates this.

Listing 2.8 Program `tm4c123_delay_intr.c`

```
// tm4c123_delay_intr.c
#include "tm4c123_aic3104_init.h"
#define BUFFER_SIZE 2000
float32_t buffer[BUFFER_SIZE];
int16_t buf_ptr = 0;
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  float32_t input_left, input_right, delayed;
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  input_left = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  input_right = (float32_t)(sample_data.bit16[0]);
  delayed = buffer[buf_ptr] + input_left;
  buffer[buf_ptr] = input_left;
  buf_ptr = (buf_ptr+1)% BUFFER_SIZE;
  sample_data.bit32 = ((int16_t)(delayed));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  sample_data.bit32 = ((int16_t)(0));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
}
int main(void)
{
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_MIC_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1){}
}
```

A delay line is implemented using the array `buffer` to store samples as they are read from the ADC. 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 `buffer`, that sample value is retrieved, added to the current input sample, and written to the DAC. Figure 2.11 shows a block diagram representation of the operation of program `tm4c123_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 `tm4c123_delay_intr.c`, using a microphone connected to the (pink) MIC IN socket on the audio booster pack as an input device.

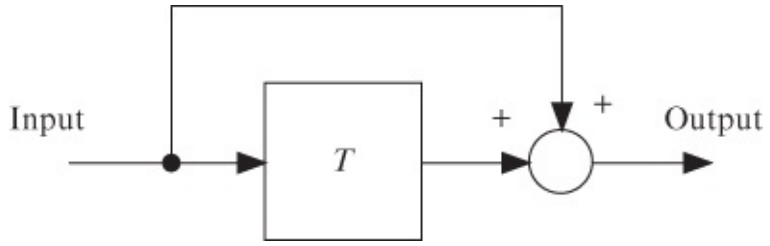


Figure 2.11 Block diagram representation of program `tm4c123_delay_intr.c`.

Example 2.8

Modifying Program `tm4c123_loop_intr.c` to create an echo (`tm4c123_echo_intr.c`).

By feeding back a fraction of the output of the delay line to its input, a fading echo effect can be realized. Program `tm4c123_echo_intr.c`, shown in Listing 2.17 and represented in block diagram form in Figure 2.12, does this.

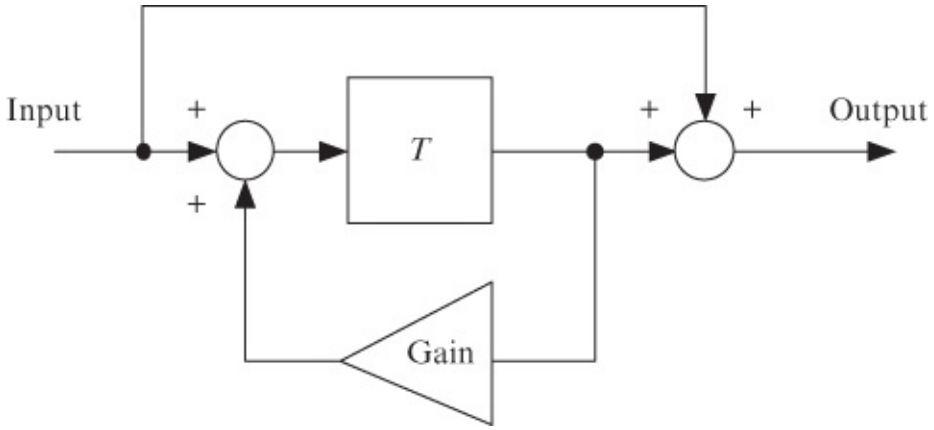


Figure 2.12 Block diagram representation of program `tm4c123_echo_intr.c`.

Listing 2.9 Program `tm4c123_echo_intr.c`

```
// tm4c123_echo_intr.c
#include “tm4c123_aic3104_init.h”
#define BUFFER_SIZE 2000
#define GAIN 0.6f
float32_t buffer[BUFFER_SIZE];
int16_t buf_ptr = 0;
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  float32_t input_left, input_right, delayed;
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  input_left = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  input_right = (float32_t)(sample_data.bit16[0]);
  delayed = buffer[buf_ptr];
  buffer[buf_ptr] = input_left + delayed * GAIN;
  buf_ptr = (buf_ptr+1) % BUFFER_SIZE;
  sample_data.bit32 = ((int16_t)(delayed));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  sample_data.bit32 = ((int16_t)(0));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
}
int main(void)
{
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_MIC_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1){}
}
```

The value of the constant `BUFFER_SIZE` in program `tm4c123_echo_intr.c` determines the number of samples stored in the array `buffer` and hence the duration of the delay implemented. 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, 1 will cause instability. Build and run this program. Experiment with different values of `GAIN` (between 0.0 and 1.0) and `BUFFER_SIZE` (between 100 and 8000). Source file `tm4c123_echo_intr.c` must be edited and the project rebuilt in order to make these changes.

Example 2.9

Modifying Program `tm4c123_loop_intr.c` to create a flanging effect (`tm4c123_flanger_intr.c`).

Flanging is an audio effect used in recording studios (and live performances) that, depending on its parameter settings, can add a whooshing sound not unlike 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 2.13. 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 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 $1/T$ Hz with the first notch located at frequency $1/2T$ 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 `tm4c123_flanger_intr.c` (Listing 2.19), 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, for example, 3 Hz gives an effect closer to that produced by a Leslie rotating speaker.

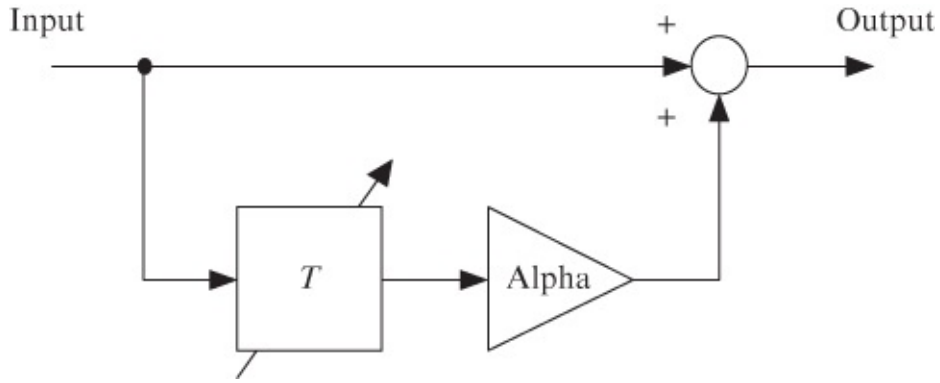


Figure 2.13 Block diagram representation of program `tm4c123_flanger_intr.c`.

Listing 2.10 Program `tm4c123_flanger_intr.c`

```
// tm4c123_flanger_intr.c
#include "tm4c123_aic3104_init.h"
#define TS 0.000020833333f                  // sampling rate 48 kHz
#define PERIOD 10.0f             // period of delay modulation
#define MEAN_DELAY 0.001f       // mean delay in seconds
#define MODULATION_MAG 0.0008f // delay modulation magnitude
#define BUFFER_SIZE 2048
```

```
#define ALPHA 0.9f
uint16_t in_ptr = 0;           // pointers into buffers
uint16_t out_ptr;
float32_t buffer[BUFFER_SIZE];
float32_t t = 0.0f;
float32_t Rxn, Lxn, Ryn, Lyn, delay_in_seconds;
uint16_t delay_in_samples;
float32_t theta;
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  Lxn = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  Rxn = (float32_t)(sample_data.bit16[0]);
  buffer[in_ptr] = Lxn;
  in_ptr = (in_ptr + 1)
  t = t + TS;
  theta = (float32_t)((2*PI/PERIOD)*t);
        delay_in_seconds = MEAN_DELAY
                    + MODULATION_MAG * arm_sin_f32(theta);
  delay_in_samples = (uint16_t)(delay_in_seconds
                                    * 48000.0);
  out_ptr = (in_ptr + BUFFER_SIZE
                - delay_in_samples) % BUFFER_SIZE;
  Lyn = Lxn + buffer[out_ptr]*ALPHA;
  sample_data.bit32 = ((int16_t)(Lyn));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  sample_data.bit32 = ((int16_t)(Lyn));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
}
int main(void)
{
  tm4c123_aic3104_init(FS_48000_HZ,
                       AIC3104_MIC_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1){}
}
```

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 the statement in program `tm4c123_flanger_intr.c` that reads

```
Lyn = Lxn + buffer[out_ptr]*ALPHA;
```

to read

```
Lyn = Lxn - buffer[out_ptr]*ALPHA;
```

Figures 2.14 and 2.15 show experimentally measured examples of the instantaneous impulse and magnitude frequency responses of the flanger program for the two cases described. Details of how these results were obtained are given later in this chapter.

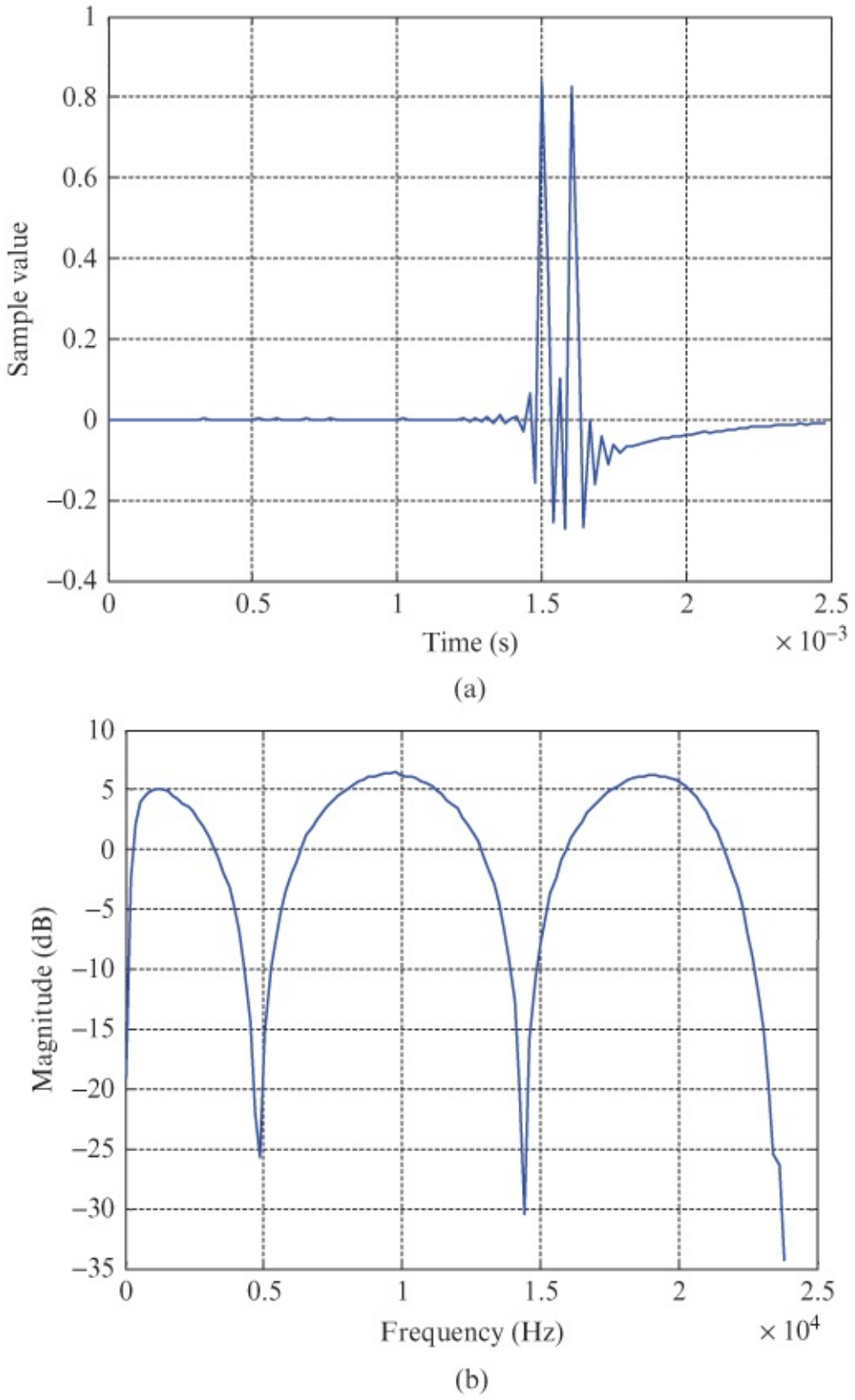


Figure 2.14 (a) impulse response and (b) magnitude frequency response of flanger implemented using program `tm4c123_flanger_intr.c` at an instant when delay T is equal to 104.2 μs. The notches in the magnitude frequency response are at frequencies 4800 and 14,400 Hz.

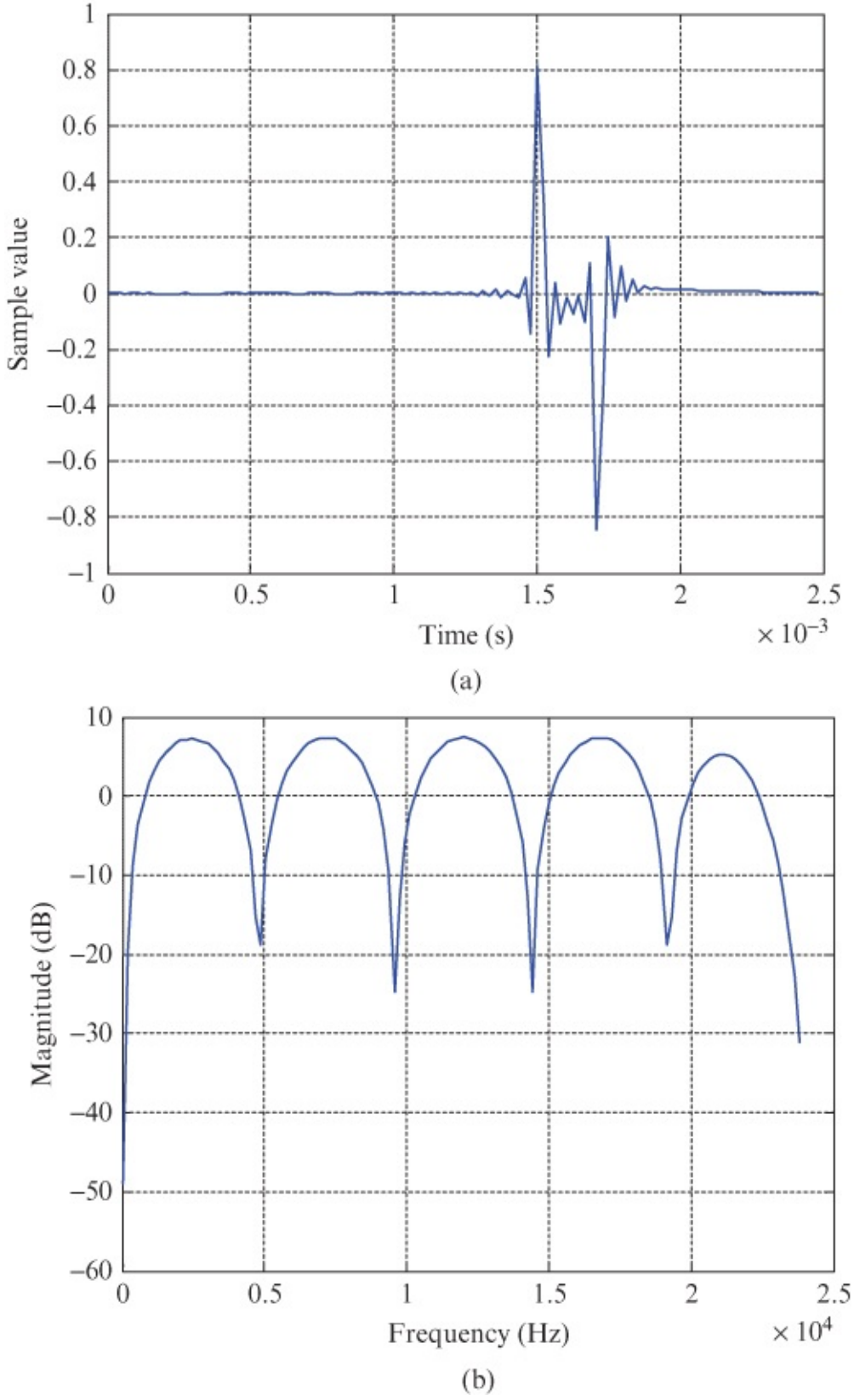


Figure 2.15 (a) Impulse response and (b) magnitude frequency response of modified flanger implemented using program `tm4c123_flanger_intr.c` at an instant when delay T is equal to 208.3 μs. The notches in the magnitude frequency response are at frequencies 0, 4800, 9600, 14,400, and 19,200 Hz.

The time-varying delay implemented by the flanger may be illustrated using program

`tm4c123_flanger_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 `tm4c123_flanger_dimpulse_intr.c` comprises the input (which passes straight through) plus a delayed and attenuated version of the input. Figure 2.16 shows an example of the output from the program captured using a *Rigol DS1052E* oscilloscope. The pulse in the centre 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 2.31.

Figure 2.16 Output waveform produced using program `tm4c123_flanger_dimpulse_intr.c` at an instant when delay T is equal to approximately 400 µs.

The time-varying magnitude frequency response of the flanger is illustrated in Figure 2.17, which shows the output of program `tm4c123_flanger_intr.c`, modified so that the input signal is a pseudorandom binary sequence, displayed as a spectrogram (frequency vs. time) using *Goldwave*. The dark bands in the spectrogram correspond to the notches in the magnitude frequency responses. An effective alternative to pseudorandom noise as an input signal is to blow gently on the microphone.

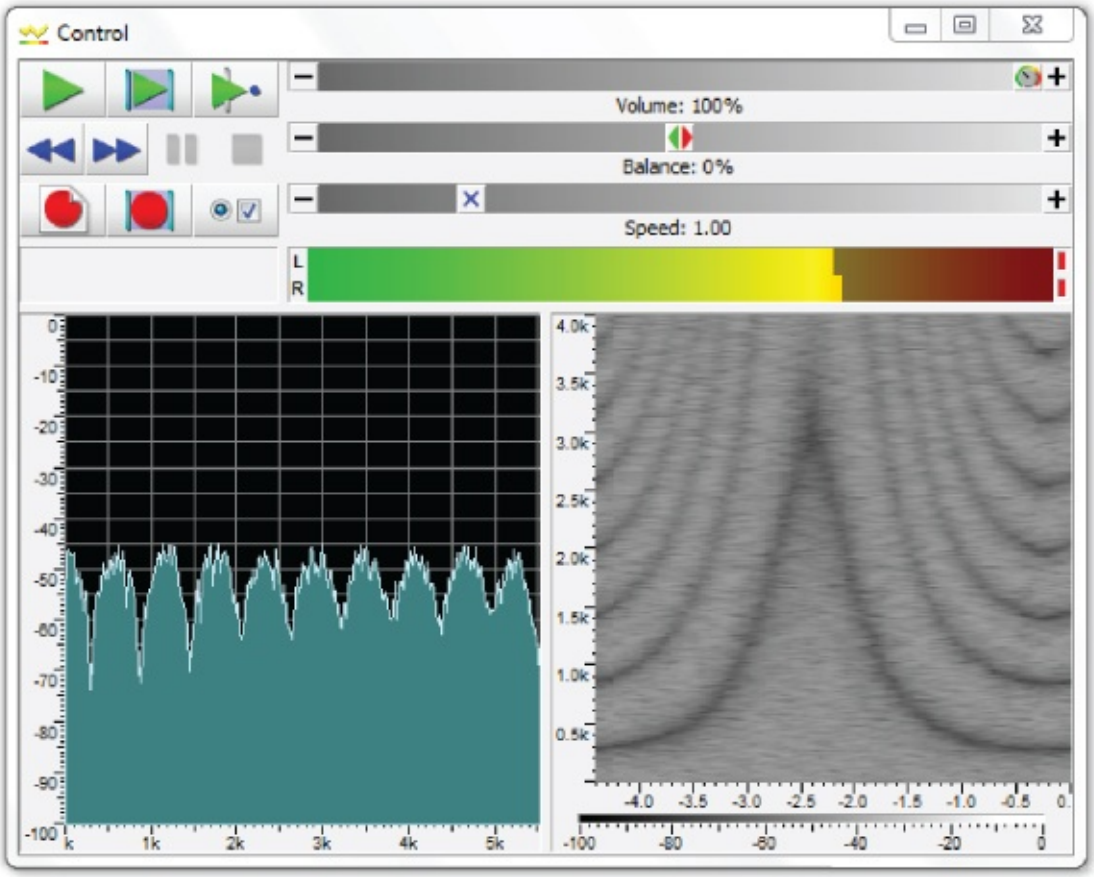


Figure 2.17 Spectrum and spectrogram of flanger output for pseudorandom noise input. In the spectrogram, the x-axis represents time in seconds and the y-axis represents frequency in Hz.

Example 2.10

Loop Program with Input Data Stored in a Buffer (`stm32f4_loop_buf_intr.c`).

Program `stm32f4_loop_buf_intr.c`, shown in Listing 2.21, is similar to program `stm32f4_loop_intr.c` except that it implements circular buffers in arrays `lbuffer` and `rbuffer`, each containing the `BUFFER_SIZE` most recent input sample values on one or other channel. Consequently, it is possible to examine this data after halting the program.

Listing 2.11 Program `stm32f4_loop_buf_intr.c`

```
// stm32f4_loop_buf_intr.c
#include "stm32f4_wm5102_init.h"
#define BUFFER_SIZE 256
float32_t rbuffer[BUFFER_SIZE];
int16_t rbufptr = 0;
float32_t lbuffer[BUFFER_SIZE];
int16_t lbufptr = 0;
void SPI2_IRQHandler()
{
  int16_t left_out_sample, right_out_sample;
  int16_t left_in_sample, right_in_sample;
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ⌋ SET)
  {
    left_in_sample = SPI_I2S_ReceiveData(I2Sx);
    left_out_sample = left_in_sample;
    lbuffer[lbufptr] = (float32_t)(left_in_sample);
    lbufptr = (lbufptr+1)
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
  }
  else
  {
    right_in_sample = SPI_I2S_ReceiveData(I2Sx);
    right_out_sample = right_in_sample;
    rbuffer[rbufptr] = (float32_t)(right_in_sample);
    rbufptr = (rbufptr+1) % BUFFER_SIZE;
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
  GPIO_ToggleBits(GPIOD, GPIO_Pin_15);
 }
int main(void)
{
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
  while(1){}
}
```

2.5.16 Running the Program

Build and run the program. Use a signal generator connected to the (pink) LINE IN socket on the Wolfson audio card to input a sinusoidal signal with a frequency between 100 and 3500 Hz. Halt the program after a short time and save the contents of the array `rbuffer` or `lbuffer` to a data file by typing

```
save <filename> <start address>, <start address + 0x400>
```

at the *Command* line in the *MDK-ARM debugger*, where `start address` is the address of array `rbuffer` or `lbuffer` (and which may be determined using *Watch* and *Memory* functions in the *MDK-ARM debugger*.

The contents of the data file can be plotted using MATLAB® function `stm32f4_logfft()`. Figure 2.18 shows an example of the contents of array `lbuffer` when the frequency of the input signal was 350 Hz. The discontinuity in the waveform at time $t = 16$ ms corresponds to the value of the index `lbufptr` into the array when program execution was halted. Recall that array `lbuffer` is used as a circular buffer in which older sample values are overwritten continually by newer ones. Program `stm32f4_loop_buf_intr.c` is used again later in this chapter in order to highlight the characteristics of the codec's antialiasing filter.

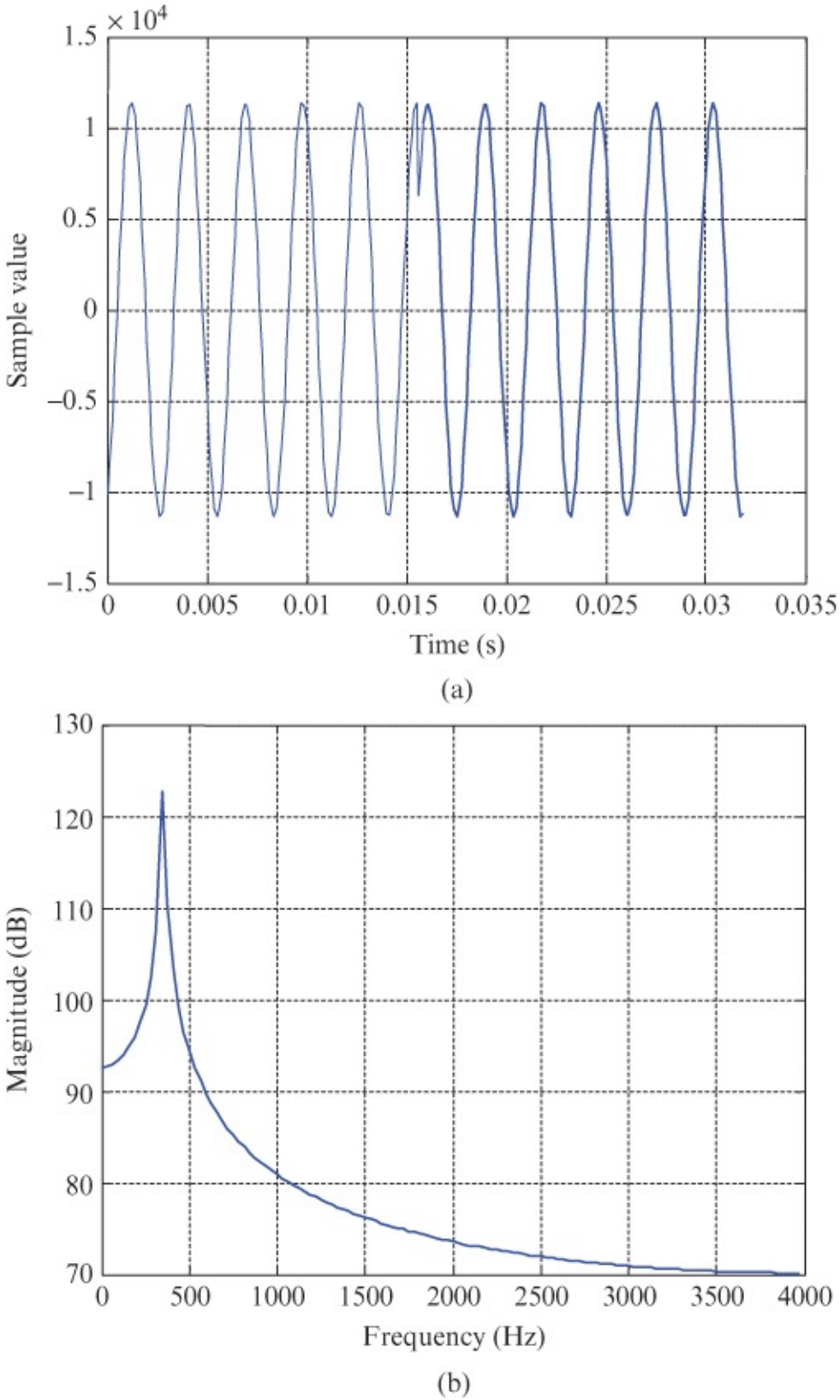


Figure 2.18 Sample values stored in array `lbuffer` by program `stm32f4_loop_buf_intr.c` plotted using MATLAB function `stm32f4_logftt()`. Input signal frequency was 350 Hz.

2.6 Real-Time Waveform Generation

The following examples are concerned with the generation of a variety of different analog output waveforms, including sinusoids of different frequencies, by writing different sequences of sample values to the DACs in the WM5102 and AIC3104 codecs. In this way, the detailed characteristics of the DACs are revealed and the concepts of sampling, reconstruction, and aliasing are illustrated. In addition, the use of the *Goldwave* shareware application is introduced. This virtual instrument is a useful alternative to an oscilloscope or spectrum analyzer and is used again in later chapters.

Example 2.11

Sine Wave Generation Using a Lookup Table (`stm32f4_sine48_intr.c`).

Program `stm32f4_sine48_intr.c`, shown in Listing 2.23, generates a sinusoidal output signal using interrupt-based i/o and a table lookup method (Figure 2.18). Its operation is as follows. A 48-point lookup table is initialized in the array `sine_table` such that the value of `sine_table[i]` is equal to

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

Array `sine_table` therefore contains 48 samples of exactly one cycle of a sinusoid. In this example, a sampling rate of 48 kHz is used, and therefore, interrupts will occur every 20.833 µs. Within interrupt service routine `SPI2_IRQHandler()`, the most important program statements are executed. Every 20.833 µs, a sample value read from the array `sine_table` is written 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.

Listing 2.12 Program `stm32f4_sine48_intr.c`

```
// stm32f4_sine48_intr.c
#include "stm32f4_wm5102_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};
int16_t sine_ptr = 0;  // pointer into lookup table
void SPI2_IRQHandler()
{
  int16_t left_out_sample, right_out_sample;
  int16_t left_in_sample, right_in_sample;
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ⌋ SET)
  {
    left_in_sample = SPI_I2S_ReceiveData(I2Sx);
    left_out_sample = sine_table[sine_ptr];
    sine_ptr = (sine_ptr+1)%LOOPLENGTH;
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
  }
  else
  {
    right_in_sample = SPI_I2S_ReceiveData(I2Sx);
    right_out_sample = 0;
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
  GPIO_ToggleBits(GPIOD, GPIO_Pin_15);
 }
int main(void)
{
  stm32_wm5102_init(FS_48000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
  while(1){}
}
```

The DAC in the WM5102 codec reconstructs a sinusoidal analogue output signal from the output sample values.

2.6.1 Running the Program

Build and run the program and verify a 1 kHz output tone on the (green) LINE OUT and (black)

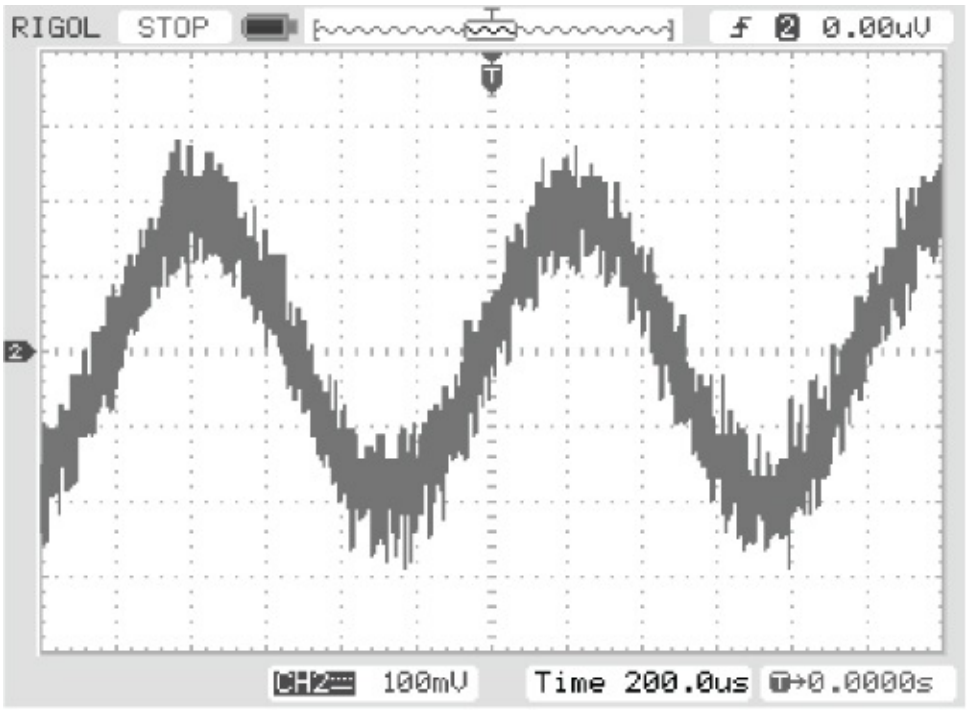
HEADSET connections on the Wolfson audio card.

Program `stm32f4_sine8_intr.c` is similar to program `stm32f4_sine48_intr.c` in that it generates a sinusoidal analog output waveform with a frequency of 1 kHz from a lookup table of sample values. However, it uses a sampling rate of 8 kHz and a lookup table containing just eight sample values, that is,

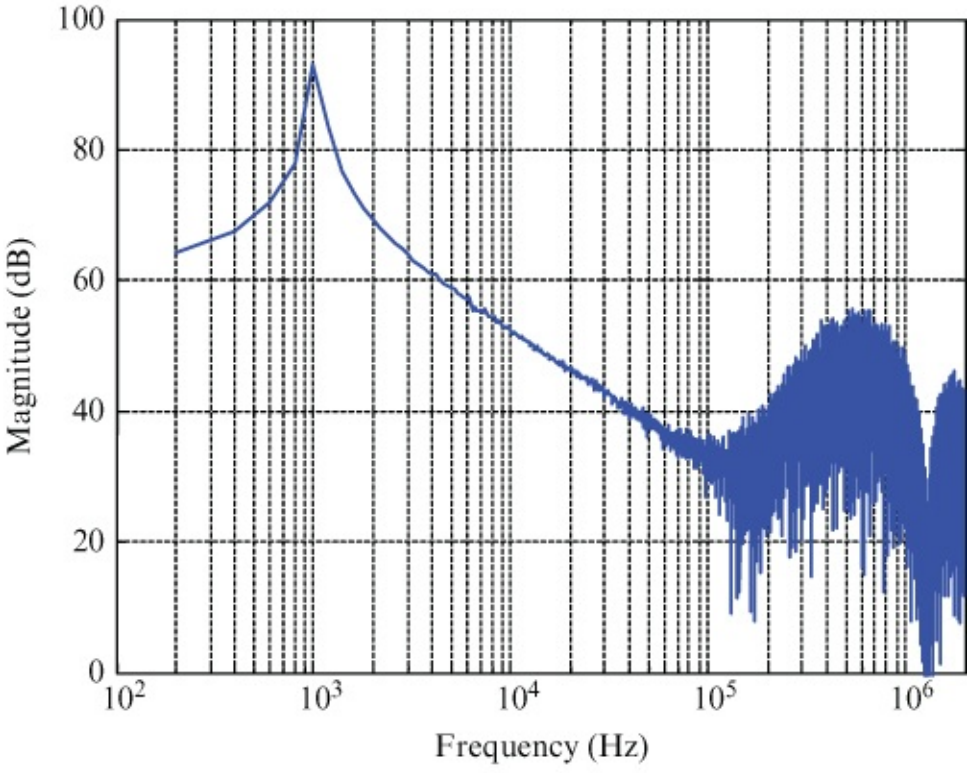
```
#define LOOPLENGTH 8
int16_t sine_table[LOOPLENGTH] = {
  0, 7071, 10000, 7071, 0, -7071, -10000, -7071};
```

2.6.2 Out-of-Band Noise in the Output of the AIC3104 Codec (`tm4c123_sine48_intr.c`)

While performing essentially similar functions, the codecs used in the two different hardware platforms differ subtly in their characteristics and functionality. If you are using the TM4C123 LaunchPad and AIC3104 audio booster pack rather than the STM32F407 Discovery and Wolfson audio card, connect an oscilloscope to the (black) LINE OUT socket on the audio booster pack. Run program `tm4c123_sine48_intr.c` and you should see a waveform similar to that shown in Figure 2.19. There is a significant level of noise superimposed on the 1-kHz sine wave, and this reveals something about the characteristics of the AIC3104 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.



(a)

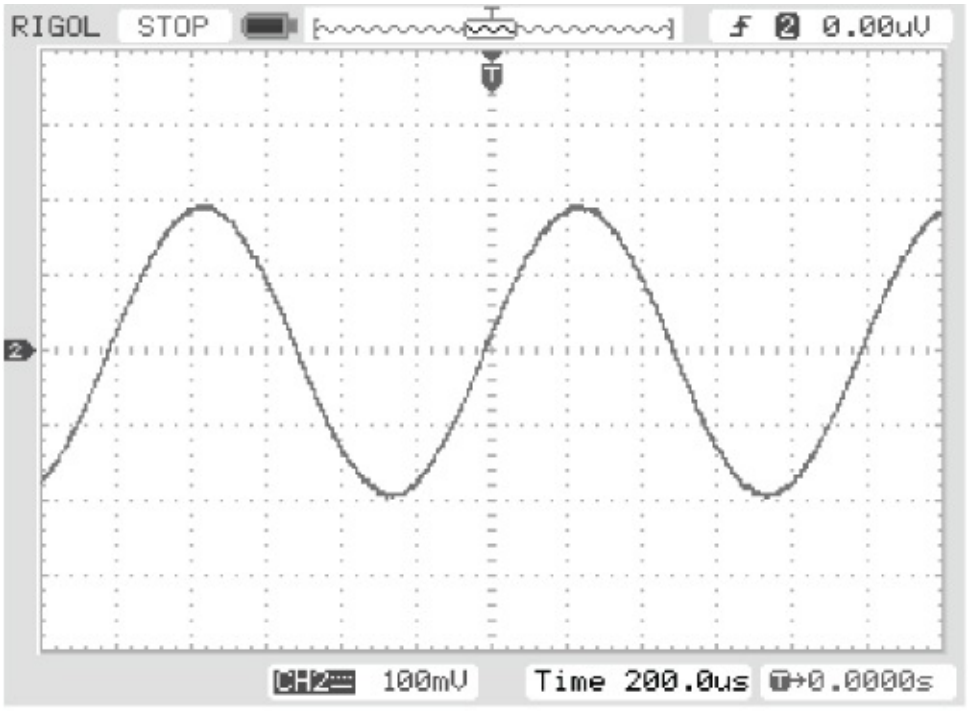


(b)

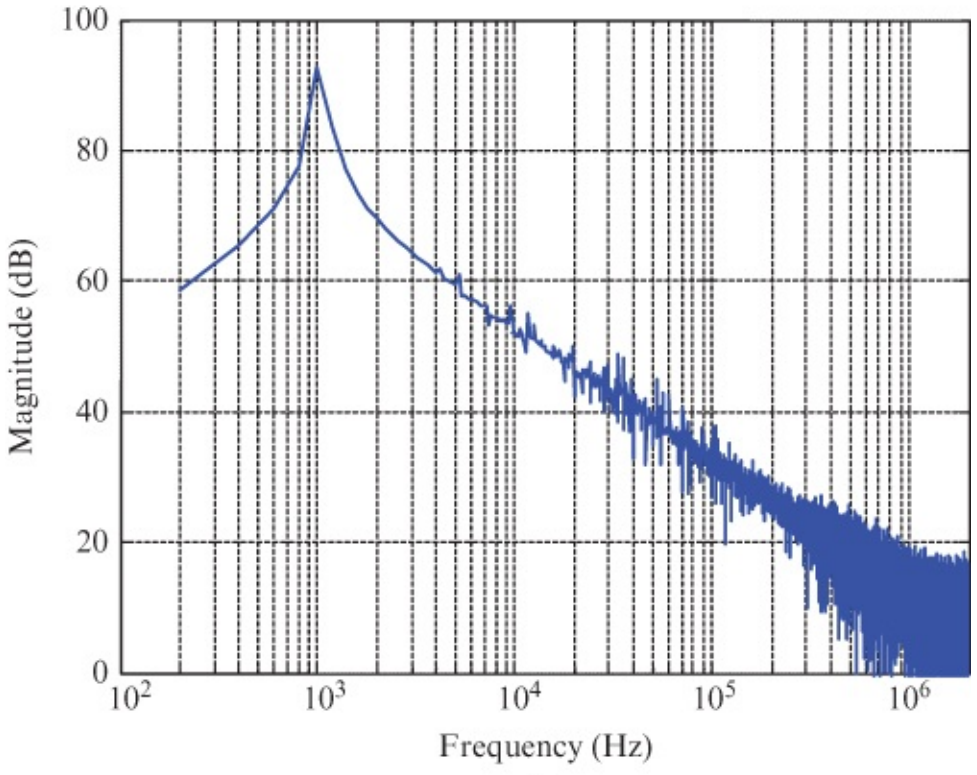
Figure 2.19 (a) 1-kHz sinusoid generated using program `tm4c123_sine48_intr.c` viewed using *Rigol DS1052E* oscilloscope connected to (black) LINE OUT connection on audio booster pack. (b) Magnitude of FFT of signal plotted using MATLAB.

The plot shown in Figure 2.19 was obtained using a *Rigol DS1052E* digital oscilloscope to capture the output waveform and then the magnitude of the FFT of the data was plotted using MATLAB. It shows clearly both 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 in-band detail of audio signals generated by the AIC3104 codec 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. This can be done by fitting jumper J6 or J7 and looking at the output signals on the scope hooks, TP2 and TP3, on the audio booster pack. Figure 2.20 shows the filtered output signal from the program in both time and frequency domains for comparison with Figure 2.19.



(a)



(b)

Figure 2.20 (a) 1-kHz sinusoid generated using program `tm4c123_sine48_intr.c` viewed using *Rigol DS1052E* oscilloscope connected to scope hook on audio booster pack. (b) Magnitude of FFT of signal plotted using MATLAB.

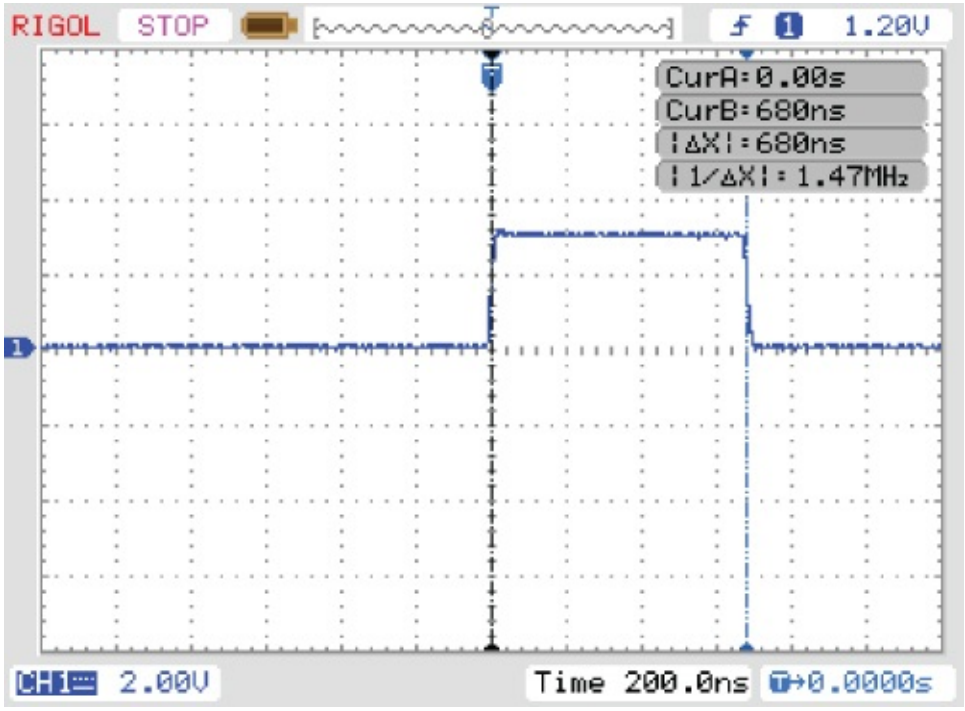


Figure 2.21 Rectangular pulse output on GPIO pin PD15 by program `stm32f4_sine_intr.c`.

In any examples in this book, in which an oscilloscope is used to view an output signal from the AIC3104 codec, it is recommended that the oscilloscope probe is connected to one of the scope hooks on the audio booster pack (TP2 or TP3) and the corresponding jumper J6 or J7 is fitted.

In contrast, the analog signals output by the WM5102 codec via the LINE OUT connection on the Wolfson audio card do not contain out-of-band-noise as described here and connections may be made directly from the (green) LINE OUT connection on the Wolfson audio card to an oscilloscope (using a 3.5-mm jack plug to RCA plug cable and an RCA to BNC adapter).

Example 2.12

Sine wave generation using a `sinf()` function call (`stm32f4_sine_intr.c`).

Sine waves of different frequencies can be generated using the table lookup method employed by programs `stm32f4_sine48_intr.c` and `stm32f4_sine8_intr.c`. For example, a 3-kHz sine wave can be generated using program `stm32f4_sine8_intr.c` by changing the program statement that reads

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

to read

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

However, changing the contents and/or size of the lookup table is not a particularly flexible way of generating sinusoids of arbitrary frequencies (Example 2.26 demonstrates how different frequency sinusoids *may* be generated from a fixed lookup table).

Program `stm32f4_sine_intr.c`, shown in Listing 2.25, takes a different approach. At each sampling instant, that is, in function `SPI2_IRQHandler()`, a new output sample value is calculated using a call to the math library function `sinf()`. 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 the value of `theta` exceeds 2π, the value 2π is subtracted from it.

Listing 2.13 Program `stm32f4_sine_intr.c`

```
// stm32f4_sine_intr.c
#include “stm32f4_wm5102_init.h”
#define SAMPLING_FREQ 8000
float32_t frequency = 1000.0;
float32_t amplitude = 10000.0;
float32_t theta_increment;
float32_t theta = 0.0;
void SPI2_IRQHandler()
{
  int16_t left_out_sample, right_out_sample;
  int16_t left_in_sample, right_in_sample;
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ⌋ SET)
  {
    left_in_sample = SPI_I2S_ReceiveData(I2Sx);
    theta_increment = 2*PI*frequency/SAMPLING_FREQ;
    theta += theta_increment;
    if (theta > 2*PI) theta -= 2*PI;
    GPIO_SetBits(GPIOD, GPIO_Pin_15);
    left_out_sample = (int16_t)(amplitude*sinf(theta));
    GPIO_ResetBits(GPIOD, GPIO_Pin_15);
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
  }
  else
  {
    right_in_sample = SPI_I2S_ReceiveData(I2Sx);
    right_out_sample = 0;
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
 }
int main(void)
{
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
  while(1){}
}
```

While program `stm32f4_sine_intr.c` has the advantage of flexibility, it also has the disadvantage, relative to program `stm32f4_sine48_intr.c`, that it requires greater computational effort. This is an important consideration in real-time applications.

2.6.3 Running the Program

Build and run this program and experiment by changing the value assigned to the variable

frequency in the range from 100.0–3800.0 (editing the program and rebuilding the project each time). The sampling frequency used by this program is 8 kHz. Program stm32f4_sine_intr.c uses GPIO pin PD15 to indicate the time taken to execute interrupt service routine SPI2_IRQHandler(). GPIO pin PD15 is set high by program statement

```
GPIO_SetBits(GPIOD, GPIO_Pin_15);
```

at the start of the interrupt service routine SPI2_IRQHandler() and reset low by program statement

```
GPIO_ResetBits(GPIOD, GPIO_Pin_15);
```

at the end of the interrupt service routine SPI2_IRQHandler(). In other words, it outputs a rectangular pulse on PD15 every 125 μs and the duration of that pulse is indicative of the time taken to compute each new output sample value. That time is determined primarily by the time taken to execute program statement

```
left_out_sample = (int16_t)(amplitude*sinf(theta));
```

Figure 2.21 shows the waveform output on GPIO pin PD15. The duration of the pulses is approximately 680 ns. Function sinf() is used because function sin() is computationally too expensive to be used in this application. A slightly more efficient alternative is provided by function arm_sin_f32(), defined in the CMSIS DSP library. Edit the program, replacing the program statement

```
left_out_sample = (int16_t)(amplitude*sinf(theta));
```

with

```
left_out_sample = (int16_t)(amplitude*arm_sin_f32(theta));
```

and you should find that the duration of the pulse output on PD15 is reduced to approximately 400 ns. In general, calls to trigonometrical functions defined in standard math libraries should be avoided in real-time DSP applications.

Example 2.13

Swept Sinusoid Using Table with 8000 Points (stm32f4_sweep_intr.c).

This example illustrates the use of a fixed lookup table of sample values in order to generate sinusoidal analog output waveforms of arbitrary frequency (without changing the sampling frequency). Listing 2.27 is of program stm32f4_sweep_intr.c, which generates a swept frequency sinusoidal signal using a lookup table containing 8000 sample values. This is a method of waveform generation commonly used in 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 exactly one cycle of a sine wave. Listing 2.28 is a partial listing of the file `sine8000_table.h`.

Listing 2.14 Program `stm32f4_sweep_intr.c`

```
// stm32f4_sweep_intr.c
#include "stm32f4_wm5102_init.h"
#include "sine8000_table.h"  //one cycle with 8000 points
#define SAMPLING_FREQ 8000.0
#define N 8000
#define START_FREQ 500.0
#define STOP_FREQ 3800.0
#define START_INCR START_FREQ*N/SAMPLING_FREQ
#define STOP_INCR STOP_FREQ*N/SAMPLING_FREQ
#define SWEEPTIME 4
#define DELTA_INCR (STOP_INCR - START_INCR)/(N*SWEEPTIME)
int16_t amplitude = 10;
float32_t float_index = 0.0;
float32_t float_incr = START_INCR;
int16_t i;
void SPI2_IRQHandler()
{
  int16_t left_out_sample, right_out_sample;
  int16_t left_in_sample, right_in_sample;
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ⌋ SET)
  {
    left_in_sample = SPI_I2S_ReceiveData(I2Sx);
    GPIO_SetBits(GPIOD, GPIO_Pin_15);
    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);
    left_out_sample = (amplitude*sine8000[i]);
    GPIO_ResetBits(GPIOD, GPIO_Pin_15);
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
  }
  else
  {
    right_in_sample = SPI_I2S_ReceiveData(I2Sx);
    right_out_sample = 0;
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
}
int main(void)
{
```

```
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
  while(1){}
}
```

Listing 2.15 Partial listing of header file `sine8000_table.h`

```
//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}
```

At each sampling instant, program `stm32f4_sweep_intr.c` reads an output sample value from the array `sine8000`, using the `float32_t` value of variable `float_index`, converted to type `int16_t`, as an index, and increments the value of `float_index` by the value `float_incr`. With `N` points in the lookup table representing one 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 `stm32f4_sweep_intr.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 sinusoid taking `SWEEPTIME` seconds to increase in frequency from `START_FREQ` to `STOP_FREQ`.

Example 2.14

Generation of DTMF Tones Using a Lookup Table (`tm4c123_sineDTMF_intr.c`).

Program `tm4c123_sineDTMF_intr.c`, listed in Listing 2.30, 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 comprising the sum of two sinusoids of different frequencies are used in telephone networks to indicate key presses. A total of 16 different combinations of frequencies each comprising one of four low-frequency components (697, 770, 852, or 941 Hz) and one of four high-frequency components (1209, 1336, 1477, or 1633 Hz) are used. Program `tm4c123_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 using different step sizes.

Listing 2.16

Program `tm4c123_sineDTMF_intr.c`.

```
// tm4c123_sineDTMF_intr.c
#include "tm4c123_aic3104_init.h"
#define TABLESIZE 512        // size of look up table
#define SAMPLING_FREQ 16000
#define STEP_770 (float32_t)(770 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1336 (float32_t)(1336 * TABLESIZE)/SAMPLING_FREQ
#define STEP_697 (float32_t)(697 * TABLESIZE)/SAMPLING_FREQ
#define STEP_852 (float32_t)(852 * TABLESIZE)/SAMPLING_FREQ
#define STEP_941 (float32_t)(941 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1209 (float32_t)(1209 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1477 (float32_t)(1477 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1633 (float32_t)(1633 * TABLESIZE)/SAMPLING_FREQ
int16_t sine_table[TABLESIZE];
float32_t loopindexlow = 0.0;
float32_t loopindexhigh = 0.0;
int16_t output;
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  float32_t input_left, input_right;
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  input_left = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  input_right = (float32_t)(sample_data.bit16[0]);
   output = (sine_table[(int16_t)loopindexlow]
             + sine_table[(int16_t)loopindexhigh]);
  loopindexlow += STEP_770;
  if (loopindexlow > (float32_t)TABLESIZE)
```

```
    loopindexlow -= (float32_t)TABLESIZE;
  loopindexhigh += STEP_1477;
  if (loopindexhigh > (float32_t)TABLESIZE)
    loopindexhigh -= (float32_t)TABLESIZE;
  sample_data.bit32 = ((int16_t)(output));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  sample_data.bit32 = ((int16_t)(output));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
}
int main(void)
{
 int16_t i;
       tm4c123_aic3104_init(FS_16000_HZ,
                            AIC3104_LINE_IN,
                            IO_METHOD_INTR,
                            PGA_GAIN_6_DB);
  for (i=0 ; i< TABLESIZE ; i++)
    sine_table[i] = (int16_t)(10000.0*sin(2*PI*i/TABLESIZE));
  while(1){}
}
```

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/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 * STEP/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 `STEP = 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 (`((int16_t)(loopindexlow))`) but a floating-point value of the pointer, or index, (`loopindexlow`) is maintained by the program and incremented by 24.64 at each sampling instant and wrapping around to 0.0 when its value exceeds 512.0 using the statements

```
loopindexlow += 24.64;
if(loopindexlow>(float32_t)TABLESIZE)loopindexlow-=(float32_t)TABLESIZE;
```

In program `tm4c123_sineDTMF_intr.c`, the floating point values by which the table lookup indices are incremented are predefined using, for example,

```
#define STEP_770 (float32_t)(770 * TABLESIZE) / SAMPLING_FREQ
```

In order to change the DTMF tone generated and simulate a different key press, edit program `tm4c123_sineDTMF_intr.c` and change the program statements.

```
loopindexlow += STEP_697;
loopindexhi += STEP_1477;
```

to, for example

```
loopindexlow += STEP_770;
loopindexhi += STEP_1209;
```

An example of the output generated by program `tm4c123_sineDTMF_intr.c` is shown in Figure 2.22.

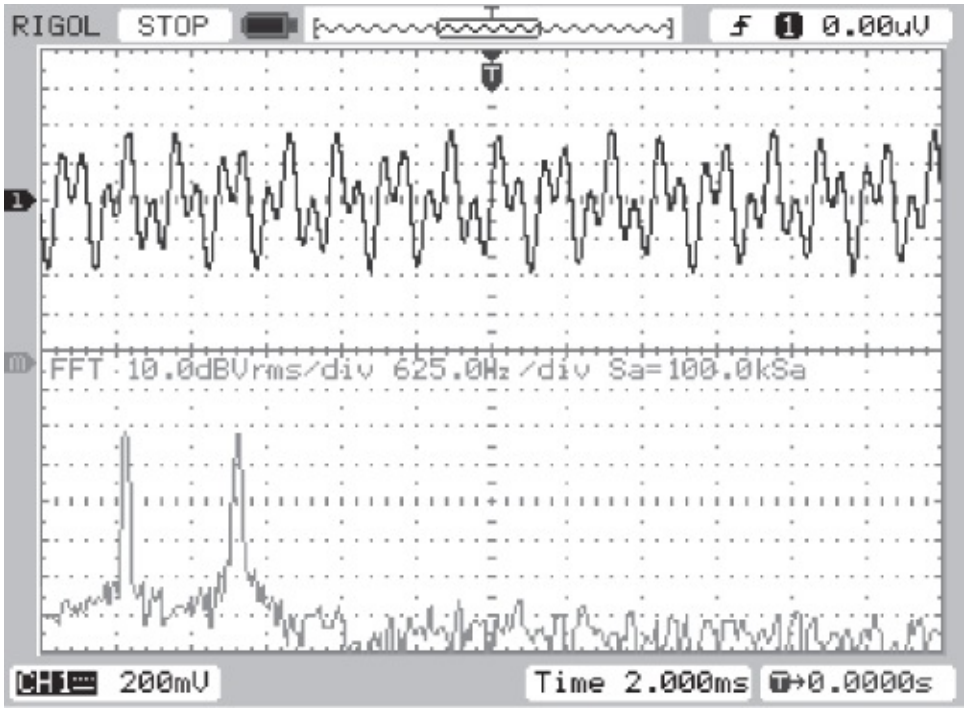


Figure 2.22 Output from program `tm4c123_sineDTMF_intr.c` viewed using *Rigol DS1052 oscilloscope*.

Example 2.15

Signal Reconstruction, Aliasing, and the Properties of the WM5102 Codec (`stm32f4_sine_intr.c`).

Generating analog output signals using, for example, program `stm32f4_sine_intr.c` is a useful means of investigating the characteristics of the WM5102 codec. If you are using the TM4C123 LaunchPad and AIC3104 audio booster pack, program `tm4c123_sine_intr.c` and others are useful means of investigating the slightly different characteristics of the AIC3104 codec.

Change the value of the variable `frequency` in program `stm32f4_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 1000 Hz sine wave is generated. The same is true if the value of `frequency` is changed to 9000.0 or 15000.0. A value of `frequency` equal to 5000.0 will

result in an output waveform with a frequency of 3000 Hz. These effects are related to the phenomenon of aliasing. Since the reconstruction (DAC) process that creates a continuous-time analog waveform from discrete-time samples is one of low-pass filtering, it follows that the bandwidth of signals output by the codec is limited (to less than half the sampling frequency). This can be demonstrated in a number of different ways.

For example, run program `stm32f4_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 analog output waveform with a frequency greater than or equal to 4000 Hz (assuming a sampling frequency of 8000 Hz). This is consistent with viewing the DAC as a low-pass filter with a cutoff frequency equal to slightly less than half its sampling frequency.

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

Example 2.16

Square wave generation using a lookup table (`stm32f4_square_intr.c`).

Program `stm32f4_square_intr.c`, shown in Listing 2.33, differs from program `stm32f4_sine8_intr.c` only in that it uses a lookup table containing 64 samples of one cycle of a square wave of frequency 125 Hz rather than 48 samples of one cycle of a sine wave of frequency 1 kHz.

Listing 2.17 Program `stm32f4_square_intr.c`

```
// stm32f4_square_intr.c
#include "stm32f4_wm5102_init.h"
#define LOOP_SIZE 64
int16_t square_table[LOOP_SIZE] = {
  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};
static int square_ptr = 0;
void SPI2_IRQHandler()
{
  int16_t left_out_sample, right_out_sample;
  int16_t left_in_sample, right_in_sample;
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ⌋ SET)
  {
    left_in_sample = SPI_I2S_ReceiveData(I2Sx);
    left_out_sample = square_table[square_ptr];
    square_ptr = (square_ptr+1)%LOOP_SIZE;
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
  }
  else
  {
    right_in_sample = SPI_I2S_ReceiveData(I2Sx);
    right_out_sample = 0;
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
 }
int main(void)
{
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
  while(1){}
}
```

2.6.4 Running the Program

Build and run the program, and using an oscilloscope connected to the (green) LINE OUT socket on the audio card, you should see an output waveform similar to that shown in Figure 2.23. 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 ringing that follows each transition in the waveform is indicative of the specific characteristics of the reconstruction filter implemented by the WM5102 DAC. The low-pass characteristic of the reconstruction filter can further be highlighted by looking at the frequency content of the output waveform. While 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 2.24.

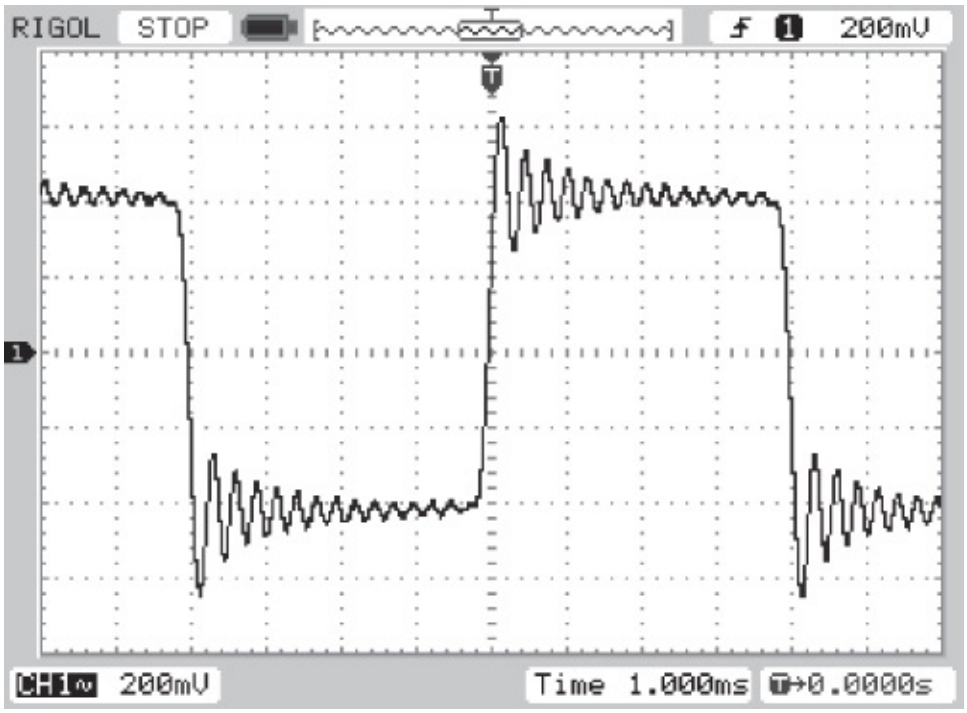


Figure 2.23 Output from program `stm32f4_square_intr.c` viewed using *Rigol DS1052 oscilloscope*.

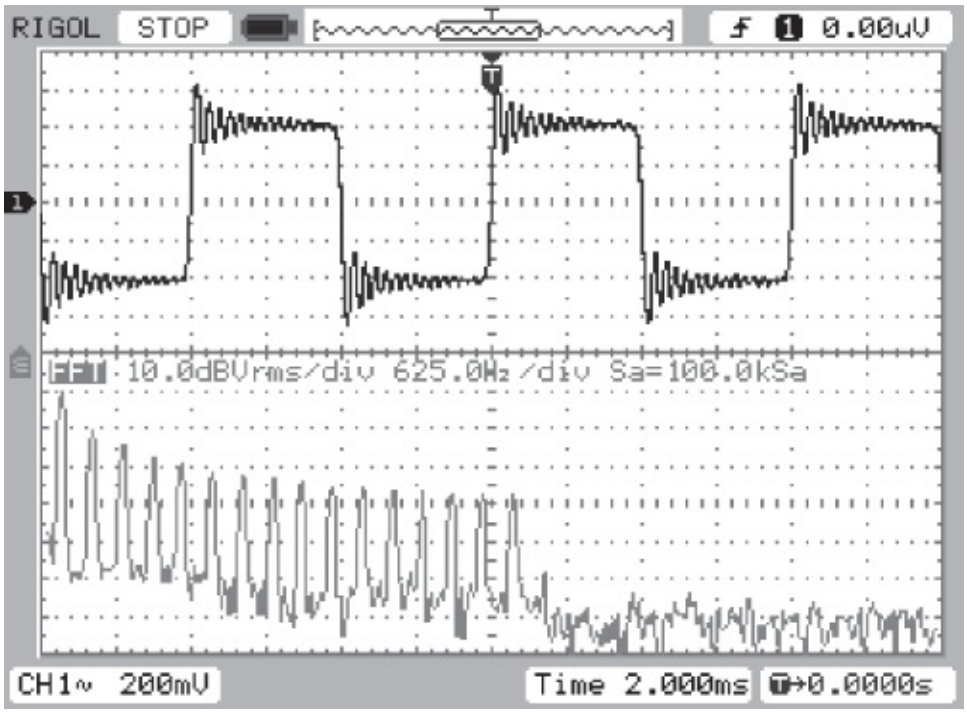


Figure 2.24 Output from program `stm32f4_square_intr.c` viewed in both time and frequency domains using *Rigol DS1052 oscilloscope*.

Example 2.17

Square Wave Generation Using a Look Up Table (`tm4c123_square_intr.c`).

Program `tm4c123_square_intr.c` is an equivalent (to `stm32f4_square_intr.c`) program that runs on the TM4C123 LaunchPad. The output waveform, as it appears on the scope hook connections on the audio booster pack and as shown in Figure 2.25, is subtly different to that shown in Figure 2.23. In this case, ringing precedes as well as follows transitions in the waveform. This suggests strongly that the reconstruction filter in the AIC3104 DAC is implemented as a digital FIR filter (at a higher sampling frequency than that at which sample values are written to the DAC by program `tm4c123_square_intr.c`). Viewed in the frequency domain (Figure 2.26), it is apparent that once again the analog output waveform contains only the harmonic components of a 125 Hz square wave at frequencies lower than 3.8 kHz.

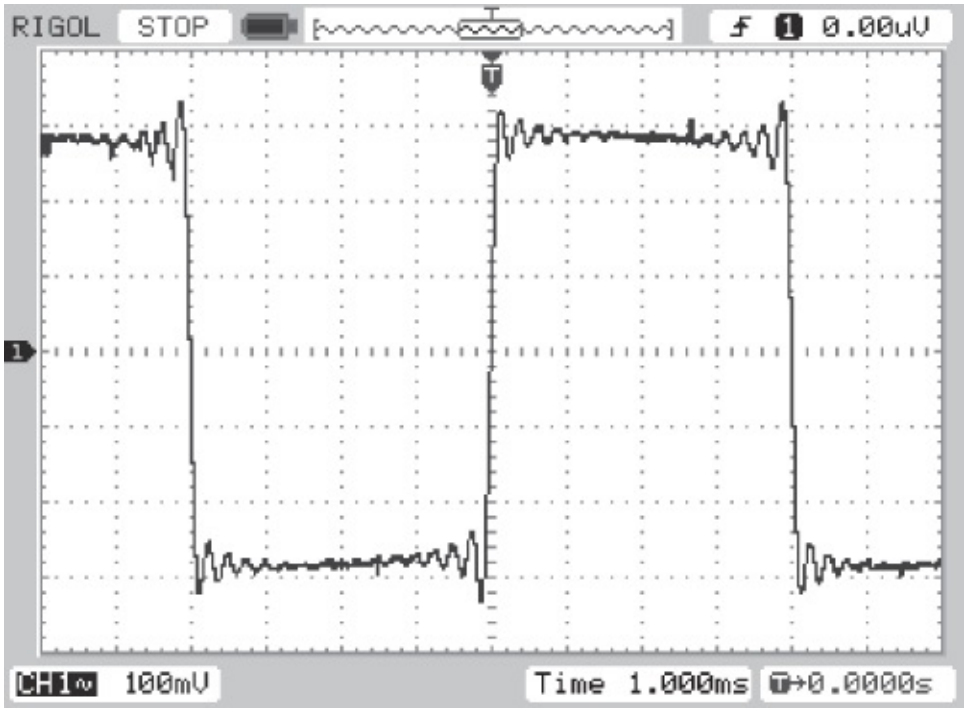


Figure 2.25 Output from program `tm4c123_square_intr.c` viewed using *Rigol DS1052 oscilloscope*.

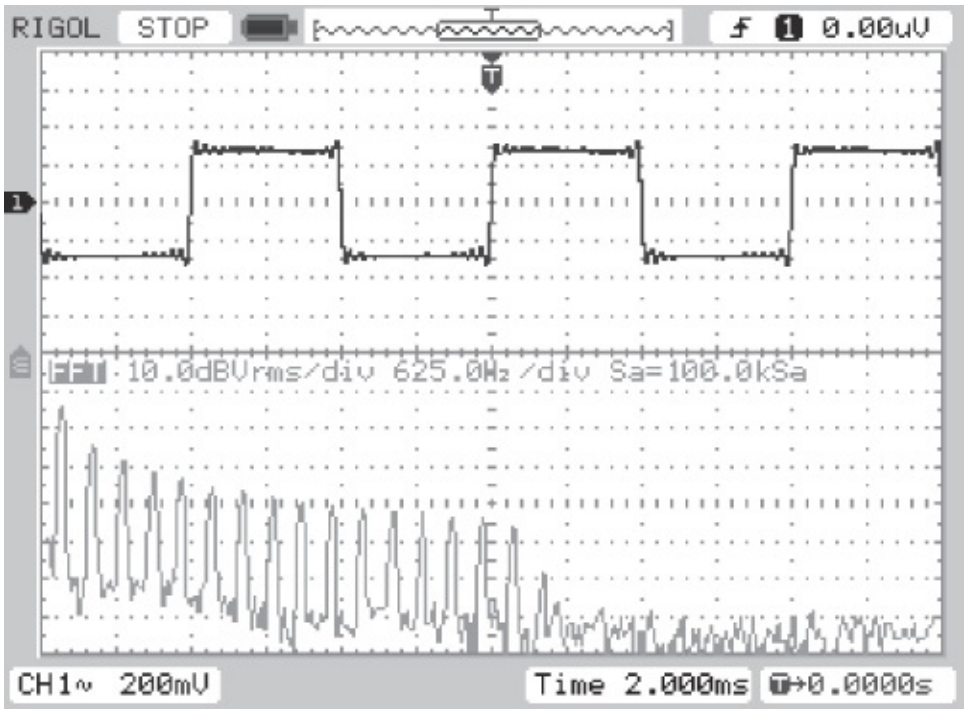


Figure 2.26 Output from program `tm4c123_square_intr.c` viewed in both time and frequency domains using *Rigol DS1052 oscilloscope*.

A further demonstration of the low-pass characteristic of the reconstruction filter is given by program `tm4c123_square_1khz_intr.c`. Here, the sequence of sample values written to the DAC at a sampling frequency of 8 kHz is given by

```
#define LOOPLENGTH 8
int16_t square_table[LOOPLENGTH] = {
  10000, 10000, 10000, 10000,
  -10000, -10000, -10000, -10000};
```

Ostensibly, these sample values represent one cycle of a 1-kHz square wave. However, the output waveform generated contains only the first two harmonic components of a 1-kHz square wave (at 1 and 3 kHz), that is, only the harmonic components of a 1-kHz square wave with frequencies lower than 3.8 kHz, the cutoff frequency of the DAC. The analog output waveform is shown in Figure 2.27 and is clearly not a square wave.

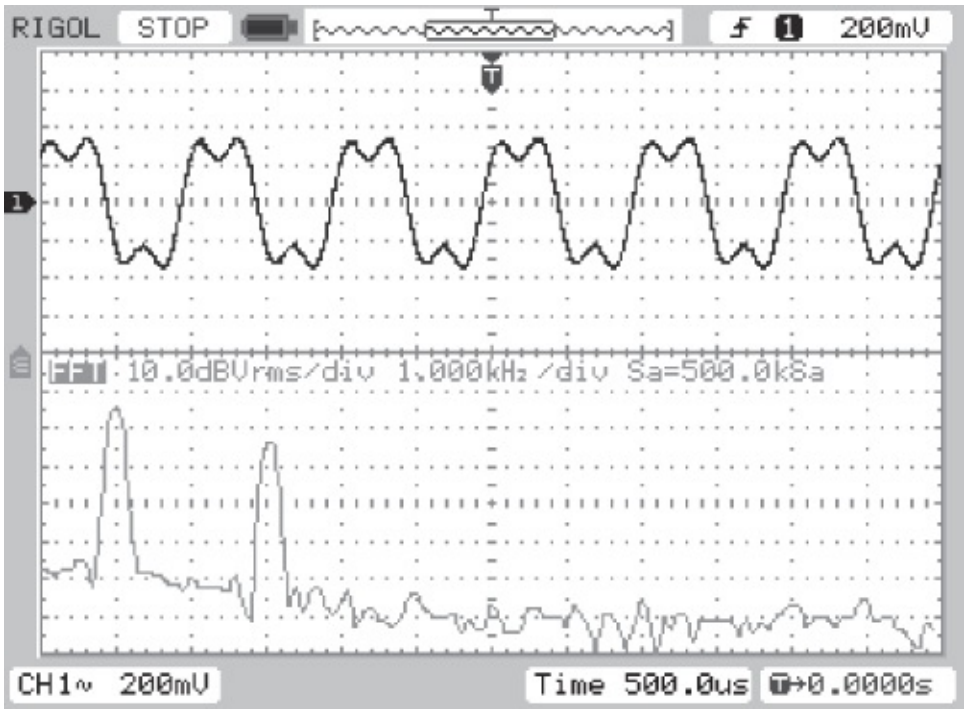


Figure 2.27 Output from program `tm4c123_square_1khz_intr.c` viewed using *Rigol DS1052 oscilloscope*.

Example 2.18

Impulse Response of the WM5102 DAC Reconstruction Filter (`stm32f4_dimpulse_intr.c`).

Each transition in the waveform generated by program `stm32f4_square_intr.c` may be considered as representative of the step response of the reconstruction filter in the WM5102 DAC. From this, the impulse response of the filter may be surmised.

That impulse response is of interest since, in accordance with linear systems theory, it characterizes the filter. Specifically, the impulse response of the WM5102 DAC is equal to the time derivative of its step response, and by inspection of Figure 2.23, it is apparent that the impulse response will have the form of a (relatively) slowly (exponentially) decaying oscillation. The impulse response can be illustrated more directly by running program `stm32f4_dimpulse_intr.c`. This program replaces the samples of a square wave in the lookup table used by program `stm32f4_square_intr.c` with a discrete impulse sequence. Figure 2.28 shows the output waveform generated by `stm32f4_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.

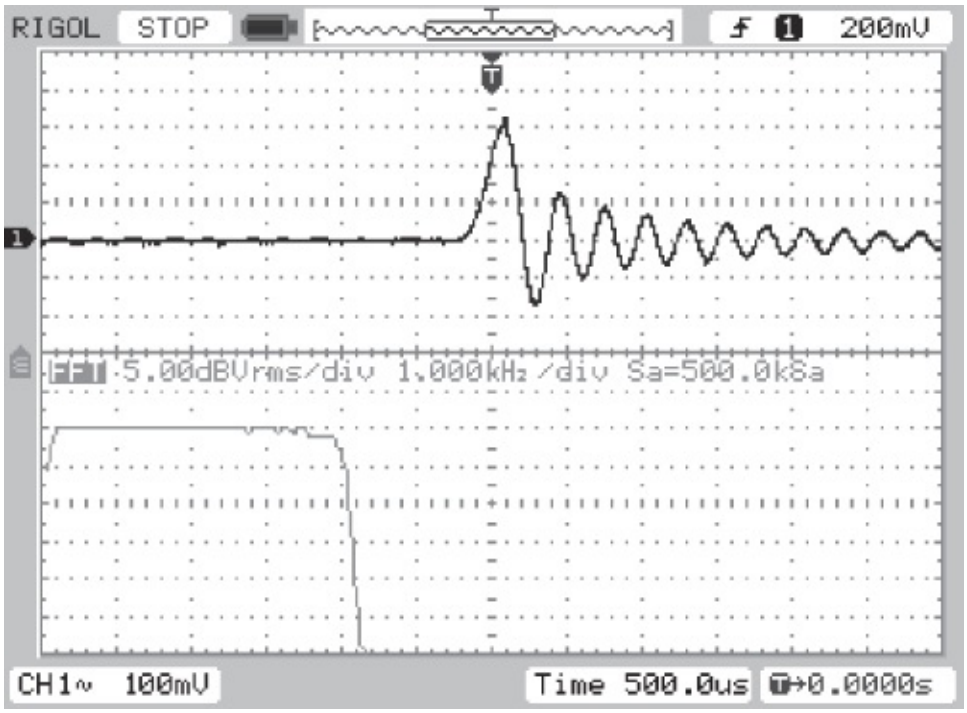


Figure 2.28 Output from program `stm32f4_dimpulse_intr.c` viewed in time and frequency domains using *Rigol DS1052 oscilloscope*.

Example 2.19

Impulse Response of the AIC3104 DAC Reconstruction Filter (`tm4c123_dimpulse_intr.c`).

This program illustrates the impulse response of the AIC3104 DAC. As suggested by the output waveform generated using program `tm4c123_square_intr.c` in the previous example, the corresponding impulse response is subtly different from that of the WM5102 DAC. The two different impulse responses, each of which corresponds to a near-ideal low-pass frequency response, are representative of those found in the majority of audio codecs. That of the WM5102 is termed a low-latency response. It is perhaps tempting to view the impulse response of the AIC3104 codec as noncausal and think of the central peak of the pulse shown in Figure 2.29 as corresponding to the same time as that of the impulse causing it. This, of course, is impossible in practice. Without knowing the detailed workings of the AIC3104 DAC, it is nonetheless apparent that the impulse causing the response shown in Figure 2.29 must have occurred before any of the ripples evident in that pulse and hence approximately 1 ms before its peak. In contrast, the impulse causing the impulse response of the WM5102 DAC shown in Figure 2.28 probably occurred approximately 300 µs before the peak.

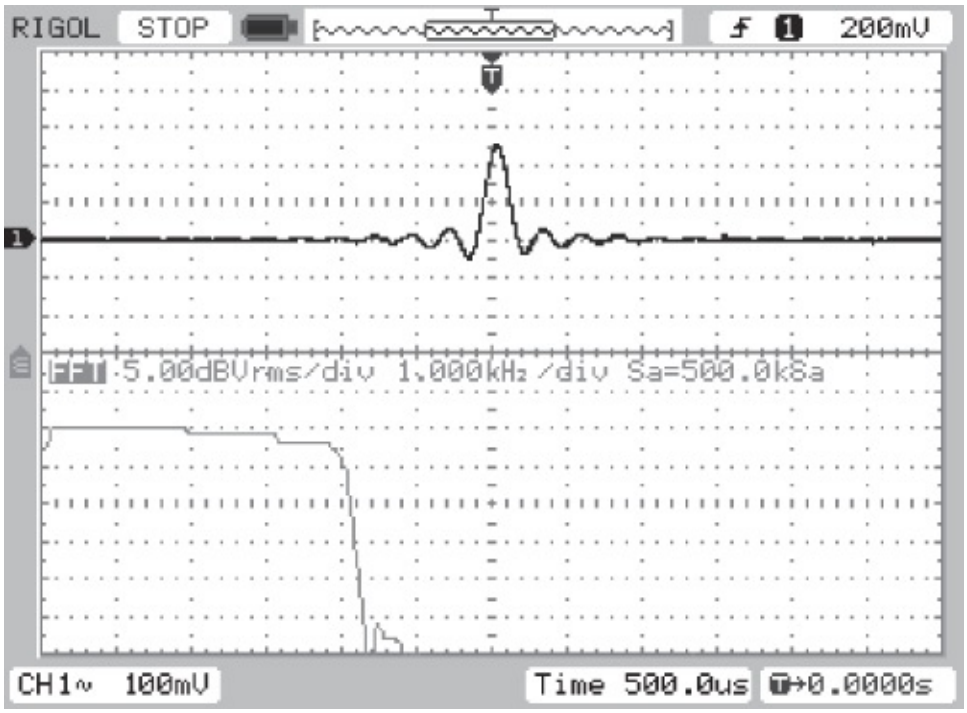


Figure 2.29 Output from program `tm4c123_dimpulse_intr.c` viewed using *Rigol DS1052 oscilloscope*.

Example 2.20

Ramp Generation (`tm4c123_ramp_intr.c`).

Listing 2.38 is of program `tm4c123_ramp_intr.c`, which generates a ramp, or sawtooth, output waveform. The value of the output sample `output_left` is incremented by 2000 every sampling instant until it reaches the value 30,000, at which point it is reset to the value −30,000. Build and run this program. Figure 2.30 shows the analog output waveform captured using an oscilloscope. The output comprises harmonic components at frequencies less than 4 kHz.

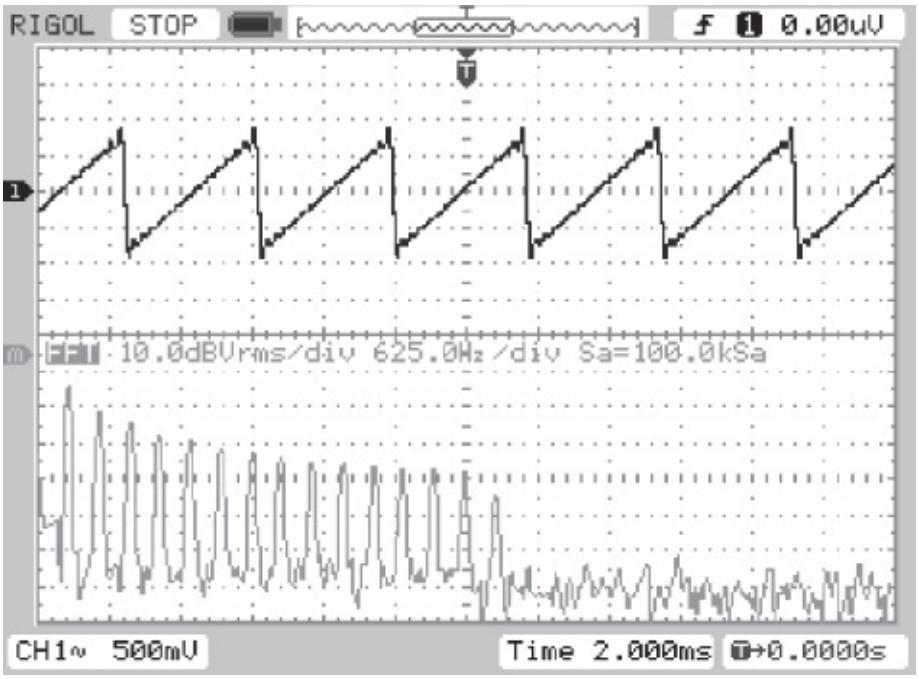


Figure 2.30 Output waveform generated by program `tm4c123_ramp_intr.c`.

Listing 2.18 Program `tm4c123_ramp_intr.c`

```
// tm4c123_ramp_intr.c
#include “tm4c123_aic3104_init.h”
int16_t output = 0;
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  float32_t input_left, input_right;
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  input_left = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  input_right = (float32_t)(sample_data.bit16[0]);
  output += 2000;
  if (output >= 30000)
  output = -30000;
  sample_data.bit32 = ((int16_t)(output));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  sample_data.bit32 = ((int16_t)(output));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
}
int main(void)
{
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1){}
}
```

Example 2.21

Amplitude Modulation (`tm4c123_am_poll.c`).

This example illustrates the basic principles of amplitude modulation (AM). Listing 2.40 is of program `tm4c123_am_poll.c`, which generates an AM signal. The array `baseband` holds 20 samples of one cycle of a cosine waveform with a frequency of fs/20 = 400 Hz (sampling frequency, fs = 8000 Hz). The array `carrier` holds 20 samples of five cycles of a sinusoidal carrier signal with a frequency of 5fs/20 = 2000 Hz. Output sample values are calculated by multiplying the baseband signal `baseband` by the carrier signal `carrier`. In this way, the baseband signal modulates the carrier signal. The variable `amp` is used to set the modulation index.

Listing 2.19 Program `tm4c123_am_poll.c`

```
// tm4c123_am poll.c
#include "tm4c123_aic3104_init.h"
short amp = 20; //index for modulation
void SSI_interrupt_routine(void){while(1){}}
int main(void)
{
  AIC3104_data_type sample_data;
  float32_t input_left, input_right;
  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;
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_POLL,
                       PGA_GAIN_6_DB);
  while(1)
  {
    for (k=0; k<20; k++)
    {
      output[k]= carrier[k] + ((amp*baseband[k]*carrier[k]/10)>>12);
      SSIDataGet(SSI1_BASE,&sample_data.bit32);
      input_left = (float32_t)(sample_data.bit16[0]);
      SSIDataGet(SSI0_BASE,&sample_data.bit32);
      input_right = (float32_t)(sample_data.bit16[0]);
      sample_data.bit32 = ((int16_t)(20*output[k]));
      SSIDataPut(SSI1_BASE,sample_data.bit32);
      SSIDataPut(SSI0_BASE,sample_data.bit32);
    }
  }
}
```

2.6.5 Running the Program

Build and run this program. Verify that the output consists of the 2-kHz carrier signal and two sideband signals as shown in Figure 2.31. The sideband signals are at the frequency of the carrier signal, +/− the frequency of the baseband 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 project will need to be rebuilt for such a change to take effect.

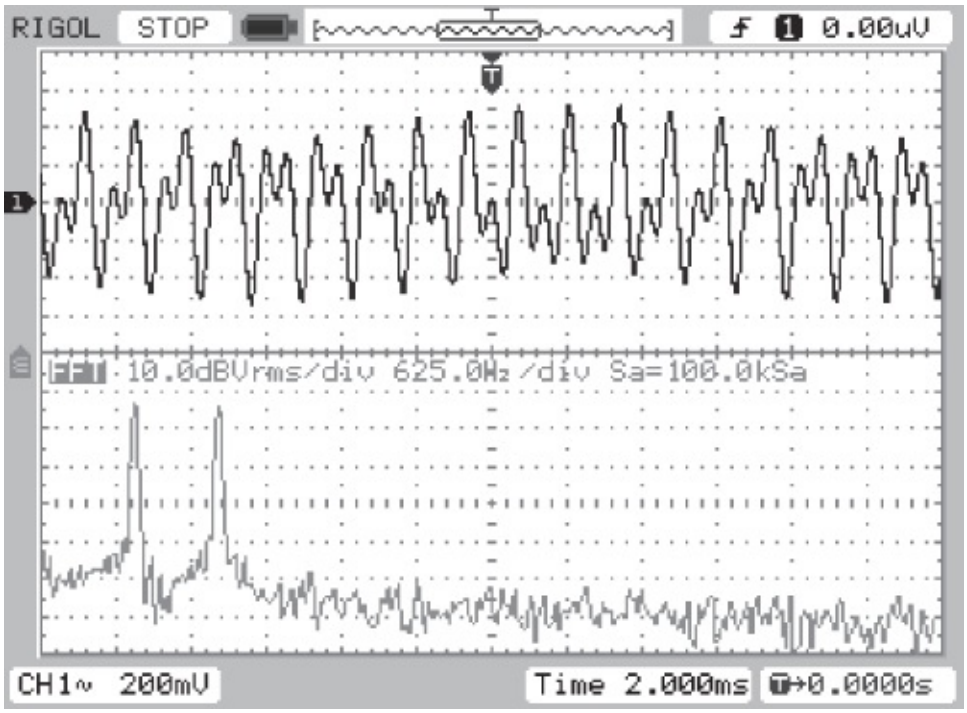


Figure 2.31 Output waveform generated by program `tm4c123_am_intr.c`.

2.7 Identifying the Frequency Response of the DAC Using Pseudorandom Noise

Example 2.22

Frequency Response of the AIC3104 DAC Reconstruction Filter Using a Pseudorandom Binary Sequence (`tm4c123_prbs_intr.c`).

Program `tm4c123_prbs_intr.c`, shown in Listing 2.42, generates a pseudorandom binary sequence (PRBS) and writes this to the AIC3104 DAC. Function `prbs()`, defined in file `tm4c123_aic3104_init.c`, and shown in Listing 2.43, uses a 16-bit linear feedback shift register (LFSR) to generate a maximal-length pseudorandom binary sequence. The least significant bit of the register is used to determine whether function `prbs()` returns either the value `noise_level` or the value `-noise_level`, where `noise_level` is a 16-bit integer value passed to the function. The value of the LFSR (`lfsr`) is initialized to `0x0001`. Each time function `prbs()` is called, a feedback value (`fb`) is formed by the modulo-2 sum of bits 15, 14, 3, and 1 in `lfsr`. The contents of the LFSR are shifted left by one bit and the value of bit 0 is assigned the feedback value `fb`. If the value of `fb` is equal to zero, then the function returns the value `-noise_level`. Otherwise, it returns the value `noise_level`.

Listing 2.20 Program `tm4c123_prbs_intr.c`

```
// tm4c123_prbs_intr.c
#include “tm4c123_aic3104_init.h”
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  float32_t input_left, input_right;
  float32_t output_left, output_right;
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  input_left = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  input_right = (float32_t)(sample_data.bit16[0]);
  output_left =  (float32_t)(prbs(8000));
  output_right = output_left;
  sample_data.bit32 = ((int16_t)(output_left));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  sample_data.bit32 = ((int16_t)(output_right));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
}
int main(void)
{
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1){}
}
```

Listing 2.21 Definition of function `prbs()`, defined in file `tm4c123_aic3104_init.c`

```
typedef union
{
  uint16_t value;
  struct
  {
    unsigned char bit0 : 1;
    unsigned char bit1 : 1;
    unsigned char bit2 : 1;
    unsigned char bit3 : 1;
    unsigned char bit4 : 1;
    unsigned char bit5 : 1;
    unsigned char bit6 : 1;
    unsigned char bit7 : 1;
    unsigned char bit8 : 1;
    unsigned char bit9 : 1;
    unsigned char bit10 : 1;
    unsigned char bit11 : 1;
    unsigned char bit12 : 1;
    unsigned char bit13 : 1;
    unsigned char bit14 : 1;
    unsigned char bit15 : 1;
  } bits;
} shift_register;
shift_register sreg = {0x0001};
short prbs(int16_t noise_level)
{
  char fb;
  fb =((sreg.bits.bit15)+(sreg.bits.bit14)+(sreg.bits.bit3)
      +(sreg.bits.bit1))%2
  sreg.value = sreg.value << 1;
  sreg.bits.bit0 = fb;
  if(fb ⌡ 0)
    return(-noise_level);
  else
    return(noise_level);
}
```

Figure 2.32 shows the analog output signal generated by the program displayed using an oscilloscope and using *Goldwave*. The theoretical power spectral density of a PRBS is constant across all frequencies. When a PRBS is written to a DAC, the spectral content of the signal output by the DAC is indicative of the magnitude frequency response of the DAC. In this case, the magnitude frequency response is flat, or constant, at frequencies up to the cutoff frequency of the DAC's reconstruction filter just below half its sampling frequency.

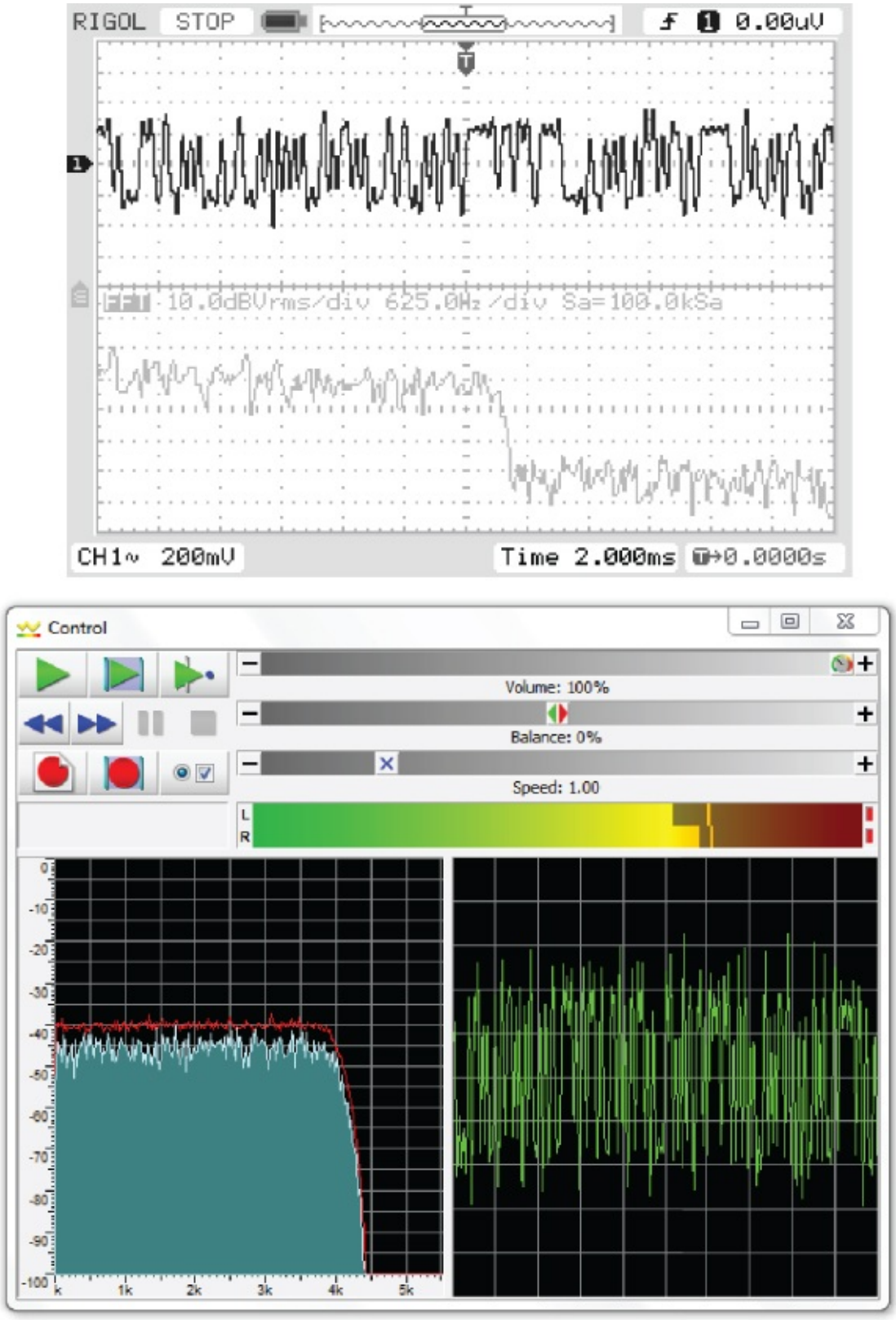


Figure 2.32 Output from program `tm4c123_prbs_intr.c` viewed using *Rigol DS1052* oscilloscope and *Goldwave*.

2.7.1 Programmable De-Emphasis in the AIC3104 Codec

The AIC3104 codec features a programmable first-order de-emphasis filter that may optionally

be switched into the signal path just before the DAC. Program `tm4c123_prbs_deemph_intr.c` demonstrates its use (see Figure 2.33). User switch SW1 on the TM4C123 LaunchPad may be used to enable or disable the de-emphasis function. 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*) using function `I2CRegWrite()`.

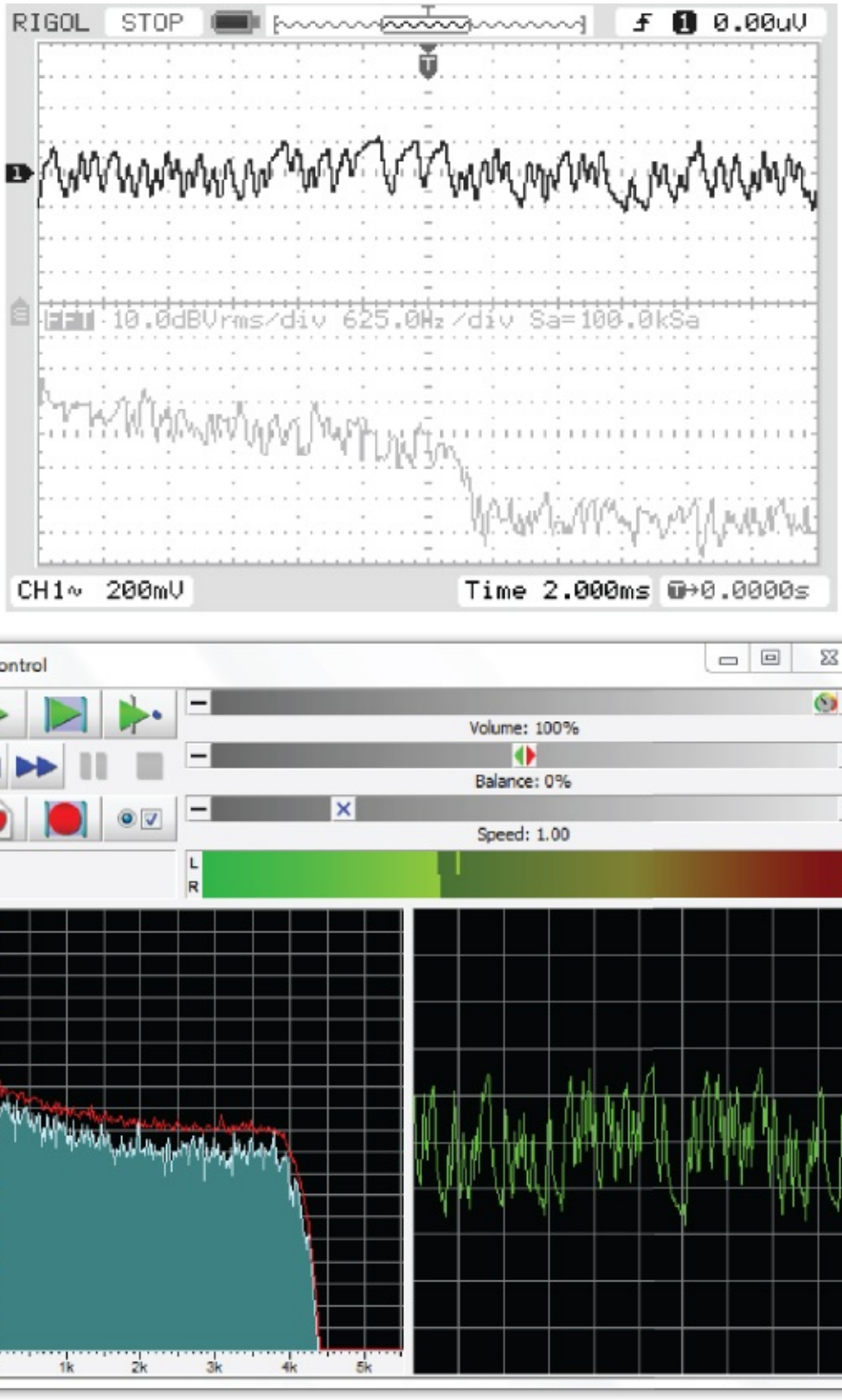


Figure 2.33 Output from program `tm4c123_prbs_deemph_intr.c` viewed using *Rigol DS1052* oscilloscope and *Goldwave*.

The coefficients of the de-emphasis filter can be reprogrammed in order to implement a different first-order IIR filter. In program `tm4c123_prbs_hpf_intr.c`, a high-pass filter has been implemented and its characteristics are apparent in Figure 2.34.

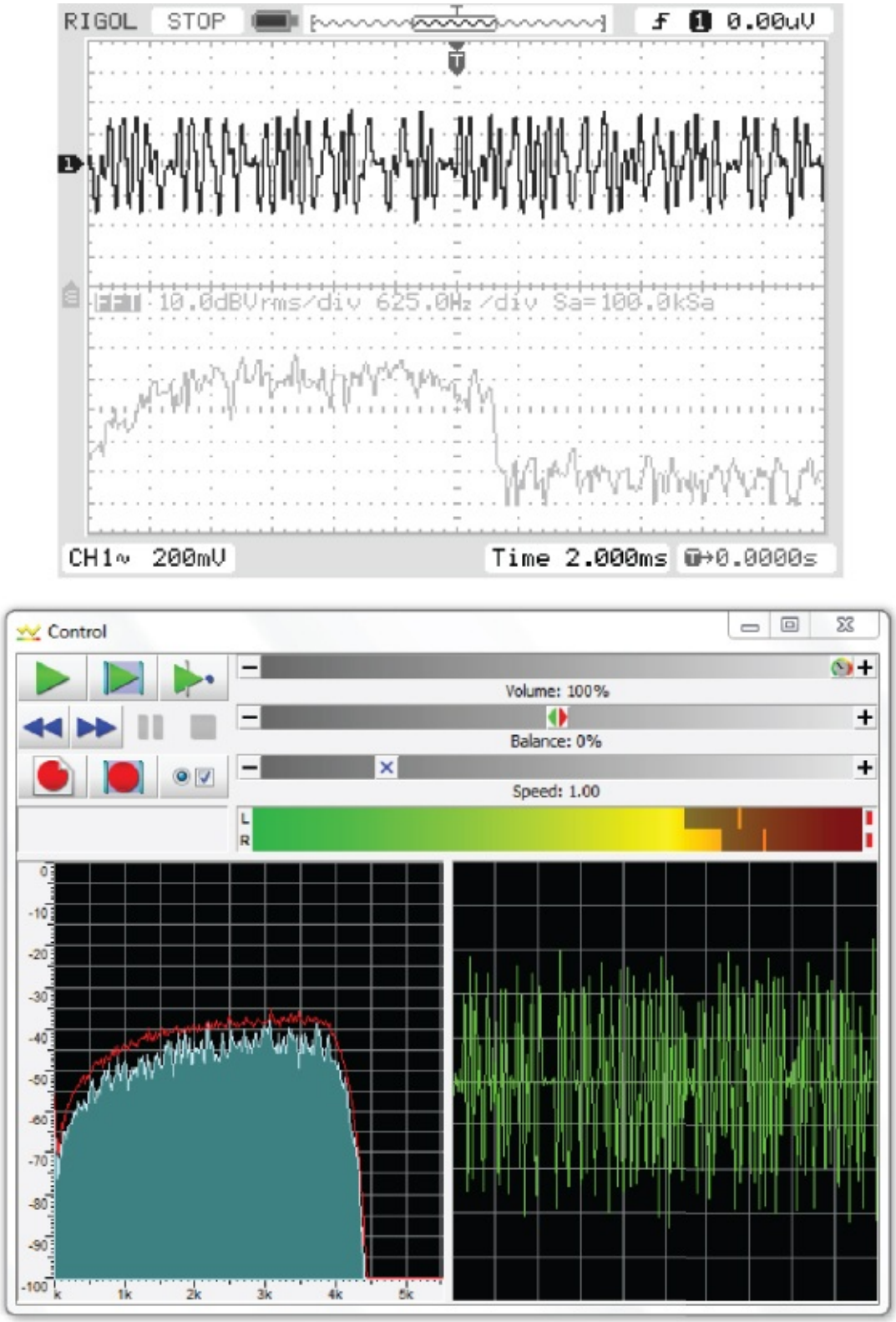


Figure 2.34 Output from program `tm4c123_prbs_hpf_intr.c` viewed using *Rigol DS1052* oscilloscope and *Goldwave*.

2.7.2 Programmable Digital Effects Filters in the AIC3104 Codec

The AIC3104 codec also features two fourth-order IIR filters that may optionally be switched into the left and right channel signal paths just before the DAC. Program

`tm4c123_prbs_biquad_intr.c` demonstrates how the characteristics of these filters can be programmed by writing filter coefficients to AIC3104 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 2.35 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 4.

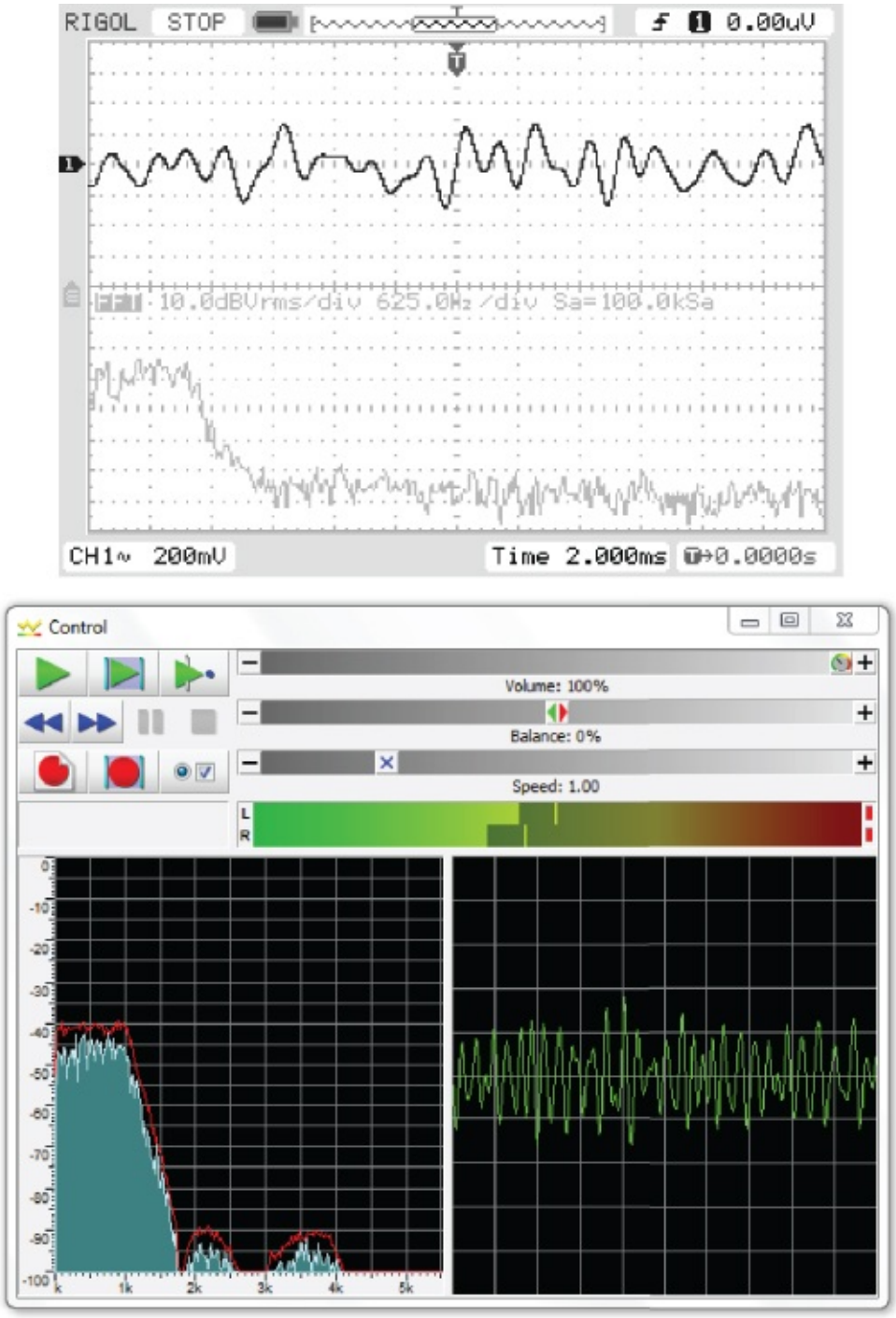


Figure 2.35 Output from program `tm4c123_prbs_biquad_intr.c` viewed using *Rigol DS1052* oscilloscope and *Goldwave*.

Example 2.23

Frequency response of the DAC reconstruction filter using pseudorandom noise (`tm4c123_prandom_intr.c`).

Program `tm4c123_prandom_intr.c` is similar to program `tm4c123_prbs_intr.c` except that it uses a Parks–Miller algorithm to generate a pseudorandom noise sequence. This may be used as an alternative to PRBS in some applications. Function `prand()` (Listing 2.45) is defined in file `tm4c123_aic3104_init.c` and returns pseudorandom values in the range +/− `noise_level`, where `noise_level` is a 16-bit integer value passed to the function. Figure 2.36 shows the waveform output by program `tm4c123_prandom_intr.c` displayed using an oscilloscope. Compare this with the output waveform generated by program `tm4c123_prbs_intr.c`, shown in Figure 2.32.

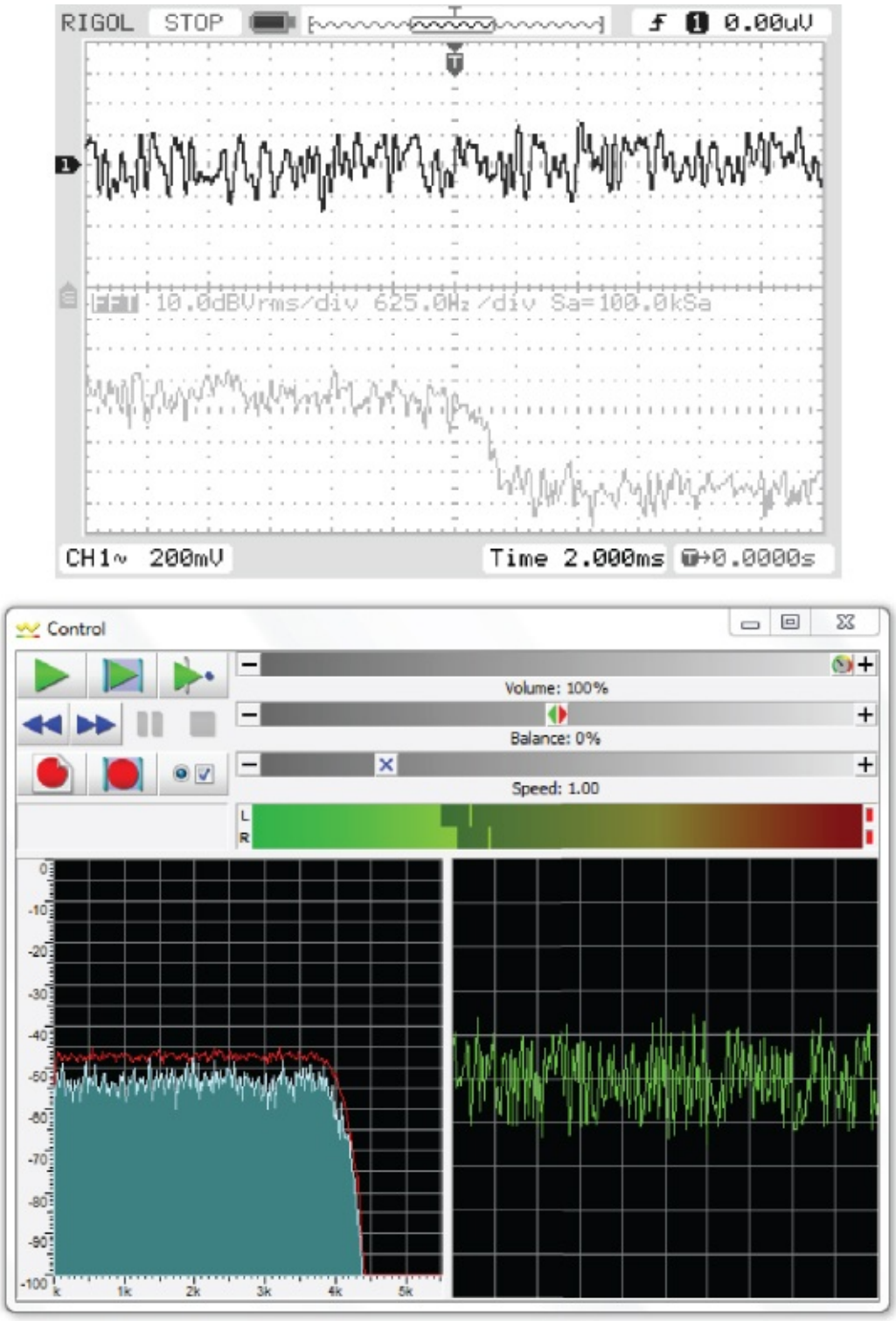


Figure 2.36 Output from program `tm4c123_prandom_intr.c` viewed using *Rigol DS1052 oscilloscope*.

Overall, the use of pseudorandom noise as an input signal is a simple and useful technique for assessing the magnitude frequency response of a system.

Listing 2.22 Definition of function `prand()`, defined in file `tm4c123_aic3104_init.c`

```
uint32_t prand_seed = 1;       // used in function prand()
uint32_t rand31_next()
{
  uint32_t hi, lo;
  lo = 16807 * (prand_seed & 0xFFFF);
  hi = 16807 * (prand_seed >> 16);
  lo += (hi & 0x7FFF) << 16;
  lo += hi >> 15;
  if (lo > 0x7FFFFFFF) lo -= 0x7FFFFFFF;
  return(prand_seed = (uint32_t)lo);
}
int16_t prand(void)
{
return ((int16_t)(rand31_next()>>18)-4096);
}
```

2.8 Aliasing

The preceding examples demonstrate that neither the AIC3104 codec nor the WM5102 codec can generate signal components that have frequencies greater than half their sampling frequency. It follows that it is inadvisable to allow analog input signal components that have 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 prior to sampling. Antialiasing filters with characteristics similar to those of the reconstruction filters in the DACs of the AIC3104 and WM5102 codecs are incorporated into these devices.

Example 2.24

Step response of the WM5102 codec antialiasing filter (`stm32f4_loop_buf_intr.c`).

In order to investigate the step response of the antialiasing filter on the WM5102, connect a signal generator to the left channel of the (pink) LINE IN socket on the Wolfson audio card. Adjust the signal generator to give a square wave output of frequency 270 Hz and amplitude 500 mV. Build and run program `stm32f4_loop_buf_intr.c`, halting the program after a few seconds. View the most recent input sample values by saving the contents of array `lbuffer` to a data file by typing

```
save <filename> <start address>, <start address + 0x400>
```

at the *Command* line in the *MDK-ARM debugger*, where `start address` is the address of array `lbuffer`, and plotting the contents of the data file using MATLAB function `stm32f4_plot_real()`. You should see something similar to that shown in Figure 2.38. Figure 2.37 shows the square wave input signal corresponding to the sample values shown in Figure 2.38. The ringing that follows each transition in the waveform represented by the samples is due to the antialiasing filter. Figure 2.39 shows the corresponding result obtained using program `tm4c123_loop_buf_intr.c`. The antialiasing filters in each of the codecs have characteristics similar to those of their corresponding reconstruction filters. Compare Figures 2.37 and 2.39 with Figures 2.23 and 2.25. The ac coupling of the LINE IN connections on both the Wolfson audio card and the audio booster pack is evident from the drooping of the signal level between transitions in Figures 2.37 and 2.39.

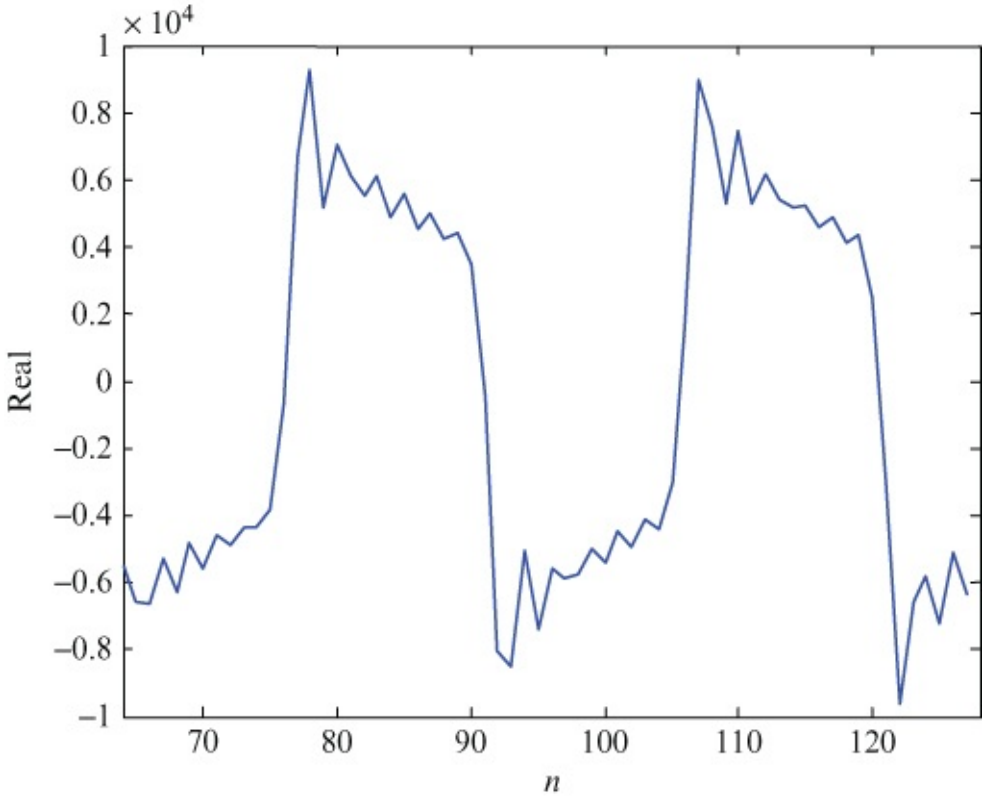


Figure 2.38 Sample values read from the WM5102 ADC and stored in array `lbuffer` by program `stm32f4_loop_buf_intr.c`.

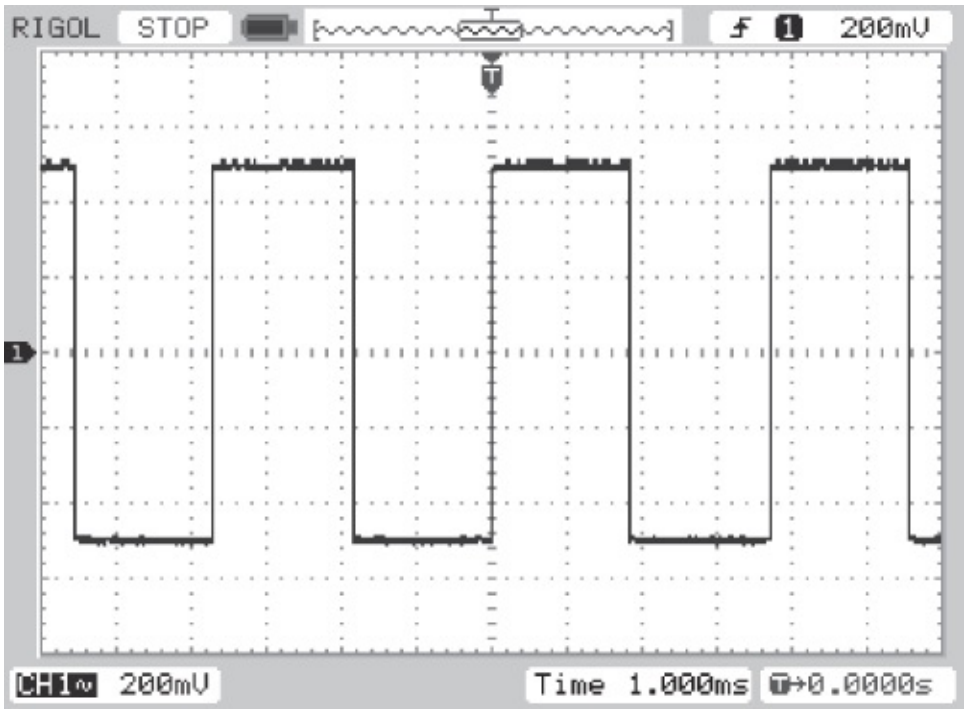


Figure 2.37 Square wave input signal used with program `stm32f4_loop_buf_intr.c`.

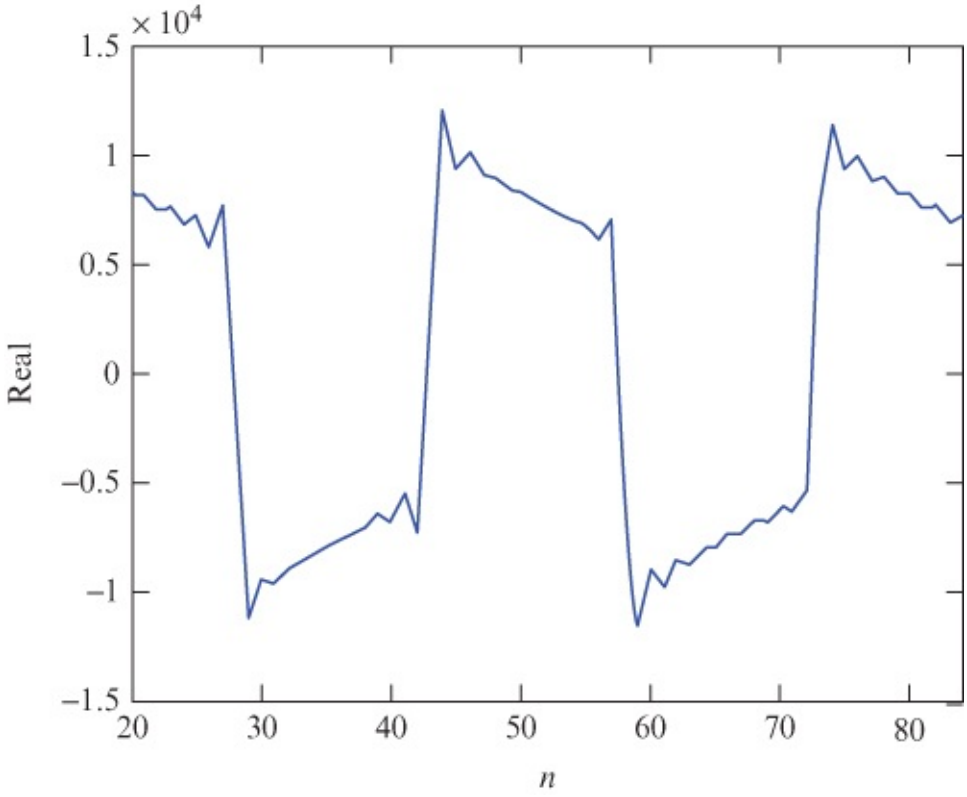


Figure 2.39 Sample values read from the WM5102 ADC and stored in array `lbuffer` by program `tm4c123_loop_buf_intr.c`.

Example 2.25

Demonstration of the Characteristics of the AIC3104 Codec Antialiasing Filter (`tm4c123_sine48_loop_intr.c`).

Program `tm4c123_sine48_loop_intr.c` is similar to program `tm4c123_sine48_intr.c` in that it generates a 1-kHz sine wave output using sample values read from a lookup table. It differs in that it reads input samples from the ADC and stores the 128 most recent in array `buffer`. Connect the (black) LINE OUT connection on the audio booster card to the (blue) LINE IN connection using a 3.5-mm jack plug to 3.5-mm jack plug cable and run the program for a short length of time. Halt the program and save the contents of array `buffer` to a data file by typing

```
save <filename> <start address>, <start address + 0x200>
```

at the *Command* line in the *MDK-ARM debugger*, where `start address` is the address of array `buffer`. Plot the contents of the data file using MATLAB function `tm4c123_plot_real()`. You should see something similar to the plot shown in Figure 2.40. The discontinuity at time $t = 5.8$ ms corresponds to the value of the index variable `bufptr` when the program was halted. At first sight, this figure may appear unremarkable. However, when you consider that the analog waveform output by the AIC3104 codec contains out-of-band-noise (readily visible on an oscilloscope as shown in Figure 2.19), it is apparent that the samples stored in array `buffer` are of a low-pass filtered version of that signal. That low-pass filtering operation has been carried out by the antialiasing filter in the codec.

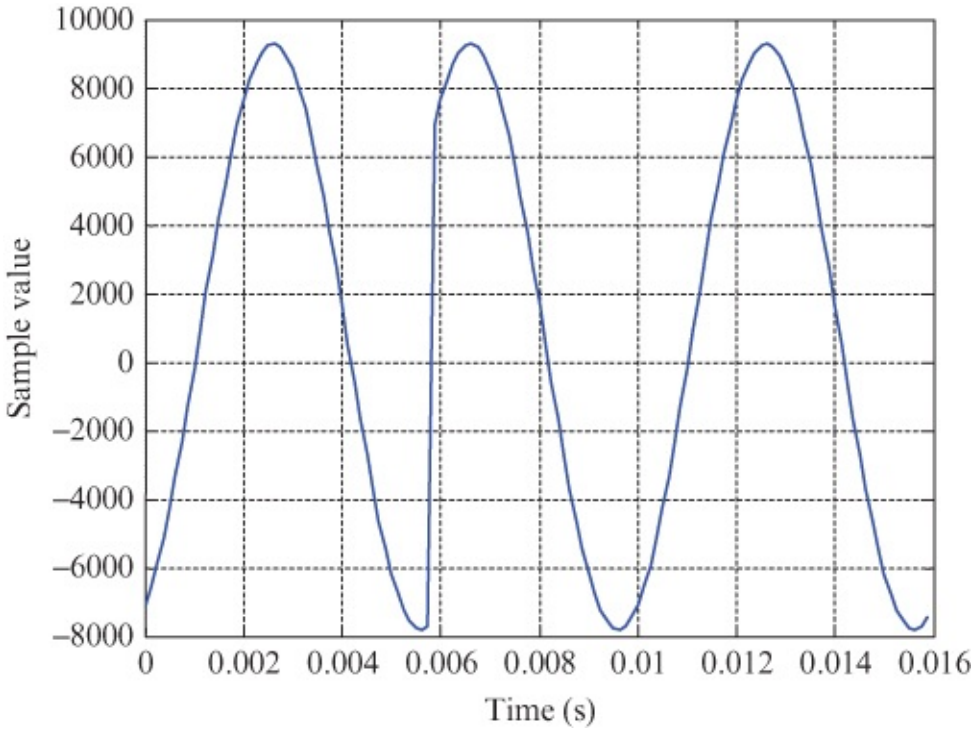


Figure 2.40 Sample values read from the AIC3104 ADC and stored in array `buffer` by program `tm4c123_sine48_loop_intr.c`.

Example 2.26

Demonstration of Aliasing (`tm4c123_aliasing_intr.c`).

The analog and digital antialiasing filters in the AIC3104 and WM5120 codecs cannot be bypassed or disabled. However, aliasing can be demonstrated within a program by downsampling a digital signal *without* taking appropriate antialiasing measures. Program `tm4c123_aliasing_intr.c`, shown in Listing 2.49 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 `tm4c123_aliasing_intr.c` uses an FIR filter with 65 coefficients defined in header file `lp6545.h` for this task. For the purposes of this example, it is unnecessary to understand the operation of the FIR filter. It is sufficient to consider simply that the program demonstrates the effect of sampling at a frequency of 8 kHz with and without using an antialiasing filter.

Listing 2.23 Program `tm4c123_aliasing_intr.c`

```
// tm4c123_aliasing_intr.c
#include “tm4c123_aic3104_init.h”
#include “lp6545.h”
#define DISCARD 0
#define SAMPLE 1
#define BLOCKSIZE 1
volatile int16_t flag = DISCARD;
int16_t antialiasing = 0;
float32_t xin[BLOCKSIZE], yin[BLOCKSIZE];
float32_t xout[BLOCKSIZE], yout[BLOCKSIZE];
float32_t stateout[N+BLOCKSIZE-1];
float32_t statein[N+BLOCKSIZE-1];
arm_fir_instance_f32 Sin, Sout;
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  float32_t input_left, input_right;
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  input_left = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  input_right = (float32_t)(sample_data.bit16[0]);
  xin[0] = input_left;
  arm_fir_f32(&Sin, xin, yin, BLOCKSIZE);
  if (flag ⌡ DISCARD)
  {
    flag = SAMPLE;
    xout[0] = 0.0f;
  }
  else
  {
    flag = DISCARD;
    if (antialiasing ⌡ 0)
      xout[0] = yin[0];
    else
      xout[0] = input_left;
  }
  arm_fir_f32(&Sout, xout, yout, BLOCKSIZE);
  sample_data.bit32 = ((int16_t)(yout[0]));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  sample_data.bit32 = ((int16_t)(0));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
}
int main(void)
{
  arm_fir_init_f32(&Sin, N, h, statein, BLOCKSIZE);
  arm_fir_init_f32(&Sout, N, h, stateout, BLOCKSIZE);
  tm4c123_aic3104_init(FS_16000_HZ,
```

```
                        AIC3104_LINE_IN,
                        IO_METHOD_INTR,
                        PGA_GAIN_6_DB);
  while(1)
  {
    ROM_SysCtlDelay(10000);
    if (!GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_4))
    {
      ROM_SysCtlDelay(10000);
      antialiasing = (antialiasing+1)
      while (!GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_4)){}
    }
  }
}
```

2.8.1 Running the Program

Build, load, and run program `tm4c123_aliasing_intr.c`. Connect a signal generator to the (blue) LINE IN socket on the audio booster pack and either connect an oscilloscope to one of the scope hooks on the audio booster pack or use *GoldWave* via the (black) LINE OUT socket. Vary the frequency of a sinusoidal input signal between 0 and 8 kHz. Press user switch SW1 on the TM4C123 LaunchPad to switch the antialiasing filter implemented by the program on and off.

With the antialiasing filter enabled, signals with frequencies greater than 4 kHz do not pass from LINE IN to LINE OUT. But with the antialiasing filter disabled, and by varying the frequency of the input signal, you should be able to verify that sinusoids with frequencies between 4 and 8 kHz are “folded back” into the frequency range 0–4 kHz.

2.9 Identifying The Frequency Response of the DAC Using An Adaptive Filter

Example 2.27

Identification of AIC3104 codec bandwidth using an adaptive filter (`tm4c123_sysid_CMSIS_intr.c`).

Another way of observing the limited bandwidth of the codec is to measure its magnitude frequency response using program `tm4c123_sysid_CMSIS_intr.c`. This program, shown in Listing 2.51 uses an adaptive FIR filter and is described in more detail in Chapter 6. However, you need not understand exactly how program `tm4c123_sysid_CMSIS_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 2.41).

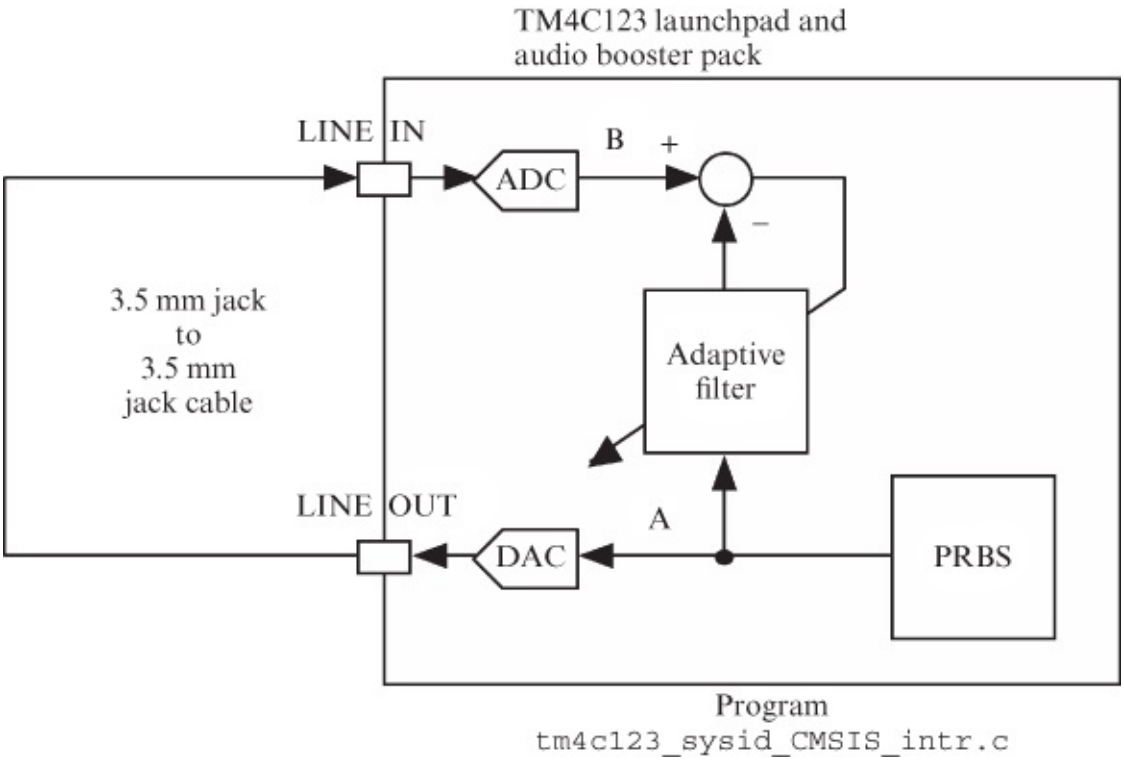


Figure 2.41 Connection diagram for program `tm4c123_sysid_CMSIS_intr.c`.

Listing 2.24 Program `tm4c123_sysid_CMSIS_intr.c`

```
// tm4c123_sysid_CMSIS_intr.c
#include “tm4c123_aic3104_init.h”
#define BETA 1E-11
#define NUM_TAPS 128
#define BLOCK_SIZE 1
float32_t firStateF32[BLOCK_SIZE + NUM_TAPS -1];
float32_t firCoeffs32[NUM_TAPS] = {0.0};
arm_lms_instance_f32 S;
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  float32_t adapt_in, adapt_out, desired;
  float32_t error, input_left, input_right;
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  input_left = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  input_right = (float32_t)(sample_data.bit16[0]);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  adapt_in = (float32_t)(prbs(8000));
  desired = input_left;
  arm_lms_f32(&S, &adapt_in, &desired,
              &adapt_out, &error, 1);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
  sample_data.bit32 = ((int16_t)(adapt_in));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  sample_data.bit32 = ((int16_t)(adapt_in));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
}
int main()
{
  arm_lms_init_f32(&S, NUM_TAPS, firCoeffs32,
                   firStateF32, BETA, BLOCK_SIZE);
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1){}
}
```

2.9.1 Running the Program

Connect the (black) LINE OUT socket on the audio booster pack to the (blue) LINE IN socket input using a 3.5 mm jack plug to 3.5 mm jack plug cable as shown in Figure 2.41. The signal path that will be identified by the program comprises the series combination of the DAC and ADC and the ac-coupling circuits between the converters and the jack socket connections on the audio booster pack. Run the program for a few seconds and then halt it and plot the values

of the weights of the adaptive filter (the identified impulse response of the signal path) by saving the 128 adaptive filter coefficients, stored in array `firCoeffs32`, and using MATLAB function `tm4c123_logfft()`.

Type

```
save <filename> <start address>, <start address + 0x200>
```

at the *Command* line in the *MDK-ARM debugger*, where `start address` is the address of array `firCoeffs32`, and plot using MATLAB function `tm4c123_logfft()`.

The adaptive filter coefficients used by the CMSIS function `arm_lms_f32()` are stored in memory in reverse order relative to the sense in which they represent the impulse response of the filter, and therefore, when using MATLAB function `stm32f4_logfft()`, respond to the prompt

```
forward (0) or reverse (1) order of time-domain samples?
```

by entering the value `1`.

The roll-off of the frequency response at very low frequencies evident in Figure 2.42 is due to the ac coupling between the codec and the 3.5 mm LINE IN and LINE OUT jack sockets. The roll-off of the frequency response at frequencies greater than 3200 Hz is due to the antialiasing and reconstruction filters in the AIC3104 ADC and DAC. A gain of approximately −6 dB due to the potential divider comprising two 5k6 ohm resistors (shown in Figure 2.04) between the (blue) LINE IN socket and LINEIN_L the codec is compensated for by the gain of +6 dB programmed into the PGA that immediately precedes the ADC in the AIC3014 (by passing parameter value `PGA_GAIN_6_DB` to function `tm4c123_aic3104_init()`).

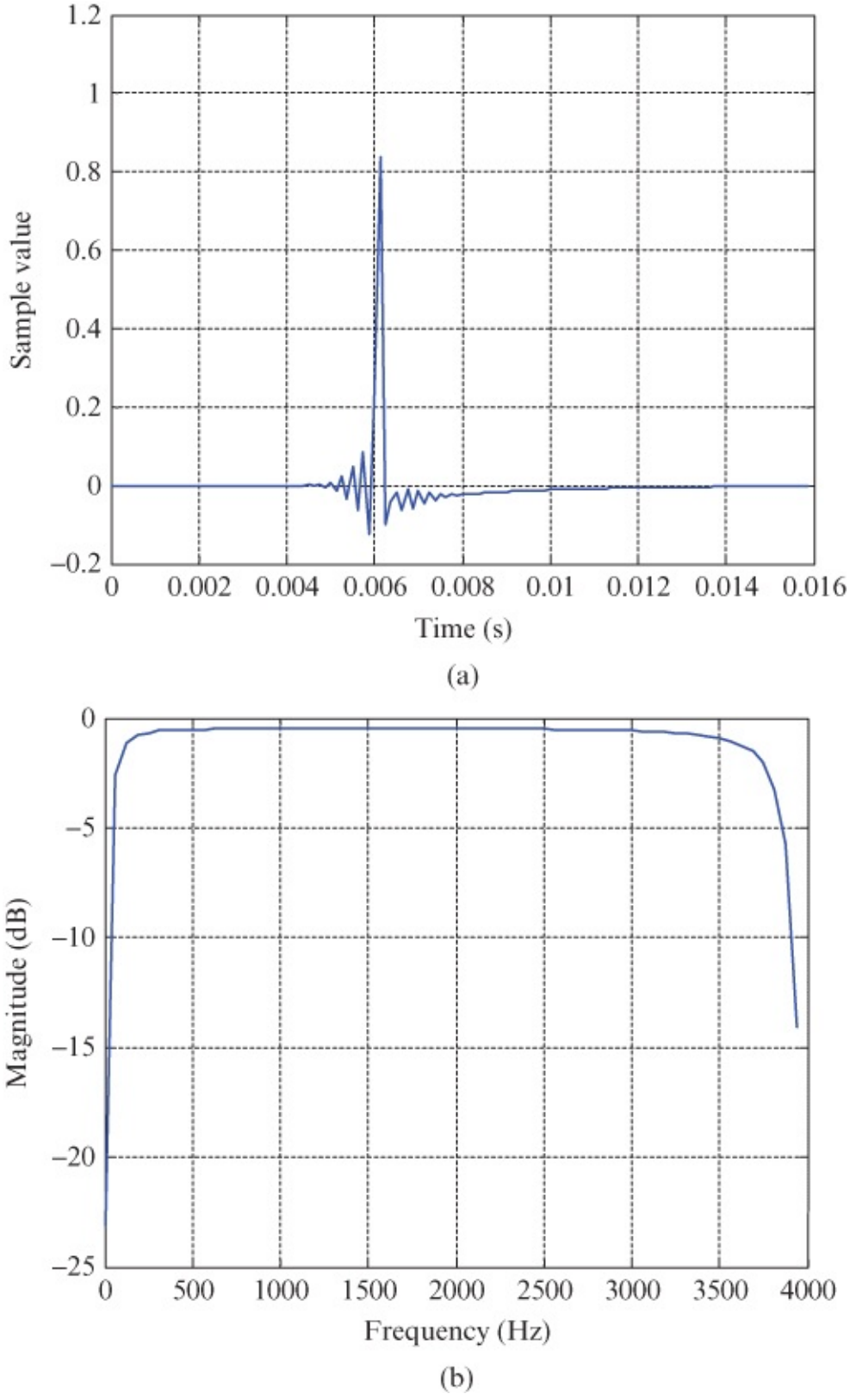
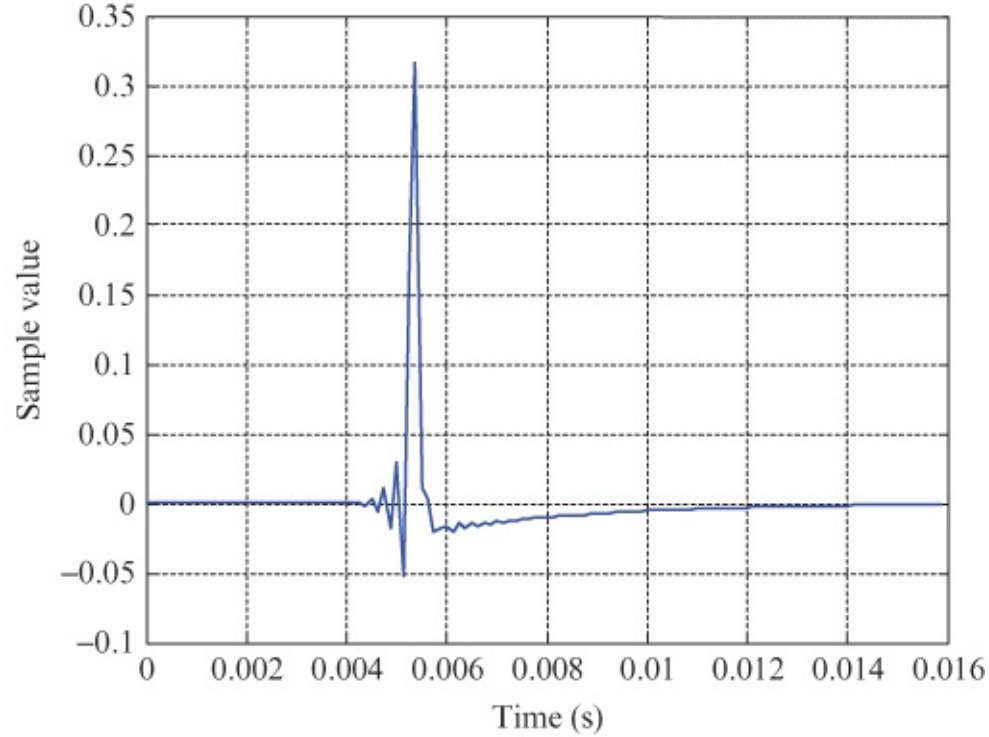


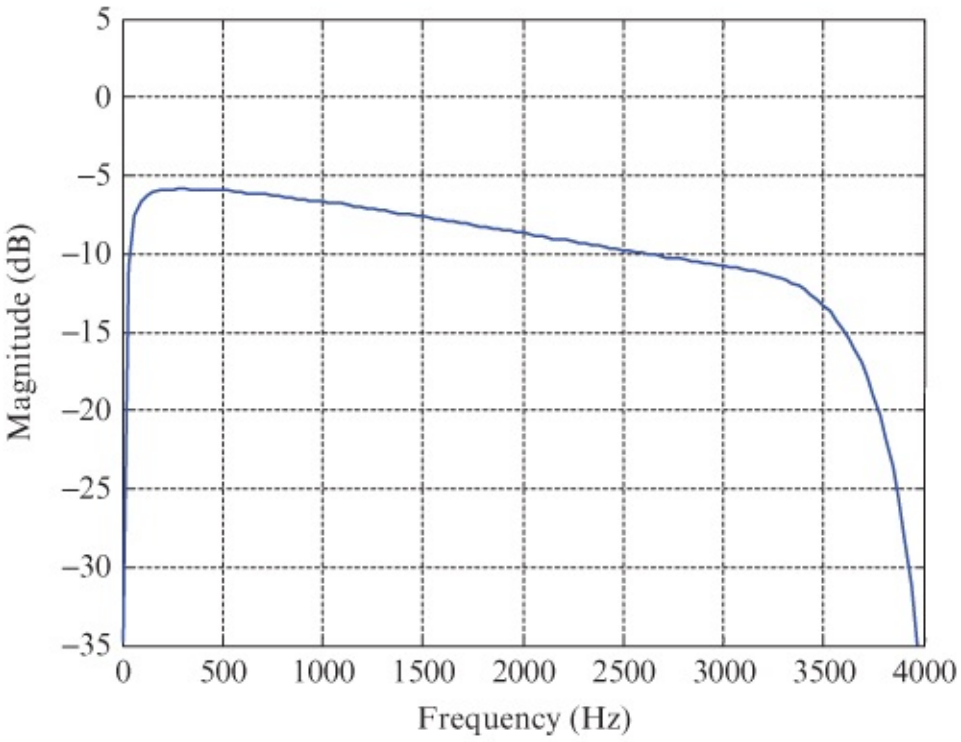
Figure 2.42 The impulse response and magnitude frequency identified using program `tm4c123_sysid_CMSIS_intr.c` with connections as shown in Figure 2.41, displayed using MATLAB function `tm4c123_logfft()`. Sampling frequency 8000 Hz, 128-coefficient adaptive filter.

Program `tm4c123_sysid_CMSIS_intr.c` may be used to identify the characteristics of

another system (as opposed to just a connecting cable) connected between LINE OUT and LINE IN on the audio booster pack. Figure 2.43 shows the result when a first-order low-pass analog filter comprising a capacitor and resistor was used.



(a)



(b)

Figure 2.43 The impulse response and magnitude frequency identified using program `tm4c123_sysid_CMSIS_intr.c` with a first-order low-pass analog filter connected between LINE IN and LINE OUT sockets, displayed using MATLAB function `tm4c123_logfft()`. Sampling frequency 8000 Hz, 128-coefficient adaptive filter.

Program `tm4c123_sysid_deemph_CMSIS_intr.c` is similar to `tm4c123_sysid_CMSIS_intr.c` but enables the digital de-emphasis filter located just before

the DAC in the AIC3104 as in Example 2.45. Figure 2.44 shows the result of running the program. Compare this with Figure 2.33.

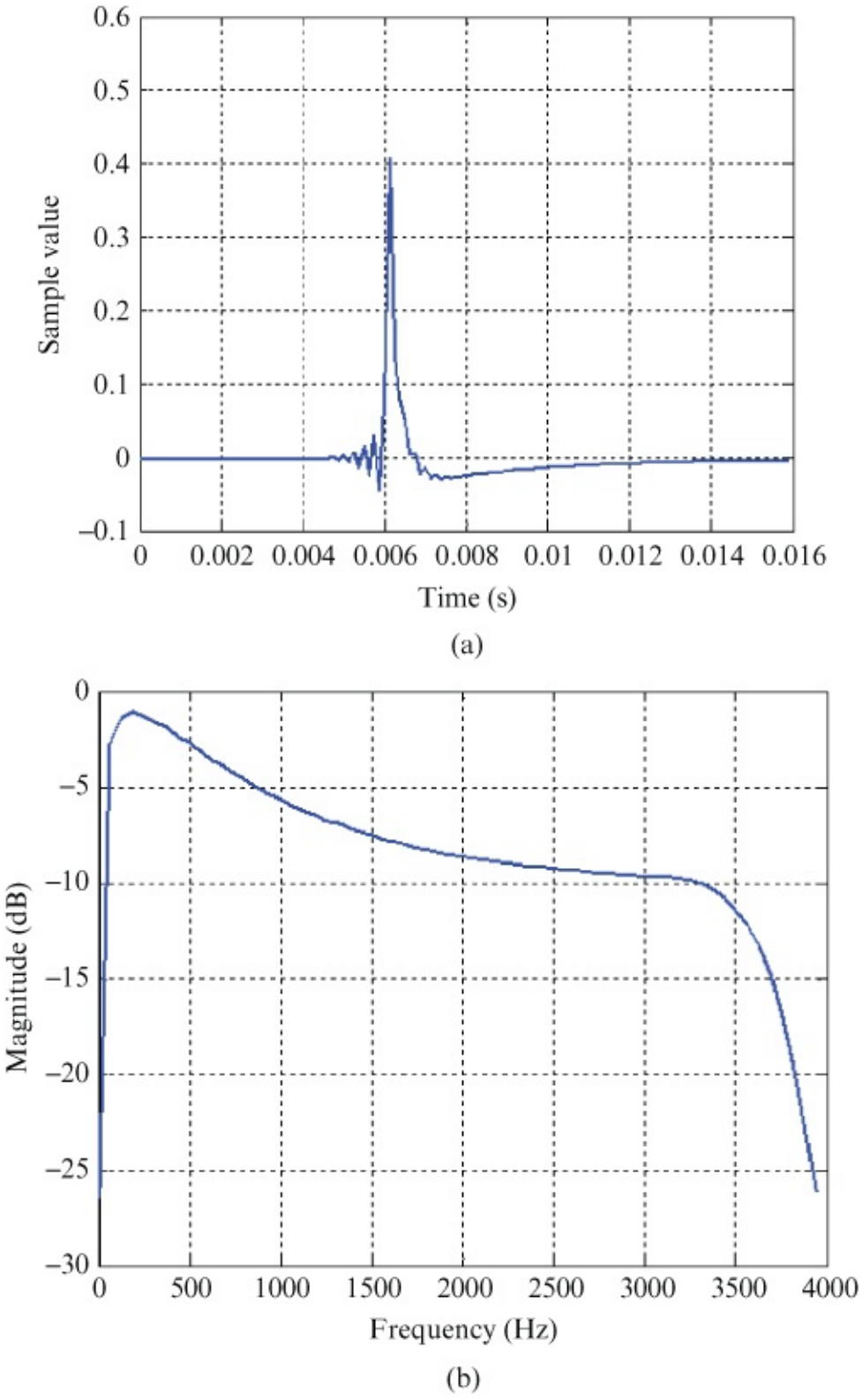


Figure 2.44 The impulse response and magnitude frequency identified using program `tm4c123_sysid_CMSIS_intr.c` with connections as shown in Figure 2.41 and de-emphasis enabled, displayed using MATLAB function `tm4c123_logfft()`. Sampling frequency 8000 Hz, 128-coefficient adaptive filter.

Figure 2.45 shows the result of running program `stm32f4_sysid_CMSIS_intr.c` on the STM32F407 Discovery with a sampling frequency of 8 kHz.

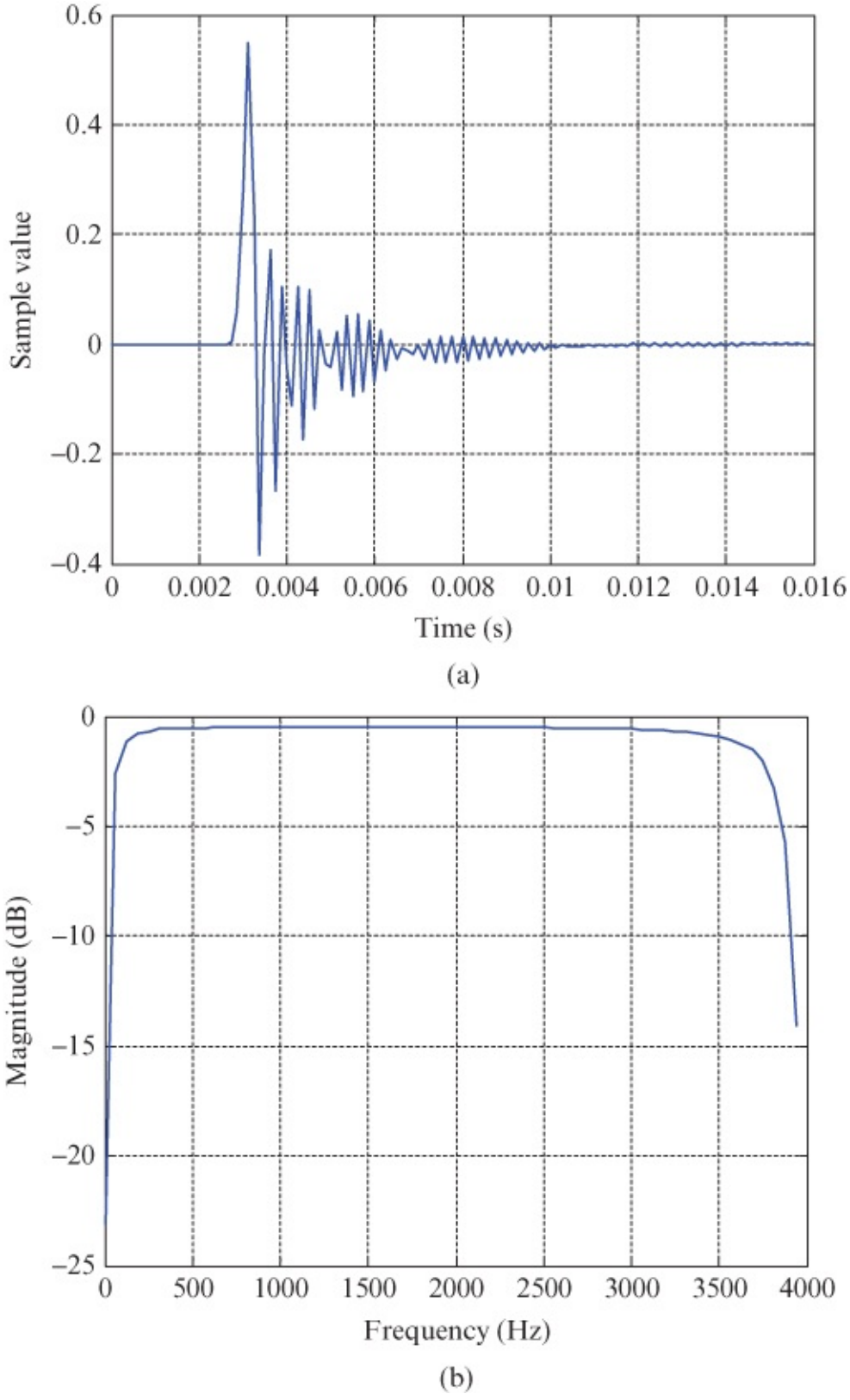


Figure 2.45 The impulse response and magnitude frequency identified using program `stm32f4_sysid_CMSIS_intr.c` with LINE OUT connected directly to LINE OUT, displayed using MATLAB function `stm32f4_logfft()`. Sampling frequency 8000 Hz, 128-coefficient adaptive filter.

The plots illustrating examples of the instantaneous frequency response of program

tm4c123_flanger_intr.c in Example 2.18 were obtained using program tm4c123_sysid_flange_intr.c, a slightly modified version of program tm4c123_sysid_intr.c that includes a flanger (Figure 2.13) with a fixed delay in the signal path identified.

Example 2.28

Identification of AIC3104 Codec Bandwidth Using Two Audio Booster Packs (tm4c123_sysid_CMSIS_intr.c).

Program tm4c123_sysid_CMSIS_intr.c can identify frequency response characteristics in the range 0 to half its sampling frequency (in the previous example, the sampling frequency was equal to 8 kHz) but the antialiasing and reconstruction filters in the codec have a bandwidth only slightly less than this. Hence, in Figures 2.42 through 2.45, only the passbands of those filters are displayed. The following example uses two sets of TM4C123 LaunchPads and audio booster packs, one running program tm4c123_loop_intr.c with a sampling frequency of 8 kHz and the other running program tm4c123_sysid_CMSIS_intr.c using a sampling frequency of 16 kHz. This allows it to identify frequency response characteristics in the range 0–8 kHz and to give a better idea of the passband, stopband, and transition band of the antialiasing and reconstruction filters on the system sampling at 8 kHz. In order to set the sampling frequency in program tm4c123_sysid_CMSIS_intr.c to 16 kHz, change the program statement that reads

```
tm4c123_aic3104_init(FS_8000_HZ,
                     AIC3104_LINE_IN,
                     IO_METHOD_INTR
                     PGA_GAIN_6_DB);
```

to read

```
tm4c123_aic3104_init(FS_16000_HZ,
                     AIC3104_LINE_IN,
                     IO_METHOD_INTR
                     PGA_GAIN_6_DB);
```

Additionally, change the number of adaptive filter coefficients used from 256 to 192 by changing the program statement that reads

```
#define NUM_TAPS 128
```

to read

```
#define NUM_TAPS 192
```

This is necessary due to the increased delay through two systems. Connect the two audio booster packs together as shown in Figure 2.46. Make sure that program

`tm4c123_loop_intr.c` is running on one launchpad before running program `tm4c123_sysid_CMSIS_intr.c` for a short time on the other. Also, make sure that program `tm4c123_loop_intr.c` is using LINE IN (as opposed to MIC IN) as its input. After running and halting the program, save the 192 adaptive filter coefficients `firCoeffs32` used by program `tm4c123_sysid_CMSIS_intr.c` to a data file by typing

```
save filename <start address>, <start address + 0x300>
```

at the *Command* line in the *MDK-ARM debugger*, where `start address` is the address of array `firCoeffs32`, and plot using MATLAB function `tm4c123_logfft()`. You should see something similar to the plots shown in Figure 2.47.

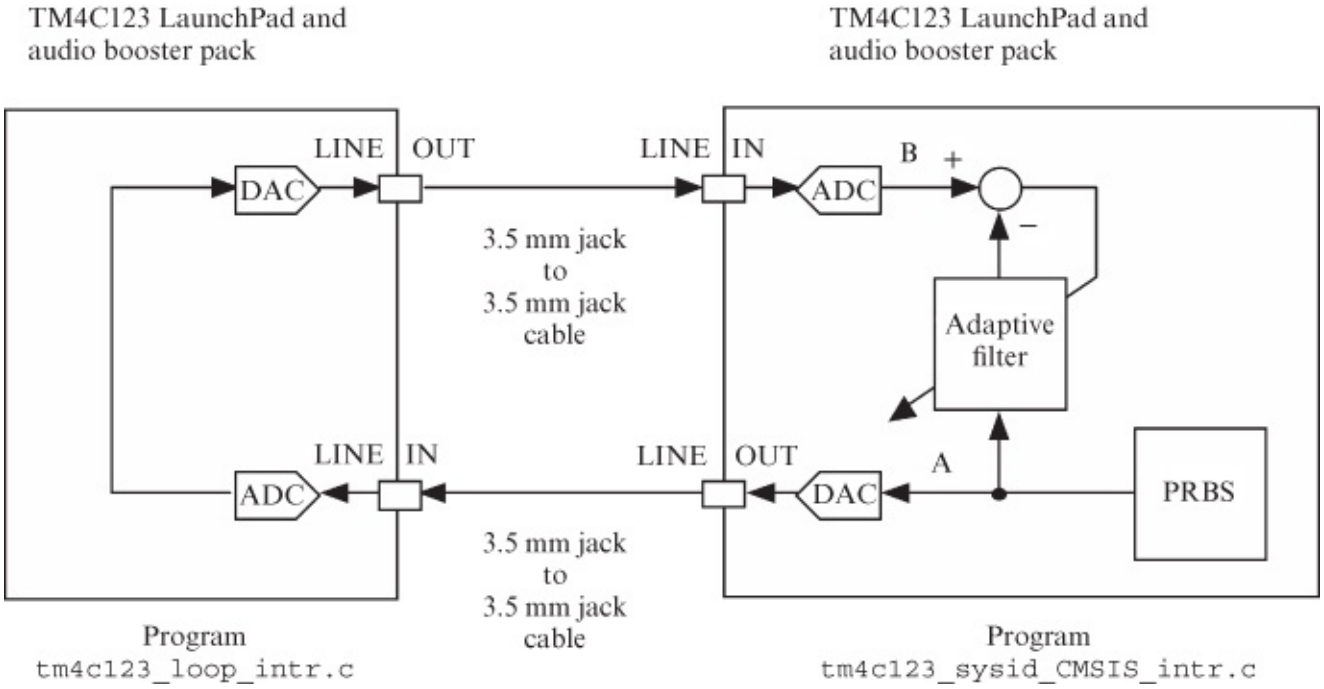


Figure 2.46 Connection diagram for program `tm4c123_sysid_CMSIS_intr.c`.

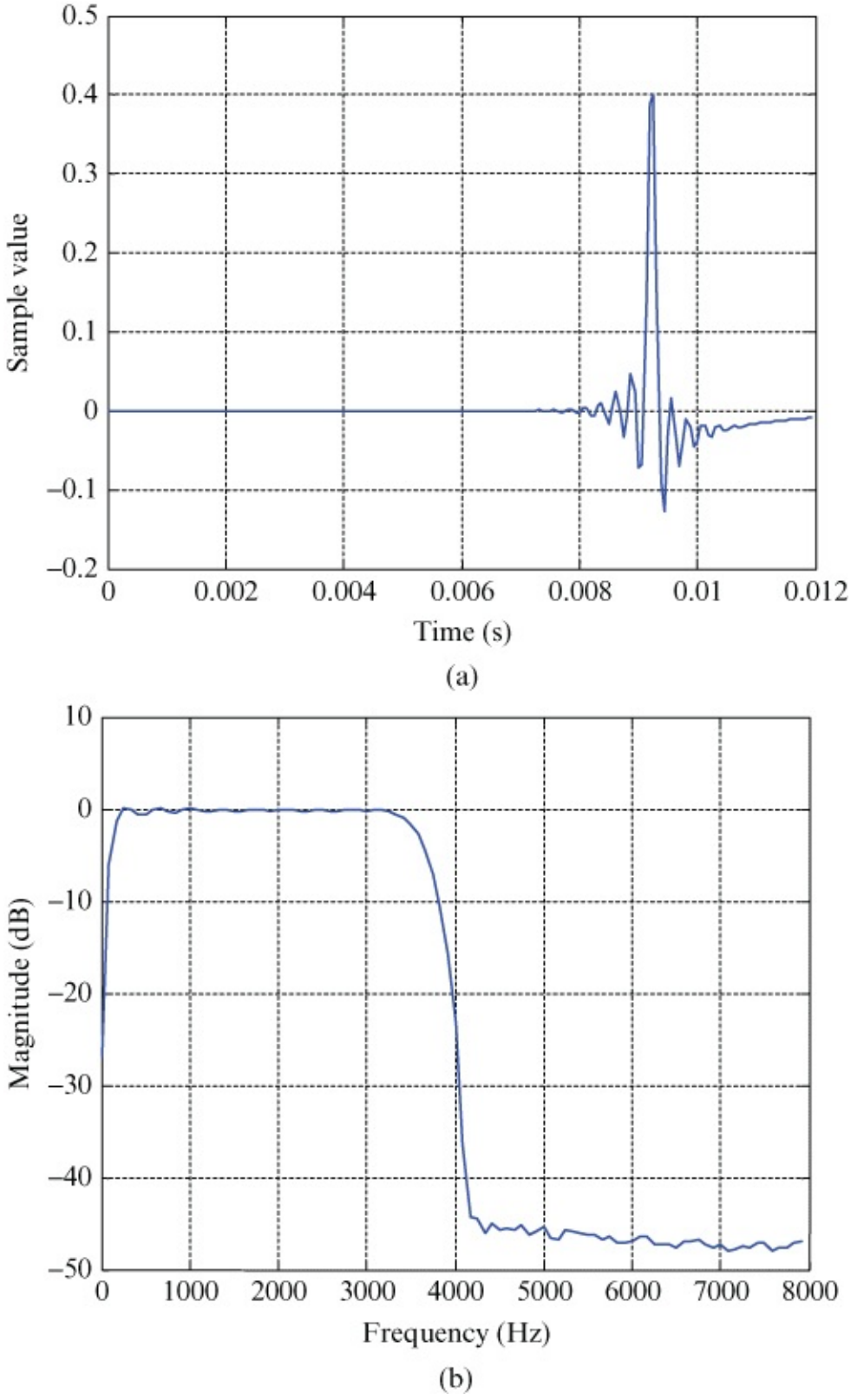


Figure 2.47 The impulse response and magnitude frequency identified using program `tm4c123_sysid_CMSIS_intr.c` with connections as shown in Figure 2.46, displayed using MATLAB function `tm4c123_logfft()`. Sampling frequency 16,000 Hz, 192-coefficient adaptive filter.

The delay in the signal path identified (9 ms) is greater than that in Figure 2.42 (6 ms). This is

consistent with the delay observed using program `tm4c123_loop_intr.c` in Example 2.6. Using 192 filter coefficients, the adaptive filter is able to identify an impulse response up to 192/16,000 = 12 ms long. Implementing a 192-coefficient adaptive filter at a sampling frequency of 16 kHz is just possible using a TM4C123 processor with a clock frequency of 84 MHz. This is evident from the oscilloscope trace shown in Figure 2.48, which shows the rectangular pulse output on GPIO pin PE2 by program `tm4c123_sysid_CMSIS_intr.c`. GPIO pin PE2 is held high during interrupt service routine function `SSI_interrupt_routine()`.

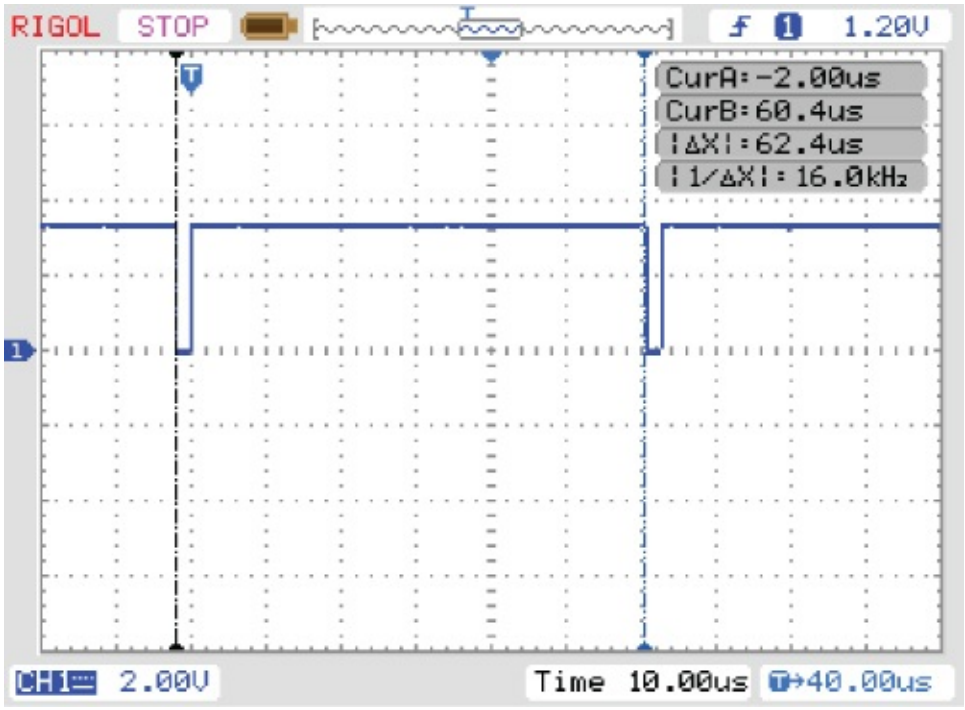


Figure 2.48 Pulse output on GPIO pin PE2 by program `tm4c123_sysid_CMSIS_intr.c` running at a sampling rate of 16 kHz and using 192 adaptive filter coefficients.

2.10 Analog Output Using the STM32F407'S 12-BIT DAC

Example 2.29

Analog Waveform Generation Using a 12-Bit Instrumentation DAC (`stm32f4_sine8_DAC12_intr.c`).

The STM32F407 processor features two 12-bit DACs that allow a comparison to be made with the DACs in the AIC3104 and WM5102 audio codecs. The analog output from one of the 12-bit DACs is routed to GPIO pin PA5 on the STM32F407 Discovery.

Programs `stm32f4_sine8_DAC12_intr.c`, `stm32f4_dimpulse_DAC12_intr.c`, `stm32f4_prbs_DAC12_intr.c`, and `stm32f4_square_DAC12_intr.c` are functionally equivalent to programs `stm32f4_sine8_intr.c`, `stm32f4_dimpulse_intr.c`, `stm32f4_prbs_intr.c`, and `stm32f4_square_intr.c` except in that they use the

STM32F407's 12-bit DAC in place of that in the WM5102 codec. The resultant output waveforms are shown in Figures 2.49 through 2.50. Figure 2.49 shows the analog output signal generated by program stm32f4_sine8_DAC12_intr.c (Listing 2.54) writing eight sample values representing one cycle of a sinusoid to the 12-bit DAC. The frequency domain clearly shows frequency components at 1 kHz, 7 kHz, 9 kHz, 15 kHz, 23 kHz, and 25 kHz. The magnitudes of these components are modulated by a sinc function with nulls at integer multiples of 8 kHz, corresponding to the 125 μs duration, rectangular impulse response of the DAC. Note that the frequency-domain representation of a 1 kHz sinusoid sampled at a rate of 8 kHz is an infinite sequence of components of equal magnitudes at frequencies $(8n \pm 1)$ kHz, where n is an integer $-\infty < n < \infty$.

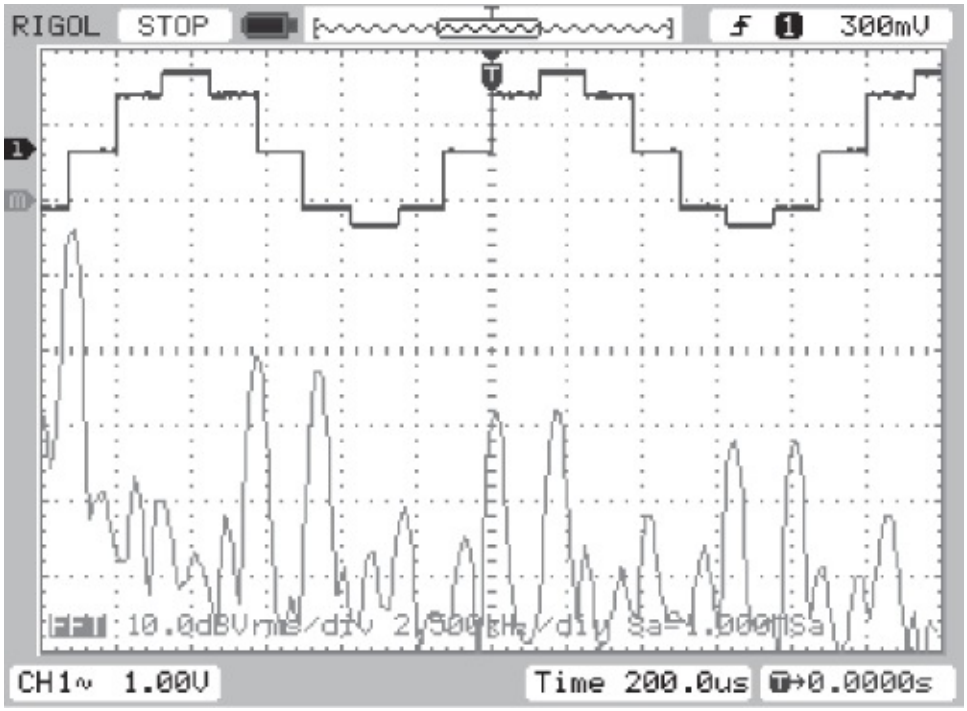


Figure 2.49 Output from program `stm32f4_sine8_dac12_intr.c` viewed using *Rigol DS1052 oscilloscope*.

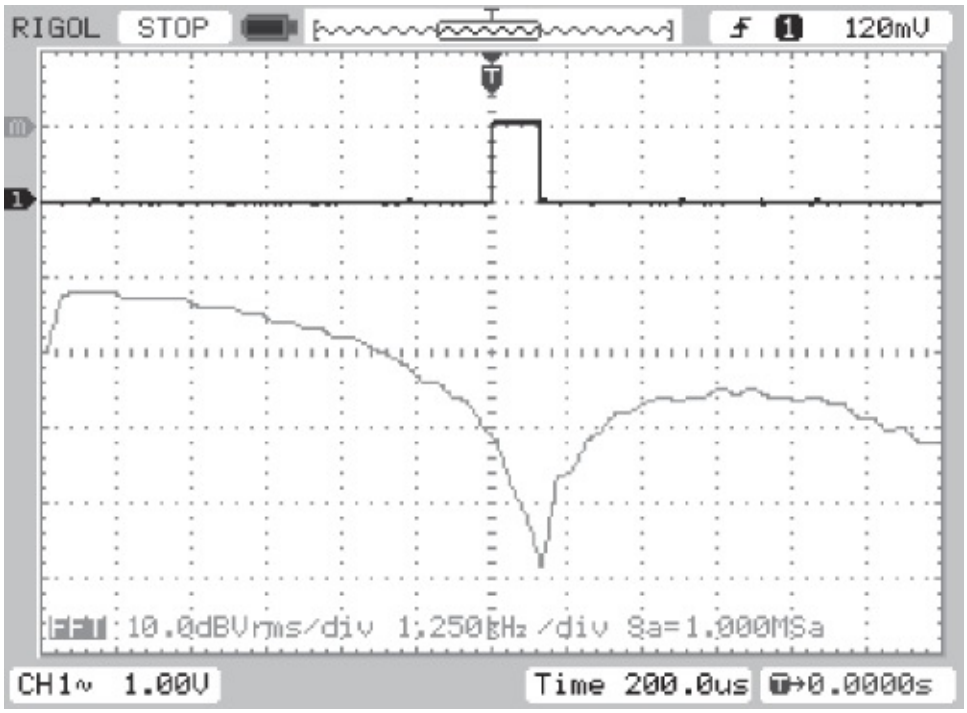


Figure 2.50 Output from program `stm32f4_dimpulse_dac12_intr.c` viewed using *Rigol DS1052 oscilloscope*.

The sinc function modulating the magnitudes of the discrete frequency components in Figure 2.49 is illustrated clearly in Figures 2.51 and 2.52. Compare the impulse response shown in Figure 2.50 with that in Figure 2.28 and the pseudorandom signal shown in Figure 2.51 with that in Figure 2.32. Both the DAC in the WM5102 codec and the DAC in the AIC3104 codec are close to ideal low-pass filters with cutoff frequencies of fs/2 (4 kHz). The low-pass characteristic of the 12-bit DAC in the STM32F407 is significantly less pronounced. Finally, compare the output waveform shown in Figure 2.52 with that in Figure 2.24. For comparison, the sample values written to the 12-bit DAC are written also to the WM5102 codec. Its analog output may be observed by connecting an oscilloscope to the (green) LINE OUT socket on the audio card.

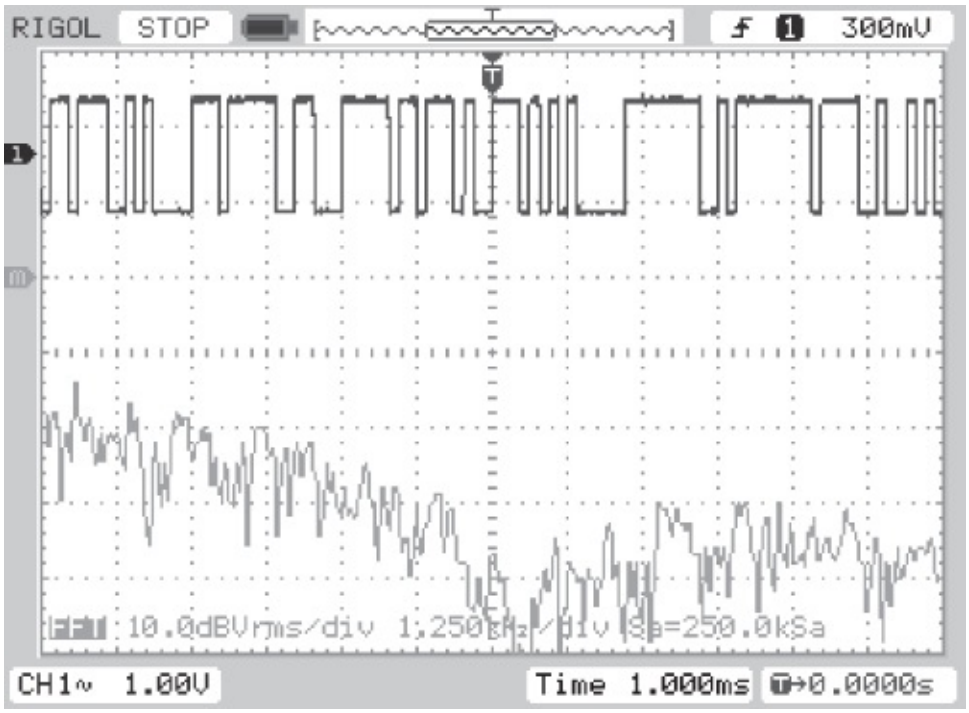


Figure 2.51 Output from program `stm32f4_prbs_dac12_intr.c` viewed using *Rigol DS1052 oscilloscope*.

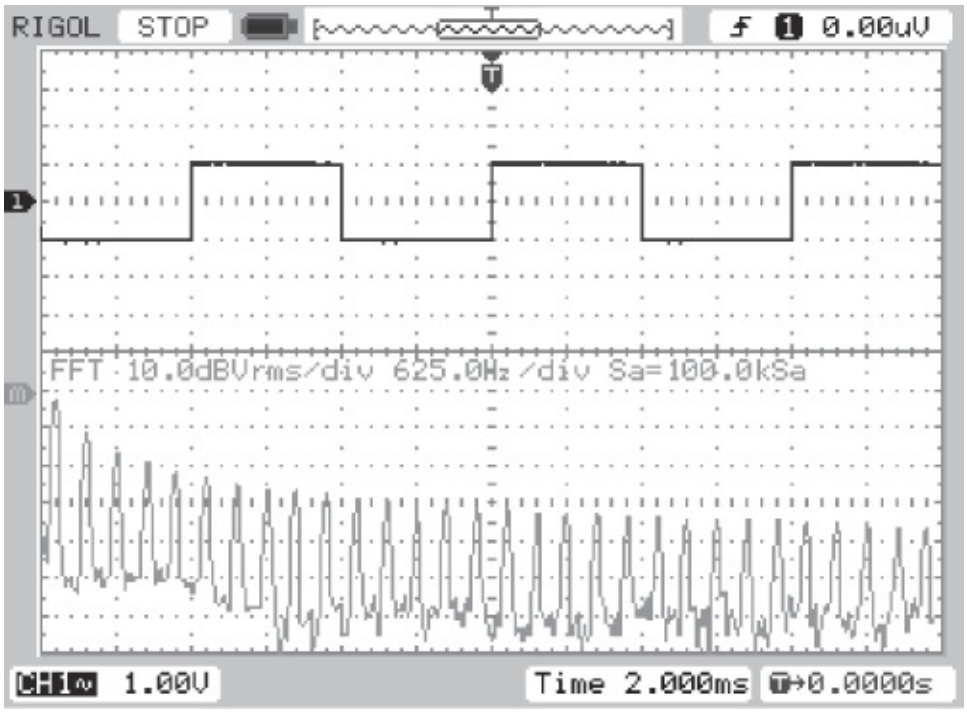


Figure 2.52 Output from program `stm32f4_square_dac12_intr.c` viewed using *Rigol DS1052 oscilloscope*.

Listing 2.25 Program `stm32f4_sine8_DAC12bit_intr.c`

```
// stm32f4_sine8_DAC12bit_intr.c
#include “stm32f4_wm5102_init.h”
```

```
#define LOOP_SIZE 8
int16_t sine_table[LOOP_SIZE] = {0, 7071, 10000, 7071,
                                 0, -7071, -10000, -7071};
static int sine_ptr = 0;
void init_DAC12bit(void)
{
  // enable clock for DAC module and GPIO port A
  RCC->AHB1ENR|=RCC_AHB1ENR_GPIOAEN;
  RCC->APB1ENR|=RCC_APB1ENR_DACEN;
  // configure GPIO pin PA5 as DAC output
  GPIOA->MODER|=GPIO_MODER_MODER5;
  GPIOA->PUPDR&=~(GPIO_PUPDR_PUPDR5);
  // enable DAC
  DAC->CR|=DAC_CR_EN2;
  // zero DAC output
  DAC->DHR12R2=0;
}
void SPI2_IRQHandler()
{
  int16_t left_out_sample, left_in_sample;
  int16_t right_out_sample, right_in_sample;
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ) SET)
  {
    left_in_sample = SPI_I2S_ReceiveData(I2Sx);
    left_out_sample = sine_table[sine_ptr];
    sine_ptr = (sine_ptr+1)
    DAC->DHR12R2=(left_out_sample+10000)/7;
    while (SPI_I2S_GetFlagStatus(I2Sxext,
           SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
  }
  else
  {
    right_in_sample = SPI_I2S_ReceiveData(I2Sx);
    right_out_sample = 0;
    while (SPI_I2S_GetFlagStatus(I2Sxext,
           SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
}
int main(void)
{
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
  init_DAC12bit();
  while(1){}
}
```

References

1 .Texas Instruments, Inc., “TLV320AIC3104 Low-Power Stereo Audio Codec for Portable

Audio and Telephony", Literature no. SLAS510D, 2014.

2 .Wolfson Microelectronics plc., "Audio Hub Codec with Voice Processor DSP", 2013.

3 .Texas Instruments, Inc., "Dual-SPI Emulating I^2S on Tiva^{TM} TM4C123x MCUs", Literature no. SPMA042B, 2013.

Chapter 3
Finite Impulse Response Filters

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 either nonrecursive, finite impulse response (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 The FIR Filter

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

1. A delay line, or buffer, in which a number of previous input samples $x(n-k)$ 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.
2. A number of blocks (multipliers) that multiply the samples stored in the delay line by a set of filter coefficients, $h(k)$.
3. A summing junction that sums the multiplier outputs to form the current filter output sample $y(n)$.

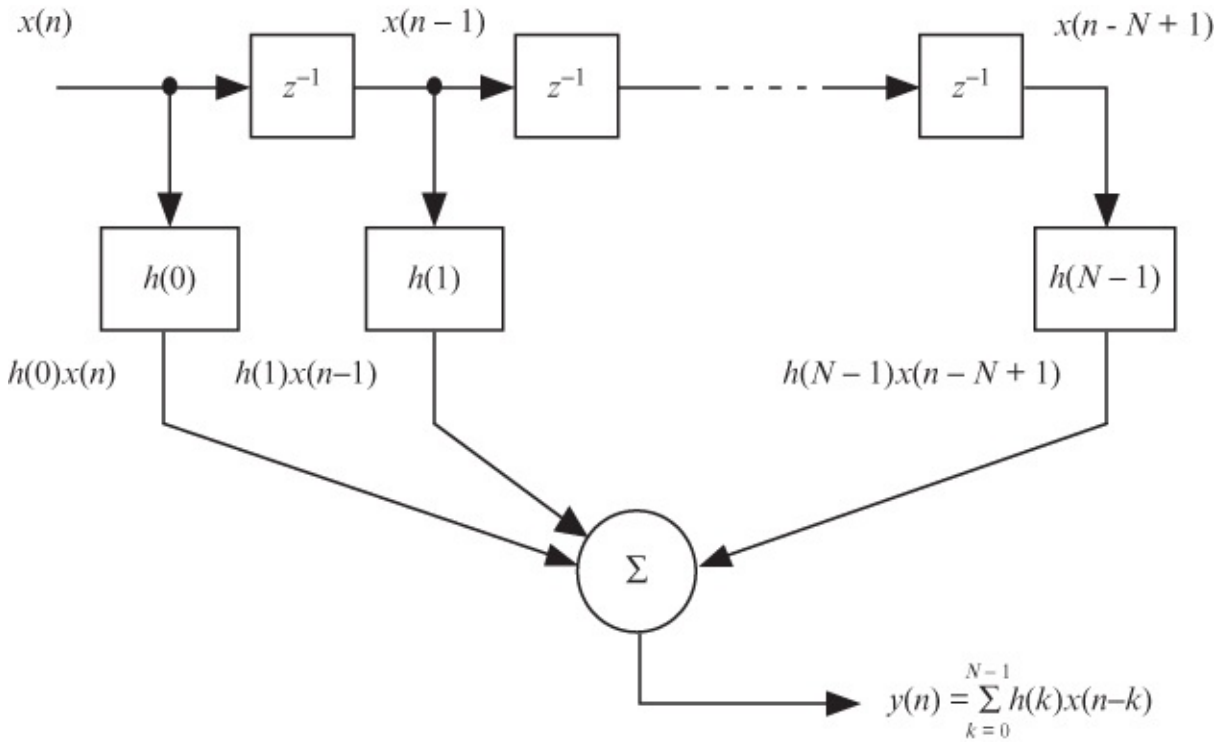


Figure 3.1 Block diagram representation of a generic FIR filter.

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 $x(n)$, $x(n-1)$, $x(n-2)$, …, $x(n-N+1)$ and the output of the filter, $y(n)$ is described by the difference equation

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

This equation is an example of a convolution sum representing the input-output relationship of a discrete-time, linear time-invariant (LTI) system having an FIR $h(n)$ 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.

3.1.1.1 Equivalence of FIR Filter Coefficients and 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. That 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

be multiplied by zero sample values from the delay line and will contribute nothing to the output at that sampling instant. Hence, the output sequence $y(n)$ (the unit impulse response of the filter) will comprise the filter coefficients $h(n)$.

3.1.1.2 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 are 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 filtersis that their implementation may be computationally expensive. Obtaining an arbitrary filter characteristic to the required accuracy may require a large number of filter coefficients.

3.1.1.3 FIR Filter Implementation

The structure and operation of an FIR filter are simple (and are 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 a sum of products or a convolution sum. For the convolution sum of Equation (3.1) to be computed directly, a DSP must have sufficient memory to store N previous input samples and N filter coefficients and have sufficient computational power to execute the required number of multiplications and additions within one sampling period. For large N, the use of FFT-based fast convolution (described in Chapter 5) is computationally more efficient.

3.1.2 Introduction to the z-Transform

The z-transform is an important tool in the design and analysis of digital filters. It is the discrete-time counterpart of the Laplace transform. In the sense that the Laplace transform is a generalization of the continuous-time Fourier transform, the z-transform is a generalization of the discrete-time Fourier transform (DTFT).

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 also 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 also be viewed as an operator, representing a shift of one sample position in a sequence.

3.1.3 Definition of the z-Transform

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

$$\mathcal{Z}\{x(n)\} = X(z) = \sum_{n=-\infty}^{\infty} x(n)z^{-n}, \tag{3.2}$$

where z is a complex variable. This form of the z-transform, applicable to a two-sided (noncausal) sequence, and hence also to left-sided (anticausal) and right-sided (causal) sequences, is referred to as the two-sided or bilateral z-transform.

$X(z)$ is a power series in z containing as many terms as there are sample values in the sequence $x(n)$. For each term in $X(z)$, the coefficient corresponding to z^{-n} is equal to the *n*th sample value in the sequence $x(n)$. In a discrete-time system, z^{-n} corresponds to the time $t = nT$, where T is the sampling period. $X(z)$ exists only for values of z for which the power series converges, that is, values of z for which $|X(z)| < \infty$.

3.1.3.1 Relationship to the Discrete-Time Fourier Transform

The DTFT is the form of Fourier analysis applicable to a discrete-time sequence $x(n)$ that is aperiodic and for which $-\infty < n < \infty$. The continuous, periodic representation of that signal in the frequency domain is given by

$$X(\hat{\omega}) = \sum_{n=-\infty}^{\infty} x(n)e^{-jn\hat{\omega}}. \tag{3.3}$$

Sometimes, $X(\hat{\omega})$ is represented as $X(e^{j\hat{\omega}})$, emphasizing that it is evaluated for different values of the complex quantity $e^{-j\hat{\omega}}$, where $\hat{\omega} = \omega T$.

Representing the complex variable z in polar form $z = re^{j\hat{\omega}}$, where $|z| = r$ and $\angle z = \hat{\omega}$, and substituting for z in Equation (3.2)

$$\begin{aligned} X(re^{j\hat{\omega}}) &= \sum_{n=-\infty}^{\infty} x(n)(re^{j\hat{\omega}})^{-n} \\ &= \sum_{n=-\infty}^{\infty} (x(n)r^{-n})e^{-jn\hat{\omega}} \end{aligned} \tag{3.4}$$

it is apparent that $X(z) = X(re^{j\hat{\omega}})$ is the DTFT of $x(n)r^{-n} = x(n)|z|^{-n}$. If $r = |z| = 1$ (corresponding to a unit circle in the z-plane), then the z-transform of $x(n)$ is equivalent to the DTFT of $x(n)$.

It is also apparent that the existence, or convergence, of $X(z)$ is dependent upon the value of $r = |z|$, that is, the magnitude of z. If $X(z)$ converges for a particular value of z, it also converges for all other values of z that lie on a circle of radius $|z|$ in the complex z-plane.

Example 3.1

z-Transform of Finite Sequence $x(n) = \{\underset{\uparrow}{3}, 2, 7, -4\}$.

From the definition of the z-transform, the z-transform of the right-sided (causal) sequence

$x(n) = \{\underset{\uparrow}{3}, 2, 7, -4\}$ is

$$X(z) = \sum_{n=-\infty}^{\infty} x(n)z^{-n} \tag{3.5}$$
$$= 3 + 2z^{-1} + 7z^{-2} - 4z^{-3}.$$

$X(z)$ exists (converges) for all values of z except $z = 0$.

Example 3.2

z-Transform of Finite Sequence $x(n) = \{\underset{\uparrow}{3}, 2, 7, -4\}$.

From the definition of the z-transform, the z-transform of the two-sided (noncausal) sequence $x(n) = \{\underset{\uparrow}{3}, 2, 7, -4\}$ is

$$X(z) = 3z + 2 + 7z^{-1} - 4z^{-2}. \tag{3.6}$$

$X(z)$ exists (converges) for all values of z except $z = 0$ and $z = \infty$.

Example 3.3

z-Transform of a Discrete Impulse Sequence $x(n) = \delta(n)$.

The z-transform of the right-sided (causal) Kronecker delta sequence $x(n) = \delta(n)$ is

$$X(z) = \sum_{n=-\infty}^{\infty} x(n)z^{-n} \tag{3.7}$$
$$= \delta(0)z^{-0}$$
$$= 1.$$

$X(z)$ exists (converges) for all values of z.

Example 3.4

z-Transform of a Time-Shifted Discrete Impulse Sequence $x(n) = \delta(n + k)$, $k > 0$.

A time-shifted Kronecker delta sequence is described by

$$\delta(n+k) = \begin{cases} 1, & n = -k, \\ 0, & n \neq -k. \end{cases} \tag{3.8}$$

This is a left-sided (anticausal) sequence. From the definition of the z-transform

$$\begin{aligned} X(z) &= \sum_{n=-\infty}^{\infty} \delta(n-k) z^{-n} \\ &= \delta(-k) z^{k} \\ &= z^{k}. \end{aligned} \tag{3.9}$$

$X(z)$ exists (converges) for all values of z except $z = \infty$.

Example 3.5

z-Transform of Exponential Function $x(n) = a^n u(n)$.

The z-transform of the right-sided (causal) sequence $x(n) = a^n u(n)$, where $u(n)$ is the unit step sequence, is

$$X(z) = \sum_{n=-\infty}^{\infty} a^n u(n) z^{-n} = \sum_{n=0}^{\infty} (az^{-1})^n. \tag{3.10}$$

Comparing this to the power series summation

$$\sum_{n=0}^{\infty} p^n = \frac{1}{1-p}, \quad \text{for } |p| < 1. \tag{3.11}$$

Equation (3.10) may be written as

$$X(z) = \frac{1}{1-az^{-1}} = \frac{z}{z-a}, \text{ for } |z| > |a|. \tag{3.12}$$

The inequality $|z| > |a|$ describes the range of values of z for which $X(z)$ exists, that is, its region of convergence (ROC). In this particular case, $|z| > |a|$ describes the part of the z-plane that lies outside a circle of radius a. $X(z)$ exists (converges) for $|z| > |a|$.

Example 3.6

z-Transform of Exponential Function $x(n) = -a^n u(-n-1)$.

This is a left-sided (anticausal) sequence that may appear to be of academic interest only but is included in order to make an important point. In this case,

$$X(z) = -\sum_{n=-\infty}^{\infty} a^n u(-n-1) z^{-n} = \sum_{n=-\infty}^{-1} a^n z^{-1}. \tag{3.13}$$

Letting $m = -n$

$$\begin{aligned} X(z) &= -\sum_{n=-1}^{\infty} a^{-m} z^m \\ &= 1 - \sum_{n=0}^{\infty} (za^{-1})^m \\ &= 1 - \frac{1}{1 - a^{-1}z} \\ &= \frac{1}{1 - az^{-1}} \\ &= \frac{z}{z - a}, \text{ for } |z| < |a|. \end{aligned} \tag{3.14}$$

The inequality $|z| < |a|$ describes the range of values of z for which $X(z)$ exists, that is, its ROC. In this particular case, $|z| < |a|$ describes the part of the z-plane that lies inside a circle of radius a. $X(z)$ exists (converges) for $|z| > |a|$.

Comparing (3.14) with (3.12), it is apparent that a similar algebraic expression for $X(z)$ corresponds to two different sequences $x(n)$. Corresponding to each of these different sequences is a different ROC. The ROC is therefore an integral part of the representation of a signal in the z-domain. In order to uniquely specify $x(n)$ from $X(z)$, we must know its ROC.

Example 3.7

z-Transform of the Unit Step Function $x(n) = u(n)$.

The unit step function may be viewed as an instance of Example 3.5, where $a = 1$, and hence, $X(z)$ exists (converges) for $|z| > 1$.

$$X(z) = \frac{1}{1 - z^{-1}} = \frac{z}{z - 1}, \text{ for } |z| > 1. \tag{3.15}$$

The inequality $|z| > 1$ describes the range of values of z for which $X(z)$ exists, that is, its ROC. In this particular case, $|z| > 1$ describes the part of the z-plane that lies outside a circle of radius 1.

Example 3.8

z-Transform of a Sinusoidal Function $x(n) = \sin(n\omega T)u(n)$.

This function is right-sided or causal. A sinusoidal function may be represented by complex exponentials according to Euler's formula $e^{j\theta} = \cos(\theta) + j\sin(\theta)$, that is,

$$\sin(n\omega T) = \frac{e^{jn\omega T}}{2j} - \frac{e^{-jn\omega T}}{2j} \tag{3.16}$$

and hence, the z-transform of the sequence $\sin(n\omega T)u(n)$ is given by

$$\begin{aligned} X(z) &= \frac{1}{2j}\sum_{n=0}^{\infty}(e^{jn\omega T}z^{-n} - e^{-jn\omega T}z^{-n}) \\ &= \frac{1}{2j}\sum_{n=0}^{\infty}e^{jn\omega T}z^{-n} - \frac{1}{2j}\sum_{n=0}^{\infty}e^{-jn\omega T}z^{-n}. \end{aligned} \tag{3.17}$$

Using the result for $x(n) = a^n u(n)$ with $a = e^{j\omega T}$,

$$\begin{aligned} X(z) &= \frac{1}{2j}\left(\frac{z}{z - e^{j\omega T}} - \frac{z}{z - e^{-j\omega T}}\right) \\ &= \frac{1}{2j}\left(\frac{z^2 - ze^{-j\omega T} - z^2 + ze^{j\omega T}}{z^2 - z(e^{-j\omega T} + e^{j\omega T}) + 1}\right) \\ &= \frac{z\sin(\omega T)}{z^2 - 2z\cos(\omega T) + 1}, \text{ for } |z| > 1. \end{aligned} \tag{3.18}$$

$X(z)$ exists (converges) for $|z| > |1|$.

In the z-plane, this function has a zero at the origin and two complex conjugate poles on the unit circle at angles $+/-\ \omega T$. Its ROC is the entire z-plane outside of, but not including, the unit circle.

Similarly, using Euler's formula to express $\cos(n\omega T)$ as the sum of two complex exponentials, it can be shown that the z-transform of the sequence $x(n) = \cos(n\omega T)u(n)$ is given by

$$X(z) = \frac{z^2 - z\cos(\omega T)}{z^2 - 2z\cos(\omega T) + 1}, \text{ for } |z| > 1. \tag{3.19}$$

3.1.3.2 Regions of Convergence

The preceding examples illustrate some important properties of the z-transform. Examples 3.5 and 3.6, for example, demonstrate that for a given z-transform $X(z)$, more than one ROC may be possible, corresponding to more than one different time-domain sequence $x(n)$. In order to transform from the z-domain back to the time domain, it is necessary to consider the ROC.

It is instructive to represent the poles and zeros of $X(z)$ and the ROCs in Examples 3.5 and 3.6 graphically, as shown in Figures 3.2 and 3.3. In each case, $X(z)$ has a zero at the origin and a pole at $z = a$.

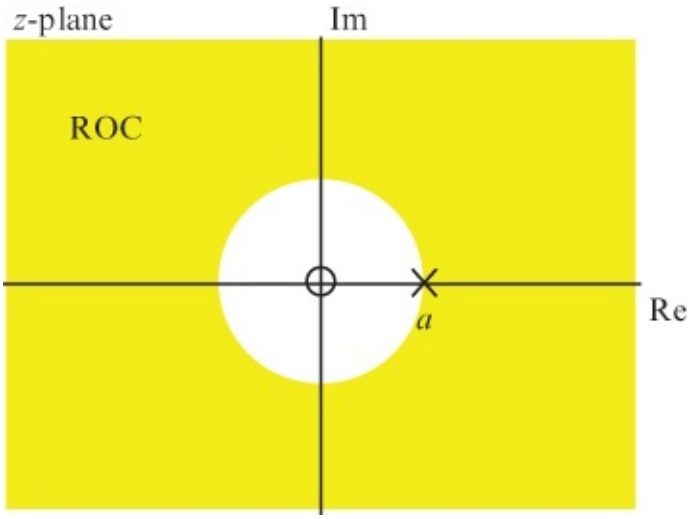


Figure 3.2 Poles and zeros and region of convergence for causal sequence $x(n) = a^n u(n)$, $X(z) = z/(z-a)$, plotted in the z-plane.

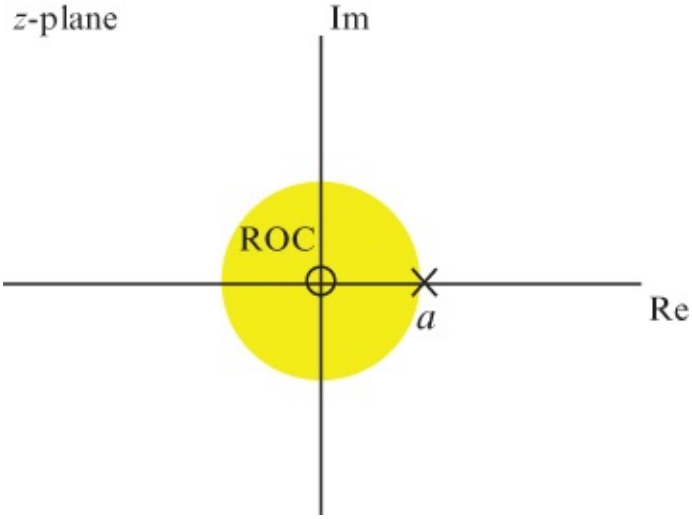


Figure 3.3 Poles and zeros and region of convergence for anticausal sequence $x(n) = -a^n u(-n-1)$, $X(z) = z/(z-a)$, plotted in the z-plane.

The two different ROCs for the z-transform $X(z) = z/(z-a)$ shown in the figures are consistent with the following ROC properties.

- An ROC is a single, connected region of the z-plane.
- Since convergence of $X(z)$ is dependent on the magnitude of z, the boundaries of an ROC are circles centered on the origin of the z-plane.
- Since regions of convergence correspond to $|X(z)| < \infty$, the poles of $X(z)$ do not lie within its ROC.
- Right-sided (causal) sequences $x(n)$ correspond to ROCs that extend outward from a circle drawn through the outermost pole of $X(z)$, that is, the pole with the greatest magnitude.
- Left-sided (anticausal) sequences $x(n)$ correspond to ROCs that extend inward from a circle drawn through the innermost pole of $X(z)$, that is, the pole with the smallest

magnitude.

- Two-sided (noncausal) sequences $x(n)$ correspond to annular ROCs.

Examples 3.1 through 3.4 illustrate the property that if $x(n)$ is finite in duration, then its ROC is the entire z-plane except possibly $z = 0$ or $z = \infty$.

In Examples 3.5 and 3.6, $X(z)$ has only one pole (at $z = a$) and, hence, the two possible ROCs (corresponding to causal and anticausal $x(n)$) extend outward and inward from a circle of radius a.

Figures 3.4 through 3.6 illustrate the case of a z-transform $X(z)$ that has more than one pole and for which more than two different ROCs, consistent with the properties listed earlier, are possible.

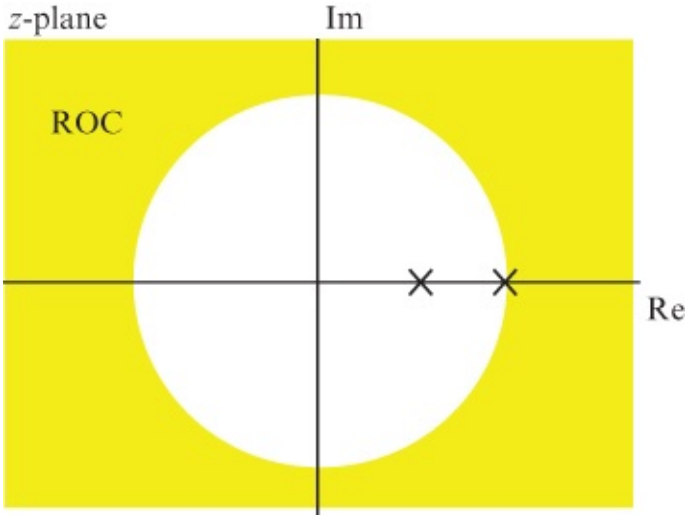


Figure 3.4 Possible region of convergence, plotted in the z-plane, corresponding to a right-sided causal sequence $x(n)$ for a system with two real-valued poles.

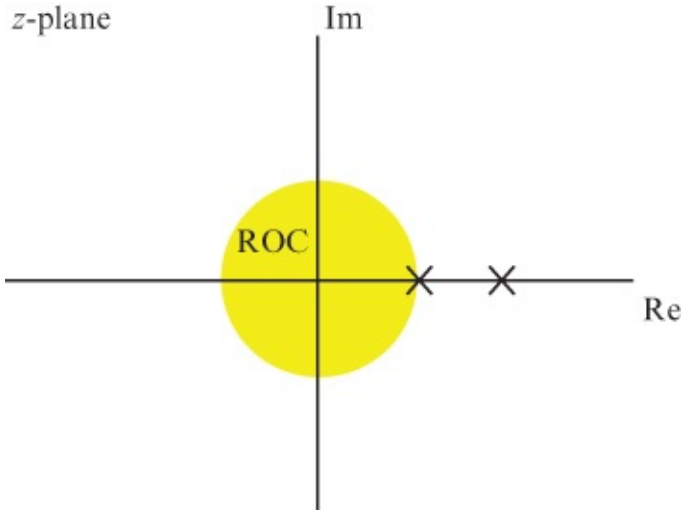


Figure 3.5 Possible region of convergence, plotted in the z-plane, corresponding to a left-sided anticausal sequence $x(n)$ for a system with two real-valued poles.

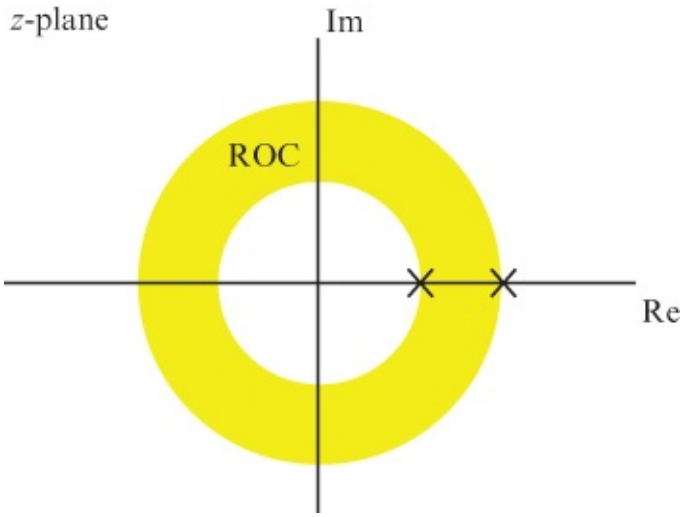


Figure 3.6 Possible region of convergence, plotted in the z-plane, corresponding to a two-sided noncausal sequence $x(n)$ for a system with two real-valued poles.

3.1.3.3 Regions of Convergence and Stability

An LTI system characterized by an impulse response $h(n)$ is BIBO stable if $h(n)$ is absolutely summable, that is, if

$$\sum_{n=-\infty}^{\infty} |h(n)| < \infty. \tag{3.20}$$

Given that the z-transform of $h(n)$ exists if

$$\sum_{n=-\infty}^{\infty} |h(n)z^{-n}| < \infty, \tag{3.21}$$

if $H(z)$ exists for $|z| = 1$, then the DTFT of $h(n)$ exists and the system is BIBO stable. In other words, if its ROC includes the unit circle, then the system represented by $H(z)$ and $h(n)$ is BIBO stable.

Consider again the example of the right-sided (causal) sequence $x(n) = a^n u(n)$ for which the z-transform $X(z) = z/(z-a)$ converges if $|z| > |a|$. Figures 3.7 through 3.9 show possible regions of convergence and corresponding sequences $x(n)$ for $a < 1$, $a = 1$, and $a > 1$.

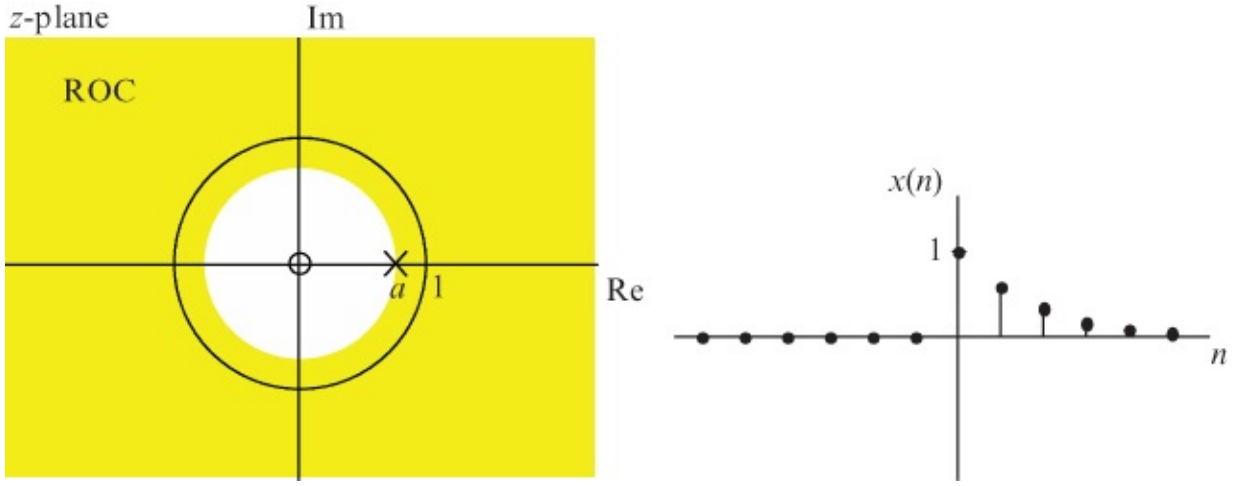


Figure 3.7 Poles and zeros and region of convergence for $X(z) = z/(z-a)$ plotted in the z-plane, for $|a| < 1$. Corresponding sequence $x(n)a^n u(n)$ is causal and stable.

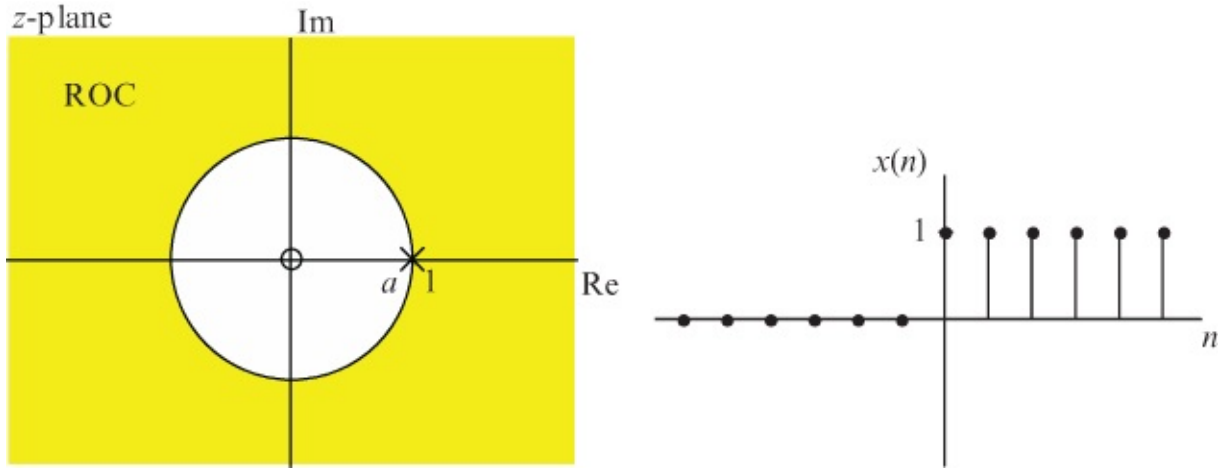


Figure 3.8 Poles and zeros and region of convergence for $X(z) = z/(z-a)$ plotted in the z-plane, for $|a| = 1$. Corresponding sequence $x(n)a^n u(n)$ is causal and unstable.

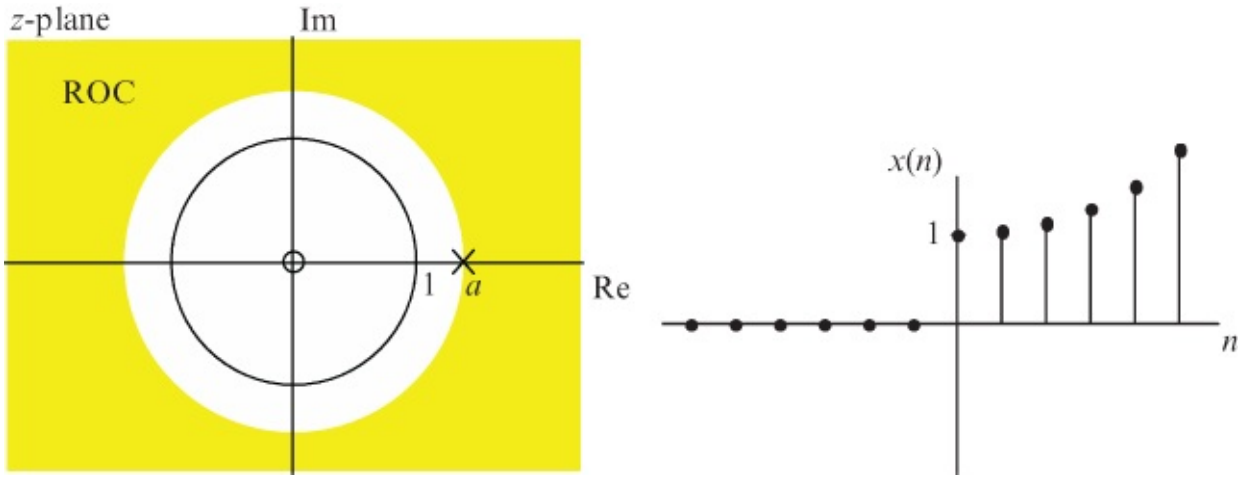


Figure 3.9 Poles and zeros and region of convergence for $X(z) = z/(z-a)$ plotted in the z-plane, for $|a| > 1$. Corresponding sequence $x(n) = a^n u(n)$ is causal and unstable.

For the left-sided (anticausal) sequence $x(n) = -a^n u(-n-1)$, the z-transform $X(z) = z/(z-a)$

converges if $|z| < |a|$. Figures 3.10 through 3.12 show possible regions of convergence and corresponding sequences $x(n)$ for $a > 1$, $a = 1$, and $a < 1$.

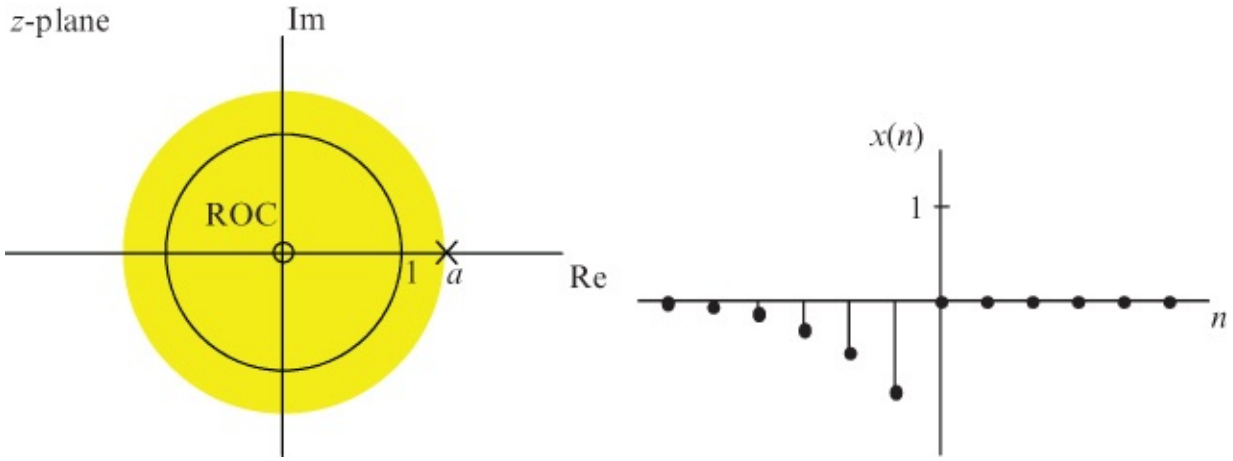


Figure 3.10 Poles and zeros and region of convergence for $X(z) = z/(z-a)$ plotted in the z-plane, for $|a| < 1$. Corresponding sequence $x(n) = -a^n u(-n-1)$ is anticausal and stable.

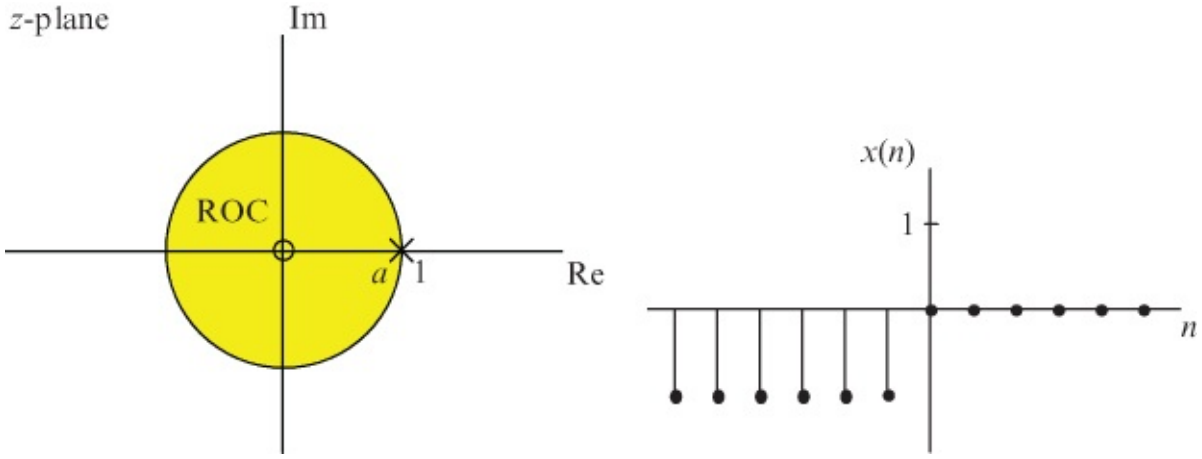


Figure 3.11 Poles and zeros and region of convergence for $X(z) = z/(z-a)$ plotted in the z-plane, for $|a| = 1$. Corresponding sequence $x(n) = -a^n u(-n-1)$ is anticausal and unstable.

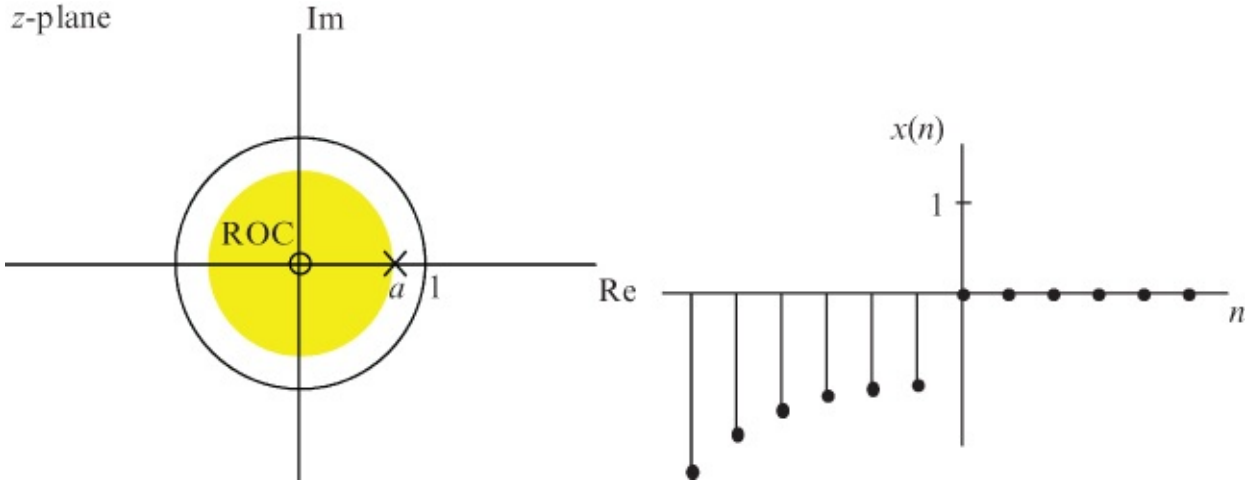


Figure 3.12 Poles and zeros and region of convergence for $X(z) = z/(z-a)$ plotted in the z-plane, for $|a| > 1$. Corresponding sequence $x(n) = -a^n u(-n-1)$ is anticausal and unstable.

3.1.3.4 Poles and Zeros

In the case of LTI causal systems, $X(z)$ may be expressed as a ratio of polynomials in z and as such has poles and zeros (values of z for which $X(z)$ is equal to zero or ∞). The poles and zeros of $X(z)$ are related to the region(s) of convergence of $X(z)$. In fact, we can deduce possible ROCs from the poles and zeros of $X(z)$ according to the rules listed earlier. In most engineering applications, we are concerned with, and will encounter, causal sequences.

3.1.4 Properties of the z-Transform

3.1.4.1 Linearity

The z-transform obeys the laws of superposition.

If $\mathcal{Z}\{x(n)\} = X(z)$ and $\mathcal{Z}\{y(n)\} = Y(z)$ then $\mathcal{Z}\{ax(n) + by(n)\} = aX(z) + bY(z)$, where $x(n)$ and $y(n)$ are arbitrary sequences and a and b are arbitrary constants.

3.1.4.2 Shifting

For a time-shifted sequence $x(n-m)$ where m is any integer

$$\mathcal{Z}\{x(n-m)\} = z^{-m}X(z). \tag{3.22}$$

From the definition of the z-transform,

$$\mathcal{Z}\{x(n-m)\} = \sum_{n=-\infty}^{\infty} x(n-m)z^{-n}. \tag{3.23}$$

Substituting $l = (n-m)$,

$$\begin{aligned}\mathcal{Z}\{x(n-m)\} &= \sum_{l=-\infty}^{\infty} x(l)z^{-(l+m)} \\ &= \sum_{l=-\infty}^{\infty} x(l)z^{-l}z^{-m} \\ &= z^{-m}\sum_{l=-\infty}^{\infty} x(l)z^{-l},\end{aligned} \tag{3.24}$$

which is recognizable as

$$\mathcal{Z}\{x(n-m)\} = z^{-m}X(z). \tag{3.25}$$

3.1.4.3 Time Delay

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.

3.1.4.4 Convolution

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)

$$y(n) = \sum_{m=0}^{\infty} h(m)x(n-m). \tag{3.26}$$

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

$$\mathcal{Z}\{y(n)\} = \mathcal{Z}\{\sum_{m=-\infty}^{\infty} h(m)x(n-m)\}, \tag{3.27}$$

$$Y(z) = \sum_{n=-\infty}^{\infty}\left[\sum_{m=-\infty}^{\infty} h(m)x(n-m)\right] z^{-n}.$$

Changing the order of summation

$$Y(z) = \sum_{m=-\infty}^{\infty}\left[\sum_{n=-\infty}^{\infty} h(m)x(n-m)\right] z^{-n} \tag{3.28}$$

$$= \sum_{m=-\infty}^{\infty} h(m)\left[\sum_{n=-\infty}^{\infty} x(n-m)\right] z^{-n}.$$

Letting $l = n - m$,

$$Y(z) = \sum_{m=-\infty}^{\infty} h(m) \sum_{n=-\infty}^{\infty} x(l)z^{-l}z - m \tag{3.29}$$

$$= \sum_{m=-\infty}^{\infty} h(m)z^{-m} \sum_{n=-\infty}^{\infty} x(l)z^{-l}$$

$$= H(z)X(z).$$

Hence,

$$\mathcal{Z}\{h(n) * x(n)\} = H(z)X(z) \tag{3.30}$$

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 $x(n)$ (as shown in Figure 3.13).

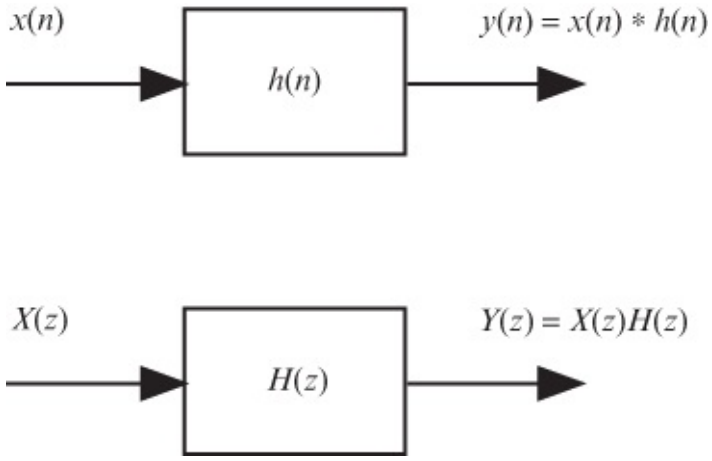


Figure 3.13 Time-domain and z-domain block diagram representations of a discrete-time LTI system.

3.1.5 z-Transfer Functions

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

3.1.6 Mapping from the s-Plane to the z-Plane

Consider the Laplace transform that is generally applied to causal systems. The Laplace transform can be used to determine the stability of a causal 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 correspond 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 exponentially and hence correspond 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 $s = \sigma + j\omega$,

$$z = e^{\sigma T} e^{j\omega T}. \tag{3.31}$$

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

3.1.6.1 $\sigma < 0$

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 a causal system represented by that z-transfer function will be stable.

3.1.6.2 $\sigma > 0$

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 ($|e^{\sigma T}| > 1$). 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 3.14. If the poles of a z-transfer function lie outside that unit circle, then a causal system represented by that z-transfer function will be unstable.

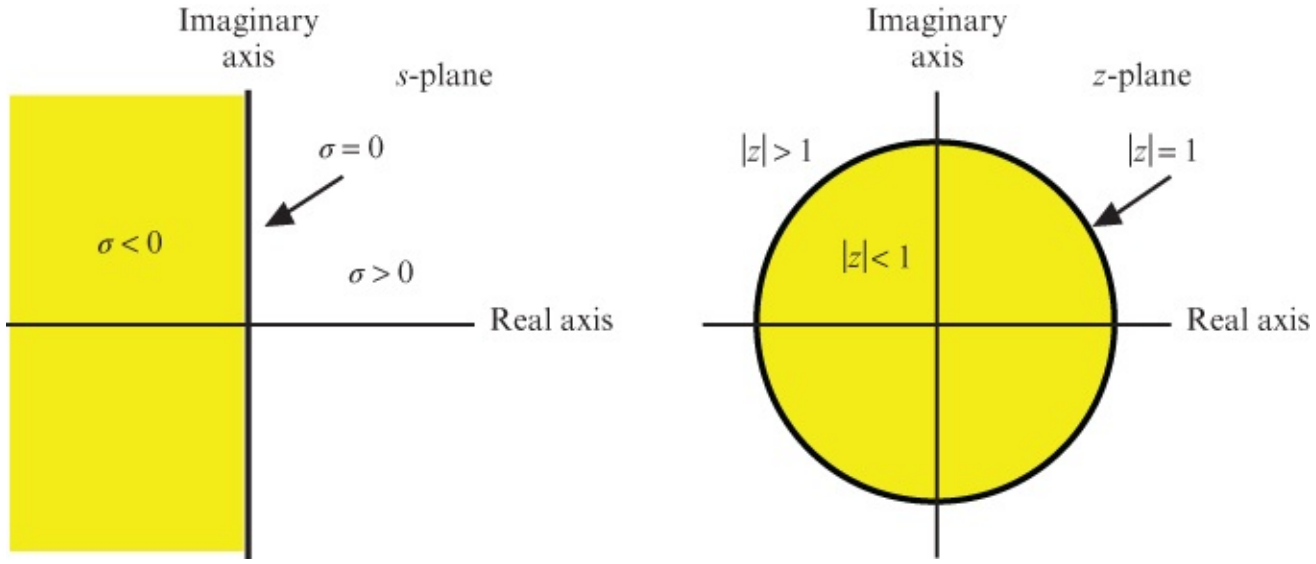


Figure 3.14 Mapping from the s-plane to the z-plane.

3.1.6.3 $\sigma = 0$

The imaginary axis in the s-plane maps to the unit circle in the z-plane. If the poles of a z-transfer function lie on the unit circle in the z-plane, then a causal system represented by that z-transfer function will have an oscillatory response and is not considered stable.

This view of the relationship between the location of system poles in the z-plane and system stability is, of course, consistent with consideration of whether or not the ROC includes the unit circle as described earlier. For causal systems, the ROC extends outward from the outermost pole, and for stability, the ROC must include the unit circle.

3.1.7 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. A differential equation may be solved using Laplace transforms, whereas a difference equation may be solved using z-transforms. In order to do this, the z-transforms of a term $x(n-k)$, which corresponds to the kth derivative with respect to time $d^k x(t)/dt^k$ of analog signal $x(t)$, must be found. From its definition, the z-transform of a right-sided, causal sequence is

$$X(z) = \sum_{n=0}^{\infty} x(n)z^{-n} = x(0) + x(1)z^{-1} + x(2)z^{-2} + \cdots \tag{3.32}$$

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

$$\begin{aligned} \mathcal{Z}\{x(n-1)\} &= \sum_{n=0}^{\infty} x(n-1)z^{-n} \\ &= x(-1) + x(0)z^{-1} + x(1)z^{-2} + x(2)z^{-3} + \cdots \\ &= x(-1) + z^{-1}[x(0) + x(1)z^{-1} + x(2)z^{-2} + \cdots] \\ &= x(-1) + z^{-1}X(z), \end{aligned} \tag{3.33}$$

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

$$\begin{aligned} \mathcal{Z}\{x(n-2)\} &= \sum_{n=0}^{\infty} x(n-2)z^{-n} \\ &= x(-2) + x(-1)z^{-1} + x(0)z^{-2} + x(1)z^{-3} + \cdots \\ &= x(-2) + x(-1)z^{-1} + z^{-2}[x(0) + x(1)z^{-1} + x(2)z^{-2} + \cdots] \\ &= x(-2) + x(-1)z^{-1} + z^{-2}X(z), \end{aligned} \tag{3.34}$$

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

$$\mathcal{Z}\{x(n-k)\} = z^{-k}\sum_{m=1}^{k} x(-m)z^{m} + z^{-k}X(z). \tag{3.35}$$

If the initial conditions are all zero, then $x(-m) = 0$ for $m = 1, 2, \cdots, k$ and Equation (3.35) reduces to

$$\mathcal{Z}\{x(n-k)\} = z^{-k}X(z). \tag{3.36}$$

3.1.8 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 of a system is found by evaluating of its z-transfer function for values of z that lie on the unit circle in the z-plane and the result will be 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 $z = e^{j\omega T}$, that is, around the unit circle in the z-plane corresponds to evaluating a Laplace transform for $s = j\omega$, that is, along the imaginary axis in the s-plane. This was considered earlier where it was stated that the frequency response of a discrete-time

system may be found by evaluating the DTFT of its impulse response. If $H(z)$ represents the z-transfer function of a discrete-time LTI system having an FIR of length N, then evaluation of that expression using $z = e^{j\hat{\omega}}$ yields the system's frequency response.

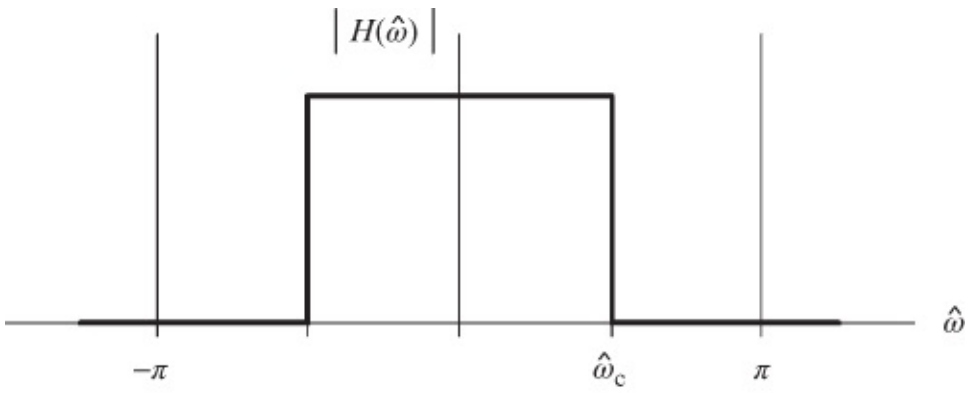
$$H(\hat{\omega}) = \sum_{n=0}^{N-1} h(n) e^{-jn\hat{\omega}}. \tag{3.37}$$

3.1.9 The Inverse z-Transform

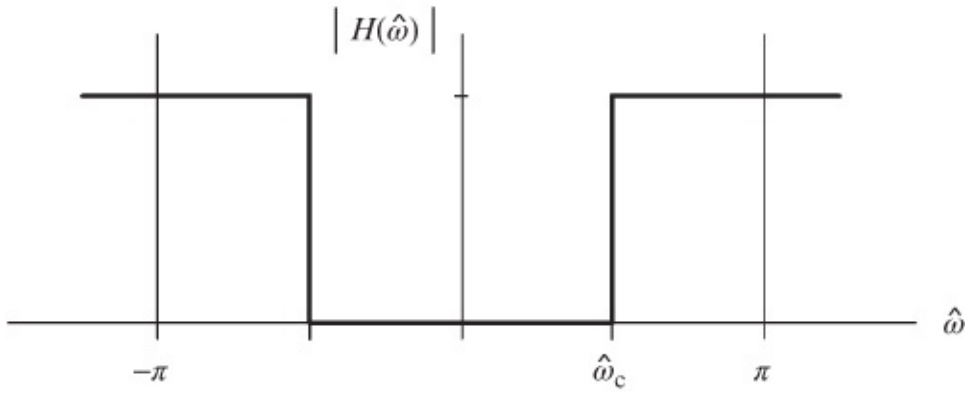
In practice, the inverse z-transform is best evaluated using tables of transform pairs, where first decomposed a complicated $X(z)$ by partial fraction expansion (PFE).

3.2 Ideal Filter Response Classifications: LP, HP, BP, BS

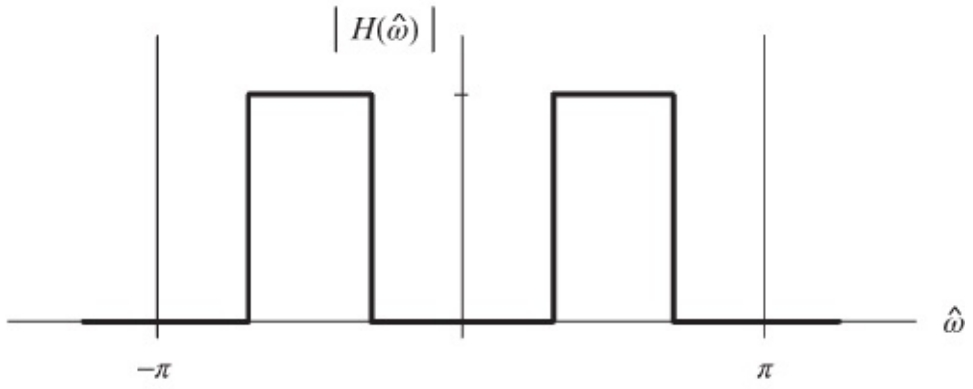
Shown in Figure 3.15 are the magnitude frequency responses of ideal low-pass, high-pass, band-pass, and band-stop filters. These are some of the most common filter characteristics used in a range of applications.


$|H(\hat{\omega})|$
$\hat{\omega}$
$-\pi$
$\hat{\omega}_c$
π

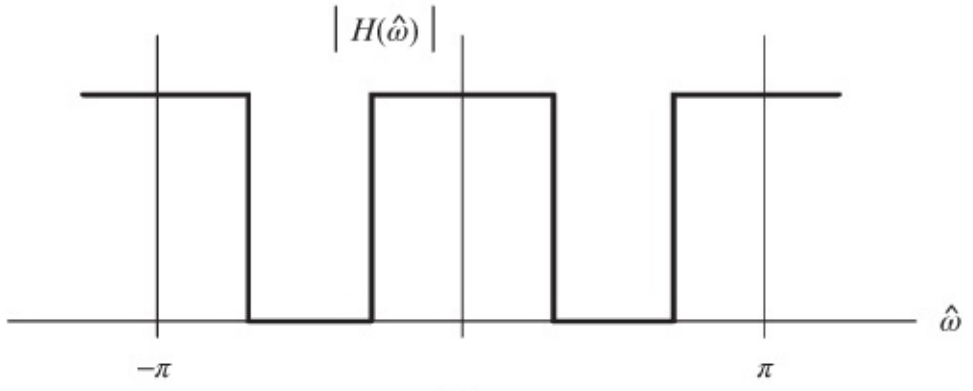
(a)


$|H(\hat{\omega})|$
$\hat{\omega}$
$-\pi$
$\hat{\omega}_c$
π

(b)


$|H(\hat{\omega})|$
$\hat{\omega}$
$-\pi$
π

(c)


$|H(\hat{\omega})|$
$\hat{\omega}$
$-\pi$
π

(d)

Figure 3.15 Ideal filter magnitude frequency responses. (a) Low-pass (LP). (b) High-pass (HP). (c) Band-pass (BP). (d) Band-stop (BS).

3.2.1 Window Method of FIR Filter Design

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

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

If the desired frequency response of the filter is specified as a continuous (periodic) function of frequency, the form of inverse Fourier analysis that will yield the discrete-time impulse response of the filter is the inverse discrete-time Fourier transform (IDTFT). In general, the inverse DTFT applied to a continuous frequency response will yield an infinite sequence in the time domain. Multiplication by a finite window function will truncate that sequence. A symmetrical 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 needs to be specified over just one period (2π radians of normalized frequency $\hat{\omega}$).

Application of the inverse DTFT is relatively straightforward for some analytic frequency responses (expressed as algebraic functions of $\hat{\omega}$) including the ideal filter characteristics shown in Figure 3.15, but for arbitrary frequency responses, applying the inverse DTFT may be problematic. In such cases, it is more practical to specify the desired frequency response of a filter as a discrete function of frequency and to compute a finite set of FIR filter coefficients using the inverse DFT.

3.2.2 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 stop bands. These effects 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.

3.2.2.1 Indexing of Filter Coefficients

In a computer program, it is likely that FIR filter coefficients $h(n)$ and window functions $w(n)$ will be indexed $0 \le n < N$. Alternatively, index n may be considered over the range $-L \le n \le L$. The following examples involve type 1 FIR filters, that is, FIR filters having N coefficients where N is odd, $-L \le n \le L$, $h(n) = h(-n)$, and $L = (N-1)/2$. The *order* of such a filter is $(N-1)$.

3.2.2.2 Common Window Functions

The window functions described in what follows are among the most commonly used. A rectangular window simply truncates an IIR to yield a finite set of FIR filter coefficients.

Rectangular Window

The rectangular window function is

$$w_{\text{RECT}}(n) = \begin{cases} 1, & -L \le n \le L, \\ 0, & \text{otherwise.} \end{cases} \tag{3.38}$$

Compared with other window functions, a rectangular window has a narrow central lobe (corresponding to a sharp transition between pass and stop bands), but its first sidelobe is only 13 dB less than the peak of its central main lobe.

Hamming Window

The Hamming window function is

$$w_{\text{HAMM}}(n) = \begin{cases} 0.54 + 0.46\cos(n\pi/L), & -L \le n \le L, \\ 0, & \text{otherwise.} \end{cases} \tag{3.39}$$

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

Hanning Window

The Hanning window function is

$$w_{\text{HANN}}(n) = \begin{cases} 0.5 + 0.5\cos(n\pi/L), & -L \le n \le L, \\ 0, & \text{otherwise.} \end{cases} \tag{3.40}$$

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

Blackman Window

The Blackman window function is

$$w_{BLACK}(n) = \begin{cases} 0.42 + 0.5\cos(n\pi/L) + 0.08\cos(2n\pi/L), & -L \le n \le L, \\ 0, & \text{otherwise.} \end{cases} \tag{3.41}$$

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

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 sidelobe relative to the main lobe. The Kaiser window function is

$$w_{\text{KAISER}}(n) = \begin{cases} I_0(b)/I_0(a), & |n| \le L, \\ 0, & \text{otherwise}, \end{cases} \tag{3.42}$$

where a is an empirically determined variable and $b = a[1 - (n/L)^2]^{0.5}$. $I_0(x)$ is a modified Bessel function of the first kind defined by

$$I_0(x) = 1 + \frac{0.25x^2}{(1!)^2} + \frac{(0.25x^2)^2}{(2!)^2} + \cdots = 1 + \sum_{n=1}^{\infty} \left[\frac{(x/2)^n}{n!}\right]^2, \tag{3.43}$$

which converges rapidly. A trade-off between the magnitude of the sidelobe 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.

The use of windows to reduce spectral leakage is discussed in Chapter 5.

Example 3.9

Design of an Ideal Low-Pass FIR Filter Using the Window Method.

The ideal low-pass filter characteristic shown in Figure 3.16 is described by

$$H(\hat{\omega}) = \begin{cases} 1, & |\hat{\omega}| < \hat{\omega}_c, \\ 0, & \text{otherwise}, \end{cases} \tag{3.44}$$

and the inverse DTFT of Equation (3.44) is

$$h(n) = \begin{cases} \dfrac{\sin(\hat{\omega}_c n)}{n\pi}, & n \neq 0, \\ \dfrac{\hat{\omega}_c}{\pi}, & n = 0, \end{cases} \tag{3.45}$$

for $-L \le n \le L$.

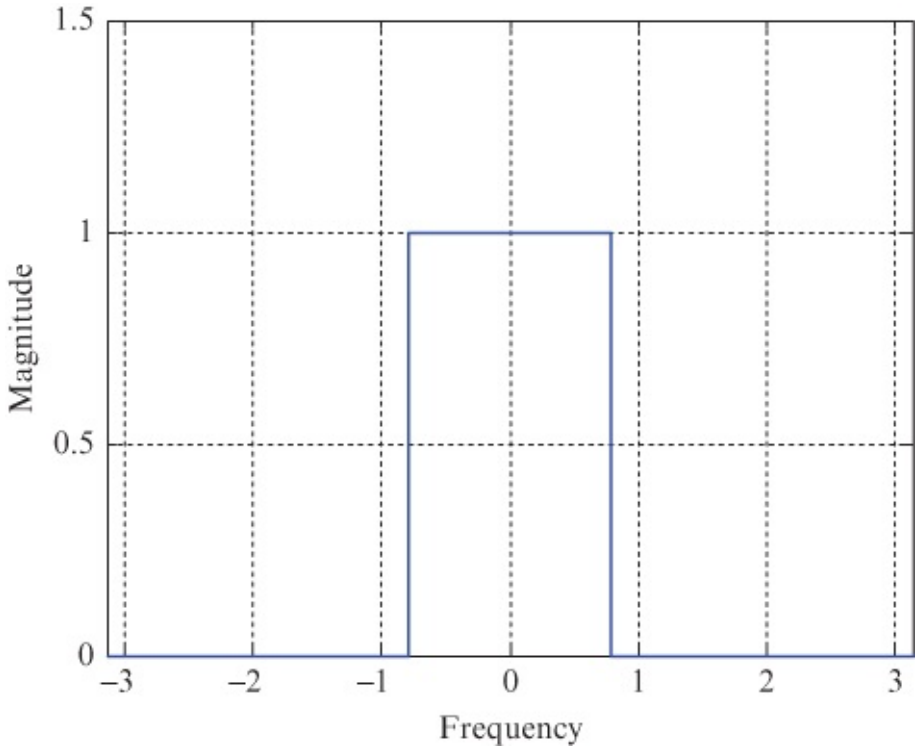


Figure 3.16 Ideal low-pass frequency response defined over normalized frequency range $-\pi \leq \hat{\omega} \leq \pi$.

This result is illustrated in Figure 3.17, 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 3.18 shows the result of truncation using a rectangular window of length $N = 33$ $(-16 \leq n \leq 16)$. This has the effect of introducing gain variations (ripple) into the corresponding frequency response as shown in Figure 3.19. This continuous (periodic) frequency response is found by taking the (forward) DTFT of the truncated impulse response shown in Figure 3.18.

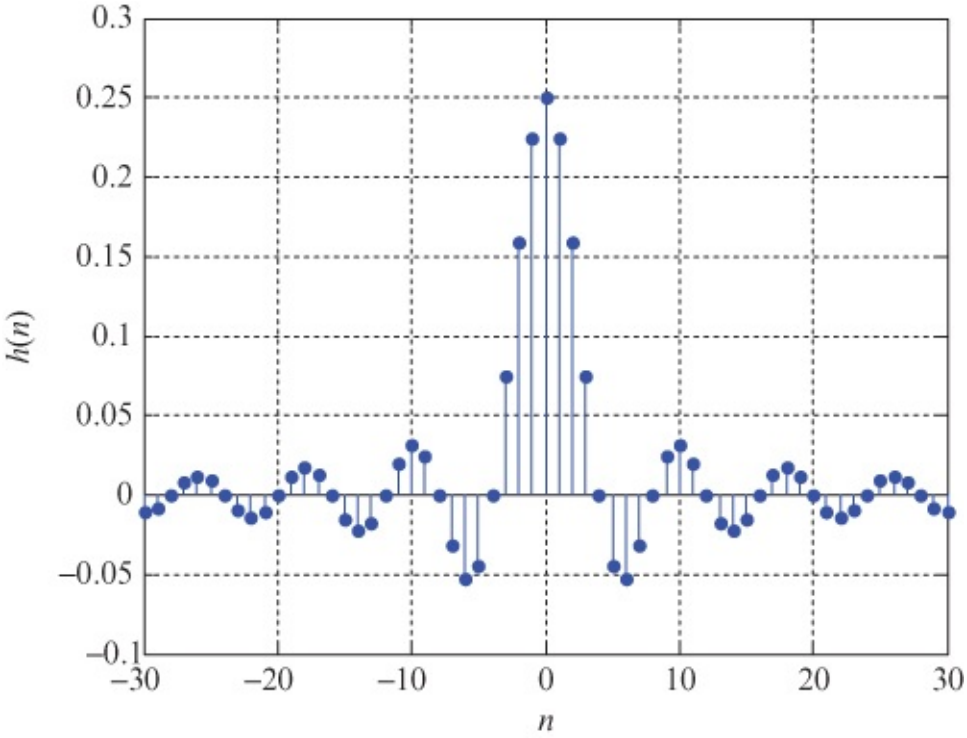


Figure 3.17 Sixty-one of the infinite number of values in the discrete-time impulse response obtained by taking the inverse DTFT of the ideal low-pass frequency response of Figure 3.16.

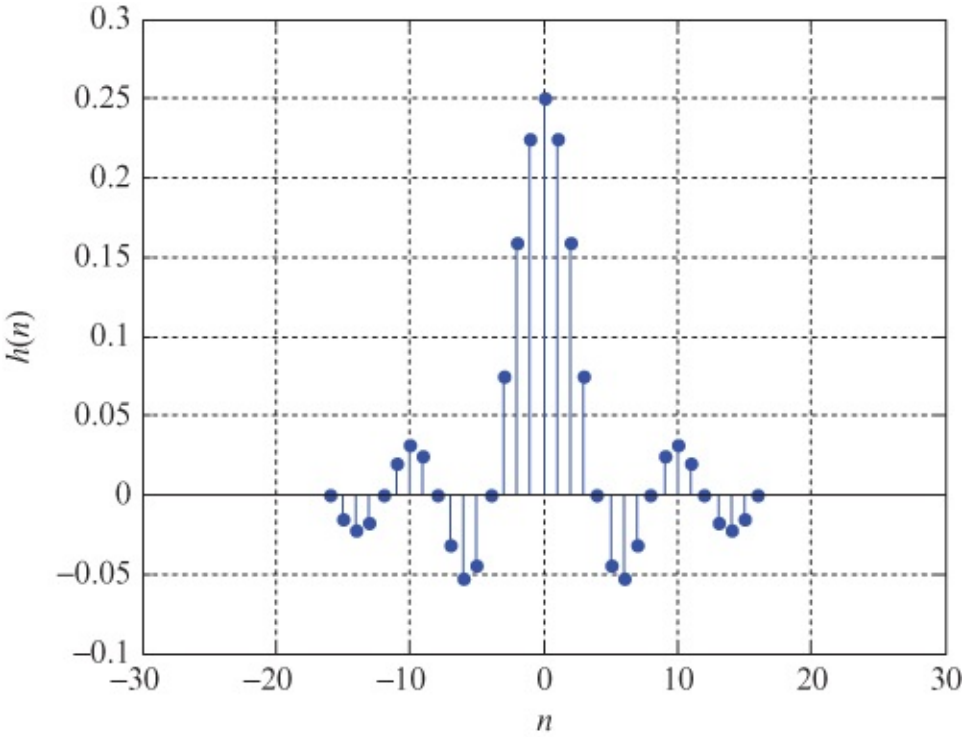


Figure 3.18 The discrete-time impulse response of Figure 3.17 truncated to $N = 33$ values.

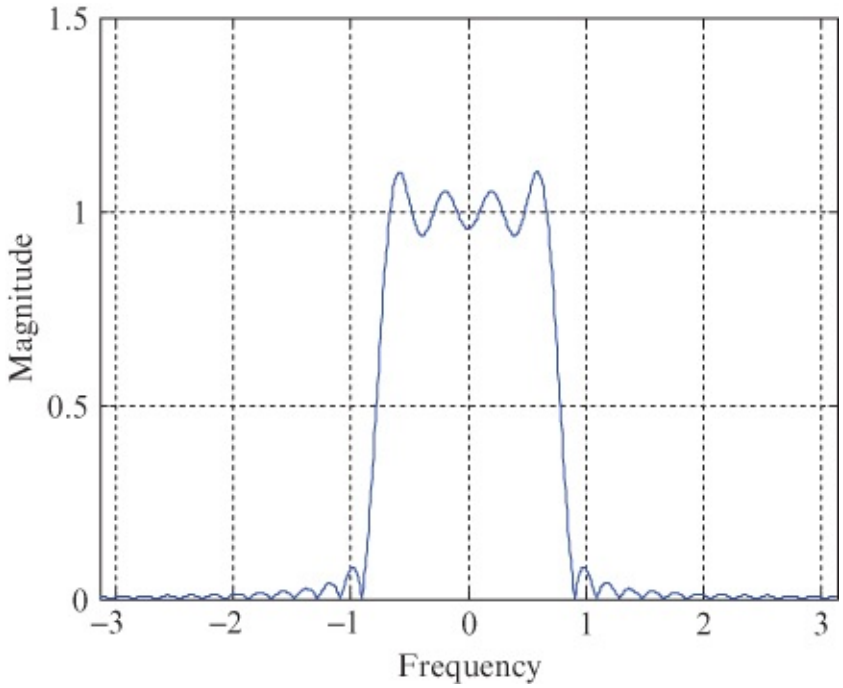


Figure 3.19 The continuous, periodic magnitude frequency response obtained by taking the DTFT of the truncated impulse response shown in Figure 3.18 (plotted against normalized frequency $\hat{\omega}$).

Multiplying the impulse response of Figure 3.17 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 pass band to stop band less sharp. Figures 3.20–3.21 show a 33-point Hanning window function, the result of multiplying the impulse response (filter coefficients) of Figure 3.19 by that window function, and the magnitude frequency response corresponding to the filter coefficients shown in Figure 3.22, respectively.

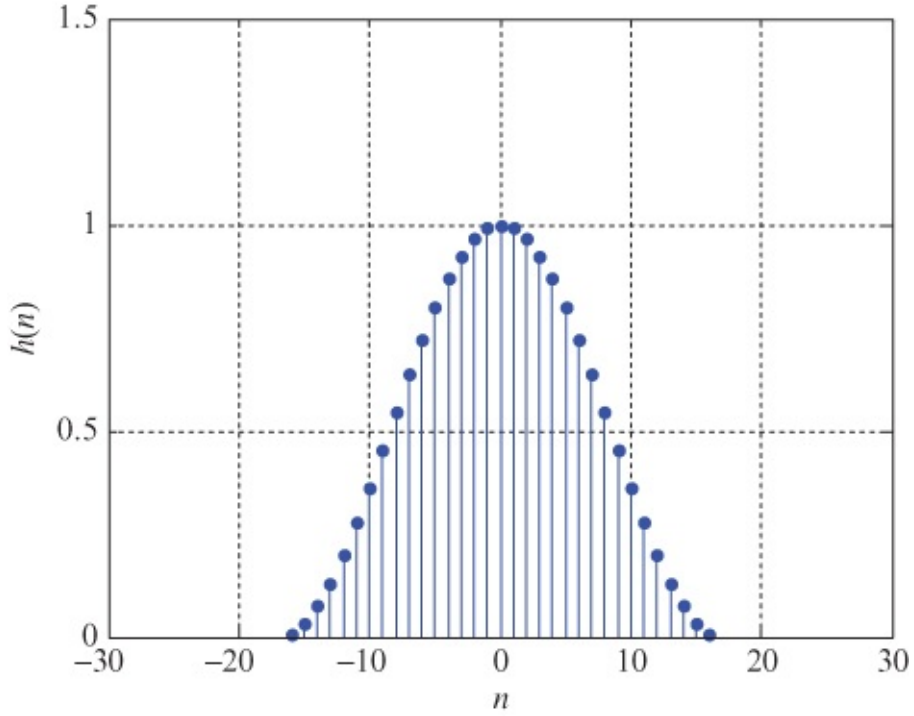


Figure 3.20 A 33-point Hanning window.

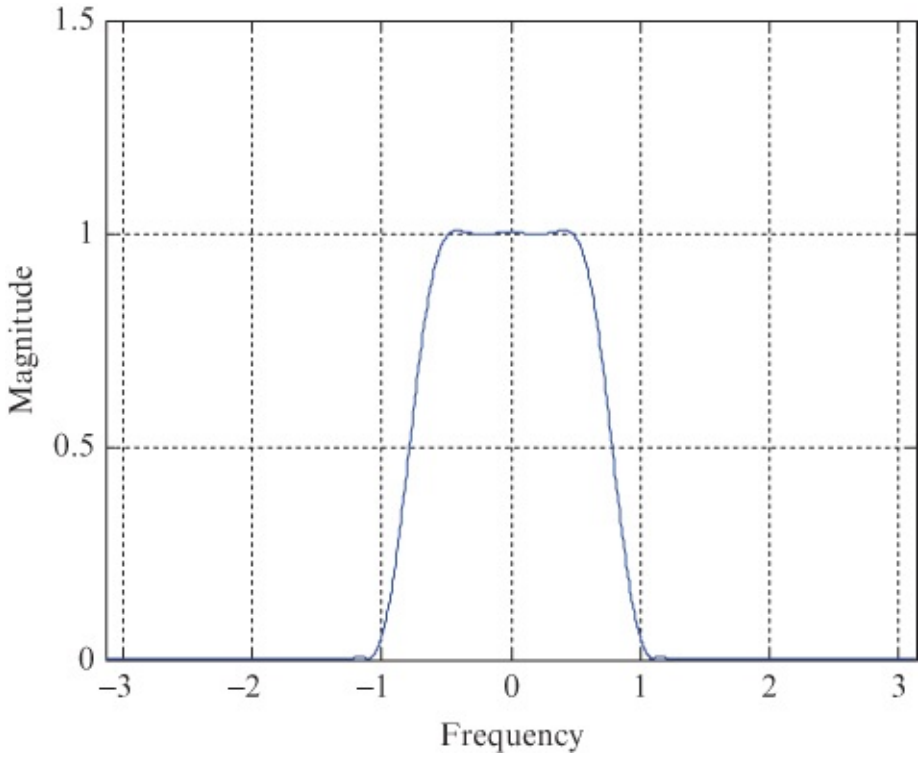


Figure 3.21 The magnitude frequency response corresponding to the filter coefficients of Figure 3.22 (plotted against normalized frequency $\hat{\omega}$).

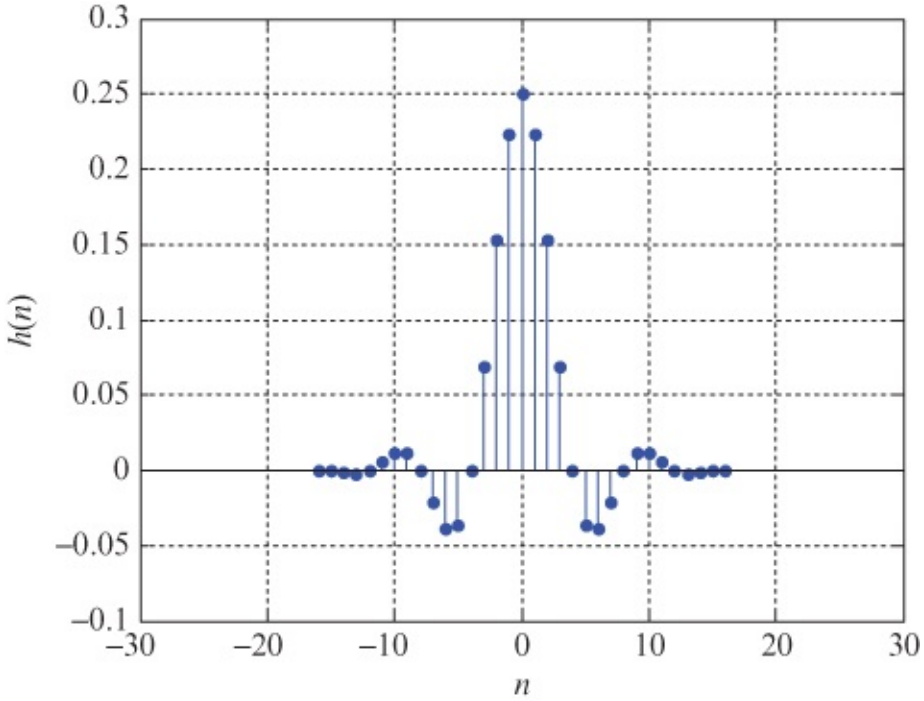


Figure 3.22 The filter coefficients of Figure 3.17 multiplied by the Hanning window of Figure 3.20.

Figure 3.23 shows the magnitude frequency responses of Figures 3.19 and 3.21 plotted together on a logarithmic scale. This emphasizes the wider main lobe and suppressed sidelobes associated with the Hanning window.

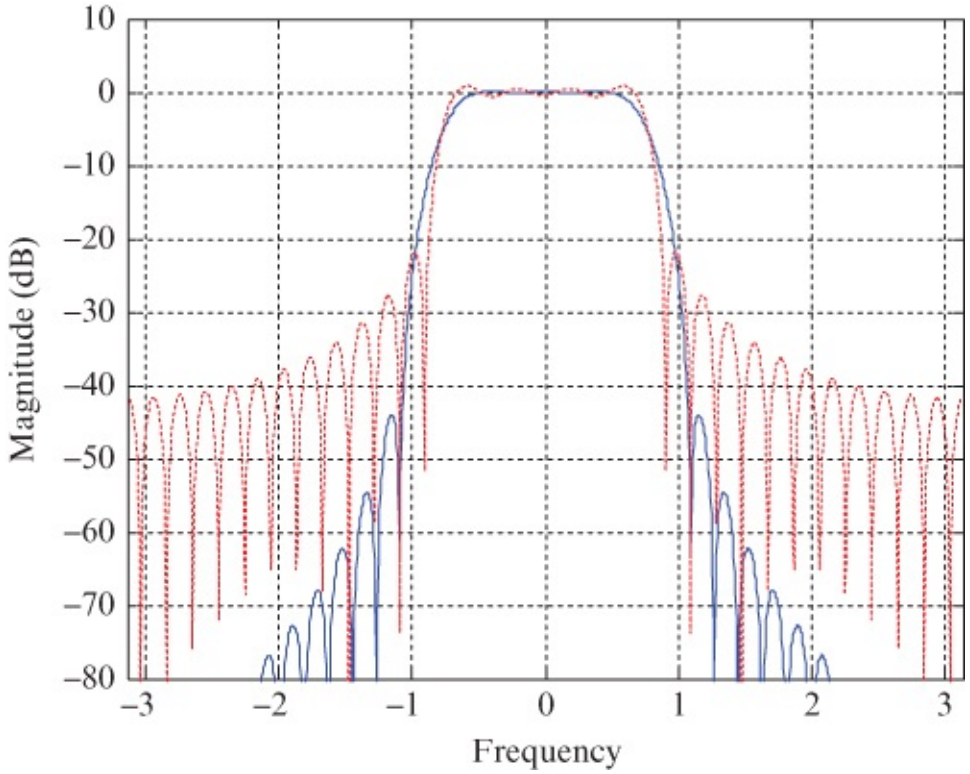


Figure 3.23 The magnitude frequency responses of Figures 3.19 and 3.21 plotted on a logarithmic scale, against normalized frequency $\hat{\omega}$.

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 specified. These correspond to real-valued filter coefficients that are even functions of time but are noncausal. 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.

3.2.3 Design of Ideal High-Pass, Band-Pass, and Band-Stop FIR Filters Using the Window Method

3.2.3.1 Ideal High-Pass Filter

The ideal high-pass filter characteristic shown in Figure 3.24 is described by

$$H(\hat{\omega}) = \begin{cases} 0, & |\hat{\omega}| < \hat{\omega}_c, \\ 1, & \text{otherwise}, \end{cases} \tag{3.46}$$

in the range $-\pi < \hat{\omega} < \pi$.

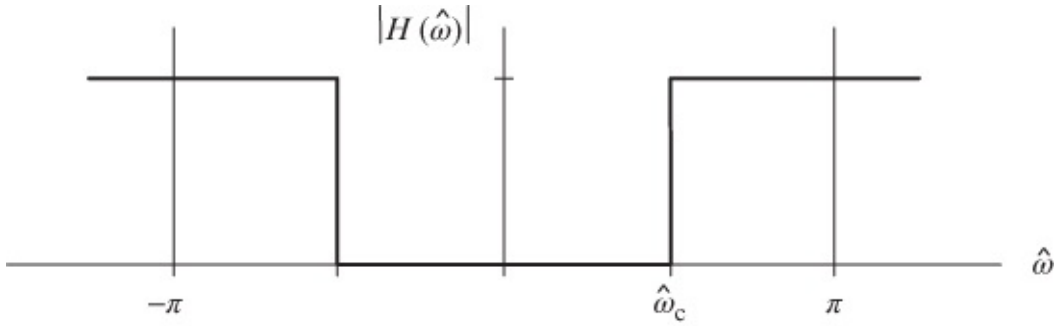


Figure 3.24 Ideal high-pass filter magnitude frequency response.

and the inverse DTFT of Equation (3.46) is

$$h(n) = \begin{cases} \dfrac{-\sin(\hat{\omega}_c n)}{n\pi}, & n \neq 0, \\ 1 - \dfrac{\hat{\omega}_c}{\pi}, & n = 0, \end{cases} \tag{3.47}$$

for $-L \leq n \leq L$.

3.2.3.2 Ideal Band-Pass Filter

The ideal band-pass filter characteristic shown in Figure 3.25 is described by

$$H(\hat{\omega}) = \begin{cases} 0, & |\hat{\omega}| < \hat{\omega}_{c1}, \\ 1, & \hat{\omega}_{c1} \leq |\hat{\omega}| \leq \hat{\omega}_{c2}, \\ 0, & |\hat{\omega}| > \hat{\omega}_{c2}, \end{cases} \tag{3.48}$$

in the range $-\pi < \hat{\omega} < \pi$.

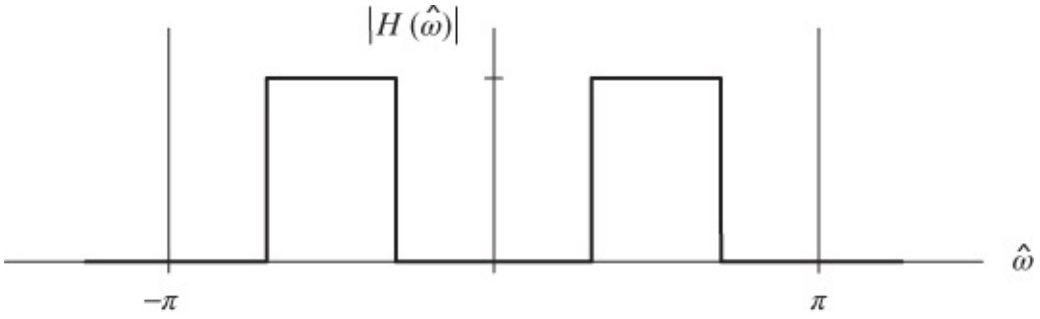


Figure 3.25 Ideal band-pass filter magnitude frequency response.

and the inverse DTFT of equation (3.48) is

$$h(n) = \begin{cases} \dfrac{\sin(\hat{\omega}_{c2}n) - \sin(\hat{\omega}_{c1}n)}{n\pi}, & n \neq 0, \\ \dfrac{\hat{\omega}_{c2} - \hat{\omega}_{c1}}{\pi}, & n = 0, \end{cases} \tag{3.49}$$

for $-L \leq n \leq L$.

3.2.3.3 Ideal band-stop filter

The ideal band-stop filter characteristic shown in Figure 3.26 is described by

$$H(\hat{\omega}) = \begin{cases} 1, & |\hat{\omega}| < \hat{\omega}_{c1}, \\ 0, & \hat{\omega}_{c1} \leq |\hat{\omega}| \leq \hat{\omega}_{c2}, \\ 1, & |\hat{\omega}| > \hat{\omega}_{c2}, \end{cases} \tag{3.50}$$

in the range $-\pi < \hat{\omega} < \pi$.

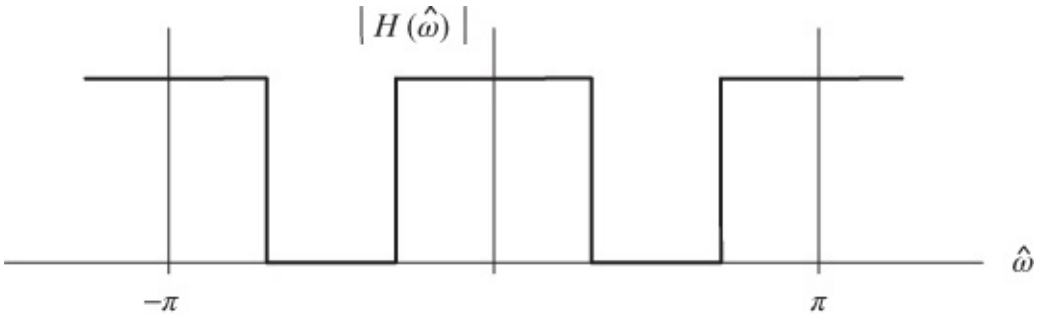


Figure 3.26 Ideal band-stop filter magnitude frequency response.

and the inverse DTFT of Equation (3.50) is

$$h(n) = \begin{cases} \dfrac{\sin(\hat{\omega}_{c1}n) - \sin(\hat{\omega}_{c2}n)}{n\pi}, & n \neq 0 \\ 1 - \dfrac{\hat{\omega}_{c2} - \hat{\omega}_{c1}}{\pi}, & n = 0 \end{cases} \tag{3.51}$$

for $-L \leq n \leq L$.

High-pass, band-pass, and band-stop filters may be designed using the window method described for the low-pass filter, but substituting Equations (3.47), (3.49), or (3.51) for Equation (3.45).

3.3 Programming Examples

The following examples illustrate the real-time implementation of FIR filters using C and functions from the CMSIS DSP library for the ARM Cortex-M4 processor. Several different methods of assessing the magnitude frequency response of a filter are presented.

Example 3.10

Moving Average Filter (`stm32f4_average_intr.c`).

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

$$y(n) = \frac{1}{N} \sum_{i=0}^{N-1} x(n-i), \tag{3.52}$$

where $x(n)$ represents the nth sample of an input signal and $y(n)$ the nth 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 which is equal to $1/N$. Equation (3.52) may be thought of as a particularly simple case of the more general convolution sum implemented by an FIR filter and introduced in Section 3.1, that is,

$$y(n) = \sum_{i=0}^{N-1} h(i)x(n-i), \tag{3.53}$$

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 nth samplinginstant, we could either

1. multiply N past input samples individually by $1/N$ and sum the N products,
2. sum N past input samples and multiply the sum by $1/N$, or
3. 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

$$y(n) = \frac{1}{N}x(n) - \frac{1}{N}x(n-N) + y(n-1) \tag{3.54}$$

is an instance of the general expression for a recursive or IIR filter

$$y(n) = \sum_{k=0}^{M} b_k x(n-k) - \sum_{l=1}^{N} a_l y(n-l). \tag{3.55}$$

Program `stm32f4_average_intr.c`, shown in Listing 3.11, uses the first of these options, even though it may not be 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.

Listing 3.1 Program `stm32f4_average_intr.c`.

```
// stm32f4_average_intr.c
#include "stm32f4_wm5102_init.h"
#define N 5
float32_t h[N];
float32_t x[N];
void SPI2_IRQHandler()
{
  int16_t left_out_sample, right_out_sample;
  int16_t left_in_sample, right_in_sample;
  int16_t i;
  float32_t yn = 0.0;
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ) SET)
  {
    left_in_sample = SPI_I2S_ReceiveData(I2Sx);
    x[0] = (float32_t)(left_in_sample);
    for (i=0 ; i<N ; i++) yn += h[i]*x[i];
    for (i=(N-1) ; i>0 ; i--) x[i] = x[i-1];
    left_out_sample = (int16_t)(yn);
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
  }
  else
  {
    right_in_sample = SPI_I2S_ReceiveData(I2Sx);
    right_out_sample = 0;
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
}
int main(void)
{
  int i;
  for (i=0 ; i<N ; i++) h[i] = 1.0/N;
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
  while(1);
}
```

Several different methods exist by which the characteristics of the five-point moving average filter may be demonstrated. A test file `mefsin.wav` 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 soundcard line output to the (green) LINE IN socket on the Wolfson audio card and listen to the filtered test signal. With program `stm32f4_average_intr.c` running, you should find that the sinusoidal tone has been blocked

and that the speech sounds muffled. Both observations are consistent with the filter having a low-pass frequency response.

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 (DTFT) of its coefficients.

$$H(\hat{\omega}) = \sum_{n=0}^{N-1} h(n) e^{-j\hat{\omega}n}. \tag{3.56}$$

Evaluated over the frequency range $-\pi < \hat{\omega} < \pi$, where $\hat{\omega} = \omega T_s$, ω is frequency in radians per second, and T_s is the sampling period in seconds. In this case,

$$\begin{aligned} H(\hat{\omega}) &= \sum_{n=0}^{4} 0.2 e^{-j\hat{\omega}n} \\ &= 0.2(1 + e^{-j\hat{\omega}} + e^{-j2\hat{\omega}} + e^{-j3\hat{\omega}} + e^{-j4\hat{\omega}}) \\ &= 0.2e^{-j2\hat{\omega}}(e^{j2\hat{\omega}} + e^{j\hat{\omega}} + 1 + e^{-j\hat{\omega}} + e^{-j2\hat{\omega}}) \end{aligned} \tag{3.57}$$

and hence

$$|H(\hat{\omega})| = |0.2(1 + 2\cos(\hat{\omega}) + 2\cos(2\hat{\omega}))|. \tag{3.58}$$

The theoretical magnitude frequency response of the filter is illustrated in Figure 3.27. This is the magnitude of a Dirichlet, or periodic sinc, function.

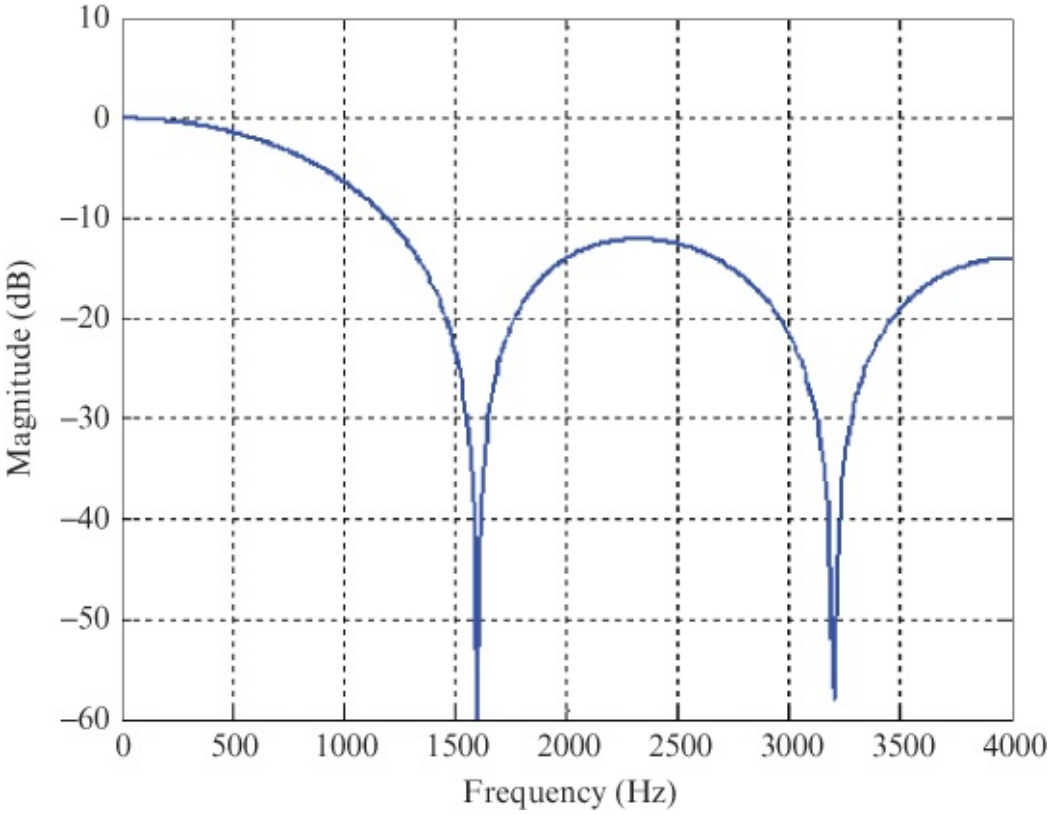


Figure 3.27 Theoretical magnitude frequency response of the five-point moving average filter (sampling rate 8 kHz).

Example 3.11

Moving Average Filter with Internally Generated Pseudorandom Noise as Input (`stm32f4_average_prbs_intr.c`).

An alternative method of assessing the magnitude frequency response of a filter is to use wideband noise as an input signal.

Program `stm32f4_average_prbs_intr.c` demonstrates this technique. A pseudorandom binary sequence (PRBS) is generated within the program (see program `tm4c123_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 magnitude frequency response of the filter. Figure 3.28 shows the output of program `stm32f4_average_prbs_intr.c` displayed using the FFT function of a *Rigol DS1052E* oscilloscope and using *Goldwave*. Compare these plots with the theoretical magnitude frequency response shown in Figure 3.27.

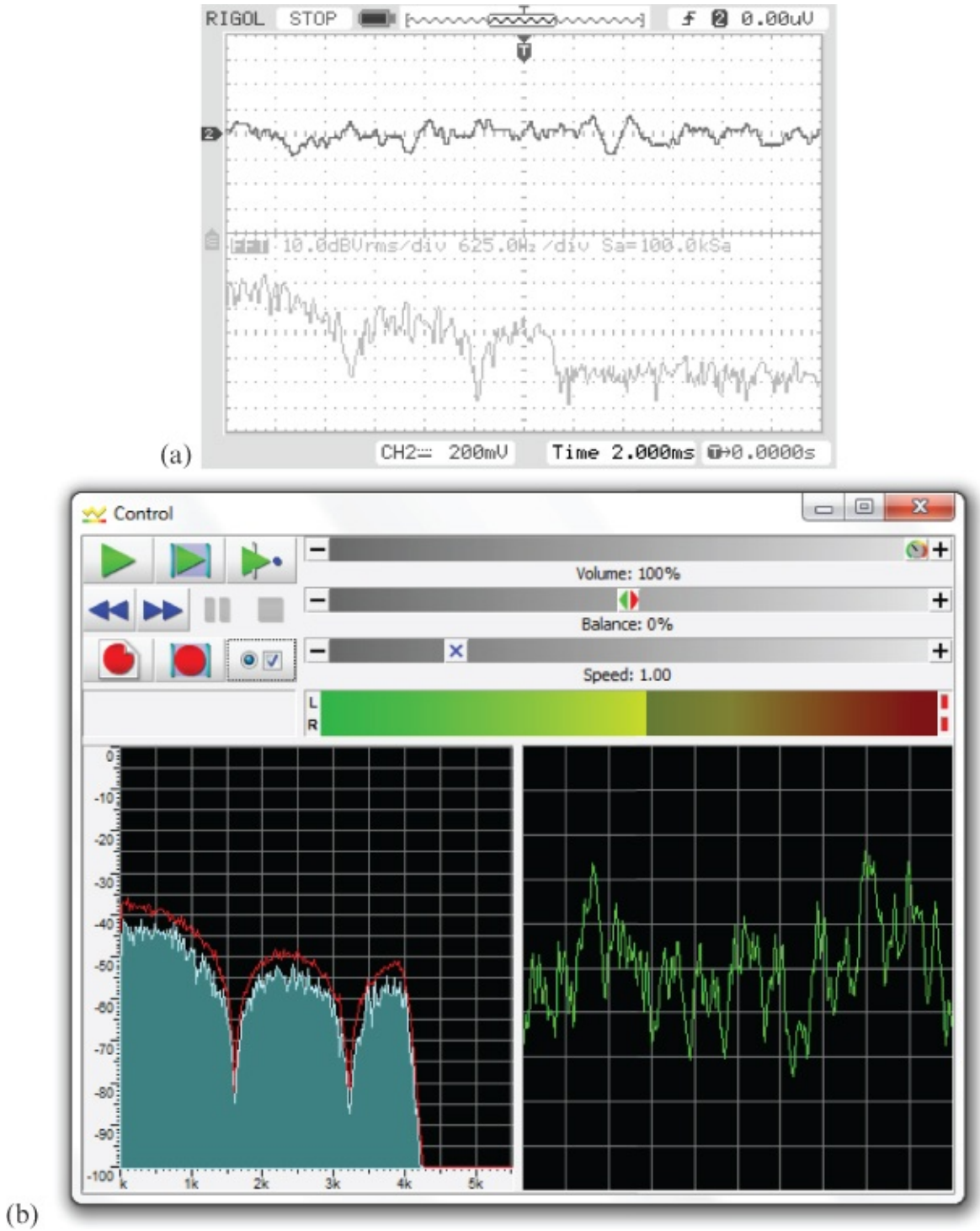


Figure 3.28 Magnitude frequency response of the five-point moving average filter demonstrated using program `stm32f4_average_prbs_intr.c` and displayed using (a) *Rigol DS1052E* oscilloscope (lower trace) and (b) *Goldwave*.

Example 3.12

Identification of Moving Average Filter Frequency Response Using an Adaptive Filter (`tm4c123_sysid_CMSIS_intr.c`).

In Chapter 2, program `tm4c123_sysid_CMSIS_intr.c` was used to identify the characteristics of the antialiasing and reconstruction filters of a codec. Here, the same program is used to identify the characteristics of a moving average filter. For this example, two sets of hardware connected as shown in Figure 3.29 are required. On one of the launchpads, run program `tm4c123_average_intr.c`, and on the other, run program `tm4c123_sysid_CMSIS_intr.c`. After program `tm4c123_sysid_CMSIS_intr.c` has run for a few seconds, halt the program and save the values of the 256 adaptive filter coefficients `firCoeffs32` to a file by typing

```
save <filename.dat> <start address>, <start address + 0x400>
```

at the *Command* line in the *MDK-ARM debugger*, where `start address` is the address of array `firCoeffs32`, and plot them using MATLAB® function `tm4c123_logfft()`.

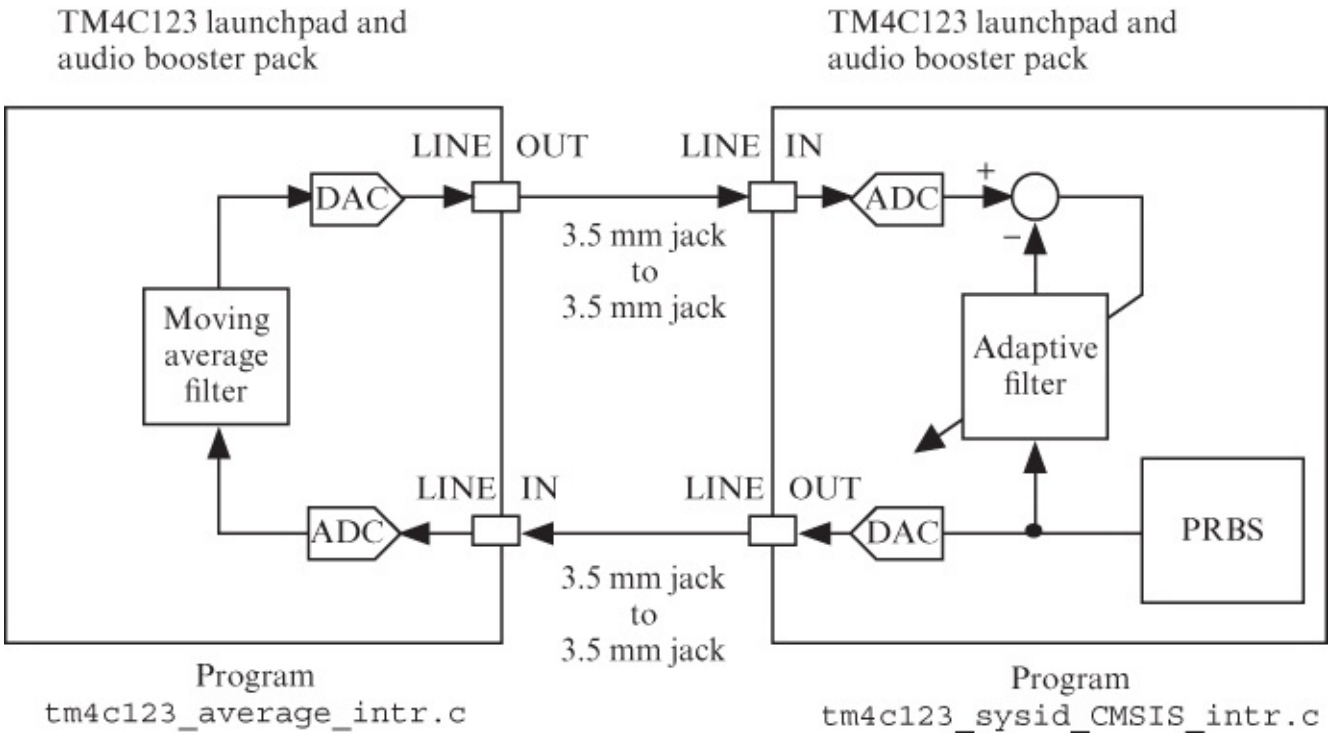


Figure 3.29 Connection diagram for use of program `tm4c123_sysid_CMSIS_intr.c` to identify the characteristics of a moving average filter implemented using two sets of hardware.

The number of adaptive filter coefficients used by the program is set by the preprocessor command

```
#define NUM_TAPS 256
```

You should see something similar to what is shown in Figures 3.30 and 3.31, that is, the impulse response and magnitude frequency response identified by the adaptive filter.

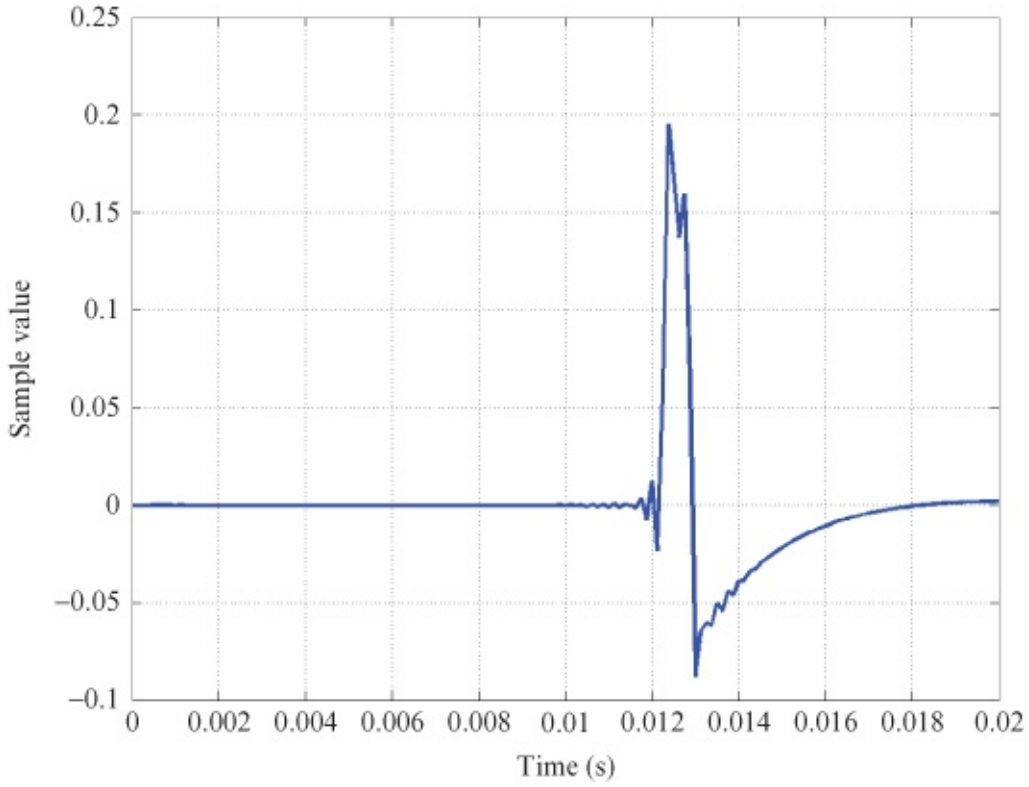


Figure 3.30 Impulse response of the five-point moving average filter identified using two launchpads and booster packs and programs `tm4c123_sysid_CMSIS_intr.c` and `tm4c123_average_intr.c`.

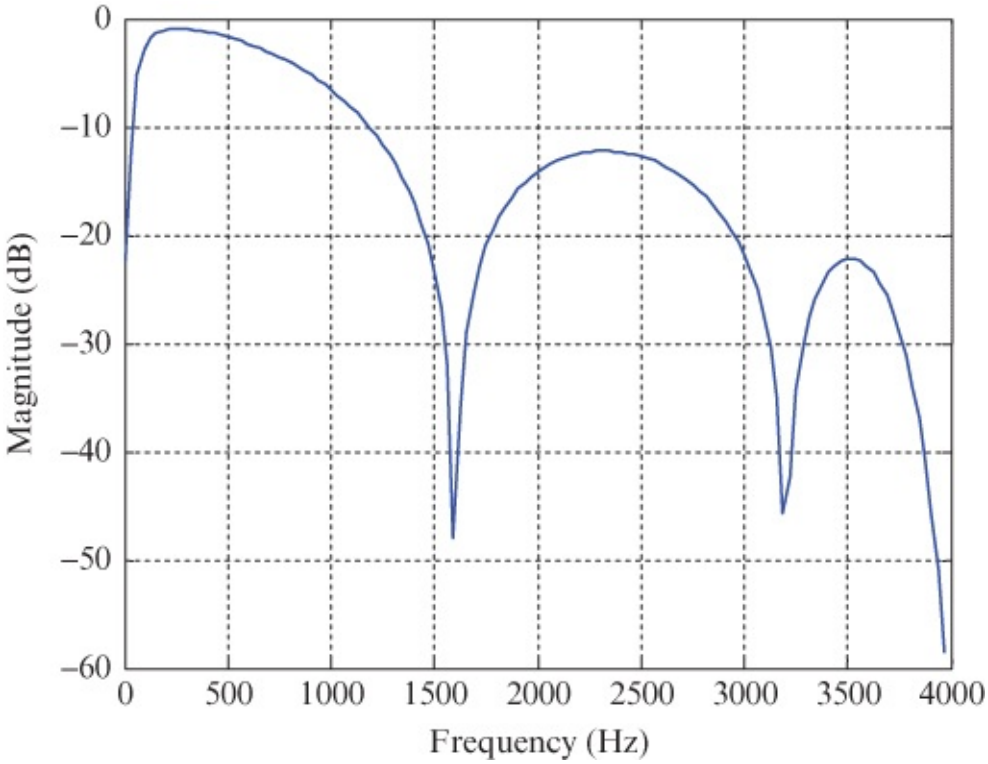


Figure 3.31 Magnitude frequency response of the five-point moving average filter identified using two sets of hardware and programs `tm4c123_sysid_CMSIS_intr.c` and `tm4c123_average_intr.c`.

The impulse response shown in Figure 3.30 differs from the theoretical (rectangular) impulse response of the moving average filter because it combines that with the responses of the reconstruction and antialiasing filters in two AIC3104 codecs and the ac coupling of the LINE IN and LINE OUT connections on the audio booster packs. The oscillations before and after transitions in the waveform are similar to those identified in Example 2.52 and shown in Figure 2.47.

In the magnitude frequency response shown in Figure 3.31, the discrepancies between theoretical and measured responses at frequencies greater than 3.5 kHz and at very low frequencies correspond to the characteristics of the antialiasing and reconstruction filters in the two AIC3104 codecs and to the ac coupling of the LINE IN and LINE OUT connections on the audio booster packs, respectively.

Example 3.14

Identification of Moving Average Filter Frequency Response Using a Single Audio Booster Pack (`tm4c123_sysid_average_CMSIS_intr.c`).

Program `tm4c123_sysid_average_CMSIS_intr.c`, shown in Listing 3.15, collapses the signal path considered in the previous example onto just one set of hardware, as shown in Figure 3.32. Build and run the program, save the 256 adaptive filter coefficients `firCoeffs32` to a file, and plot them using MATLAB function `tm4c123_logfft()`. The results should differ only subtly from those shown in Figures 3.30 and 3.31, because the identified signal path contains just one antialiasing filter and one reconstruction filter (as opposed to two of each). However, the delay before the peak in the impulse response identified should be shorter than the 12 ms apparent in Figure 3.30.

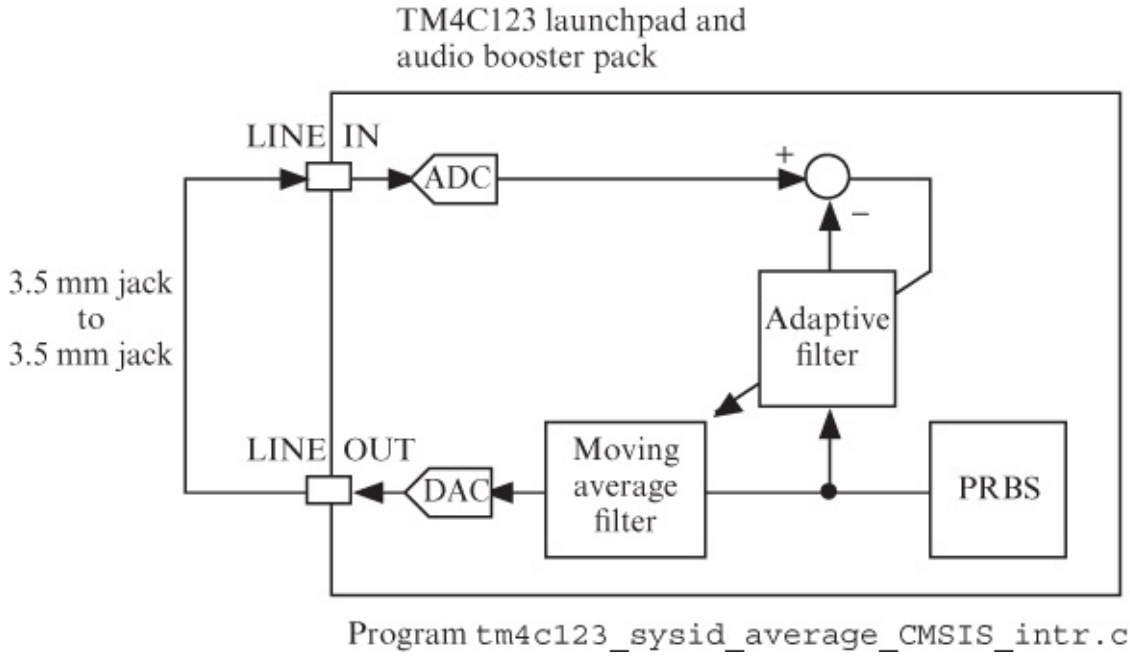


Figure 3.32 Connection diagram for program `tm4c123_sysid_average_CMSIS_intr.c`.

Listing 3.2 Program
tm4c123_sysid_average_CMSIS_intr.c.,

```
// tm4c123_sysid_average_CMSIS_intr.c
#include "tm4c123_aic3104_init.h"
#define BETA 1E-11
#define NUM_TAPS 256
#define BLOCK_SIZE 1
#define N 5
float32_t x[N];
float32_t h[N];
float32_t firStateF32[BLOCK_SIZE + NUM_TAPS -1];
float32_t firCoeffs32[NUM_TAPS] = {0.0};
arm_lms_instance_f32 S;
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  float32_t adapt_in, adapt_out, desired;
  float32_t error, input_left, input_right;
  float32_t yn = 0.0f;
  int16_t i;
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  input_left = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  input_right = (float32_t)(sample_data.bit16[0]);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  adapt_in = (float32_t)(prbs(8000));
  arm_lms_f32(&S, &adapt_in, &input_left, &adapt_out,
              &error, BLOCK_SIZE);
  x[0] = adapt_in;
  for (i=0 ; i<N ; i++) yn += h[i]*x[i];
  for (i=(N-1) ; i>0 ; i--) x[i] = x[i-1];
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
  sample_data.bit32 = ((int16_t)(yn));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
}
int main()
{
  int16_t i;
  for(i=0 ; i<N ; i++) h[i] = 1.0/N;
  arm_lms_init_f32(&S, NUM_TAPS, firCoeffs32,
                   firStateF32, BETA, BLOCK_SIZE);
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1);
}
```

3.3.1 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 `tm4c123_average_prbs_intr.c` so that it implements an eleven-point moving average filter, by changing the preprocessor command 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 3.33. Alternatively, you can make a similar change to the number of points in the moving average filter in program `tm4c123_sysid_average_CMSIS_intr.c`.

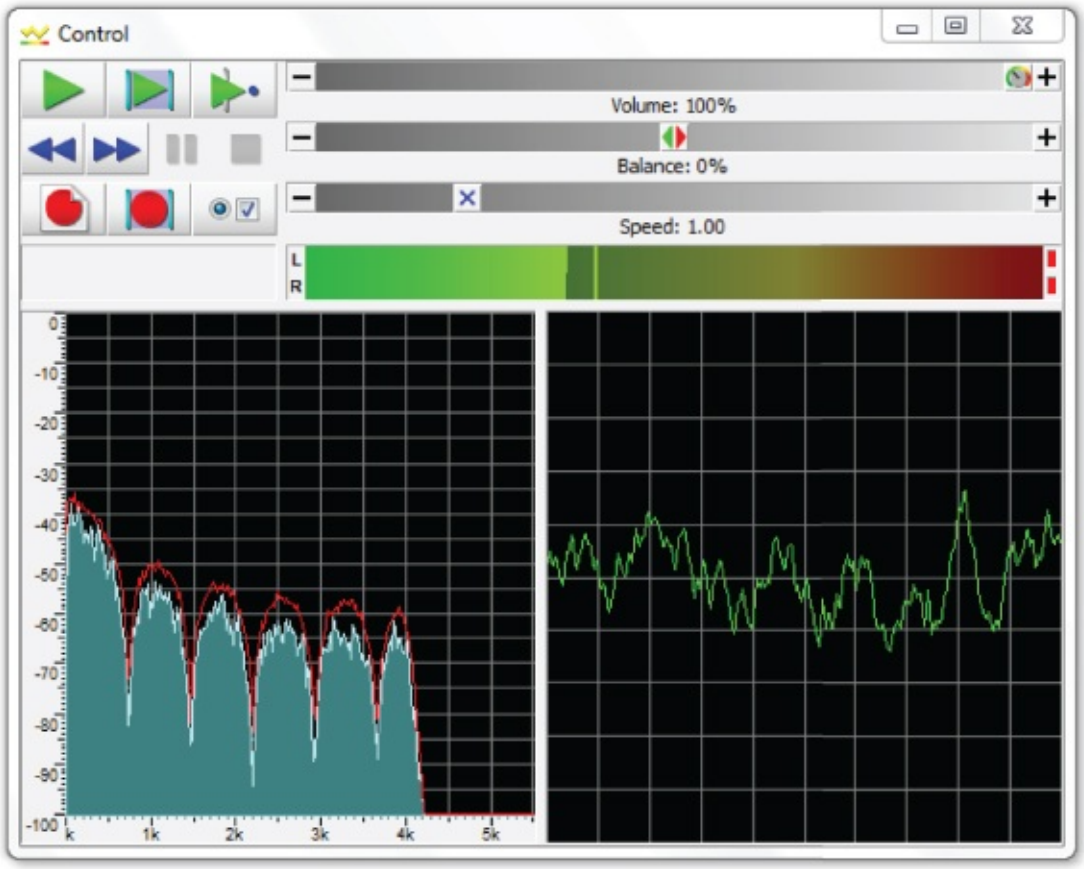


Figure 3.33 Magnitude frequency response of an eleven-point moving average filter implemented using program `tm4c123_average_prbs_intr.c` and displayed using *Goldwave*.

The frequency response of the eleven-point moving average filter has the same 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, and 2909 Hz.

The frequency response of the filter can also be changed by altering the relative values of the coefficients. Modify program `tm4c123_sysid_average_CMSIS_intr.c` again, changing the

preprocessor command and program statements 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 statement

```
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 `tm4c123_average_prbs_intr.c`) or by saving and plotting filter coefficients `firCoeffs32` (in the case of program `tm4c123_sysid_average_CMSIS_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 have disappeared as shown in Figure 3.34. You have effectively applied a Hanning window to the coefficients of the five-point moving average filter.

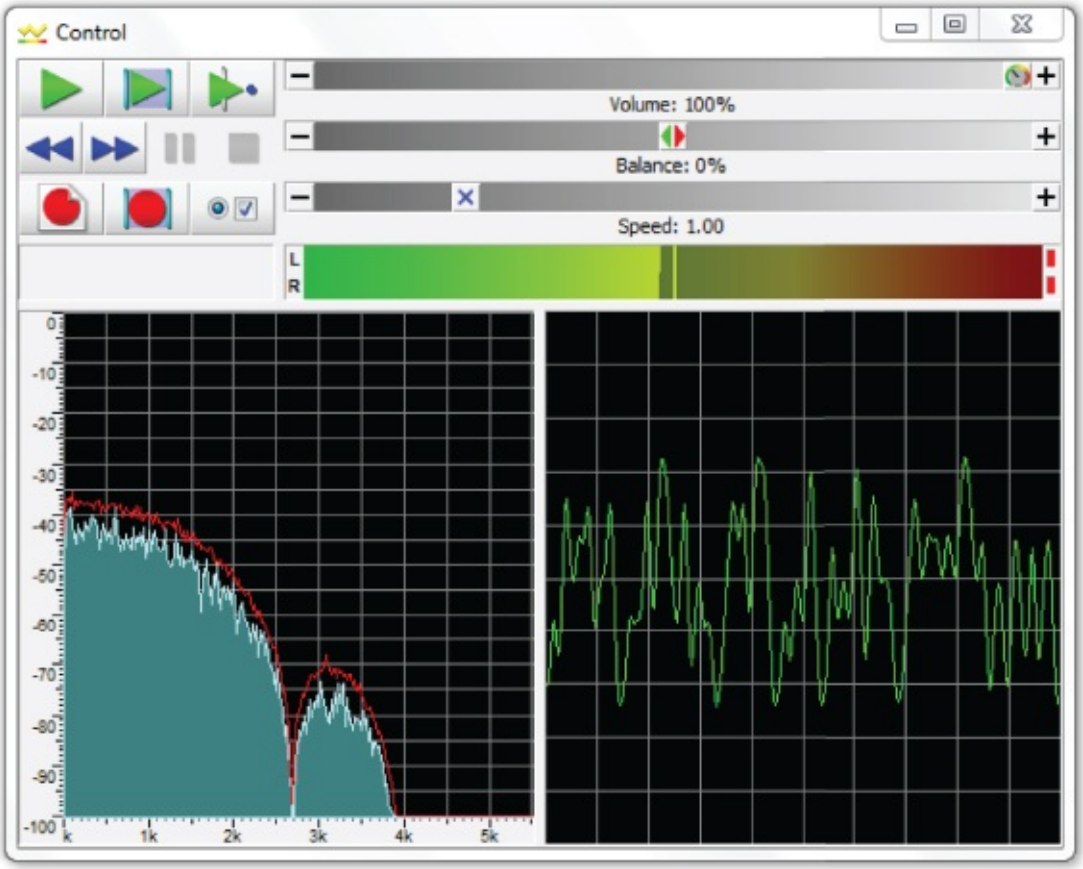


Figure 3.34 Magnitude frequency response of a five-point moving average filter with Hanning window implemented using program `stm32f4_average_prbs_intr.c` and displayed using *Goldwave*.

The N-point Hanning window is described by the equation

$$w(n) = 0.5\left(1 - \cos\left(2\pi\frac{n}{N-1}\right)\right), \quad 0 \le n < N \tag{3.59}$$

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-point Hanning window function, have been used.

The important, if rather obvious, point illustrated by this example, however, is that a five-coefficient FIR filter may exhibit different frequency response characteristics, depending on the values of its coefficients. The theoretical magnitude frequency response of the filter may be found by taking the DTFT of its coefficients.

$$\begin{aligned} H(\hat{\omega}) &= \sum_{n=0}^{4} h(n)e^{-j\hat{\omega}n} \\ &= 0.0833 + 0.25e^{j\hat{\omega}} + 0.3333e^{j2\hat{\omega}} + 0.25e^{j3\hat{\omega}} + 0.0833e^{j4\hat{\omega}} \\ &= e^{-j2\hat{\omega}}(0.0833e^{-j2\hat{\omega}} + 0.25e^{-j\hat{\omega}} + 0.3333 + 0.25e^{j\hat{\omega}} + 0.0833e^{j2\hat{\omega}}). \end{aligned} \tag{3.60}$$

Hence,

$$|H(\hat{\omega})| = |0.3333 + 0.5\cos(\hat{\omega}) + 0.1666\cos(2\hat{\omega})|. \quad \textbf{3.61}$$

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 3.2.2. Specifically, the Hanning window has a wider main lobe and relatively smaller sidelobes than a rectangular window.

Example 3.14

FIR Filter with Filter Coefficients Specified in Separate Header Files (`stm32f4_fir_intr.c` and `tm4c123_fir_intr.c`).

The algorithm used by programs `stm32f4_fir_intr.c` and `tm4c123_fir_intr.c` to calculate each output sample are identical to those employed by programs `stm32f4_average_intr.c` and `tm4c123_average_intr.c`. The interrupt service routine functions `SPI2_IRQHandler()` and `SSI_interrupt_routine()` have exactly the same definitions in each program. However, programs `stm32f4_average_intr.c` and `tm4c123_average_intr.c` calculated the values of their filter coefficients in function `main()`, whereas programs `stm32f4_fir_intr.c` (shown in Listing 3.17) and `tm4c123_fir_intr.c` read the values of their filter coefficients from separate header files.

Listing 3.3 Program `stm32f4_fir_intr.c`

```
// stm32f4_fir_intr.c
#include "stm32f4_wm5102_init.h"
#include "maf5.h"
float32_t x[N];
void SPI2_IRQHandler()
{
  int16_t left_out_sample, right_out_sample;
  int16_t left_in_sample, right_in_sample;
  int16_t i;
  float32_t yn = 0.0;
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ⌋ SET)
  {
    left_in_sample = SPI_I2S_ReceiveData(I2Sx);
    x[0] = (float32_t)(prbs(8000));
    for (i=0 ; i<N ; i++) yn += h[i]*x[i];
    for (i=(N-1) ; i>0 ; i--) x[i] = x[i-1];
    left_out_sample = (int16_t)(yn);
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
  }
  else
  {
    right_in_sample = SPI_I2S_ReceiveData(I2Sx);
    right_out_sample = 0;
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
}
int main(void)
{
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
  while(1){}
}
```

3.3.1.1 Five-Point Moving Average (`maf5.h`)

Coefficient file `maf5.h` is shown in Listing 3.18. Using that header file, programs `stm32f4_fir_intr.c` and `tm4c123_fir_intr.c` implement the same five-point moving average filter implemented by program `stm32f4_average_intr.c` in Example 3.10. The number of filter coefficients is specified by the value of the constant `N`, defined in the header file, and the coefficients are specified as the initial values in an `N` element array, `h`, of type `float32_t`. Build and run program `stm32f4_fir_intr.c` and verify that it implements a five-point moving average filter.

Listing 3.4 Coefficient header file `maf5.h`

```
// maf5.h
// this file was generated using function stm32f4_fir_coeffs.m
#define N 5
float32_t h[N] = {
2.0000E-001,2.0000E-001,2.0000E-001,2.0000E-001,2.0000E-001
};
```

3.3.1.2 Low-Pass Filter, Cutoff at 2000 Hz (`lp55.h`)

Edit source file `stm32f4_fir_intr.c` or `tm4c123_fir_intr.c`, changing the preprocessor command that reads

```
#include ave5.h
```

to read

```
#include lp55.h
```

Build and run the program. Use a signal generator connected to the (pink) LINE IN socket on the Wolfson audio card to input a sinusoidal signal and verify that this is attenuated significantly at the (green) LINE OUT socket if its frequency is greater than 2 kHz.

3.3.1.3 Band-Stop Filter, Centered at 2700 Hz (`bs2700.h`)

Edit source file `stm32f4_fir_intr.c`, changing the line that reads

```
#include ave5.h
```

to read

```
#include bs2700.h
```

Build and run program `stm32f4_fir_intr.c` or `tm4c123_fir_intr.c`. Input a sinusoidal signal and vary the input frequency slightly below and above 2700 Hz. Verify that the magnitude of the output is a minimum at 2700 Hz. The values of the coefficients for this filter were calculated using the MATLAB filter design and analysis tool, `fdatool`, as shown in Figure 3.35.

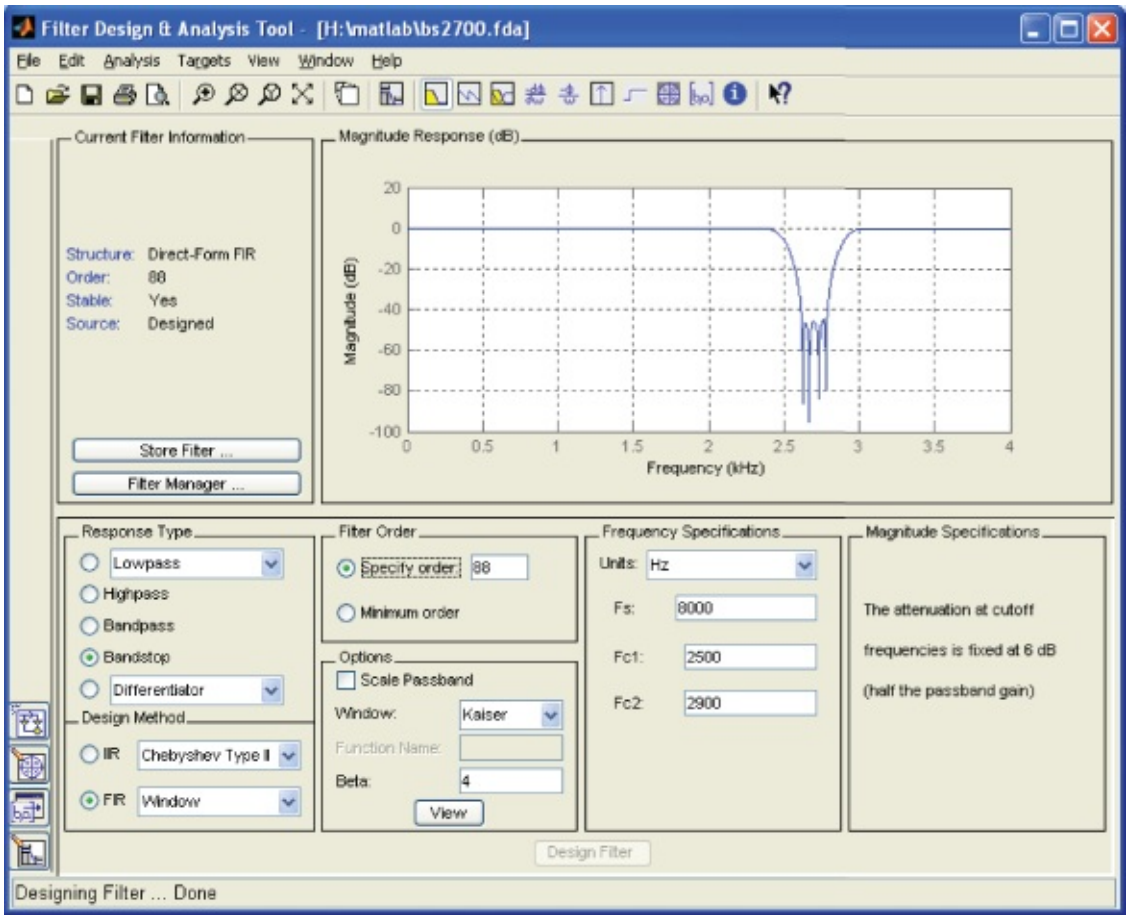


Figure 3.35 MATLAB `fdatool` window corresponding to design the of an FIR band-stop filter centered at 2700 Hz.

3.3.1.4 Band-Pass Filter, Centered at 1750 Hz (`bp1750.h`)

Edit source file `stm32f4_fir_intr.c` or `tm4c123_fir_intr.c` again to include the coefficient file `bp1750.h` in place of `bs2700.h`. File `bp1750.h` represents an FIR band-pass filter (81 coefficients) centered at 1750 Hz, as shown in Figure 3.36. Again, this filter was designed using `fdatool`. Build and run the program again and verify that it implements a band-pass filter centered at 1750 Hz.

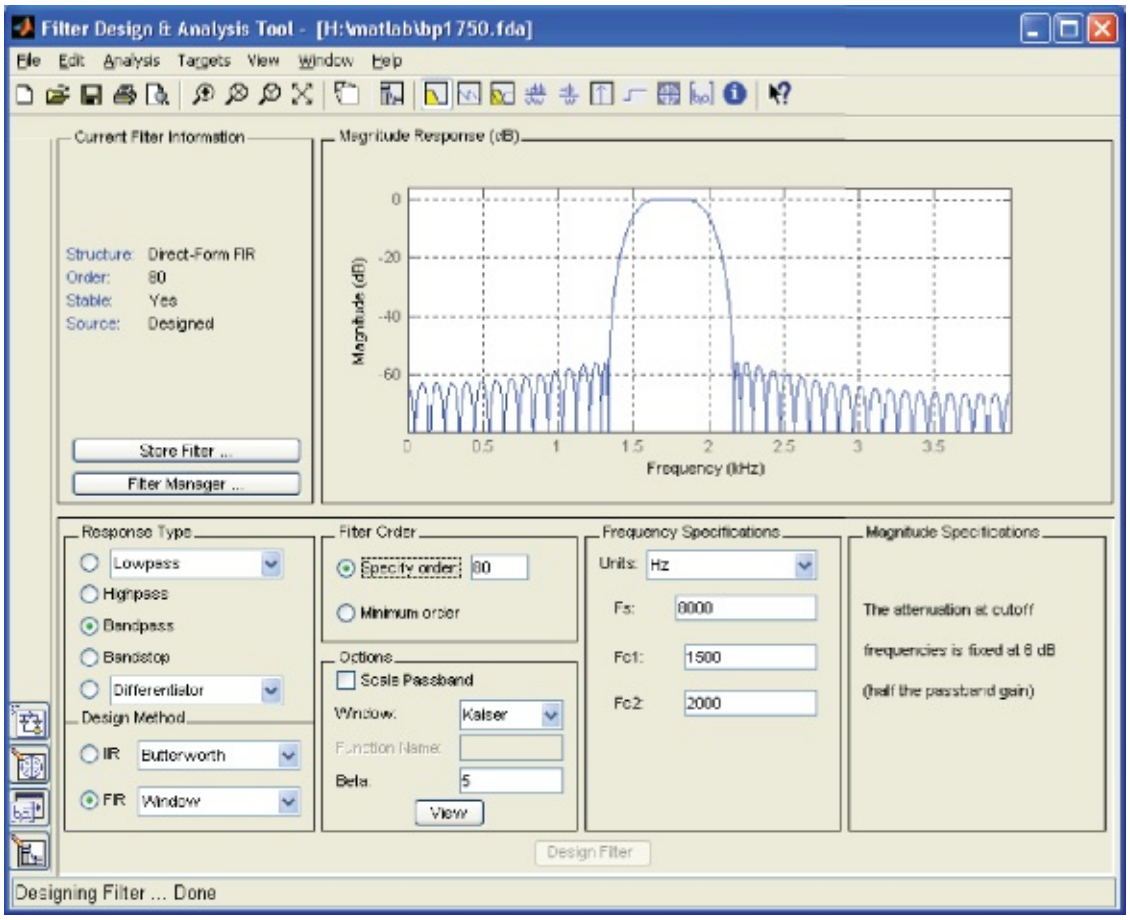


Figure 3.36 MATLAB `fdatool` window corresponding to design of FIR band-pass filter centered at 1750 Hz.

3.3.2 Generating FIR Filter Coefficient Header Files Using MATLAB

If the number of filter coefficients is small, the coefficient header file may be edited by hand. To be compatible with programs `stm32f4_fir_intr.c` and `tm4c123_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 `stm32f4_fir_coeffs()`, supplied as file `stm32f4_fir_coeffs.m`, or the MATLAB function `tm4c123_fir_coeffs()`, supplied as file `tm4c123_fir_coeffs.m`, can be used. Function `stm32f4_fir_coeffs()`, shown in Listing 3.19, should be passed a MATLAB vector of coefficient values and will prompt the user for an output filename. For example, the coefficient file `maf5.h`, shown in Listing 3.18, was created by typing the following at the MATLAB command prompt:

```
>> x = [0.2, 0.2, 0.2, 0.2, 0.2];
>> stm32f4_fir_coeffs(x)
enter filename for coefficients maf5.h
```

Note that the coefficient filename must be entered in full, including the suffix `.h`. 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 `stm32f4_fir_coeffs()` or `tm4c123_fir_coeffs()` can be used to create a coefficient

header file compatible with programs `stm32f4_fir_intr.c` and `tm4c123_fir_intr.c`.

Listing 3.5 MATLAB m-file `stm32f4_fir_coeffs.m`

```
% STM32F4_FIR_COEFFS.M
% MATLAB function to write FIR filter coefficients
% in format suitable for use in STM32F407 Discovery programs
% stm32f4_fir_intr.c and stm32f4_fir_prbs_intr.c
% written by Donald Reay
%
function stm32f4_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 stm32f4_fir_coeffs.m\n');
fprintf(fid,'\n#define N %d\n',coefflen);
% j is used to count coefficients written to current line
% in output file
fprintf(fid,'\nfloat32_t h[N] = { \n');
j=0;
% i is used to count through coefficients
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};\n');
fclose(fid);
}
```

Example 3.15

FIR Implementation with Pseudorandom Noise as Input (`tm4c123_fir_prbs_intr.c`).

Program `tm4c123_fir_prbs_intr.c`, shown in Listing 3.21, implements an FIR filter and uses an internally generated pseudorandom noise sequence as input. In all other respects, it is similar to program `tm4c123_fir_intr.c`. The coefficient file `bs2700.h` is used initially.

Listing 3.6 Program `tm4c123_fir_prbs_intr.c`

```
// tm4c123_fir_prbs_intr.c
#include "tm4c123_aic3104_init.h"
#include "bs2700.h"
float32_t x[N];
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  float32_t input_left, input_right;
  int16_t i;
  float32_t yn = 0.0f;
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  input_left = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  input_right = (float32_t)(sample_data.bit16[0]);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  x[0] = (float32_t)(prbs(8000));
  for (i=0 ; i<N ; i++) yn += h[i]*x[i];
  for (i=N-1 ; i>0 ; i--) x[i] = x[i-1];
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
  sample_data.bit32 = ((int16_t)(yn));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
}
int main(void)
{
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1){}
}
```

3.3.2.1 Running the Program

Build and run the program and verify that the output signal is pseudorandom noise filtered by an FIR band-stop filter centered at 2700 Hz. This output signal is shown using *GoldWave* and using the FFT function of a *Rigol DS1052E* oscilloscope in Figure 3.37.

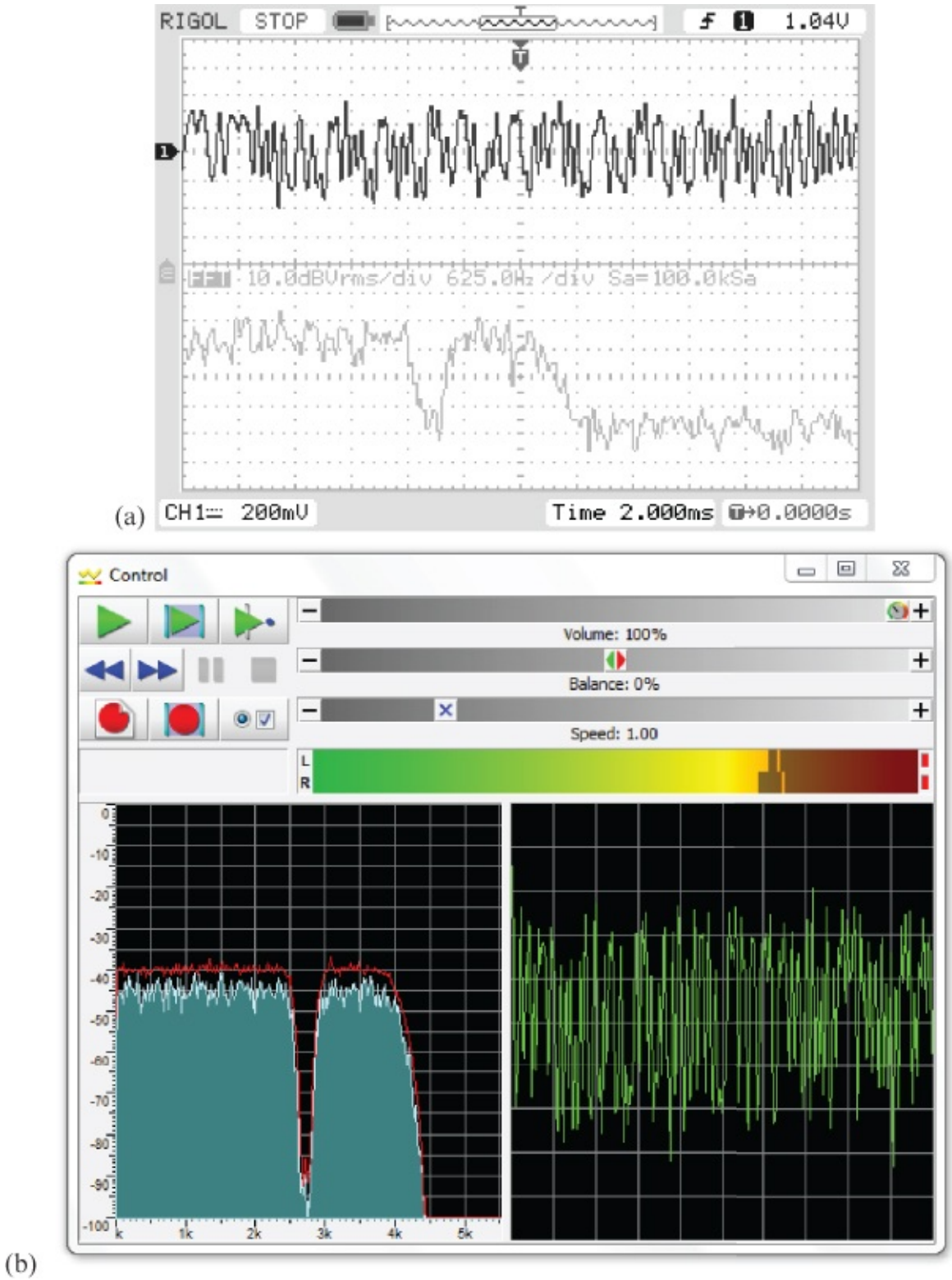


Figure 3.37 Output generated using program `tm4c123_fir_prbs_intr.c` and coefficient file `bs2700.h` displayed using (a) *Rigol DS1052E* oscilloscope and (b) *GoldWave*.

Testing Different FIR Filters

Edit the C source file `tm4c123_fir_prbs_intr.c` to include and test different coefficient files representing different FIR filters. Each of the following coefficient files, except `comb14.h`, contains 55 coefficients.

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

Figure 3.38(a) shows the filtered noise output by an FIR filter with two pass bands, using the coefficient file `pass2b.h`. Figure 3.38(b) shows the filtered noise output by a high-pass FIR filter using the coefficient file `hp55.h`. In the cases of high-pass and band-stop filters in particular, the low-pass characteristic of the reconstruction filter in the AIC3104 codec is apparent.

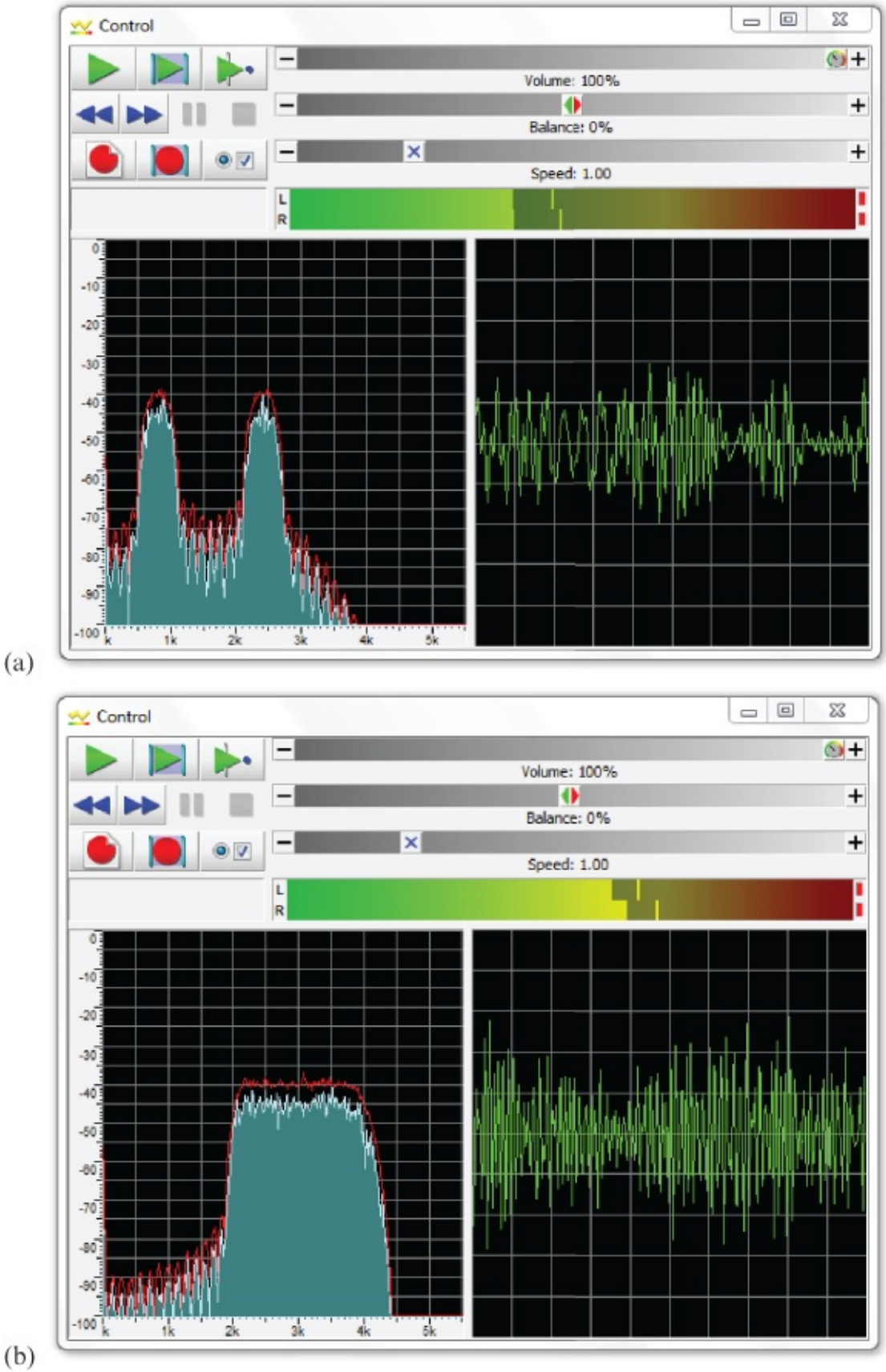


Figure 3.38 Output generated using program `tm4c123_fir_prbs_intr.c` using coefficient files (a) `pass2b.h` and (b) `hp55.h`.

Example 3.16

FIR Filter with Internally Generated Pseudorandom Noise as Input and Output Stored in Memory (`stm32f4_fir_prbs_buf_intr.c`).

This example extends the previous one by storing the 256 most recent output samples in memory. Program `stm32f4_fir_prbs_buf_intr.c` is shown in Listing 3.23. The coefficient file `bp1750.h` represents an 81-coefficient FIR band-pass filter centered at 1750 Hz.

Listing 3.7 Program `stm32f4_fir_prbs_buf_intr.c`

```
// stm32f4_fir_prbs_buf_intr.c
#include "stm32f4_wm5102_init.h"
#include "bp1750.h"
#define YNBUFLENGTH 256
float32_t ynbuffer[YNBUFLENGTH];
int16_t ynbufptr = 0;
float32_t x[N];
void SPI2_IRQHandler()
{
  int16_t left_out_sample, right_out_sample;
  int16_t left_in_sample, right_in_sample;
  int16_t i;
  float32_t yn = 0.0;
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ⌡ SET)
  {
    left_in_sample = SPI_I2S_ReceiveData(I2Sx);
    x[0] = (float32_t)(prbs(8000));
    for (i=0 ; i<N ; i++) yn += h[i]*x[i];
    for (i=(N-1) ; i>0 ; i--) x[i] = x[i-1];
    left_out_sample = (int16_t)(yn);
    ynbuffer[ynbufptr] = yn;
    ynbufptr = (ynbufptr+1) % YNBUFLENGTH;
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
  }
  else
  {
    right_in_sample = SPI_I2S_ReceiveData(I2Sx);
    right_out_sample = 0;
    while(SPI_I2S_GetFlagStatus(I2Sxext,
          SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
}
int main(void)
{
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
  while(1){}
}
```

3.3.2.2 Running the Program

Build and run the program. Verify that the output signal is bandlimited noise. Then halt the program and save the contents of array `ynbuffer` to a data file by typing

```
save <filename.dat> <start address>, <start address + 0x400>
```

where `start` address is the address of array `ynbuffer`. Use MATLAB function `stm32f4_logfft()` in order to display the frequency content of the 256 stored output samples, as shown in Figure 3.39 (Figure 3.40).

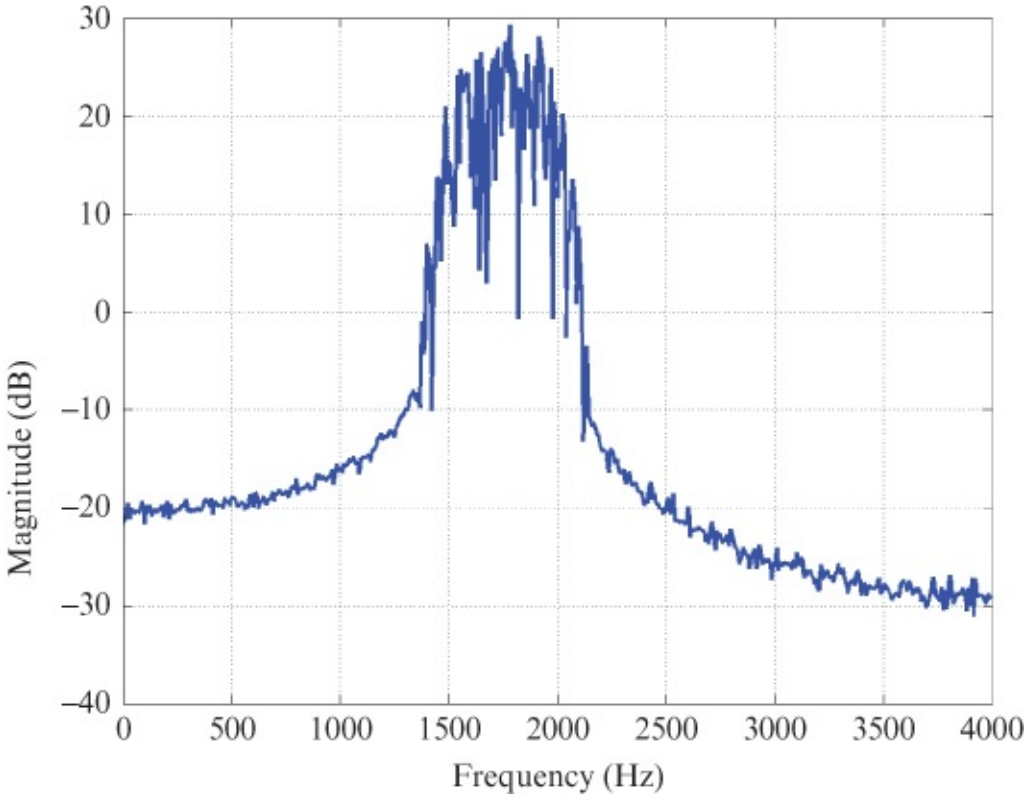


Figure 3.39 Magnitude of the FFT of the output from program `stm32f4_fir_prbs_buf_intr.c` using coefficient header file `bp1750.h`.

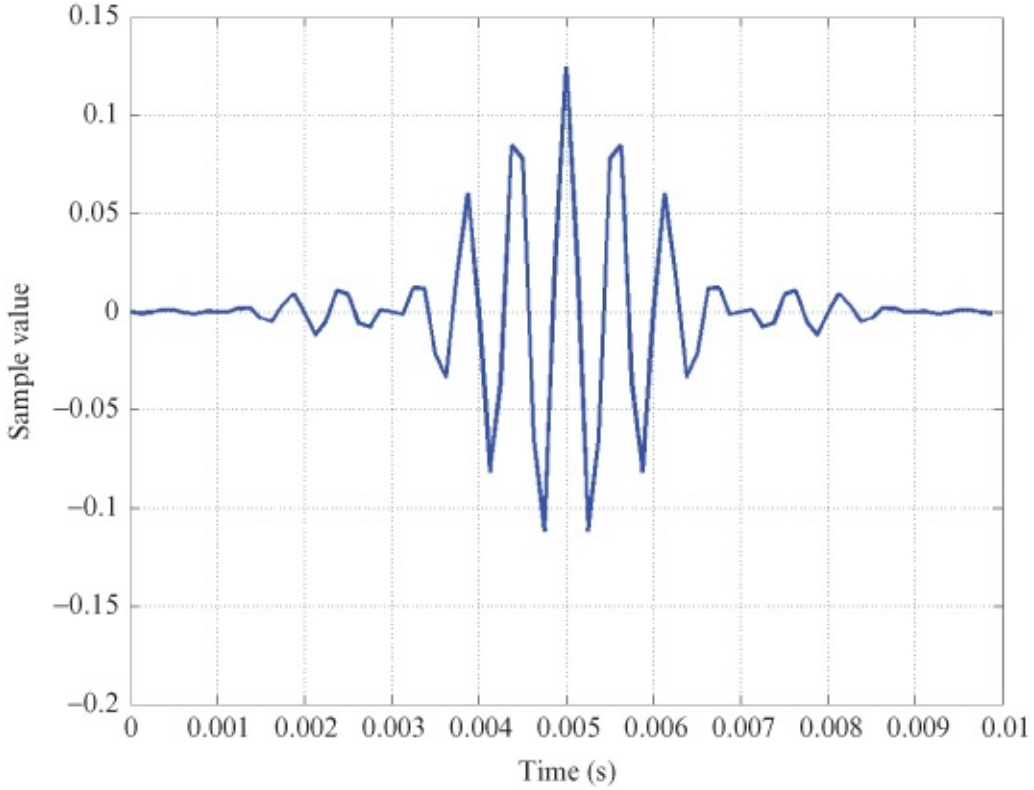


Figure 3.40 Filter coefficients used in program `stm32f4_fir_prbs_buf_intr.c` (`bp1750.h`).

Figure 3.41 shows the magnitude of the FFT of the `bp1750.h` filter coefficients for comparison.

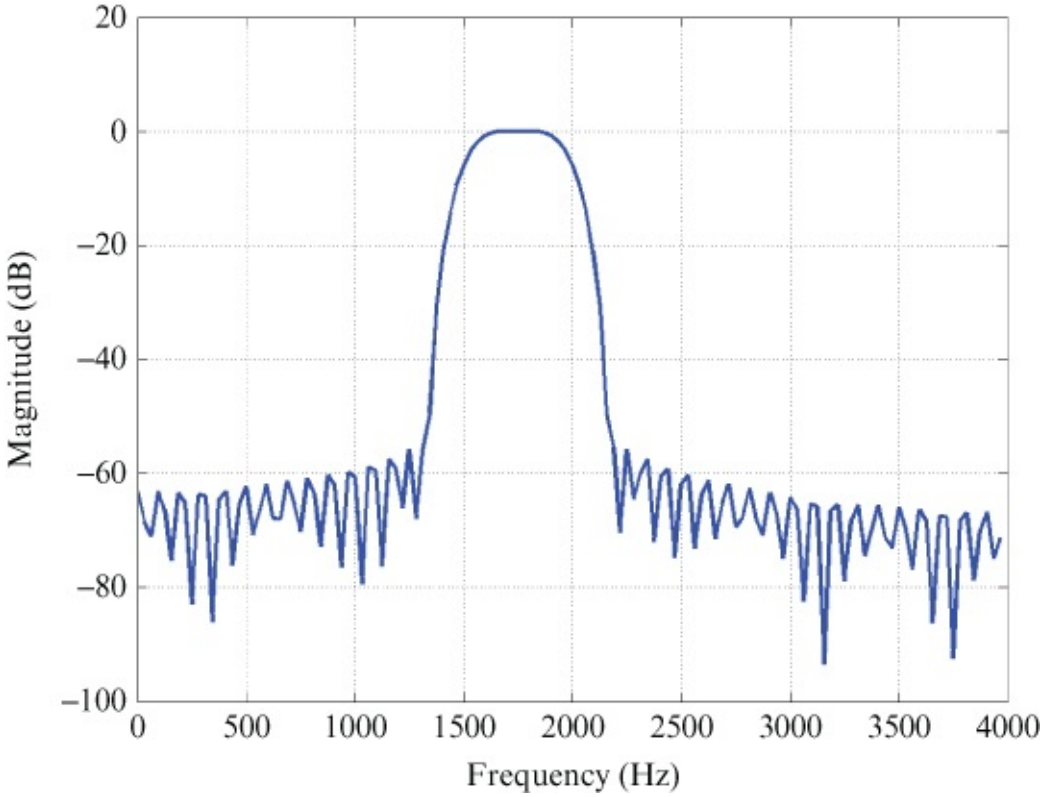


Figure 3.41 Magnitude of the FFT of the filter coefficients used in program `stm32f4_fir_prbs_buf_intr.c`.

Program `stm32f4_fir_prbs_buf_intr.c` allows the user to switch the signal written to the WM5102 DAC between filtered and unfiltered noise so as to emphasize the action of the filter. The filter is toggled on and off by pressing the (blue) user pushbutton on the Discovery board.

Example 3.17

Effects on Voice or Music Using Three FIR Low-Pass Filters (`tm4c123_fir3lp_intr.c`).

Listing 3.25 show program `stm32f4_fir3lp_intr.c`, which implements three different FIR low-pass filters with cutoff frequencies at 600, 1500, and 3000 Hz. Filter coefficients designed using MATLAB are read from file `fir3lp_coeffs.h` and, during initialization, copied into a single, two-dimensional array `h`. While the program is running, variable `FIR_number` selects the desired low-pass filter to be implemented. For example, if `FIR_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 `FIR_number` can be cycled through the values 0 through 2 to implement the 600, 1500, or 3000 Hz low pass filter, using switch SW1 on the launchpad while the program is running.

Listing 3.8 Program `tm4c123_fir3lp_intr.c`

```
// tm4c123_fir3lp_intr.c
#include "tm4c123_aic3104_init.h"
#include "L138_fir3lp_coeffs.h"
float32_t x[N];
float32_t h[3][N];
int16_t FIR_number = 0;
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  float32_t input_left, input_right;
  int16_t i;
  float32_t yn = 0.0f;
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  input_left = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  input_right = (float32_t)(sample_data.bit16[0]);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  x[0] = input_left;
  for (i=0 ; i<N ; i++) yn += h[FIR_number][i]*x[i];
  for (i=(N-1) ; i>0 ; i--) x[i] = x[i-1];
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
  sample_data.bit32 = ((int16_t)(yn));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
}
int main(void)
{
  int16_t i;
  for (i=0; i<N; i++)
  {
    x[i] = 0.0;
    h[0][i] = hlp600[i];
    h[1][i] = hlp1500[i];
    h[2][i] = hlp3000[i];
  }
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1)
  }
    ROM_SysCtlDelay(10000);
    if (!GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_4))
    {
      ROM_SysCtlDelay(10000);
      FIR_number = (FIR_number+1) % 3;
      while (!GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_4)){}
    }
  }
```

```
}
```

As supplied, the program configures the AIC3104 codec to accept input from the (blue) LINE IN socket on the audio booster pack. In order to test the effect of the filters using a microphone as an input device, change the program statement that reads

```
tm4c123_aic3104_init(FS_8000_HZ,
                     AIC3104_LINE_IN,
                     IO_METHOD_INTR,
                     PGA_GAIN_6_DB);
```

to read

```
tm4c123_aic3104_init(FS_8000_HZ,
                     AIC3104_MIC_IN,
                     IO_METHOD_INTR,
                     PGA_GAIN_6_DB);
```

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 3.42 shows a 200 Hz square wave that has been passed through the three different low-pass filters. The slope on the sections of the waveforms between transitions is due to the ac coupling of the LINE OUT and LINE IN connections on the audio booster pack.

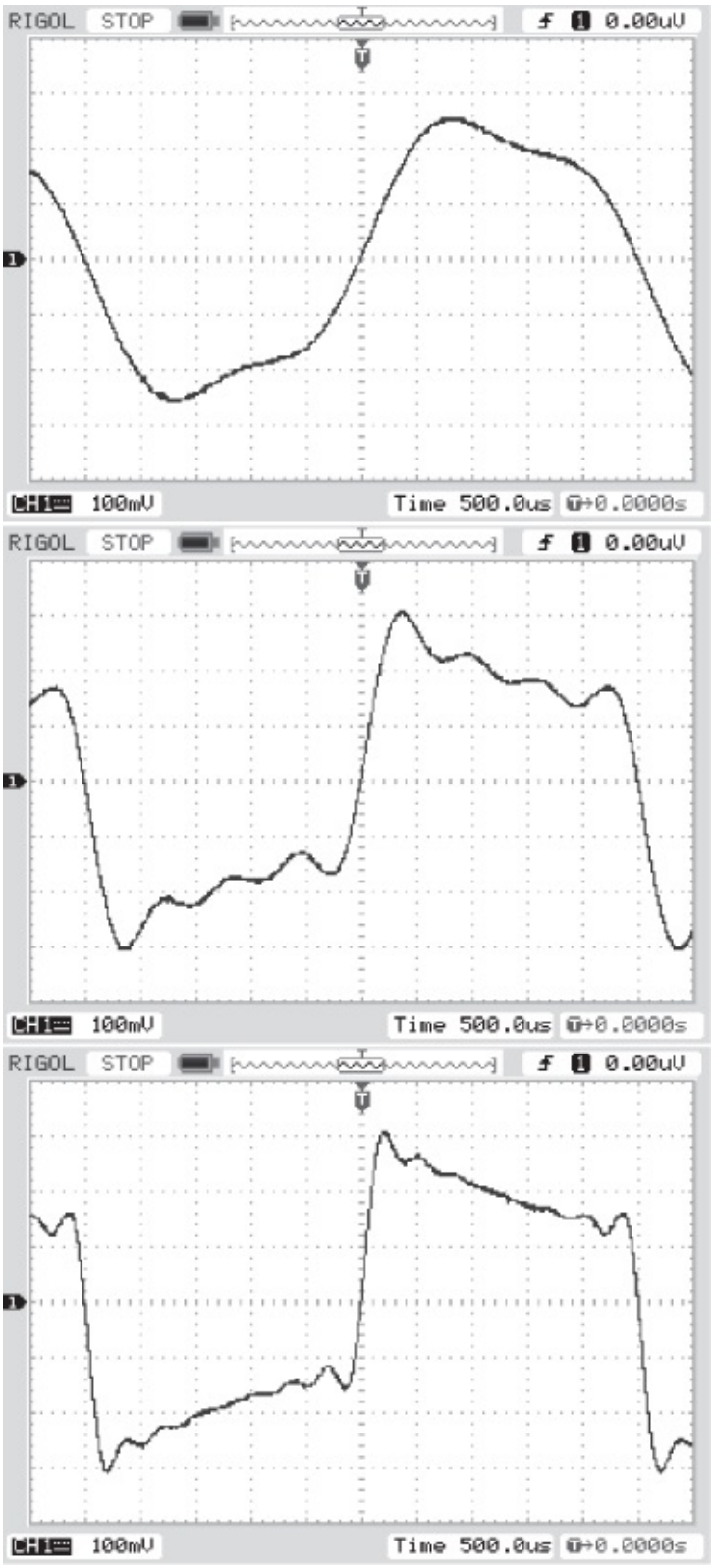


Figure 3.42 A 200 Hz square wave passed through three different low-pass filters implemented using program `tm4c123_fir3lp_intr.c`.

The magnitude frequency response of the three different filters may be observed using *Goldwave* and by changing the program statement that reads

```
x[0] = input_left;
```

to read

```
x[0] = (float32_t)(prbs(8000));
```

or, alternatively, by blowing gently on the microphone.

Example 3.18

Implementation of Four Different Filters: Low-Pass, High-Pass, Band-Pass, and Band-Stop (`tm4c123_fir4types_intr.c`).

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 `tm4c123_fir4types_intr.c` is running using switch SW1 on the launchpad. All four 81-coefficient filters were designed using MATLAB. They are

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 3.43 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.

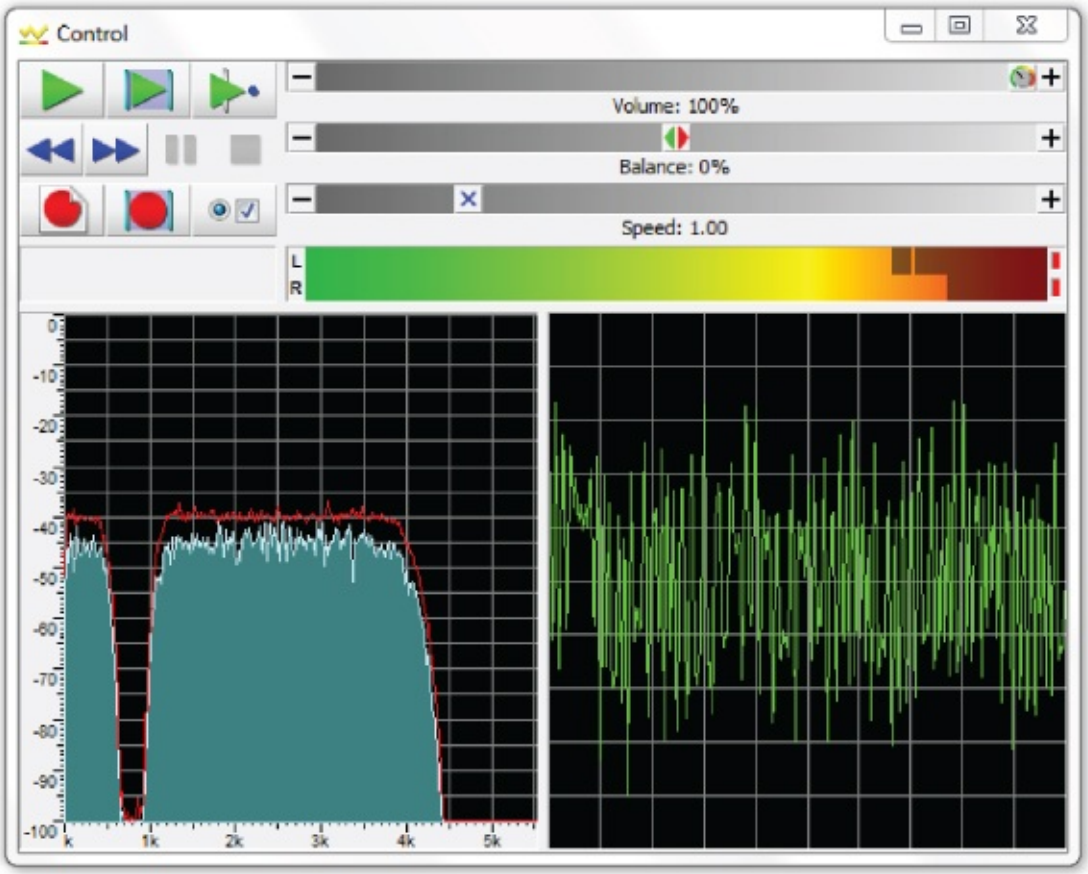


Figure 3.43 Output generated using program `tm4c123_fir_4types_intr.c`.

Example 3.19

Two Notch Filters to Recover a Corrupted Speech Recording (`tm4c123_notch2_intr.c`).

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 `tm4c123_notch2_intr.c` is shown in Listing 3.28. 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` 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 (blue) LINE IN socket on the audio booster pack and listen to the filtered test signal on (black) LINE OUT or (green) HP OUT connections. Switch SW1 on the launchpad can be used to select the output of either the first or the second of the notch filters. Compare the results of this example with those obtained in Example 3.10 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 may sound brighter because the notch filters used here do not have an overall low-pass characteristic. Figure 3.44 shows the output of the filter using pseudorandom noise as an input signal.

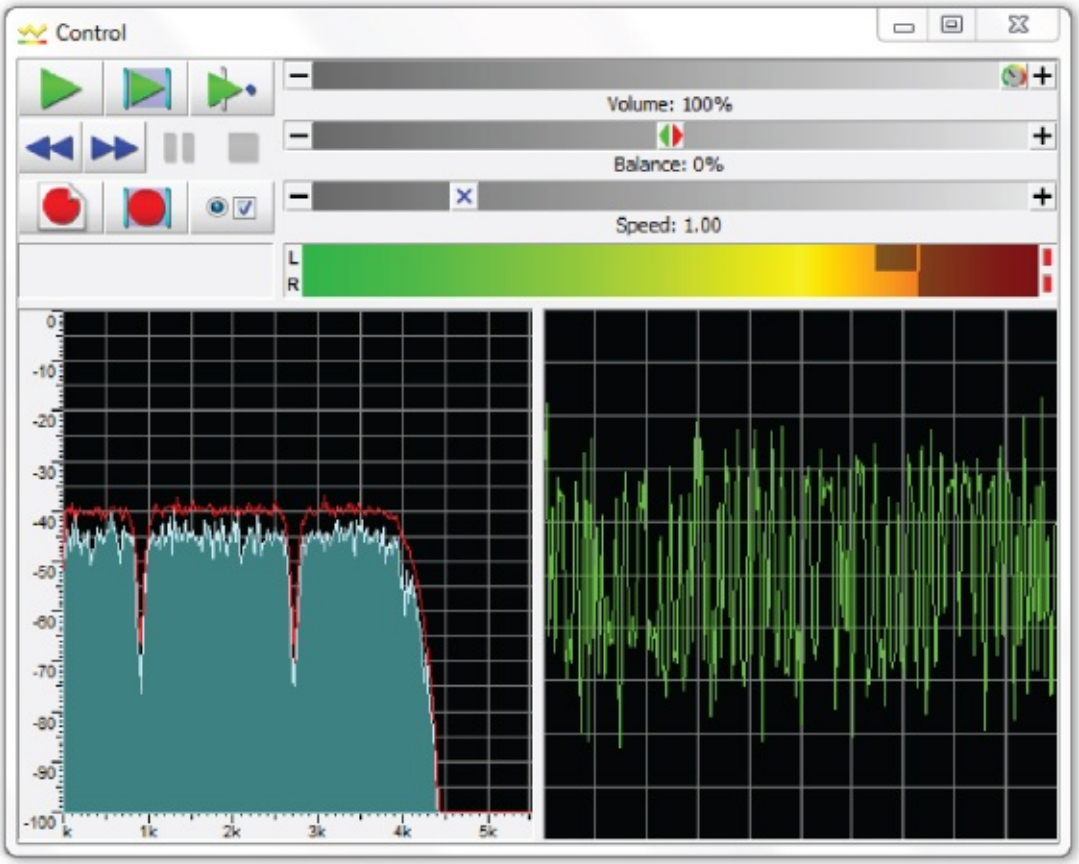


Figure 3.44 Pseudorandom noise filtered using program `tm4c123_notch2_intr.c`.

Listing 3.9 Program `tm4c123_notch2_intr.c`

```
// tm4c123_notch2_intr.c
#include “tm4c123_aic3104_init.h”
#include “notch2_coeffs.h”
float32_t x[N][2];      // filter delay lines
int16_t out_type = 0;
AIC3104_data_type sample_data;
void SSI_interrupt_routine(void)
{
  float32_t input_left, input_right;
  int16_t i;
  float32_t yn[2];
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  input_left = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  input_right = (float32_t)(sample_data.bit16[0]);
  x[0][0] = input_left;
```

```
  yn[0] = 0.0;              // compute filter 1 output
  for (i = 0; i< N; i++) yn[0] += h900[i]*x[i][0];
  x[0][1] = (yn[0]);
  yn[1] = 0.0;              // compute filter 2 output
  for (i = 0; i< N; i++) yn[1] += h2700[i]*x[i][1];
  for (i = N-1; i > 0; i--) // shift delay lines
  {
    x[i][0] = x[i-1][0];
    x[i][1] = x[i-1][1];
  }
  sample_data.bit32 = ((int16_t)(yn[out_type]));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
}
int main(void)
{
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1)
  {
    ROM_SysCtlDelay(10000);
    if (!GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_4))
    {
      ROM_SysCtlDelay(10000);
      out_type = (out_type+1) %2;
      while (!GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_4)){}
    }
  }
}
```

Example 3.20

Voice Scrambling Using Filtering and Modulation (`tm4c123_scrambler_intr.c`).

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 scrambled voice is used as the input to a second system 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 to 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, which represents the audible speech range, the low tones are high tones, and vice versa.

Figure 3.45 shows a block diagram of the scrambling scheme. At point A, the input signal has been bandlimited (low pass filtered) to 3 kHz. At point B, a double-sideband signal with suppressed carrier has been formed. 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. Listing 3.30 is of program `tm4c123_scrambler_intr.c`, which operates at a sampling rate 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.h`. The filtering algorithm used is identical to that used in, for example, programs `tm4c123_fir_intr.c` and `stm32f4_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 3.45) 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.h`) 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 system running the same algorithm, the original descrambled input may be recovered.

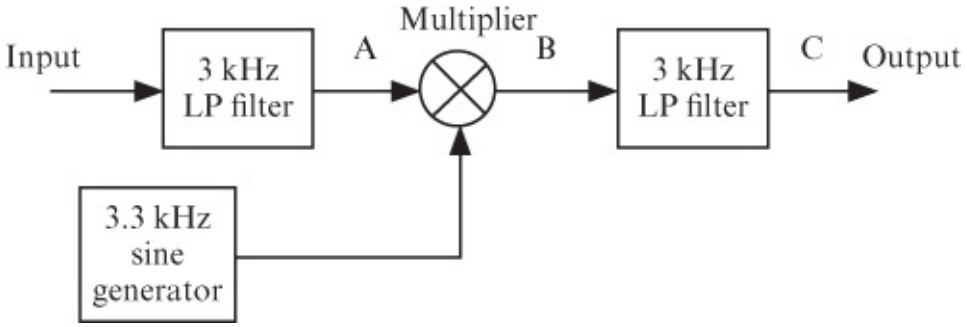


Figure 3.45 Block diagram representation of scrambler implemented using program `tm4c123_scrambler_intr.c`.

3.3.2.3 Running the Program

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 300–3000 Hz appear as output frequencies in the inverted range 3000–300 Hz.

Listing 3.10 Program `tm4c123_scrambler_intr.c`

```
// tm4c123_scrambler_intr.c
#include “tm4c123_aic3104_init.h”
#include “sine160.h”   // 3300 Hz sinusoid
#include “lp3k64.cof”  // low-pass filter coefficients
float xa[N],xb[N];     // filter delay lines
int sine_ptr = 0;      // pointer to sinusoid samples
AIC3104_data_type sample_data;
void SSI_interrupt_routine(void)
{
  float32_t input_left, input_right;
  float32_t yn;
  int16_t i;
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  input_left = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  input_right = (float32_t)(sample_data.bit16[0]);
  xa[0] = (input_left);               // filter a
  yn = 0.0;
  for (i=0 ; i<N ; i++) yn += h[i]*xa[i];
  for (i=(N-1) ; i>0 ; i--) xa[i] = xa[i-1];
  yn *= sine160[sine_ptr];            // mix with 3300 Hz
  sine_ptr = (sine_ptr+1)  % NSINE;
  xb[0] = yn;                         // filter b
  yn = 0.0;
  for (i=0 ; i<N ; i++) yn += h[i]*xb[i];
  for (i=(N-1) ; i>0 ; i--) xb[i] = xb[i-1];
  sample_data.bit32 = ((int16_t)(yn));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
}
int main(void)
{
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1){}
}
```

A second hardware system running the same program can be used to recover the original signal (simulating the receiving end). Use the output of the first audio card as the input to the second. In order to test the scrambler and descrambler using speech from a microphone as the input, change the program statement that reads

```
tm4c123_aic3104_init(FS_8000_HZ,
```

```
                        AIC3104_LINE_IN,
                        IO_METHOD_INTR,
                        PGA_GAIN_6_DB);
```

to read

```
tm4c123_aic3104_init(FS_8000_HZ,
                        AIC3104_MIC_IN,
                        IO_METHOD_INTR,
                        PGA_GAIN_6_DB);
```

at the transmitting end.

Connect LINE OUT (black) on the transmitting end system (scrambler) to LINE IN (blue) on the receiving end system (descrambler). Interception and descrambling of the scrambled 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.

Example 3.21

FIR filter implemented using DMA-based i/o (`tm4c123_fir_dma.c` and `stm32f4_fir_dma.c`).

Programs `tm4c123_fir_dma.c`, `stm32f4_fir_dma.c`, `tm4c123_fir_prbs_dma.c` and `stm32f4_fir_prbs_dma.c` have similar functionality to programs `tm4c123_fir_intr.c`, `stm32f4_fir_intr.c`, `tm4c123_fir_prbs_intr.c` and `stm32f4_fir_prbs_intr.c` but use DMA-based as opposed to interrupt-based i/o (as illustrated previously in Examples 2.5 and 2.6). As supplied, they implement band-pass filters centered on 1750 Hz. The filter characteristics implemented by the programs can be changed by including different coefficient header files.

Example 3.22

FIR Filter Implemented Using a CMSIS DSP Library Function (`tm4c123_fir_prbs_CMSIS_dma.c` and `stm32f4_fir_prbs_CMSIS_dma.c`).

Function `arm_fir_f32()` supplied as part of the CMSIS DSP library is an optimized floating-point implementation of an FIR filter. The function is passed a block of input samples and computes a corresponding block of output samples, and hence, it is suited to DMA-based i/o (although it is possible to set the size of the block of samples to one and to call the function in a program using interrupt-based i/o). Apart from using the CMSIS DSP library function, programs `tm4c123_fir_prbs_CMSIS_dma.c` and `stm32f4_fir_prbs_CMSIS_dma.c` (shown in Listing 3.33) are similar to programs `tm4c123_fir_prbs_dma.c` and `stm32f4_fir_prbs_dma.c`.

3.3.2.4 Number of Sample Values in Each DMA Transfer

The number of sample values in each DMA transfer is set in header files `stm32f4_wm5102_init.h` and `tm4c123_aic3104_init.h` by the value of the constant `BUFSIZE`. However, in the example programs in this book and as described in Chapter 2, whereas the TM4C123 is configured so that four DMA transfers take place concurrently (one in each direction for each of the two audio channels L and R), whereas STM32F4 is configured so that two DMA transfers take place concurrently (one in each direction, but each one transferring both L and R channel sample values). In each case, the value of the constant `BUFSIZE` determines the number of 16-bit values per DMA transfer.

In the case of TM4C123, each DMA transfer block processed by function `Lprocess_buffer()` contains `BUFSIZE` 16-bit L channel samples and one call to function `arm_fir_f32()` processes these `BUFSIZE` sample values.

In the case of STM32F4, each DMA transfer block processed by function `process_buffer()` contains `BUFSIZE/2` 16-bit L channel samples interleaved with `BUFSIZE/2` 16-bit R channel samples and one call to function `arm_fir_f32()` is used to process `BUFSIZE/2` L channel sample values.

Listing 3.11 Program `stm32f4_fir_prbs_CMSIS_dma.c`

```
// stm32f4_fir_prbs_CMSIS_dma.c
#include "stm32f4_wm5102_init.h"
#include "bp1750.h"
extern uint16_t pingIN[BUFSIZE], pingOUT[BUFSIZE];
extern uint16_t pongIN[BUFSIZE], pongOUT[BUFSIZE];
int rx_proc_buffer, tx_proc_buffer;
volatile int RX_buffer_full = 0;
volatile int TX_buffer_empty = 0;
float32_t x[BUFSIZE/2], y[BUFSIZE/2], state[N+(BUFSIZE/2)-1];
arm_fir_instance_f32 S;
void DMA1_Stream3_IRQHandler()
{
  if(DMA_GetITStatus(DMA1_Stream3,DMA_IT_TCIF3))
  {
    DMA_ClearITPendingBit(DMA1_Stream3,DMA_IT_TCIF3);
    if(DMA_GetCurrentMemoryTarget(DMA1_Stream3))
      rx_proc_buffer = PING;
    else
      rx_proc_buffer = PONG;
    RX_buffer_full = 1;
  }
}
void DMA1_Stream4_IRQHandler()
{
  if(DMA_GetITStatus(DMA1_Stream4,DMA_IT_TCIF4))
  {
    DMA_ClearITPendingBit(DMA1_Stream4,DMA_IT_TCIF4);
    if(DMA_GetCurrentMemoryTarget(DMA1_Stream4))
```

```
      tx_proc_buffer = PING;
    else
      tx_proc_buffer = PONG;
    TX_buffer_empty = 1;
  }
}
void process_buffer()
{
  int i;
  uint16_t *rxbuf, *txbuf;
  if (rx_proc_buffer ≠ PING)
    rxbuf = pingIN;
  else
    rxbuf = pongIN;
  if (tx_proc_buffer ≠ PING)
    txbuf = pingOUT;
  else
    txbuf = pongOUT;
// place BUFSIZE/2 prbs values in x[]
  for (i=0 ; i<(BUFSIZE/2) ; i++)
  {
    x[i] = (float32_t)(prbs(8000));
  }
// compute BUFSIZE/2 filter output values in y[]
  arm_fir_f32(&S,x,y,BUFSIZE/2);
// write BUFSIZE/2 samples to output channels
  for (i=0 ; i<(BUFSIZE/2) ; i++)
  {
    *txbuf++ = (int16_t)(y[i]);
    *txbuf++ = (int16_t)(y[i]);
  }
  TX_buffer_empty = 0;
  RX_buffer_full = 0;
}
int main(void)
{
  arm_fir_init_f32(&S, N, h, state, BUFSIZE/2);
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_DMA);
  while(1)
  {
    while (!(RX_buffer_full && TX_buffer_empty)){}
    GPIO_SetBits(GPIOD, GPIO_Pin_15);
    process_buffer();
    GPIO_ResetBits(GPIOD, GPIO_Pin_15);
  }
}
```

Example 3.23

Comparison of Execution Times for Three Different FIR Filter Implementations (`tm4c123_fir3ways_intr.c`).

A straightforward method of measuring the time taken to compute each FIR filter output sample is to toggle a GPIO output pin setting it high in a program statement immediately preceding computation and resetting it low in a program statement immediately following computation. Most of the example programs in this chapter do this. On the TM4C123 Launchpad, GPIO pin PE2 is used, and on the STM32F407 Discovery, GPIO pin PD15 is used.

Figure 3.46 shows the signal output by program `tm4c123_fir_prbs_intr.c` on GPIO pin PE2. This program uses interrupt-based i/o and the rectangular pulse is repeated every $125\ \mu s$, reflecting the 8-kHz sampling frequency. The duration of the pulse, that is $13.6\ \mu s$, indicates that the program takes that time in order to compute the value of each output sample value. The higher the order of the FIR implemented by this program (as determined by the coefficient header file used), the longer it will take to compute each output sample value. In order for the program to work, that computation time must be less than the sampling period.

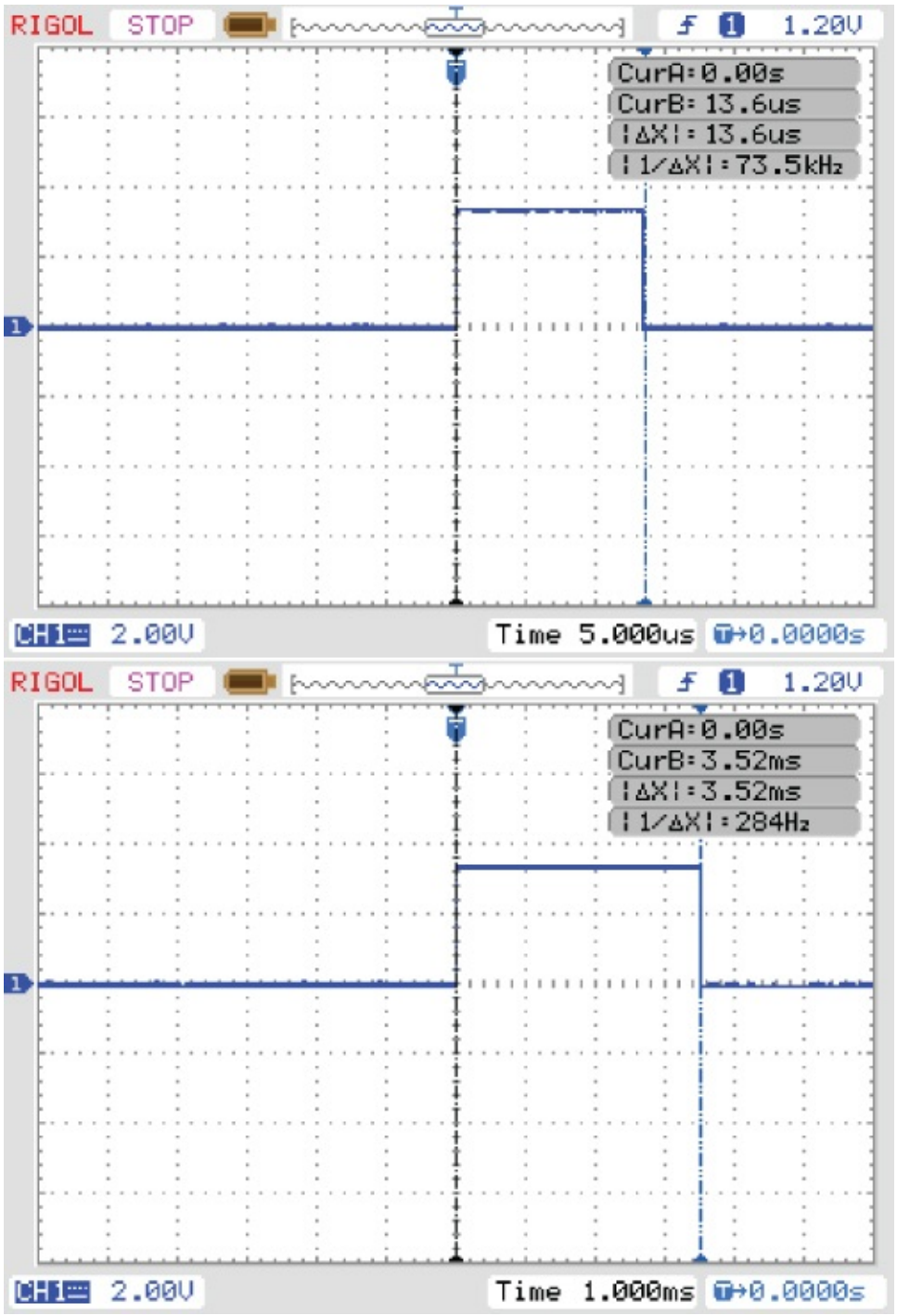


Figure 3.46 Pulses output on GPIO pin PE2 by programs `tm4c123_fir_prbs_intr.c` and `tm4c123_fir_prbs_dma.c`.

The duration of the pulses output on GPIO pin PE2 by program `tm4c123_fir_prbs_dma.c`, which uses DMA-based i/o, indicates the time taken to compute a block of `BUFSIZE` output sample values. These pulses are repeated every `BUFSIZE*125` µs. The higher the order of the FIR implemented by this program (as determined by the coefficient header file used), the longer it will take to compute `BUFSIZE` output sample values. In order for the program to work, that computation time must be less than `BUFSIZE` times the sampling period. The duration of the pulse shown in Figure 3.46 is 3.52 ms, which, given that the value of `BUFSIZE` in this example was equal to 256, represents a time of $13.7\ \mu s$ to compute the value of each output sample.

The program statements used to implement the FIR filtering operation affect its execution time. Program `tm4c123_fir3ways_intr.c`, shown in Listing 3.35, gives the user the option of switching between different FIR filter implementations while the program is running. Using user switch SW1 on the TM4C123 LaunchPad, the user can cycle through three alternatives.

Listing 3.12 Program `tm4c123_fir3ways_intr.c`

```
// tm4c123_fir3ways_intr.c
#include "tm4c123_aic3104_init.h"
#include "bp1750.h"
#define NUM_METHODS 3
float32_t x[2*N];
int16_t k = 0;
int16_t METHOD_number = 0;
AIC3104_data_type sample_data;
float32_t inputl, inputr;
void SSI_interrupt_routine(void)
{
  int16_t i;
  float32_t yn = 0.0;
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  inputl = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  inputr = (float32_t)(sample_data.bit16[0]);
  switch(METHOD_number)
  {
    case 0:
      GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
      x[0] = (float32_t)(prbs(8000));
      for (i=0 ; i<N ; i++) yn += h[i]*x[i];
      for (i=N-1 ; i>0 ; i--) x[i] = x[i-1];
      GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
      break;
    case 1:
      GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
      x[k++] = (float32_t)(prbs(8000));
      if (k>= N) k = 0;
      for (i=0 ; i<N ; i++)
      {
        yn += h[i]*x[k++];
        if (k>= N) k = 0;
      }
      GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
      break;
    case 2:
      GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
      x[k] = (float32_t)(prbs(8000));
      x[k+N] = x[k];
      k = (k+1) % N;
      for (i=0 ; i<N ; i++) yn += h[i]*x[k+i];
      GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
      break;
```

```
  }
  sample_data.bit32 = ((int16_t)(yn));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
}
int main(void)
}
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1)
  {
    ROM_SysCtlDelay(10000);
    if (!GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_4))
    {
      ROM_SysCtlDelay(10000);
      METHOD_number = (METHOD_number+1) % NUM_METHODS;
      while (!GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_4)){}
    }
  }
}
```

The first implementation method is straightforward and uses two separate loops. The first loop

```
for (i=0 ; i< N ; i++) yn += h[i]*x[i];
```

is used in order to compute the convolution sum of the `N` previous input samples stored in the filter delay line `x` and its `N` filter coefficients `h`, placing the result in `yn`. The second loop

```
for (i=N-1 ; i>0 ; i--) x[i] = x[i-1];
```

is used to shift the contents of the filter delay line `x` by one. Intuitively, it is wasteful of computational effort to repeatedly move each input sample from one memory location to another. On the other hand, the computational effort involved in that straightforward operation is unlikely to be great.

The second implementation method

```
for (i=0 ; i<N ; i++)
{
  yn += h[i]*x[k++];
  if (k>= N) k = 0;
}
```

involves treating array `x` as a circular buffer in which to store input sample values. Once a sample has been stored in array `x`, it is not moved. Variable `k` is used to keep track of where in array `x` the most recent input sample is stored. The value of `k` is incremented once per sampling instant. Each time the value of `k` is incremented, its value is tested, and if greater than or equal to `N`, it is reset to zero. The value of `k` is also incremented `N` times during computation of the

convolution sum returning to the same value it started at (before computation of the convolution sum). This implementation method uses just one loop but requires that the value of `k` is tested `(N+1)` times in total.

The third implementation method

```
for (i=0 ; i<N ; i++) yn += h[i]*x[k+i];
```

eliminates the need to test the value of `k` (`N` times) during computation of the convolution sum, at the expense of requiring twice as much memory to store input sample values. Each input sample value is stored in two different locations, `x[k]` and `x[k+N]`, in an array `x` of length `2`. Effectively, this is an alternative method of implementing circular buffering.

3.3.2.5 Running the Program

Build and run the program and observe the pulses output on GPIO pin PE2 using an oscilloscope. Press user switch SW1 on the launchpad to cycle through the three different implementation methods. In the case of an 81-coefficient FIR filter, defined in header file `bp1750.h`, the three different implementation methods take 13.7, 16.0, and $10.2\ \mu s$ to compute each output sample value.

Example 3.24

Comparison of Execution Times for FIR Filters Implemented in C and Using CMSIS DSP library function `arm_fir_f32()`.

CMSIS DSP library function `arm_fir_f32()` is an efficient method of implementing an FIR filter and was demonstrated in Example 3.32. The duration of the pulse output on GPIO pin PE2 by program `tm4c123_fir_prbs_CMSIS_dma.c` is 936 µs (`BUFSIZE = 256`), indicating that it takes approximately 3.6 µs to compute the value of each output sample. It is most computationally efficient to implement an FIR filter using DMA-based i/o to provide blocks of input data to the CMSIS library function `arm_fir_f32()`. However, this approach incurs a greater real-time delay (latency) than a sample-by-sample interrupt-based approach.

Chapter 4
Infinite Impulse Response Filters

The FIR filter discussed in Chapter 3 has no analog counterpart. In this chapter, we discuss the infinite impulse response (IIR) filter that, typically, makes use of the vast knowledge that exists concerning analog filters. The design procedure described in this chapter involves the conversion of an analog filter into an equivalent discrete filter. Either the impulse invariance or bilinear transformation (BLT) technique may be used to effect that conversion. 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

$$y(n) = \sum_{k=0}^{M} b_k x(n-k) - \sum_{l=1}^{N} a_l y(n-l) \tag{4.1}$$

or, equivalently,

$$\begin{aligned} y(n) = {} & b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) + \cdots + b_M x(n-M) \\ & - a_1 y(n-1) - a_2 y(n-2) - \cdots - a_N y(n-N). \end{aligned} \tag{4.2}$$

This recursive difference equation represents an IIR filter. The output $y(n)$, at instant n, depends not only on the current input $x(n)$, at instant n, and on past inputs $x(n-1)$, $x(n-2)$, …, $x(n-M)$, but also on past outputs $y(n-1)$, $y(n-2)$,…, $y(n-N)$. If we assume all initial conditions to be zero in Equation (4.2), its z-transform is

$$\begin{aligned} Y(z) = {} & (b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_M z^{-M}) X(z) \\ & - (a_1 z^{-1} + a_2 z^{-2} + \cdots + a_N z^{-N}) Y(z). \end{aligned} \tag{4.3}$$

Letting $M = N$ in (4.3), the transfer function $H(z)$ of the IIR filter is

$$H(z) = \frac{X(z)}{Y(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_N z^{-N}} = \frac{N(z)}{D(z)}, \tag{4.4}$$

where $N(z)$ and $D(z)$ represent the numerator and denominator polynomials, respectively. Multiplying and dividing by z^N, $H(z)$ becomes

$$H(z) = \frac{b_0 z^N + b_1 z^{N-1} + b_2 z^{N-2} + \cdots + b_N}{z^N + a_1 z^{N-1} + a_2 z^{N-2} + \cdots + a_N} = C \prod_{i=1}^{N} \frac{z - z_i}{z - p_i}, \tag{4.5}$$

where C is a constant is a transfer function with N zeros and N poles. If all of the coefficients a_l 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 causal 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 $h(n) \to 0$, as $n \to \infty$, yielding a stable system.
2. If $|p_i| > 1$, then $h(n) \to \infty$, as $n \to \infty$, yielding an unstable system.
3. If $|p_i| = 1$, the system is marginally stable, yielding an oscillatory response.

4.2 IIR Filter Structures

Several different structures may be used to represent an IIR filter.

4.2.1 Direct Form I Structure

Using the direct form I structure shown in Figure 4.1, the filter in Equation (4.2) can be realized. For an Nth-order filter, this structure contains $2N$ delay elements, each represented by a block labeled z^{-1}. For example, a second-order filter with $N = 2$ will contain four delay elements.

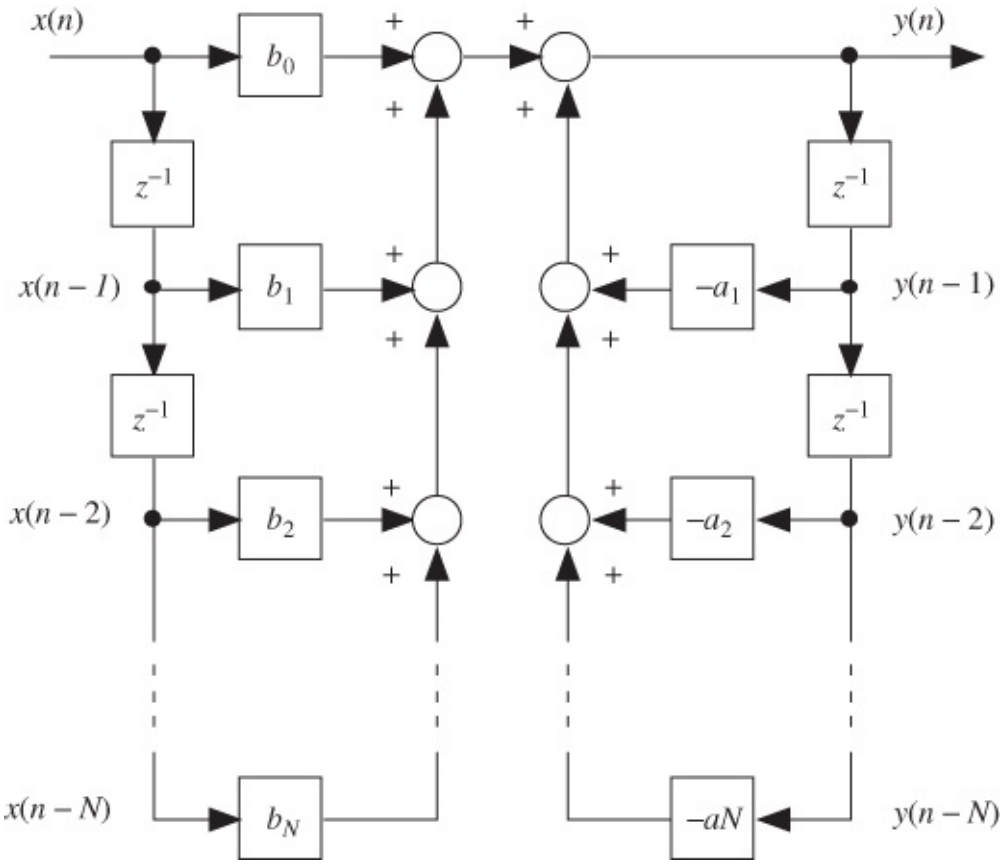


Figure 4.1 Direct form I IIR filter structure.

4.2.2 Direct Form II Structure

The direct form II structure shown in Figure 4.2 is one of the structures most commonly used to represent an IIR filter. It requires half as many delay elements as direct form I. For example, a second-order filter requires two delay elements, as opposed to four with the direct form I structure. From the block diagram of Figure 4.2, it can be seen that

$$w(n) = x(n) - a_1 w(n-1) - a_2 w(n-2) - \cdots - a_N w(n-N) \tag{4.6}$$

and that

$$y(n) = b_0 w(n) + b_1 w(n-1) + b_2 w(n-2) + \cdots + b_N w(n-N). \tag{4.7}$$

Taking z-transforms of Equations (4.6) and (4.7)

$$W(z) = X(z) - a_1 z^{-1} W(z) - a_2 z^{-2} W(z) - \cdots - a_N z^{-N} W(z) \tag{4.8}$$

and hence,

$$X(z) = (1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_N z^{-N}) W(z) \tag{4.9}$$

and

$$Y(z) = (b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_N z^{-N}) W(z). \tag{4.10}$$

Thus

$$H(z) = \frac{X(z)}{Y(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_N z^{-N}}, \tag{4.11}$$

which is similar to Equation (4.5).

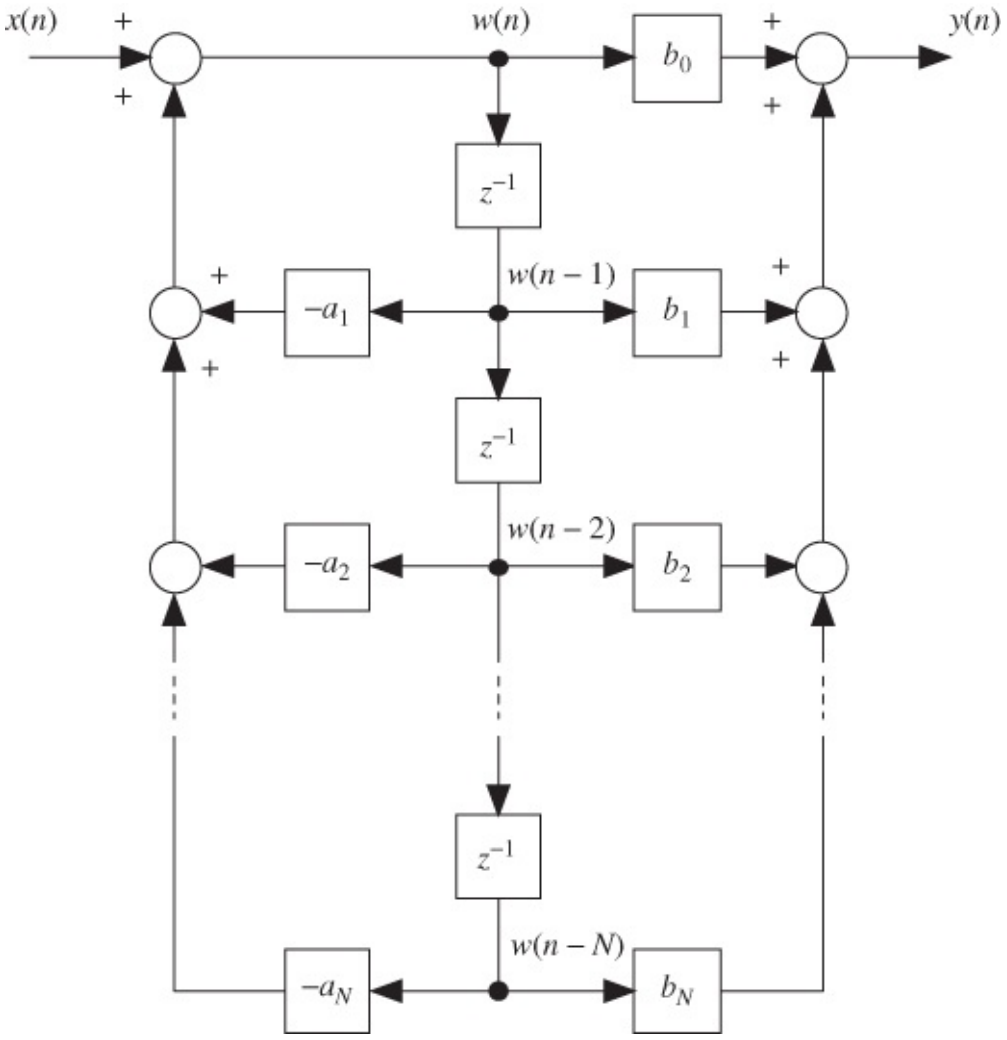


Figure 4.2 Direct form II IIR filter structure.

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, $w(n), w(n-1), w(n-2), \ldots$ are set to zero. At instant 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 structure and requires the same number of delay elements.

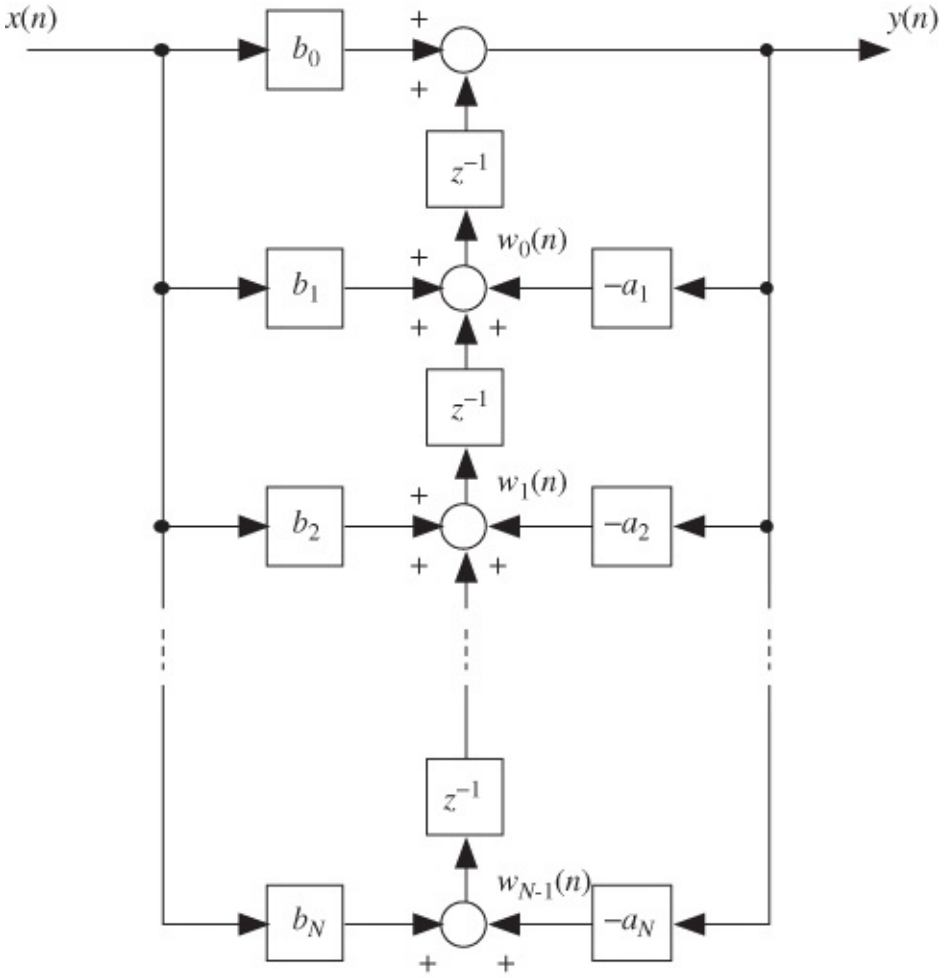


Figure 4.3 Direct form II transpose IIR filter structure.

From inspection of the block diagram, it is apparent that the filter output can be computed using

$$y(n) = b_0 x(n) + w_0(n-1). \tag{4.12}$$

Subsequently, the contents of the delay line can be updated using

$$w_0(n) = b_1 x(n) + w_1(n-1) - a_1 y(n), \tag{4.13}$$

and

$$w_1(n) = b_2 x(n) + w_2(n-1) - a_2 y(n), \tag{4.14}$$

and so on until finally

$$w_{N-1}(n) = b_N x(n) - a_N y(n), \tag{4.15}$$

Using Equation (4.13) to find $w_0(n-1)$,

$$w_0(n-1) = b_1 x(n-1) + w_1(n-2) - a_1 y(n-1), \tag{4.16}$$

Equation (4.12) becomes

$$y(n) = b_0x(n) + [b_1x(n-1) + w_1(n-2) - a_1y(n-1)]. \quad \textbf{4.17}$$

Similarly, using Equation (4.14) to find $w_1(n-2)$,

$$w_1(n-2) = b_2x(n-2) + w_2(n-3) - a_2y(n-3), \quad \textbf{4.18}$$

Equation (4.12) becomes

$$y(n) = b_0x(n) + [b_1x(n-1) + [b_2x(n-2) + w_2(n-3) - a_2y(n-2)] - a_1y(n-1)]. \quad \textbf{4.19}$$

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 of the filter first and then the poles, whereas the direct form II structure implements the poles first.

4.2.4 Cascade Structure

The transfer function in Equation (4.5) can be factorized as

$$H(z) = CH_1(z)H_2(z)\cdots H_r(z), \quad \textbf{4.20}$$

in terms of first- or second-order transfer functions, $H_i(z)$. This cascade (or series) structure is shown in Figure 4.4. An overall transfer function can be represented with cascaded transfer functions. For each section, either 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 in this case be written as

$$H(z) = \prod_{i=1}^{N/2} \frac{b_{0i} + b_{1i}z^{-1} + b_{2i}z^{-2}}{1 + a_{1i}z^{-1} + a_{2i}z^{-2}}, \quad \textbf{4.21}$$

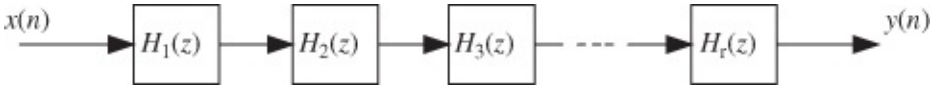


Figure 4.4 Cascade form IIR filter structure.

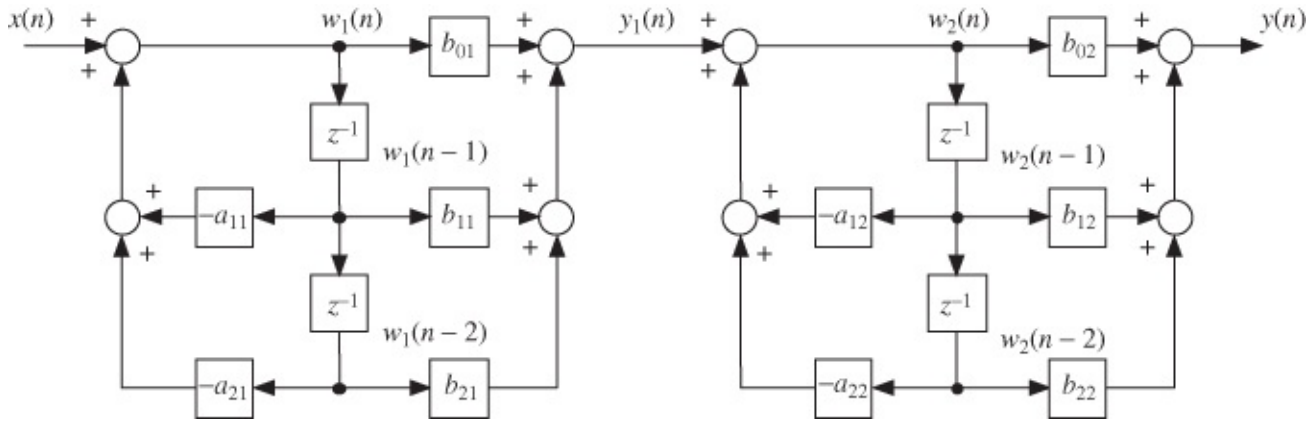


Figure 4.5 Fourth-order IIR filter with two direct form II sections in cascade.

where the constant C in Equation (4.20) is incorporated into the coefficients. For example, if $N = 4$ for a fourth-order transfer function, then Equation (4.18) becomes

$$H(z) = \frac{(b_{01} + b_{11}z^{-1} + b_{21}z^{-2})(b_{02} + b_{12}z^{-1} + b_{22}z^{-2})}{(1 + a_{11}z^{-1} + a_{21}z^{-2})(1 + a_{12}z^{-1} + a_{22}z^{-2})}, \quad \textbf{4.22}$$

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 Equation (4.11) can be represented as

$$H(z) = C + H_1(z) + H_2(z) + \cdots + H_r(z), \quad \textbf{4.23}$$

which can be obtained using a partial fraction expansion (PFE) of Equation (4.11). This parallel form structure is shown in Figure 4.6. Each of the transfer functions can be either first- or second-order function.

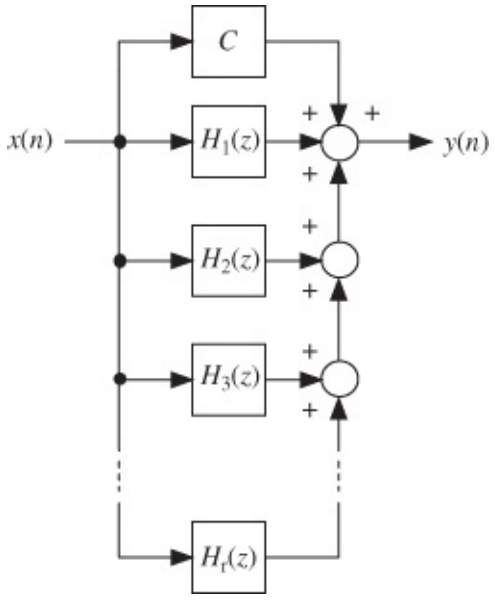


Figure 4.6 Parallel form IIR filter structure.

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

$$H(z) = C + \sum_{i=1}^{N/2} \frac{b_{0i} + b_{1i}z^{-1} + b_{2i}z^{-2}}{1 + a_{1i}z^{-1} + a_{2i}z^{-2}}. \quad \textbf{4.24}$$

For example, for a fourth-order transfer function, $H(z)$ in Equation (4.24) becomes

$$H(z) = C + \frac{b_{01} + b_{11}z^{-1} + b_{21}z^{-2}}{1 + a_{11}z^{-1} + a_{21}z^{-2}} + \frac{b_{02} + b_{12}z^{-1} + b_{22}z^{-2}}{1 + a_{12}z^{-1} + a_{22}z^{-2}}. \quad \textbf{4.25}$$

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

$$y(n) = Cx(n) + \sum_{i=1}^{N/2} y_i(n). \quad \textbf{4.26}$$

Typically, N th-order IIR filters are implemented as cascaded second-order sections.

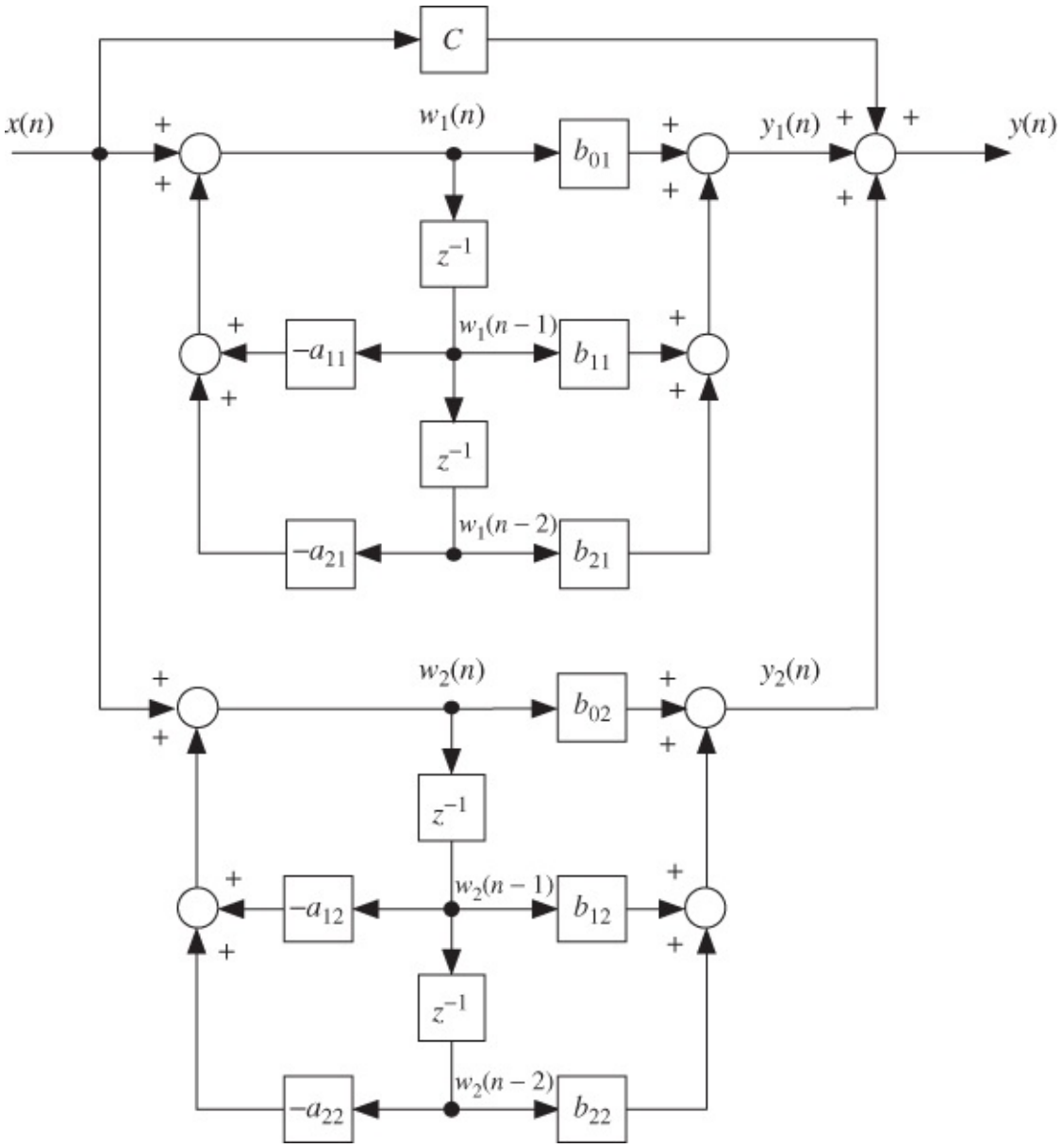


Figure 4.7 Fourth-order IIR filter with two direct form II sections in parallel.

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. PFE of $H(s)$ and substitution $(1 - e^{-p_k t_s} z^{-1})$ 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 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 detail of continuous-time impulse response, 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 the substitution

$$s = K\frac{(z-1)}{(z+1)}. \tag{4.27}$$

The constant K in Equation (4.27) 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 BLT 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 $z = e^{j\omega_D T}$, Equation (4.27) becomes

$$j\omega_A = K\frac{(e^{j\omega_D T}-1)}{(e^{j\omega_D T}+1)} = K\frac{e^{j\omega_D T/2}(e^{j\omega_D T/2}-e^{-j\omega_D T/2})}{e^{j\omega_D T/2}(e^{j\omega_D T/2}+e^{-j\omega_D T/2})}. \tag{4.28}$$

Using Euler's formulae for sine and cosine in terms of complex exponential functions, ω_A in Equation (4.28) becomes

$$\omega_A = K\tan\left(\frac{\omega_D T}{2}\right), \tag{4.29}$$

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 ($\omega_s = 2\pi/T$).

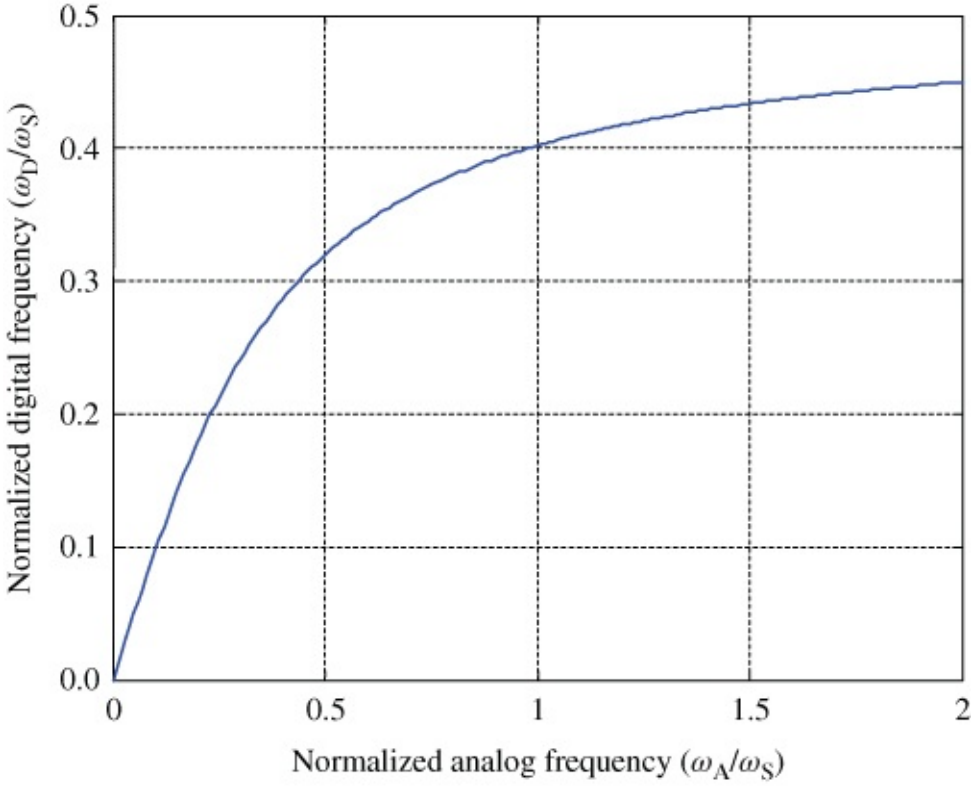


Figure 4.8 Relationship between analog and digital frequencies, ω_A and ω_D, due to frequency warping in the bilinear transform.

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

$$H(z) = H(s)|_{s=K\frac{(z-1)}{(z+1)}}. \quad \textbf{4.30}$$

$H(s)$ 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 $0–\omega_s/2$, to exactly the same digital frequency $\omega_D = \omega_c$. This is achieved by choosing

$$K = \frac{\omega_c}{\tan(\pi\omega_c/\omega_s)}. \quad \textbf{4.31}$$

4.5 Programming Examples

The examples in this section introduce and illustrate the implementation of IIR filtering. Many different approaches to the design of IIR filters 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 filters are widely used. In this example, a second-order, type 1 Chebyshev, low-pass filter with 2 dB of pass-band ripple and a cutoff frequency of 1500 Hz (9425 rad/s) is used.

The continuous-time transfer function of this filter is

$$H(s) = \frac{58,072,962}{s^2 + 7576s + 73,109,527}, \tag{4.32}$$

and its frequency response is shown in Figure 4.9.

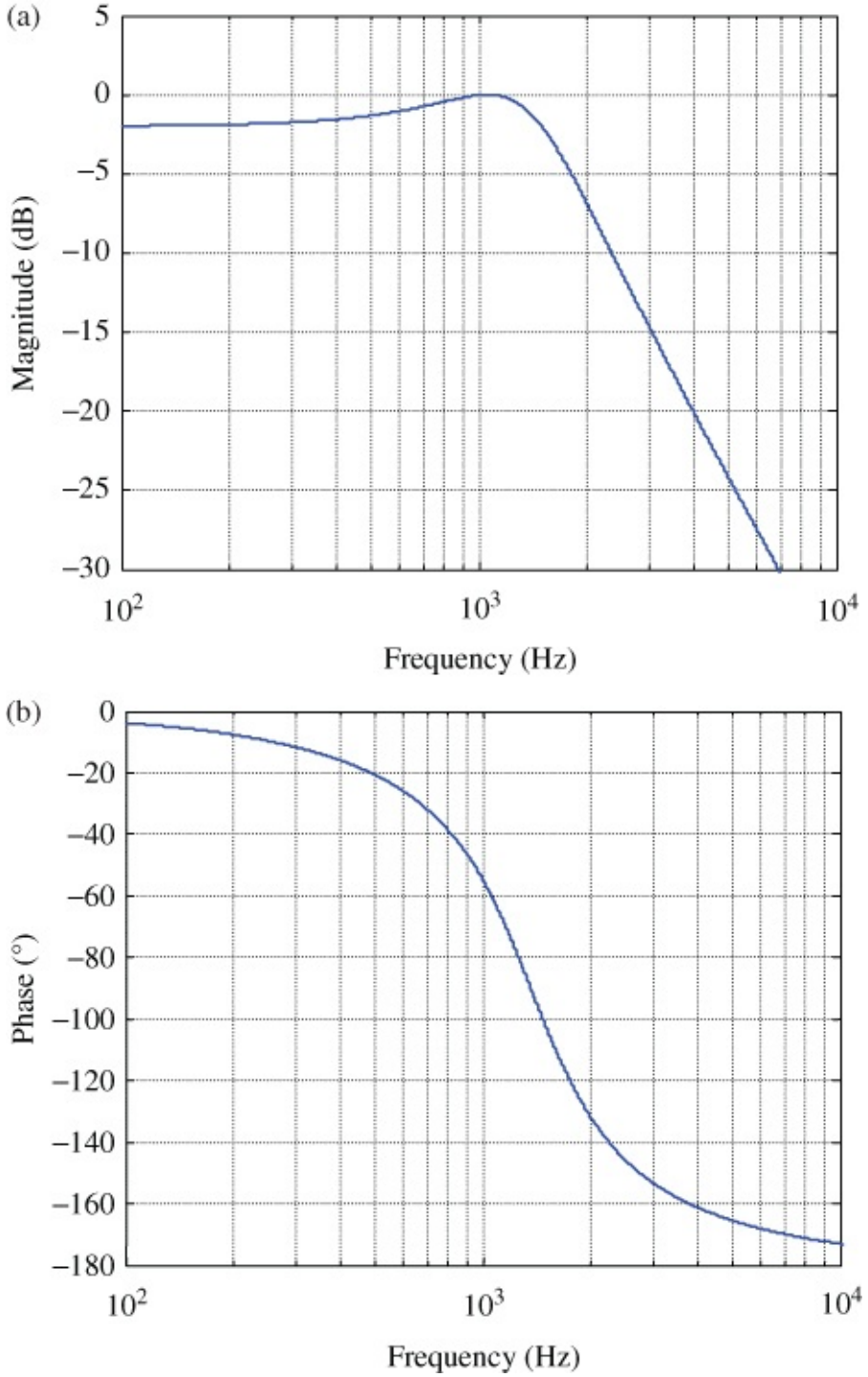


Figure 4.9 (a) Magnitude frequency response of filter $H(s)$. (b) Phase response of filter $H(s)$.

Using MATLAB®, the coefficients of this s-transfer function may 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.

4.5.1.1 Impulse Invariance Method

Starting with the filter transfer function of Equation (4.32), we can make use of the Laplace

transform pair

$$L\{Ae^{-\alpha t}\sin(\omega t)\} = \frac{A\omega}{s^2 + 2\alpha s + (\alpha^2 + \omega^2)} \quad \textbf{4.33}$$

(the s-transfer function of the filter is equal to the Laplace transform of its impulse response) and use the values

$$\alpha = 7576/2 = 2787.9,$$

$$\omega = \sqrt{73,109,527 - 3787.9^2} = 7665.6,$$

$$A = 58,072,962/7665.6 = 7575.8.$$

Hence, the impulse response of the filter in this example is given by

$$h(t) = 7575.8e^{-3788t}\sin(7665.6t) \quad \textbf{4.34}$$

The z-transform pair

$$\mathcal{Z}\{Ae^{-\alpha t}\sin(\omega t)\} = \frac{Ae^{-\alpha t_s}\sin(\omega t_s)z^{-1}}{1 - 2(Ae^{-\alpha t_s}\cos(\omega t_s))z^{-1} + e^{-2\alpha t_s}z^{-2}} \quad \textbf{4.35}$$

yields the following discrete-time transfer function when we substitute for ω, A, α and set $t_s = 0.000125$ in Equation (4.35).

$$H(z) = \frac{Y(z)}{X(z)} = \frac{0.48255z^{-1}}{1 - 0.71624315z - 1 + 0.38791310z^{-2}}. \quad \textbf{4.36}$$

From $H(z)$, the following difference equation may be derived.

$$y(n) = 0.48255x(n-1) + 0.71624315y(n-1) - 0.3879131y(n-2). \quad \textbf{4.37}$$

With reference to Equation (4.2), we can see that $a_1 = 0.71624315$, $a_2 = -0.3879131$, $b_0 = 0.0000$, and $b_1 = 0.48255$.

In order to apply the impulse invariant method using MATLAB, type

```
>> [b,a] = cheby1(2,2,2*pi*1500,'s');
>> [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, t_s), as shown in Figure 4.10. 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. It is perhaps worth noting that, counterintuitively, the definition of an IIR filter is not that its impulse response is infinite in duration but rather that it makes use of previous output sample values in order to calculate its current output. In theory, it is possible for an IIR filter to have a finite impulse response. 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 $h(n)$.

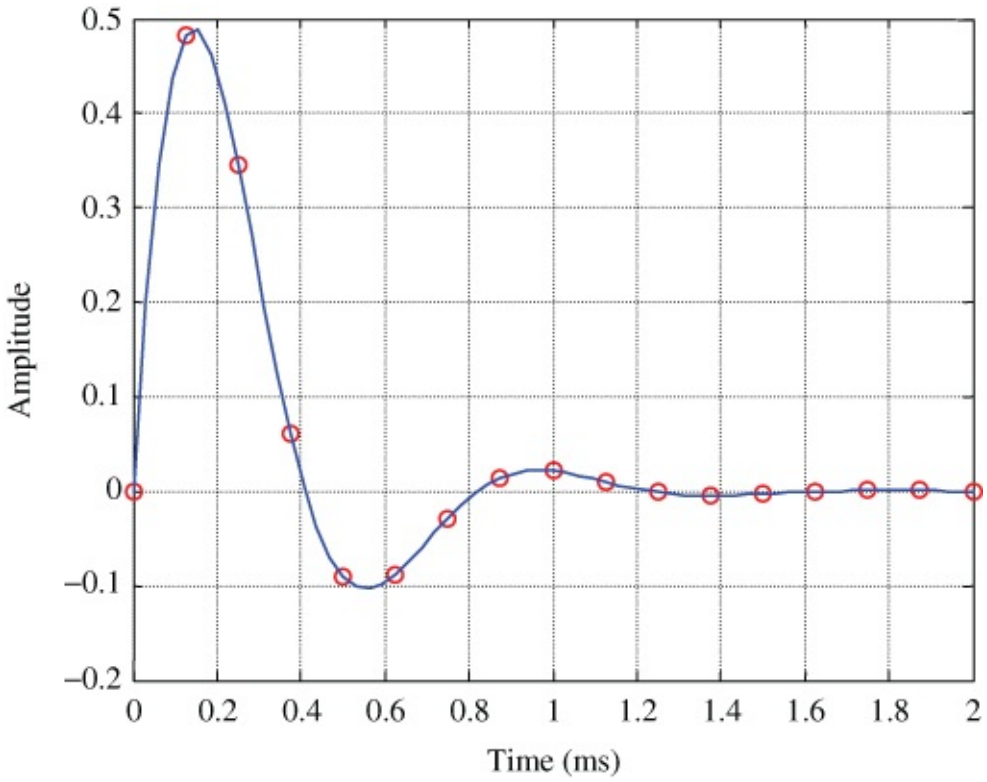


Figure 4.10 Impulse responses $h(t)$ (scaled by sampling period t_s) and $h(n)$ of continuous-time filter $H(s)$ and its impulse-invariant digital implementation.

Example 4.1

Implementation of an IIR Filter Using Cascaded Second-Order Direct Form II Sections (`stm32f4_iirsos_intr.c`).

Program `stm32f4_iirsos_intr.c`, shown in Listing 4.1, implements a generic IIR filter using cascaded direct form II second-order sections, and coefficient values stored in a separate header 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

$$w(n) = x(n) - a_1 w(n-1) - a_2 w(n-2) \qquad \textbf{4.38}$$

and

$$y(n) = b_0 w(n) + b_1 w(n-1) + b_2 w(n-2) \quad \textbf{4.39}$$

With reference to Figure 4.5 and to (4.18), the coefficients are stored by the program 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 $w_i(n-2)$ in Equations (4.38) and (4.39).

Listing 4.1 IIR filter program using second-order sections in cascade (`stm32f4_iirsos_intr.c`)

```
// stm32f4_iirsos_intr.c
#include "stm32f4_wm5102_init.h"
#include "elliptic.h"
float w[NUM_SECTIONS][2] = {0};
void SPI2_IRQHandler()
{
  int16_t left_out_sample, left_in_sample;
  int16_t right_out_sample, right_in_sample;
  int16_t section;    // second order section number
  float32_t input;    // input to each section
  float32_t wn, yn;   // intermediate and output values
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ⌡ SET)
  {
    left_in_sample = SPI_I2S_ReceiveData(I2Sx);
    input =(float32_t)(left_in_sample);
    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;
    }
    left_out_sample = (int16_t)(yn);
    while (SPI_I2S_GetFlagStatus(I2Sxext,
           SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
  }
  else
  {
    right_in_sample = SPI_I2S_ReceiveData(I2Sx);
    while (SPI_I2S_GetFlagStatus(I2Sxext,
           SPI_I2S_FLAG_TXE ) != SET){}
    right_out_sample = 0;
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
}
int main(void)
{
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
  while(1){}
}
```

The impulse invariant filter may be implemented using program `stm32f4_iirsos_intr.c` by including the coefficient header file `impinv.h`, shown in Listing 4.2. The number of cascaded second-order sections is defined as `NUM_SECTIONS` in that file.

Listing 4.2 Coefficient header file `impinv.h`

```
// impinv.h
// second-order type 1 Chebyshev LPF with 2 dB pass-band ripple
// and cutoff frequency 1500 Hz
#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} };
```

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 `stm32f4_iirsos_intr.c` uses interrupt-based i/o. A DMA-based i/o version of the program, `stm32f4_iirsos_dma.c`, is also provided. It can be used by removing program `stm32f4_iirsos_intr.c` from the project and adding program `stm32f4_iirsos_dma.c` before rebuilding the project.

Example 4.2

Implementation of IIR Filter Using Cascaded Second-Order Transposed Direct Form II Sections (`stm32f4_iirsostr_intr.c`).

A transposed direct form II structure can be implemented using program `stm32f4_iirsos_intr.c` simply by replacing the 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];
w[section][1] = w[section][0];
w[section][0] = wn;
```

with the following program statements

```
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 already in program `stm32f4_iirsostr_intr.c`. You should not notice any difference in the characteristics of the filters implemented using programs `stm32f4_iirsos_intr.c` and `stm32f4_iirsostr_intr.c`.

Example 4.3

Estimating the Frequency Response of an IIR Filter Using Pseudorandom Noise as Input (`tm4c123_iirsos_prbs_intr.c`).

Program `tm4c123_iirsos_prbs_intr.c` is closely related to program `tm4c123_fir_prbs_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 AIC3104 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 4.11 and 4.12 show the output of the example filter (using coefficient file `impinv.h`) displayed using the FFT function of a *Rigol DS1052E* oscilloscope and using *Goldwave*.

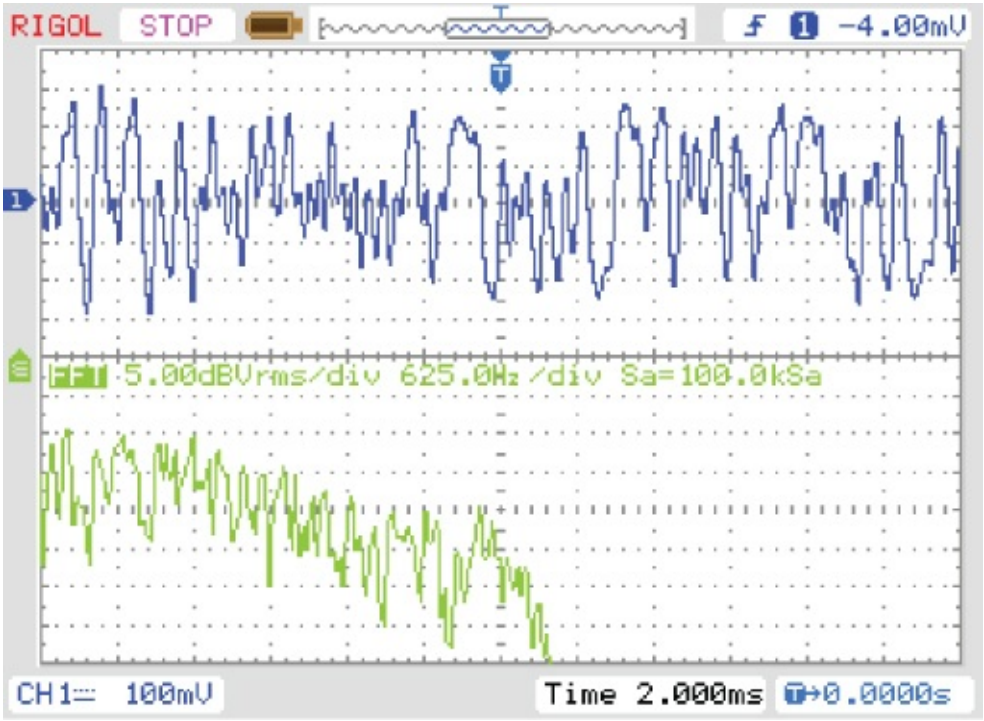


Figure 4.11 Output from program `tm4c123_iirsos_prbs_intr.c` using coefficient file `impinv.h`, viewed using the FFT function of a *Rigol DS1052E* oscilloscope.

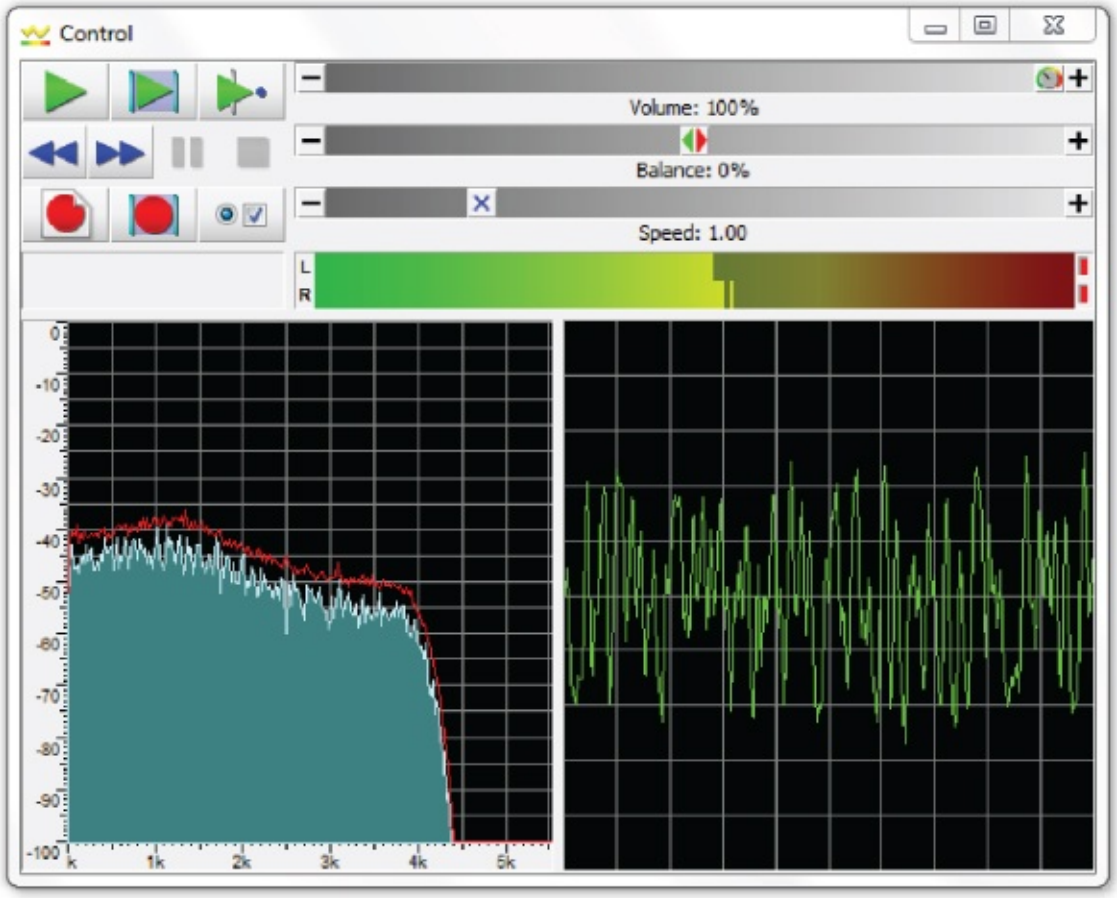


Figure 4.12 Output from program `tm4c123_iirsos_prbs_intr.c` using coefficient file `impinv.h`, viewed using *Goldwave*.

In Figure 4.11, 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 AIC3104 codec.

Example 4.4

Estimating the Frequency Response of an IIR Filter Using a Sequence of Impulses as Input (`tm4c123_iirsos_delta_intr.c`).

Instead of a pseudorandom binary sequence, program `tm4c123_iirsos_delta_intr.c`, shown in Listing 4.3, uses a sequence of discrete-time impulses as the input to an IIR filter. The resultant 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 AIC3104 codec and the resulting analog signal may be analyzed using an oscilloscope, spectrum analyzer, or

Goldwave. In addition, program tm4c123_iirsos_delta_intr.c stores the BUFFERSIZE most recent samples of the filter output yn in array response, and by saving the contents of that array to a data file and using the MATLAB function tm4c123_logfft(), the response of the filter may be viewed in both time and frequency domains.

Listing 4.3 Program tm4c123_iirsos_delta_intr.c

```
// tm4c123_iirsos_delta_intr.c
#include "tm4c123_aic3104_init.h"
#include "elliptic.h"
#define BUFFERSIZE 256
#define AMPLITUDE 10000.0f
float32_t w[NUM_SECTIONS][2] = {0};
float32_t dimpulse[BUFFERSIZE];
float32_t response[BUFFERSIZE];
int16_t bufptr = 0;
AIC3104_data_type sample_data;
void SSI_interrupt_routine(void)
{
  float32_t inputl, inputr;
  int16_t i;
  int16_t section;   // index for section number
  float32_t input;   // input to each section
  float32_t wn,yn;   // intermediate and output values
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  inputl = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  inputr = (float32_t)(sample_data.bit16[0]);
  input = dimpulse[bufptr];
  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;
  }
  response[bufptr++] = yn;
  if (bufptr >= BUFFERSIZE) bufptr = 0;
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
  sample_data.bit32 = ((int16_t)(yn*AMPLITUDE));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
}
int main(void)
{
  int i;
  for (i=0 ; i< BUFSIZE ; i++) dimpulse[i] = 0.0;
```

```
  dimpulse[0] = 10.0;
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1){}
}
```

Figure 4.13 shows the analog output signal generated by the program, captured using a *Rigol DS1052E* oscilloscope connected to one of the scope hooks on the audio booster pack. The upper trace shows the time-domain impulse response of the filter ($500\ \mu s$ 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 both by the IIR filter and by the AIC3104 codec reconstruction filter. The codec reconstruction filter is responsible for the steep roll-off of gain at frequencies above 4 kHz. Below that frequency, but at frequencies higher than 1.5 kHz, less pronounced roll-off of gain due to the IIR filter is discernible. In the upper trace, 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 save the contents of array `response`. Figure 4.14 shows the magnitude of the FFT of the contents of that array plotted using MATLAB function `tm4c123_logfft()`.

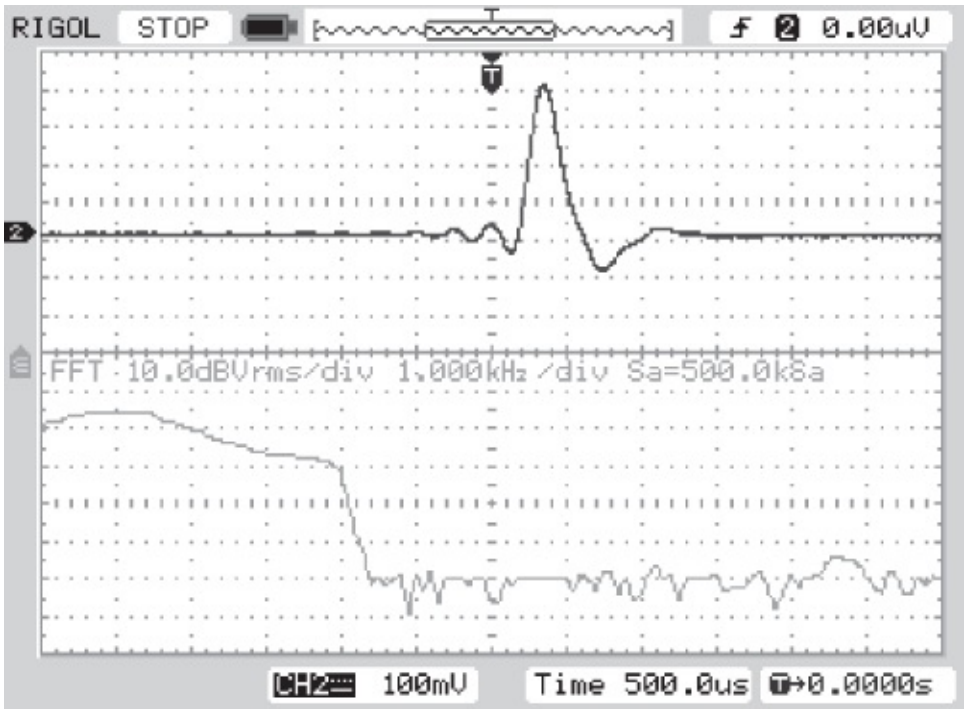


Figure 4.13 Output from program `tm4c123_iirsos_delta_intr.c` using coefficient file `impinv.h`, viewed using the FFT function of a *Rigol DS1052E* oscilloscope.

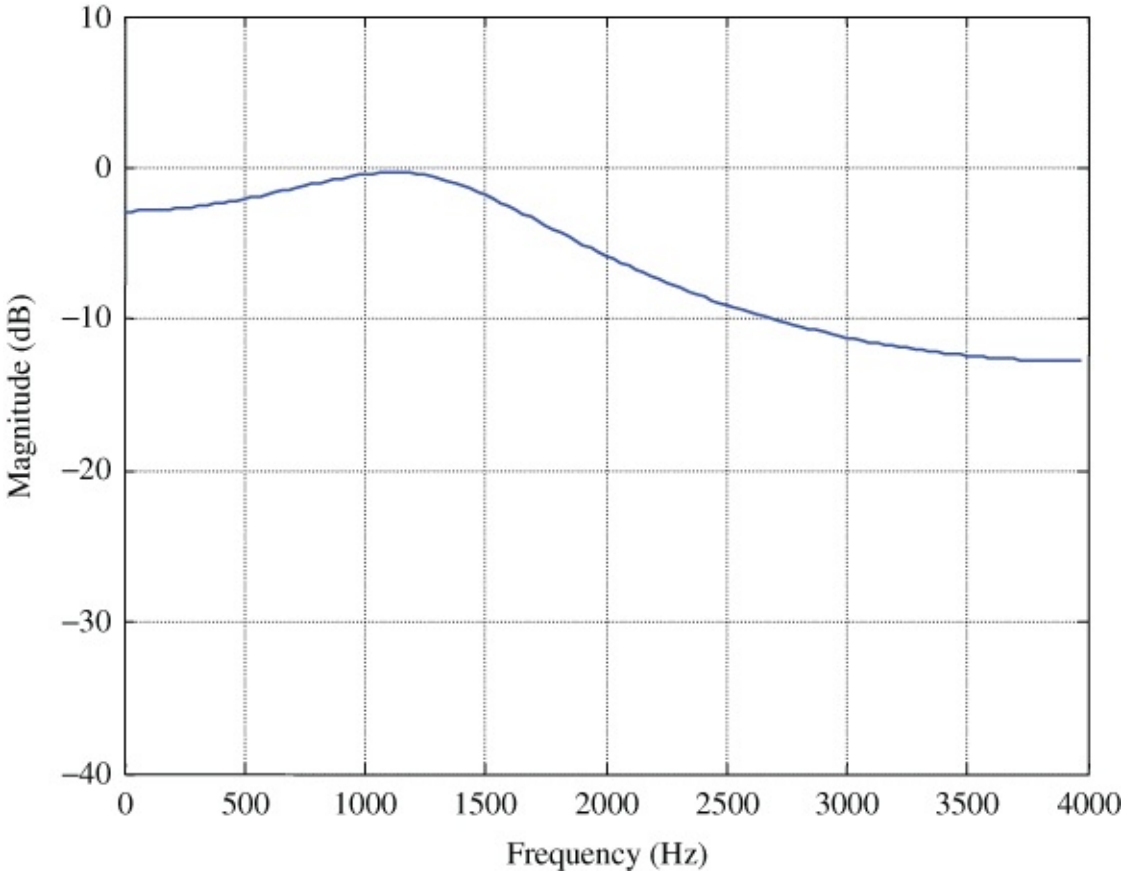


Figure 4.14 The magnitude frequency response of the filter implemented by program `tm4c123_iirsos_delta_intr.c` using coefficient file `impinv.h`, plotted using MATLAB function `tm4c123_logfft()`.

4.5.1.2 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 implementation (Figure 4.14). The gain of the analog prototype has a magnitude of -15 dB at 3000 Hz, whereas, according to Figure 4.14, 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 effects only if the continuous-time impulse response $h(t)$ contains no frequency components at frequencies greater than or equal to half the sampling frequency.

In this 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

```
>> [b,a] = cheby1(2,2,2*pi*1500,'s');
>> [bz,az] = impinvar(b,a,8000);
>> freqz(bz,az);
```

at the MATLAB command line in order to view the theoretical frequency response of the filter, and compare this with Figure 4.14.

An alternative method of transforming an analog filter design to a discrete-time implementation, which eliminates this effect, is the use of the bilinear transform.

4.5.1.3 Bilinear Transform Method of Digital Filter Implementation

The bilinear transform method of converting an analog filter design into discrete time is relatively straightforward, often involving less algebraic manipulation than the impulse invariant method. It is achieved by making the substitution

$$s = \frac{2(z-1)}{T(z+1)} \tag{4.40}$$

in $H(s)$, where T is the sampling period of the digital filter, that is,

$$H(z) = H(s)|_{s=\frac{2(z-1)}{T(z+1)}}. \tag{4.41}$$

Applying this to the s-transfer function of (4.32) results in the following z-transfer function.

$$H(z) = \frac{Y(z)}{X(z)} = \frac{0.12895869 + 0.25791738z^{-1} + 0.12895869z^{-2}}{1 - 0.81226498z - 1 + 0.46166249z^{-2}}. \tag{4.42}$$

From Equation (4.42), the following difference equation may be derived.

$$\begin{aligned} y(n) = {} & 0.1290x(n) + 0.2579x(n-1) + 0.1290x(n-2) \\ & +0.8123y(n-1) - 0.4617y(n-2). \end{aligned} \tag{4.43}$$

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 `stm32f4_iirsos_intr.c`, `tm4c123_iirsos_prbs_intr.c`, and `tm4c123_iirsos_delta_intr.c` from `impinv.h` to `bilinear.h`. In each case, change the line that reads

```
#include “impinv.h”
```

to read

```
#include “bilinear.h”
```

before building, loading, and running the programs. Figures 4.15 through 4.18 show results obtained using programs `tm4c123_iirsos_prbs_intr.c` and

`tm4c123_iirsos_delta_intr.c` with coefficient file `bilinear.h`. The attenuation 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.

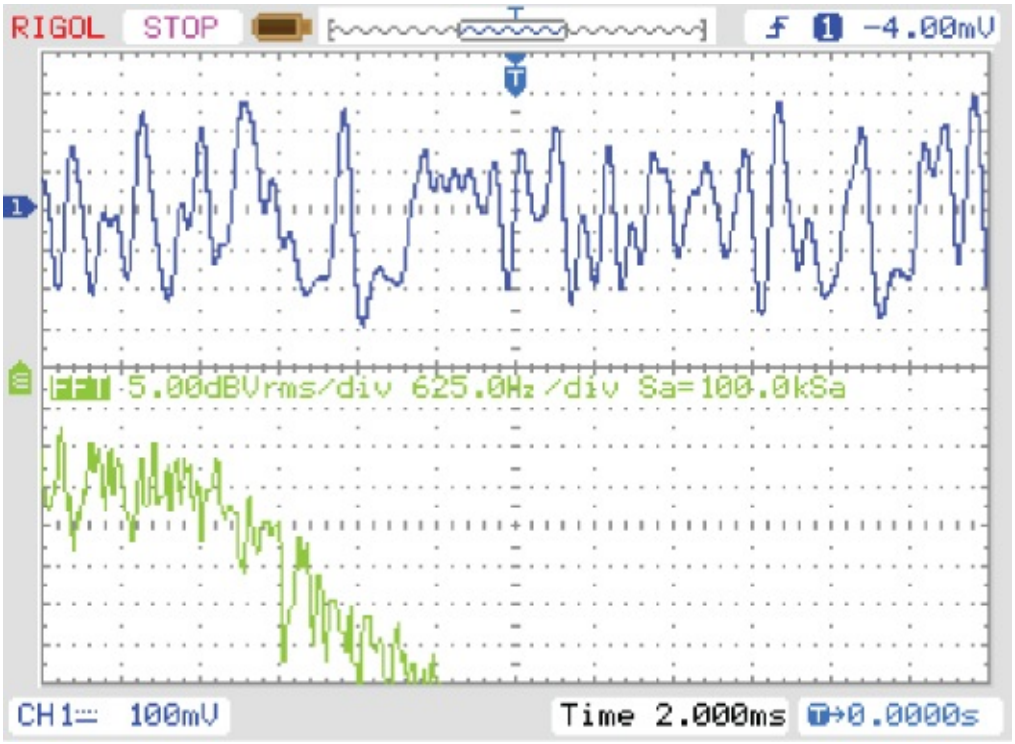


Figure 4.15 Output from program `tm4c123_iirsos_prbs_intr.c` using coefficient file `bilinear.h`, viewed using the FFT function of a *Rigol DS1052E* oscilloscope.

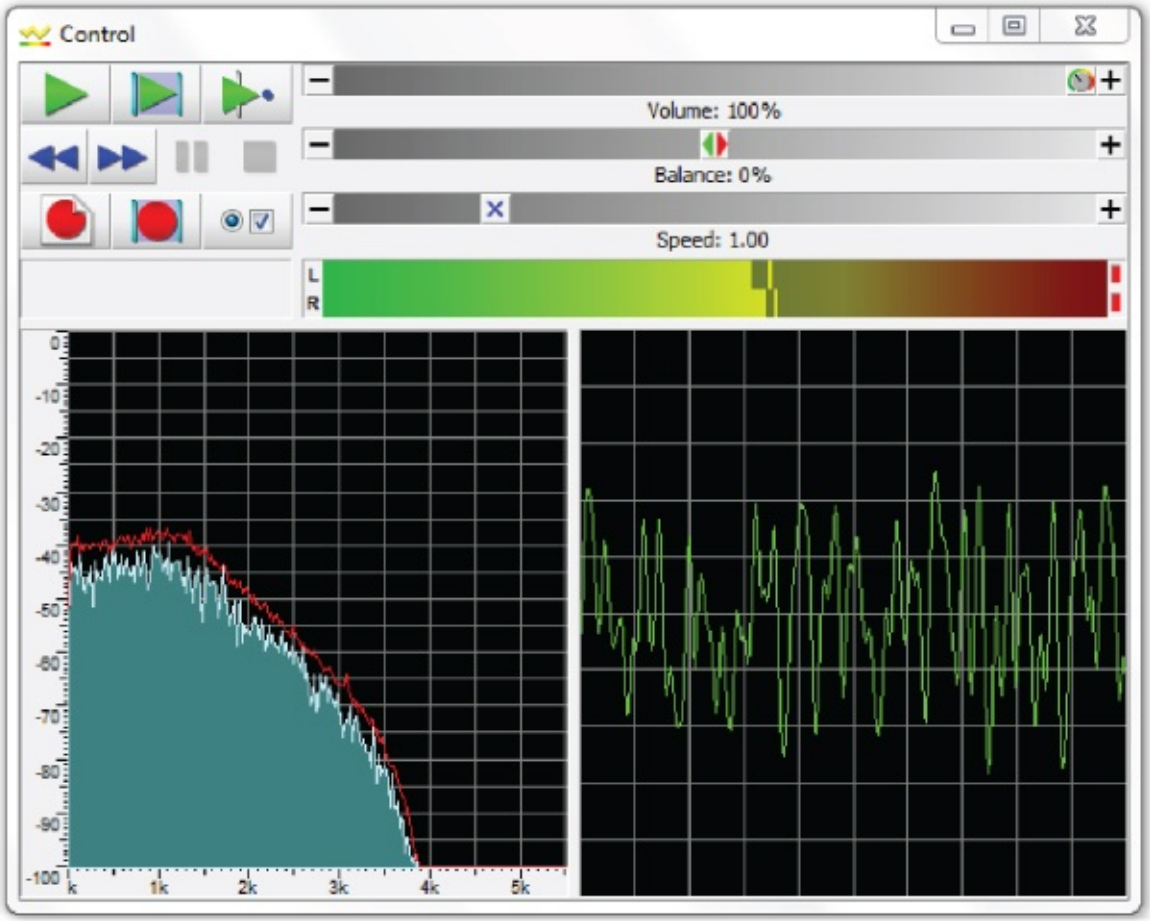


Figure 4.16 Output from program `tm4c123_iirsos_prbs_intr.c` using coefficient file `bilinear.h`, viewed using *Goldwave*.

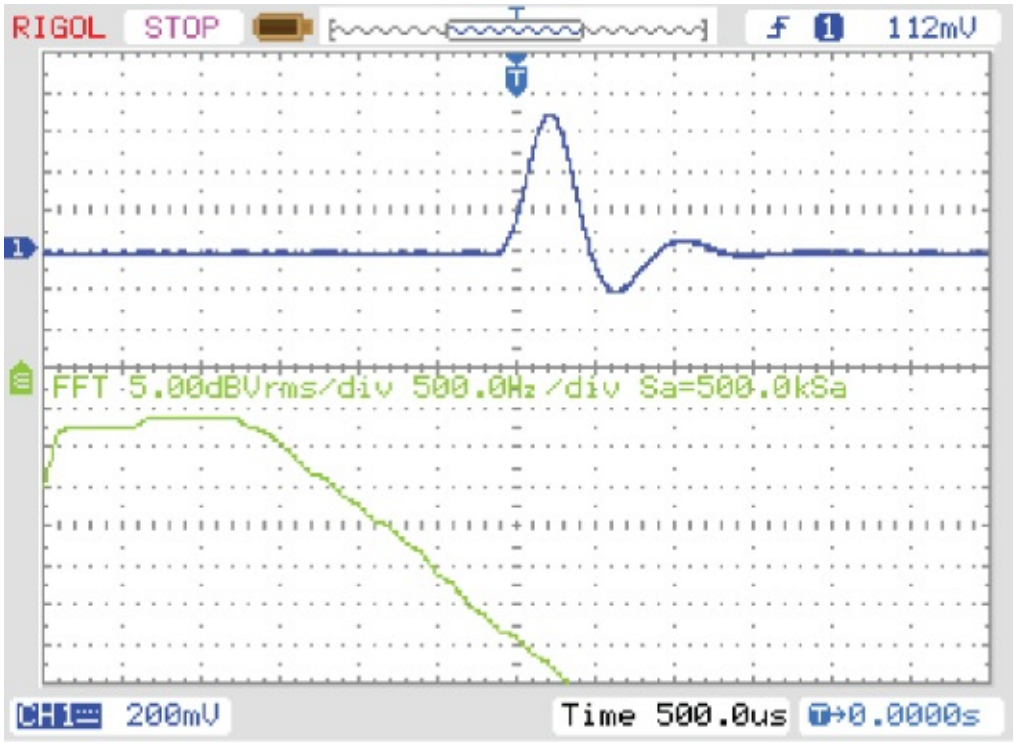


Figure 4.17 Output from program `tm4c123_iirsos_delta_intr.c` using coefficient file `bilinear.h`, viewed using the FFT function of a *Rigol DS1052E* oscilloscope.

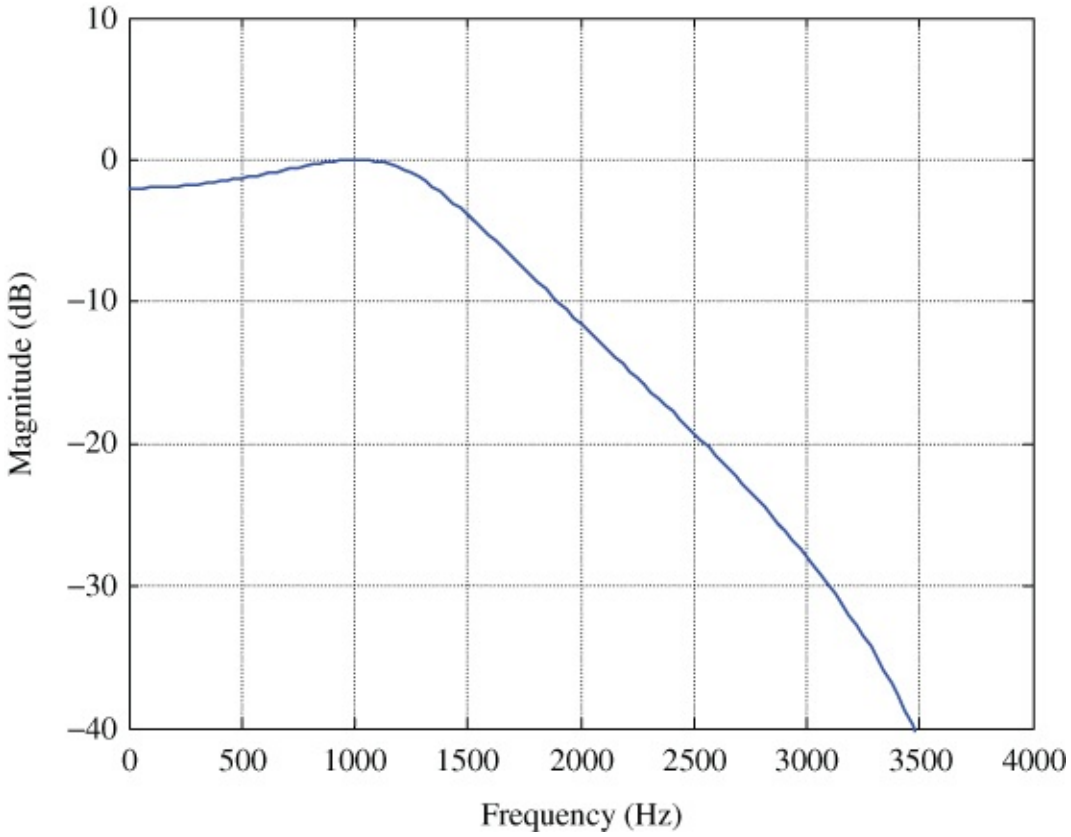


Figure 4.18 The magnitude frequency response of the filter implemented by program `tm4c123_iirsos_delta_intr.c` using coefficient file `bilinear.h`, plotted using MATLAB function `tm4c123_logfft()`.

4.5.1.4 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 infinity is mapped into the frequency range from zero to half the sampling frequency of the digital filter. This may be represented by

$$f_D = \frac{\arctan(\pi f_A T_s)}{\pi T_s} \text{ or } \omega_D = \frac{2}{T_s}\arctan\left(\frac{\omega_A T_s}{2}\right) \quad \textbf{4.44}$$

and

$$f_A = \frac{\tan(\pi f_D T_s)}{\pi T_s} \text{ or } \omega_A = \frac{2}{T_s}\tan\left(\frac{\omega_D T_s}{2}\right), \quad \textbf{4.45}$$

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 4.19. 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 4.19 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.

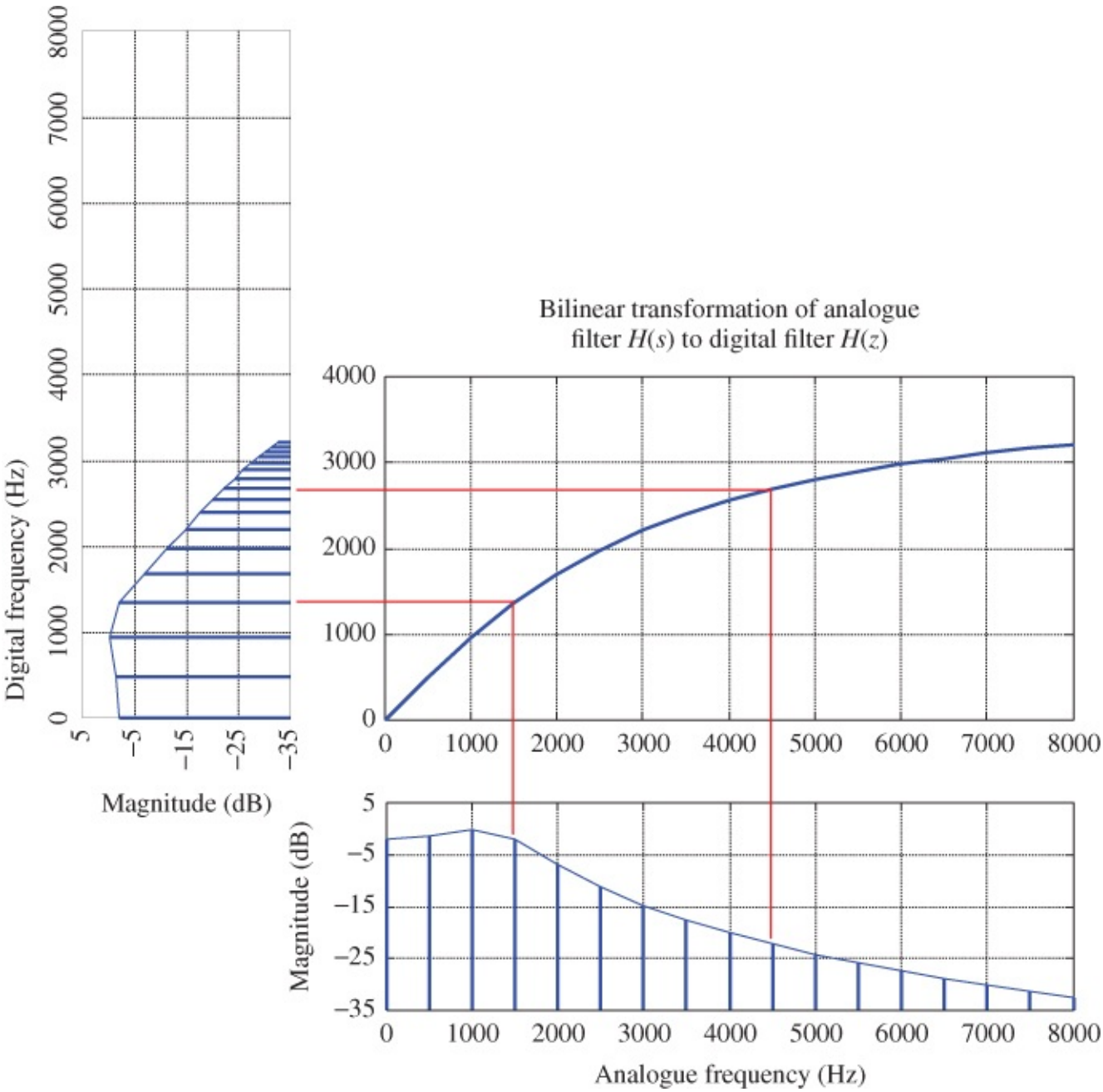


Figure 4.19 The effect of the bilinear transform on the magnitude frequency response of the example filter.

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.

$$H(s) = \frac{747,467,106}{s^2 + 8596s + 94,126,209} \qquad 4.46$$

that is

$$H(z) = H(s)|_{s=\frac{2(z-1)}{(z+1)}} \qquad \textbf{4.47}$$

$$= \frac{0.15325788 + 0.30651556z^{-1} + 0.15325788z^{-2}}{1 - 0.66423178z^{-1} + 0.43599223z^{-2}}$$

The analog filter represented by Equation (4.46) can be produced using the MATLAB command

```
>> [bb,aa] = cheby1(2,2,2*pi*1702,`s');
```

and the BLT applied by typing

```
>> [bbd,aad] = bilinear(bb,aa,8000);
```

to yield the result given by Equation (4.47). 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.h` contains the coefficients obtained as described earlier.

4.5.1.5 Using MATLAB's Filter Design and Analysis Tool

MATLAB provides a filter design and analysis tool, `fdatool`, which makes the calculation of IIR filter coefficient values simple. Coefficients can be exported to the MATLAB workspace in direct form II, second-order section format, and MATLAB function `stm32f4_iirsos_coeffs()` or `tm4c123_iirsos_coeffs()`, supplied with this book as files `stm32f4_iirsos_coeffs.m` and `tm4c123_iirsos_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 pass band, and 50 dB of stop-band attenuation. Click on *Design Filter* and then look at the characteristics of the filter using options from the *Analysis* menu (Figure 4.20).

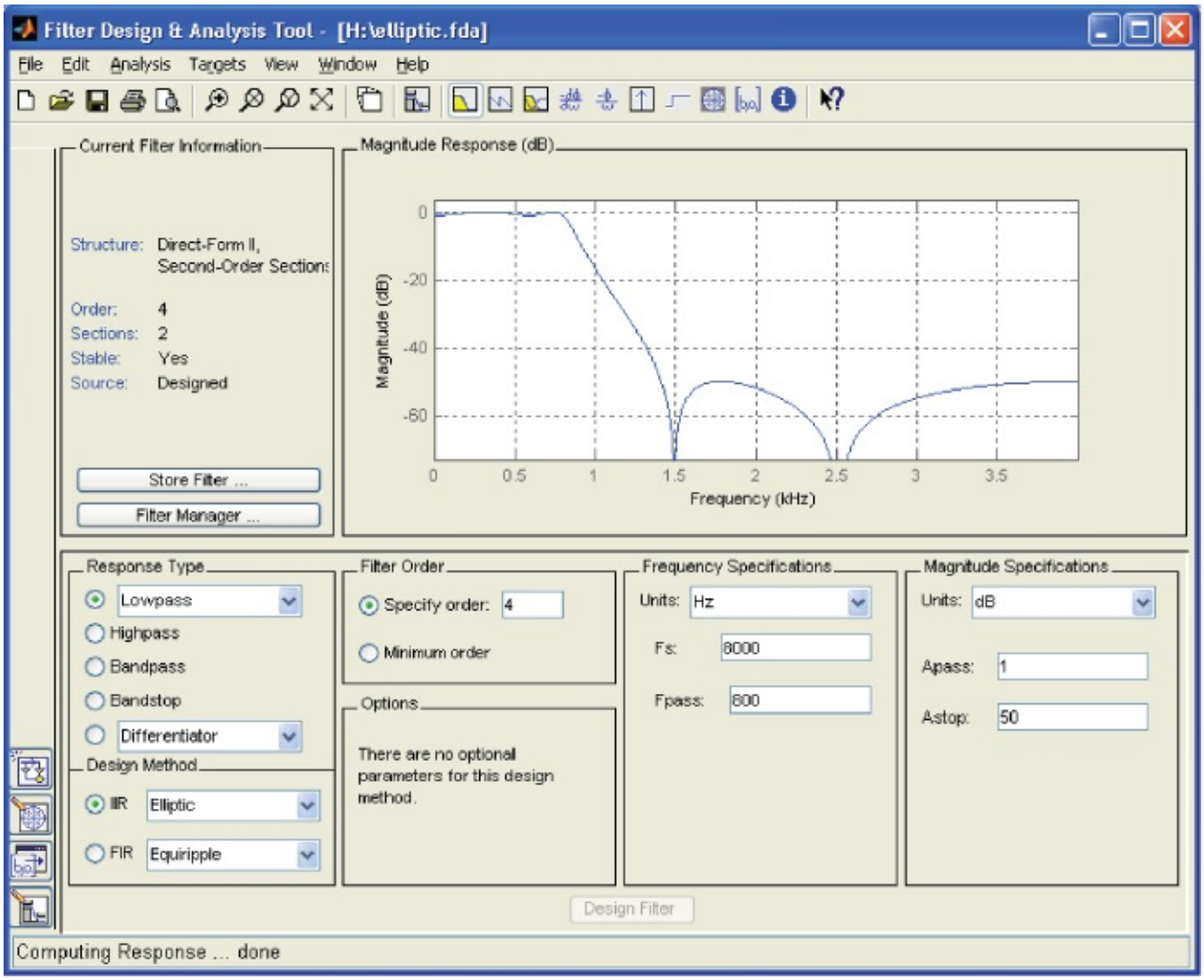


Figure 4.20 MATLAB `fdatool` window showing the magnitude frequency response of a fourth-order elliptic low-pass filter.

This example illustrates the steep transition from pass to stop bands of an IIR filter 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 4.21 shows the use of `fdatool` to design the Chebyshev filter considered in the preceding examples. Notice that the magnitude frequency response decreases more and more rapidly with frequency approaching half the sampling frequency, and compare this with Figure 4.16. This is characteristic of filters designed using the bilinear transform.

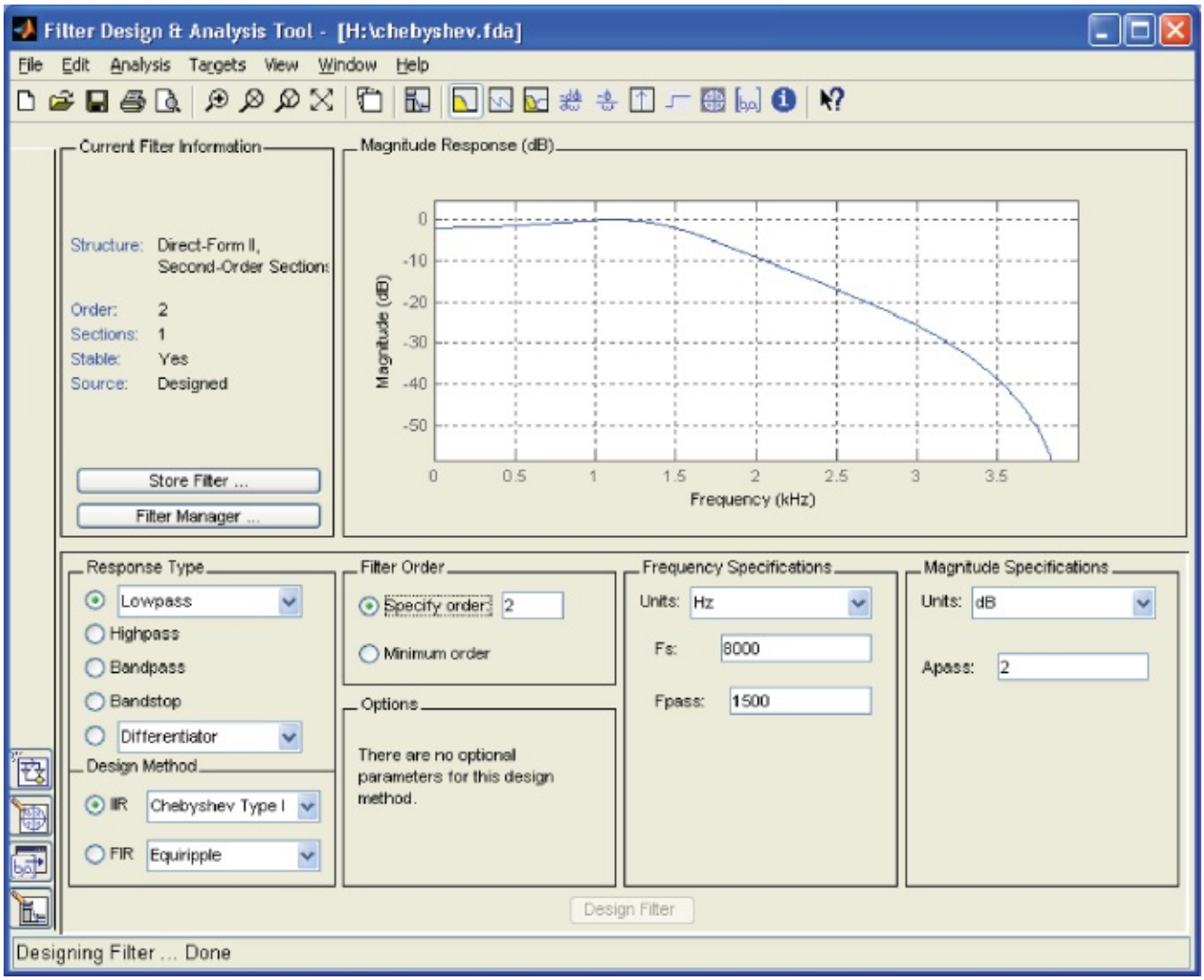


Figure 4.21 MATLAB `fdatool` window showing the magnitude frequency response of a second-order Chebyshev low-pass filter.

4.5.1.6 Implementing a Filter Designed Using `fdatool`

In order to implement a filter designed using `fdatool`, carry out the following 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 either `stm32f4_iirsos_coeffs(SOS,G)` or `tm4c123_iirsos_coeffs(SOS,G)` and enter a filename, for example, `elliptic.h`.

Listing 4.4 shows an example of a coefficient file produced using MATLAB function `stm32f4_iirsos_coeffs()` (Listing 4.5).

Listing 4.4 Coefficient header file `elliptic.h`

```
// elliptic.h
// this file was generated automatically using function
stm32f4_iirsos_coeffs.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] = {
{1.00000000E+000, -1.52872987E+000, 6.37029381E-001},
{1.00000000E+000, -1.51375731E+000, 8.68678568E-001} };
```

Listing 4.5 MATLAB function `stm32f4_iirsos_coeffs()`

```
% STM32F4_IIRSOS_COEFFS.M
%
% MATLAB function to write SOS IIR filter coefficients
% in format suitable for use in STM32F4 Discovery programs
% including stm32f4_iirsos_intr.c,
% stm32f4_iirsos_prbs_intr.c and stm32f4_iirsosdelta_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 STM32F4_iirsos_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 STM32F4_iirsos_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);
```

Program `tm4c123_iirsos_intr.c`, introduced in Example 4.1, can be used to implement the filter. Edit the line in the program that reads

```
#include “bilinear.h”
```

to read

```
#include “elliptic.h”
```

and build and run the program. The coefficient header file is also compatible with programs `tm4c123_iirsos_prbs_intr.c` and `tm4c123_iirsos_delta_intr.c`. Figures 4.22 and 4.23 show results obtained using program `tm4c123_iirsos_delta_intr.c` and coefficient file `elliptic.h`.

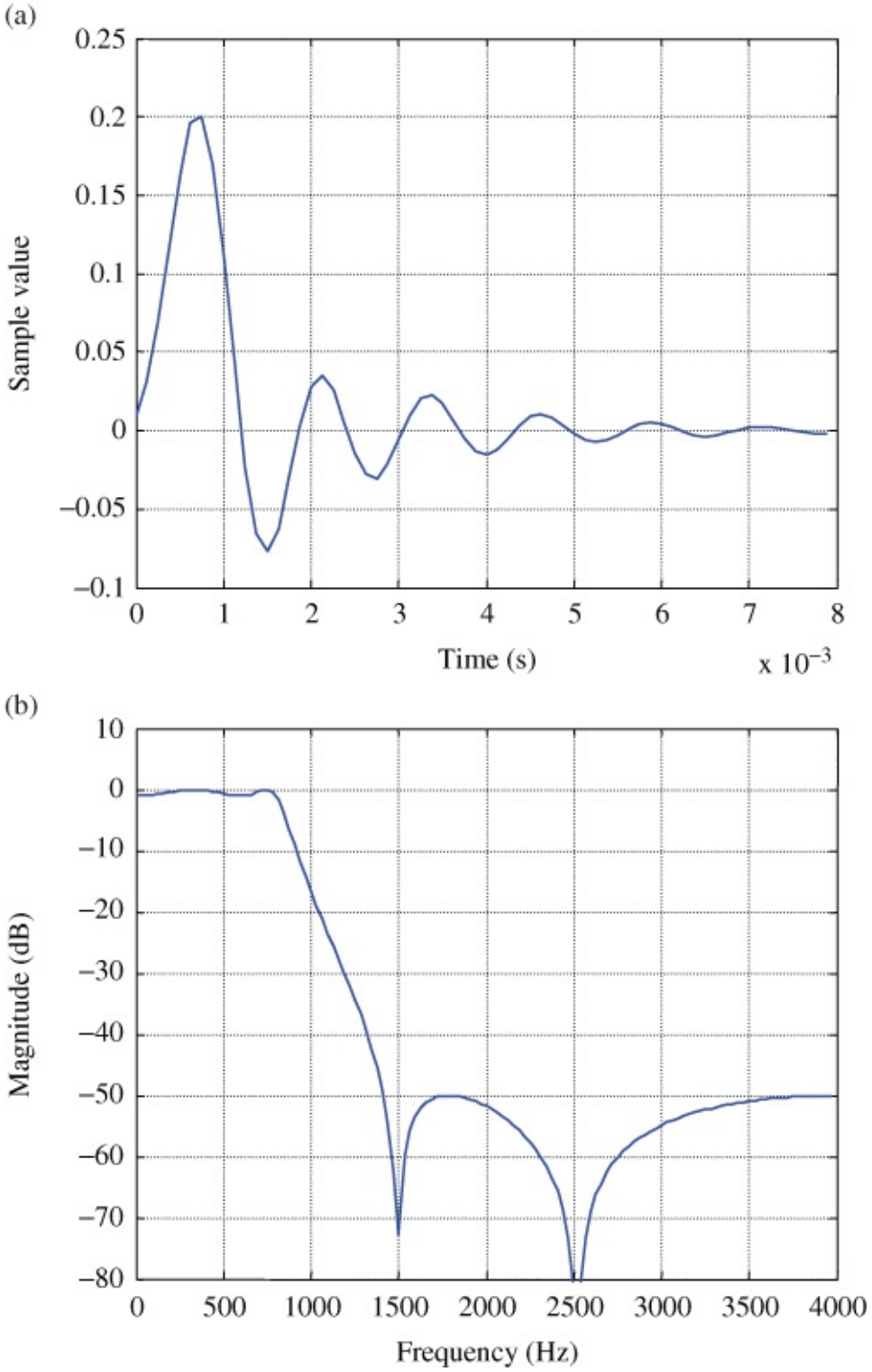


Figure 4.22 Impulse response and magnitude frequency response of the filter implemented by program `tm4c123_iirsos_delta_intr.c`, using coefficient file `elliptic.h`, plotted using MATLAB function `tm4c123_logfft()`.

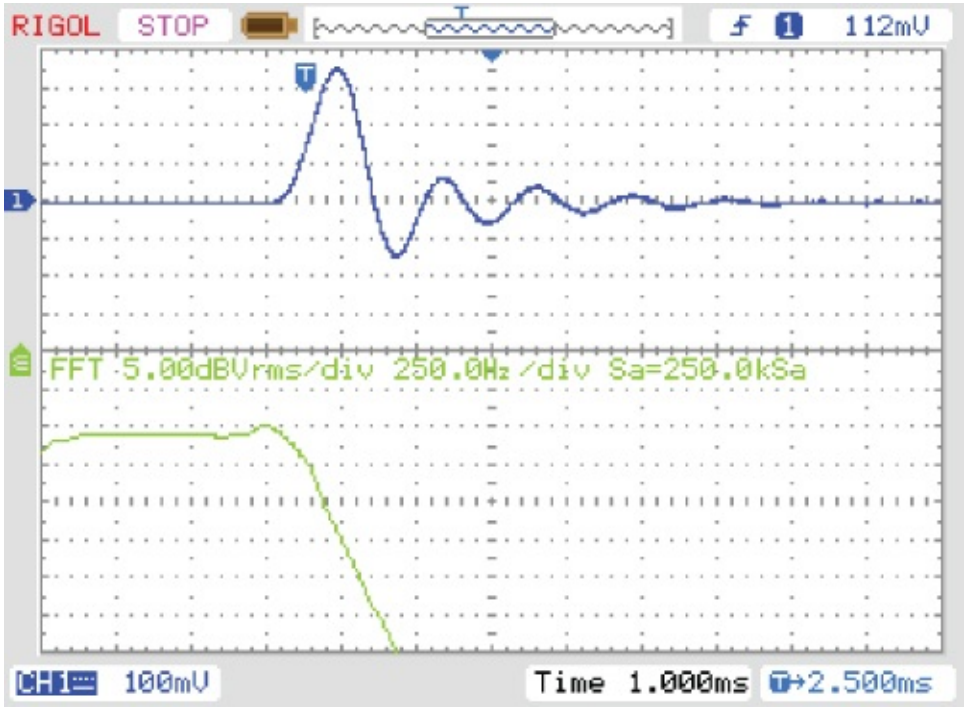


Figure 4.23 Output from program `tm4c123_iirsos_delta_intr.c`, using coefficient file `elliptic.h` viewed using a *Rigol DS1052E* oscilloscope.

Example 4.6

Band-Pass Filter Design Using `fdatool`.

Figure 4.24 shows `fdatool` used to design an 18th-order Chebyshev type 2 IIR band-pass filter centered at 2000 Hz. The filter coefficient file `bp2000.h` is compatible with programs `tm4c123_iirsos_intr.c`, `tm4c123_iirsos_delta_intr.c`, and `tm4c123_iirsos_prbs_intr.c`. Figures 4.25 and 4.26 show the output from program `tm4c123_iirsos_prbs_intr.c` using these coefficients.

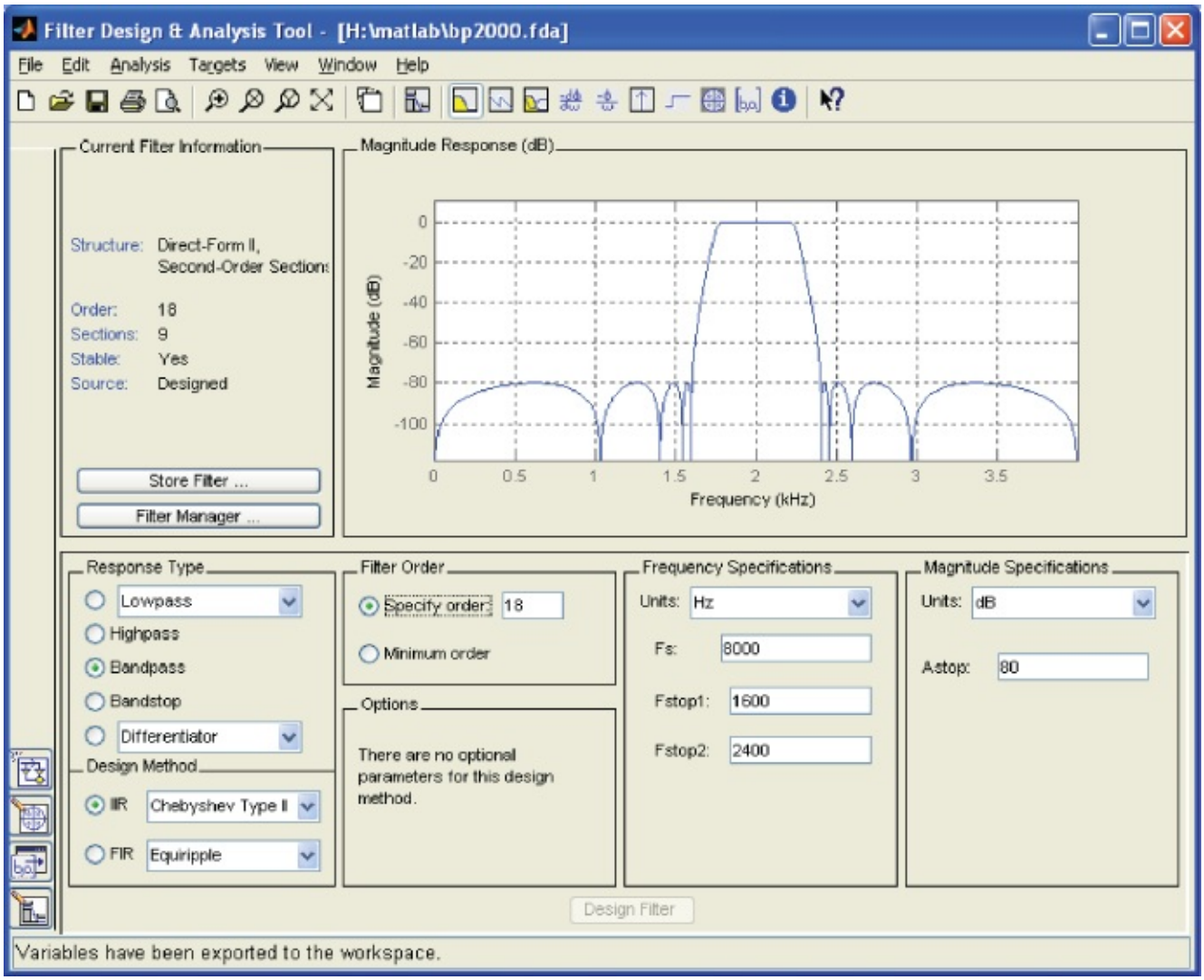


Figure 4.24 MATLAB `fdatool` window showing the magnitude frequency response of an 18th-order band-pass filter centered on 2000 Hz.

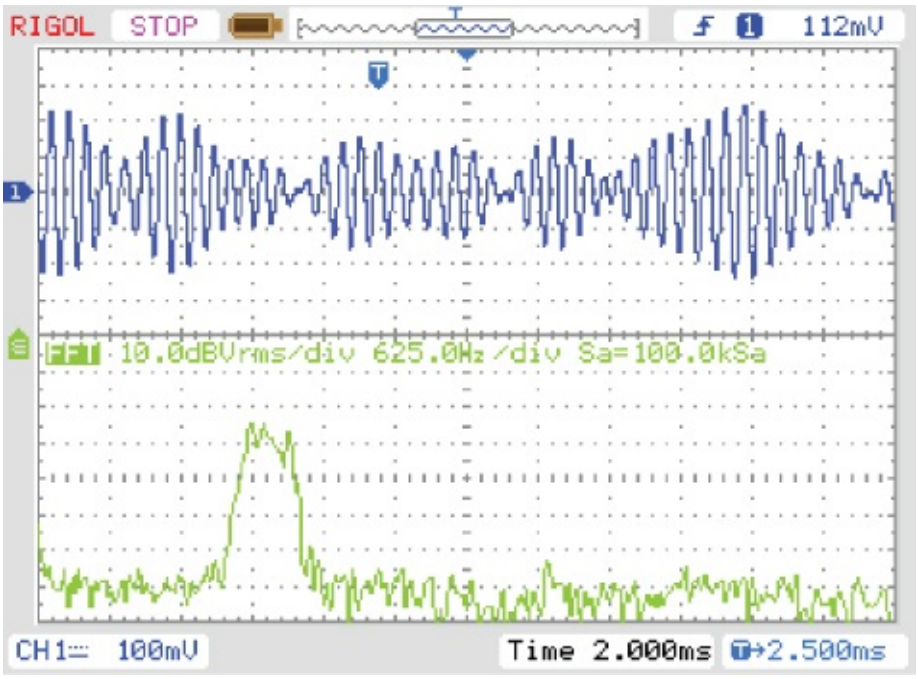


Figure 4.25 Output from program `tm4c123_iirsos_prbs_intr.c`, using coefficient file `bp2000.h` viewed using a *Rigol DS1052E* oscilloscope.

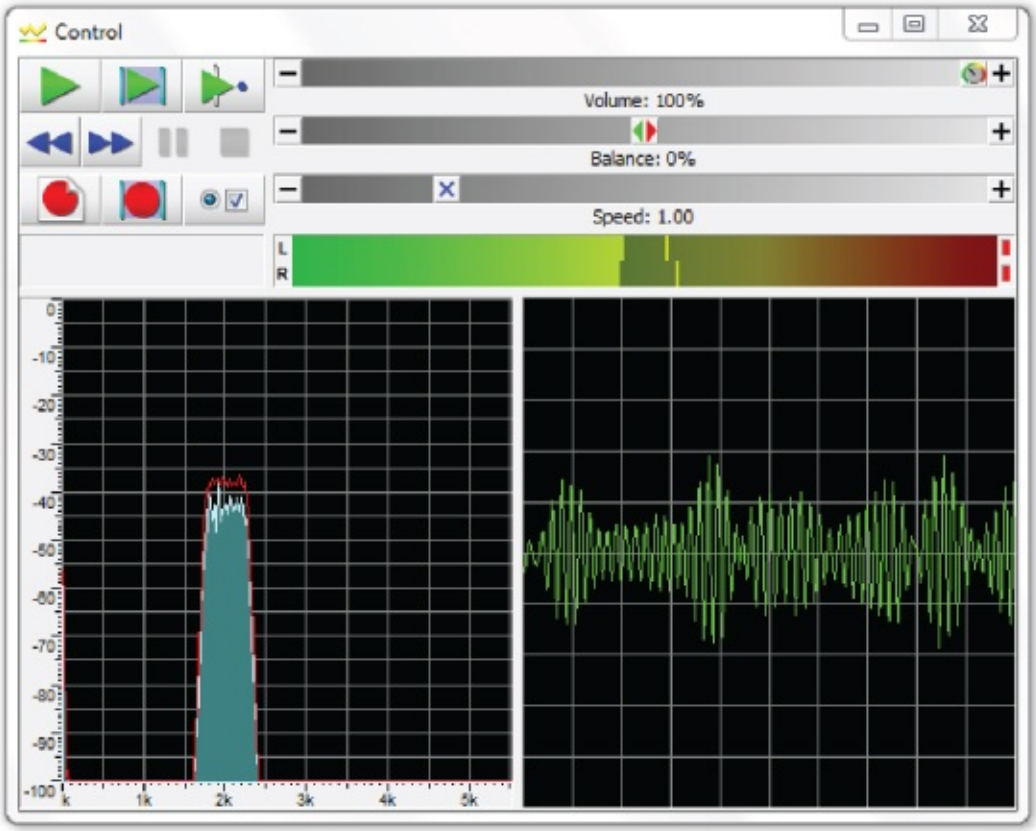


Figure 4.26 Output from program `tm4c123_iirsos_prbs_intr.c`, using coefficient file `bp2000.h` viewed using *Goldwave*.

Example 4.7

Implementation of IIR Filter Using CMSIS Function arm_biquad_cascade_f32() (`stm32f4_iirsos_CMSIS_intr.c`).

This example demonstrates the use of the CMSIS DSP library IIR filtering function `arm_biquad_cascade_df1_f32()`.

Function `arm_biquad_cascade_df1_f32()` implements a second-order IIR filter section (a biquad) using single precision (32-bit) floating point arithmetic. As the function name suggests, the filter is implemented as a direct form I structure. 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. However, program `stm32f4_iirsos_CMSIS_intr.c` uses sample-by-sample interrupt-based i/o and a block size of one sample (Listing 4.6).

Listing 4.6 Program `stm32f4_iirsos_CMSIS_intr.c`

```
// stm32f4_iirsos_CMSIS_intr.c
#include "stm32f4_wm5102_init.h"
#include "elliptic.h"
float32_t coeffs[5*NUM_SECTIONS] = {0};
float32_t state[4*NUM_SECTIONS] = {0};
arm_biquad_casd_df1_inst_f32 S;
void SPI2_IRQHandler()
{
  int16_t left_out_sample, left_in_sample;
  int16_t right_out_sample, right_in_sample;
  float32_t xn, yn;
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ⌋ SET)
  {
    left_in_sample = SPI_I2S_ReceiveData(I2Sx);
    xn =(float32_t)(left_in_sample);
    arm_biquad_cascade_df1_f32(&S, &xn, &yn, 1);
    left_out_sample = (int16_t)(yn);
    while (SPI_I2S_GetFlagStatus(I2Sxext,
           SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
  }
  else
  {
    right_in_sample = SPI_I2S_ReceiveData(I2Sx);
    while (SPI_I2S_GetFlagStatus(I2Sxext,
           SPI_I2S_FLAG_TXE ) != SET){}
    right_out_sample = 0;
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
}
```

```
int main(void)
{
  int i,k;
  k = 0;
  for (i=0; i<NUM_SECTIONS ; i++)
  {
    coeffs[k++] = b[i][0];
    coeffs[k++] = b[i][1];
    coeffs[k++] = b[i][2];
    coeffs[k++] = -a[i][1];
    coeffs[k++] = -a[i][2];
  }
  arm_biquad_cascade_df1_init_f32(&S, NUM_SECTIONS,
                                  coeffs, state);
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
  while(1){{
}
```

As supplied, the program implements the fourth-order elliptic low-pass filter used in Example 4.5, reading filter coefficients from the header file `elliptic.h`.

Within function `SPI2_IRQHandler()`, the value of a new left channel input sample is copied to `float32_t` variable `xn`.

Function `arm_biquad_cascade_df1_f32()` is passed pointers to

1. An instance of an IIR filter structure of type `arm_biquad_casd_inst_df1_f32`.
2. An array containing a sequence of input sample values of type `float32_t`.
3. An array in which to place a sequence of output sample values of type `float32_t`.
4. The number of input and output sample values.

An IIR filter structure `S` is declared in program statement

```
arm_biquad_casd_inst_df1_f32 S;
```

and, after the filter coefficients specified in header file `elliptic.h` as arrays `a` and `b` have been copied into the array `coeffs` used by function `arm_biquad_cascade_df1_f32()`, it is initialized in program statement

```
arm_biquad_cascade_df1_init_f32(&S, NUM_SECTIONS, coeffs, state);
```

Each second-order stage of a filter is represented by five coefficient values, of type `float32_t`, stored in array `coeffs` in the order $b_0, b_1, b_2, -a_1, -a_2$. These values correspond to a transfer function of the form

$$H(z) = \frac{X(z)}{Y(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}. \qquad \textbf{4.48}$$

The `arm_biquad_casd_inst_df1_f32` filter structure keeps track of the internal state of each of the `NUM_SECTIONS` second-order IIR filter stages, that is, it stores and updates previous input and output sample values $x(n-1)$, $x(n-2)$, $y(n-1)$, and $y(n-2)$.

4.5.1.7 Testing the Filter Using a Pseudorandom Input Signal

In program `stm32f4_iirsos_prbs_CMSIS_intr.c`, the statement in program `stm32f4_iirsos_CMSIS_intr.c` that reads

```
xn =(float32_t)(left_in_sample);
```

is replaced by

```
xn = (float32_t)(prbs(8000));
```

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.

In program `stm32f4_iirsos_delta_CMSIS_intr.c`, the input signal applied to the filter is a sequence of discrete impulses read from array `dimpulse`.

Example 4.8

Implementation of a Fourth-Order IIR Filter Using the AIC3104 Digital Effects Filter (`tm4c123_sysid_biquad_intr.c`).

The AIC3104 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 through 20 (*Left Channel Audio Effects Filter Coefficient Registers*) and 27 through 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

$$H(z) = \left(\frac{N_0 + 2N_1z^{-1} + N_2z^{-2}}{32768 - 2D_1z^{-1} - D_2z^{-2}}\right)\left(\frac{N_3 + 2N_4z^{-1} + N_5z^{-2}}{32768 - 2D_4z^{-1} - D_5z^{-2}}\right), \quad \textbf{4.49}$$

where coefficients N_i and D_i are 16-bit signed integers. Full details of these filters are given in the AIC3104 data sheet [1].

Program `tm4c123_sysid_biquad_intr.c` is almost identical to program `tm4c123_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 earlier.

Function `I2CRegWrite()` may be used to program the 8-bit control registers of the AIC3104 and each of the 16-bit coefficients of the filters must therefore be split into two 8-bit bytes. MATLAB function `tm4c123_aic3104_biquad()`, shown in Listing 4.7, has been provided to automate the generation of these program statements following calculation of filter coefficients using `fdatool`.

Listing 4.7 MATLAB function `tm4c123_aic3104_biquad()`

```
% TM4C123_AIC3104_BIQUAD.M
%
% MATLAB function to write C program statements
% to program left and right channel biquads in AIC3104 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 tm4c123_make_hex_str() defined in
% file TM4C123_MAKE_HEX_STR.M.
%
% Details of AIC3104 registers in TI document SLAS509E.
%
function tm4c123_aic310_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 automatically using');
fprintf(fid,' function tm4c123_aic3104_biquad.m\n',fname);
fn_name = 'I2CRegWrite';
parameters = 'I2C1_BASE, AIC3104_SLAVE_ADDRESS';
fsref divisors% left channel audio effects filter coefficients
a = tm4c123_make_hex_str(round(coeff(1,1)*gain(1)*(2^15 - 1)));
fprintf(fid,'%s(%s, 1,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s, 2,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(coeff(1,2)*gain(1)*2^14));
fprintf(fid,'%s(%s, 3,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s, 4,0x%s%s);\n',fn_name,parameters,a(1),a(2));
```

```
a = tm4c123_make_hex_str(round(coeff(1,3)*gain(1)*(2^15 - 1)));
fprintf(fid,'%s(%s, 5,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s, 6,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(coeff(2,1)*gain(2)*(2^15 - 1)));
fprintf(fid,'%s(%s, 7,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s, 8,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(coeff(2,2)*gain(2)*2^14));
fprintf(fid,'%s(%s, 9,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,10,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(coeff(2,3)*gain(2)*(2^15 - 1)));
fprintf(fid,'%s(%s,11,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,12,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(-coeff(1,5)*2^14));
fprintf(fid,'%s(%s,13,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,14,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(-coeff(1,6)*(2^15 - 1)));
fprintf(fid,'%s(%s,15,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,16,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(-coeff(2,5)*2^14));
fprintf(fid,'%s(%s,17,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,18,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(-coeff(2,6)*(2^15 - 1)));
fprintf(fid,'%s(%s,19,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,20,0x%s%s);\n',fn_name,parameters,a(1),a(2));
% right channel audio effects filter coefficients
a = tm4c123_make_hex_str(round(coeff(1,1)*gain(1)*(2^15 - 1)));
fprintf(fid,'%s(%s,27,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,28,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(coeff(1,2)*gain(1)*2^14));
fprintf(fid,'%s(%s,29,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,30,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(coeff(1,3)*gain(1)*(2^15 - 1)));
fprintf(fid,'%s(%s,31,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,32,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(coeff(2,1)*gain(2)*(2^15 - 1)));
fprintf(fid,'%s(%s,33,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,34,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(coeff(2,2)*gain(2)*2^14));
fprintf(fid,'%s(%s,35,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,36,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(coeff(2,3)*gain(2)*(2^15 - 1)));
fprintf(fid,'%s(%s,37,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,38,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(-coeff(1,5)*2^14));
fprintf(fid,'%s(%s,39,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,40,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(-coeff(1,6)*(2^15 - 1)));
fprintf(fid,'%s(%s,41,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,42,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(-coeff(2,5)*2^14));
fprintf(fid,'%s(%s,43,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,44,0x%s%s);\n',fn_name,parameters,a(1),a(2));
a = tm4c123_make_hex_str(round(-coeff(2,6)*(2^15 - 1)));
fprintf(fid,'%s(%s,45,0x%s%s);\n',fn_name,parameters,a(1),a(2));
fprintf(fid,'%s(%s,46,0x%s%s);\n',fn_name,parameters,a(1),a(2));
```

```
fclose(fid);
```

Connect LINE OUT (black) on the audio booster pack to LINE IN (blue) as shown in Figure 4.27 and build and run the program `tm4c123_sysid_biquad_intr.c` as supplied. Initially, the AIC3104 biquad filters are neither programmed nor enabled. Halt the program after a few seconds and save the 256 adaptive filter coefficients to a data file by typing

```
save <filename> start address, (start address + 0x400)
```

in the *Command* window of the *MDK-ARM debugger*, where `start address` is the address of array `firCoeffs32`. Plot the contents of the data file (the impulse response identified by the adaptive filter) using MATLAB function `tm4c123_logfft()`. You should see something similar to the graph shown in Figure 4.28.

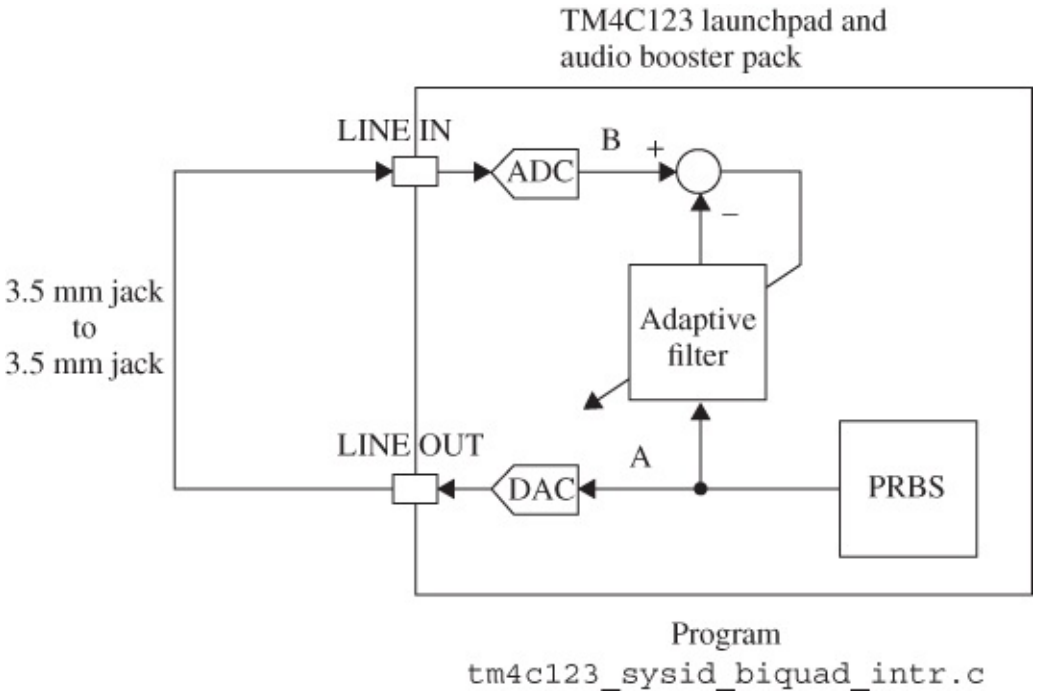


Figure 4.27 Connection diagram for program `tm4c123_sysid_biquad_intr.c`.

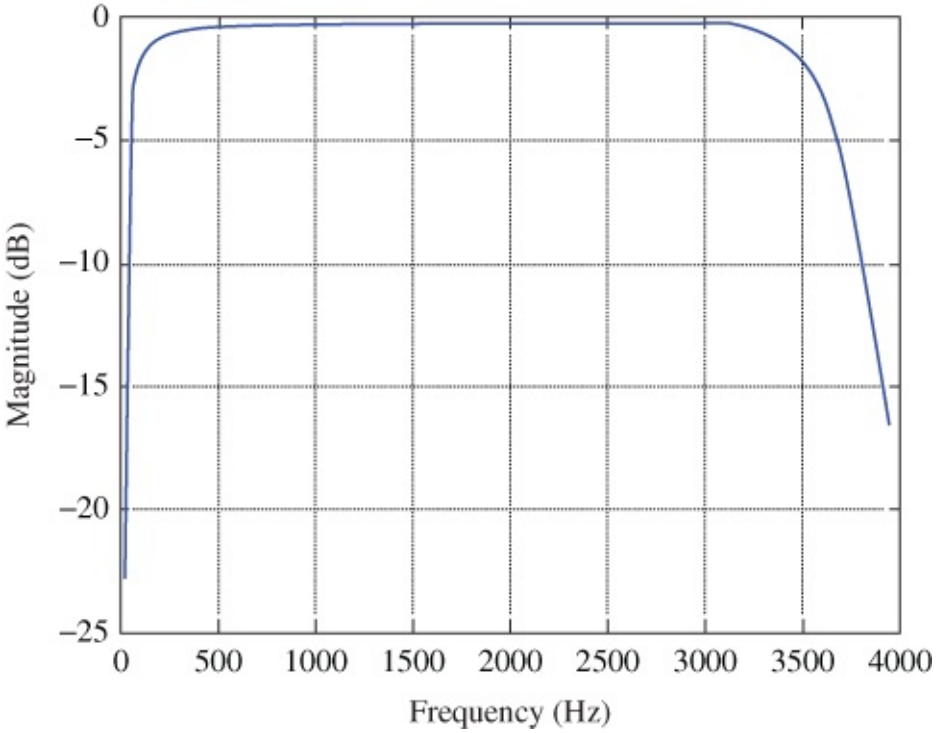


Figure 4.28 Frequency response of signal path through DAC, connecting cable, and ADC shown in Figure 4.27 with biquad filters disabled.

Listing 4.8 File `elliptic_coeffs_biquad.h`

```
// elliptic_coeffs_biquad.h
//
// this file was generated automatically using m-file
// tm4c123_aic3106_biquad.m
//
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS,  1,0x01);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS,  2,0x49);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS,  3,0x00);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS,  4,0x82);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS,  5,0x01);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS,  6,0x49);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS,  7,0x7f);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS,  8,0xff);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS,  9,0xce);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 10,0x47);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 11,0x7f);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 12,0xff);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 13,0x61);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 14,0xd7);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 15,0xae);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 16,0x76);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 17,0x60);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 18,0xe1);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 19,0x90);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 20,0xd0);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 27,0x01);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 28,0x49);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 29,0x00);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 30,0x82);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 31,0x01);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 32,0x49);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 33,0x7f);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 34,0xff);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 35,0xce);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 36,0x47);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 37,0x7f);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 38,0xff);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 39,0x61);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 40,0xd7);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 41,0xae);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 42,0x76);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 43,0x60);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 44,0xe1);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 45,0x90);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS, 46,0xd0);
```

Listing 4.8 shows the contents of file `elliptic_coeffs_biquad.h`. This file was generated using MATLAB function `aic3104_biquad()` to process filter coefficients of a fourth-order elliptic low-pass filter designed using `fdatool`. Cut and paste these statements into program

`tm4c123_sysid_biquad_intr.c` just after the program statement

```
tm4c123_aic3104_init(FS_8000_HZ,
                     INPUT_LINE,
                     IO_METHOD_INTR,
                     PGA_GAIN_6_DB);
```

Add the statement

```
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS,   0, 0x01);
```

immediately preceding the statements cut and pasted from file `elliptic_coeffs_biquad.h` and add the statements

```
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS,   0, 0x00);
I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS,  12, 0x0A);
```

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. Halt the program after a few seconds and save the coefficients and plot using MATLAB. You should now see the magnitude frequency response of the biquad filter, as shown in Figure 4.29.

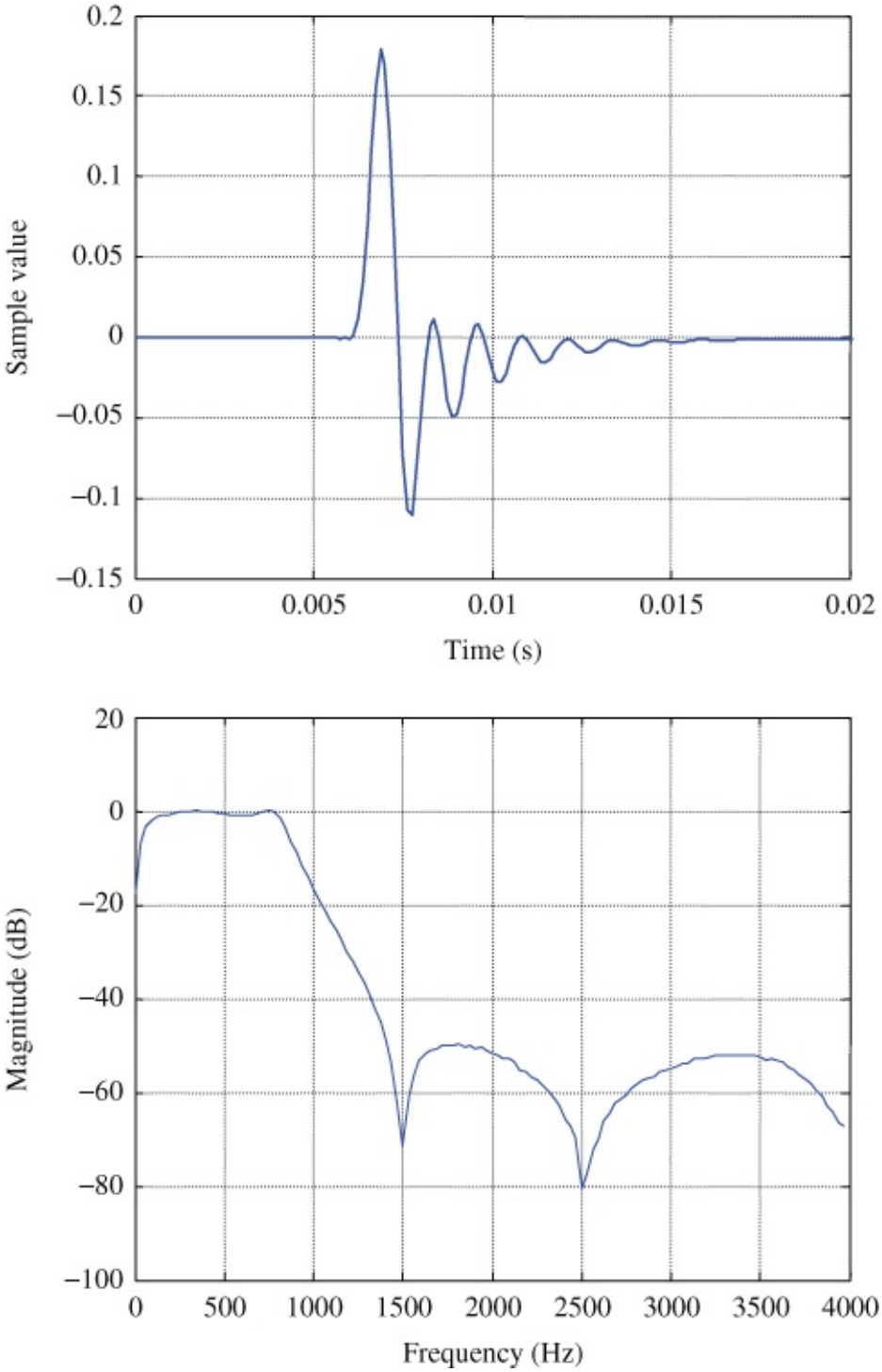


Figure 4.29 Frequency response of signal path through DAC, connecting cable, and ADC shown in Figure 4.27 with biquad filters programmed as a fourth-order elliptic low-pass filter and enabled.

4.5.1.8 Changing the Response of the AIC3104 Digital Effects Filter

In order to program an alternative filter response into the digital effects filter,

1. Design an alternative fourth-order IIR filter using fdatool. Figure 4.30 shows the design of a fourth-order 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 `aic3104_biquad(SOS,G)` and enter a filename, for example, `bandpass_coeffs_biquad.h`.
4. Cut and paste the statements contained in file `bandpass_coeffs_biquad.h` into program `tm4c123_sysid_biquad_intr.c` (in place of those used in the previous example) immediately following the call to function `tm4c123_aic3104_init()` and immediately preceding the program statements

   ```
   I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS,   0, 0x00);
   I2CRegWrite(I2C1_BASE, AIC3104_SLAVE_ADDRESS,  12, 0x0A);
   ```

   replacing the previous calls to function `I2CRegWrite()`.
5. Build and run the program as before.
6. Observe the response identified using MATLAB function `tm4c123_logfft()` to plot the values stored in array `firCoeffs32`.

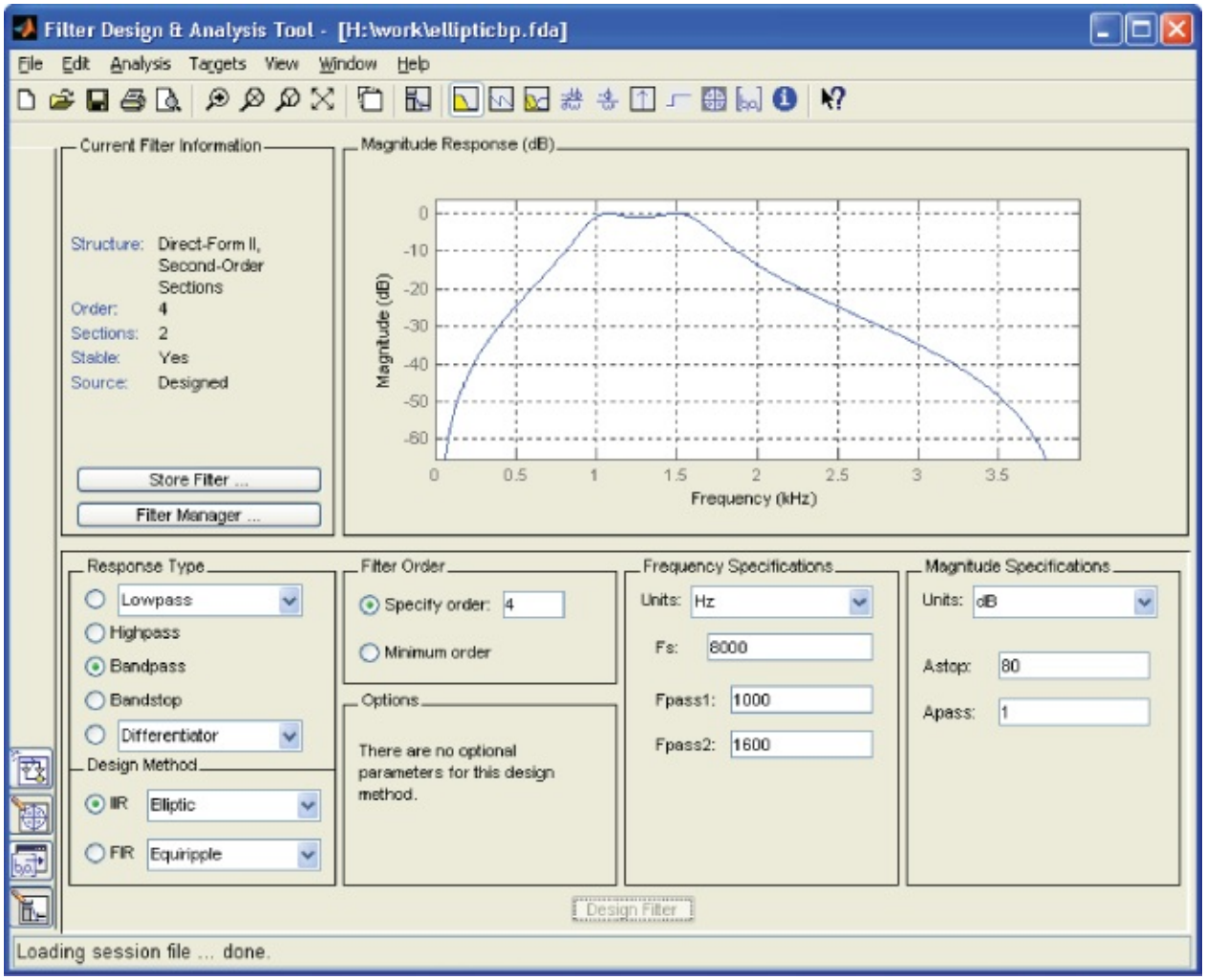


Figure 4.30 `fdatool` used to design a fourth-order elliptic band-pass filter.

Figure 4.31 shows the identified frequency response for the coefficients contained in file

`bandpass_coeffs_biquad.h` using MATLAB function `tm4c123_logfft()`.

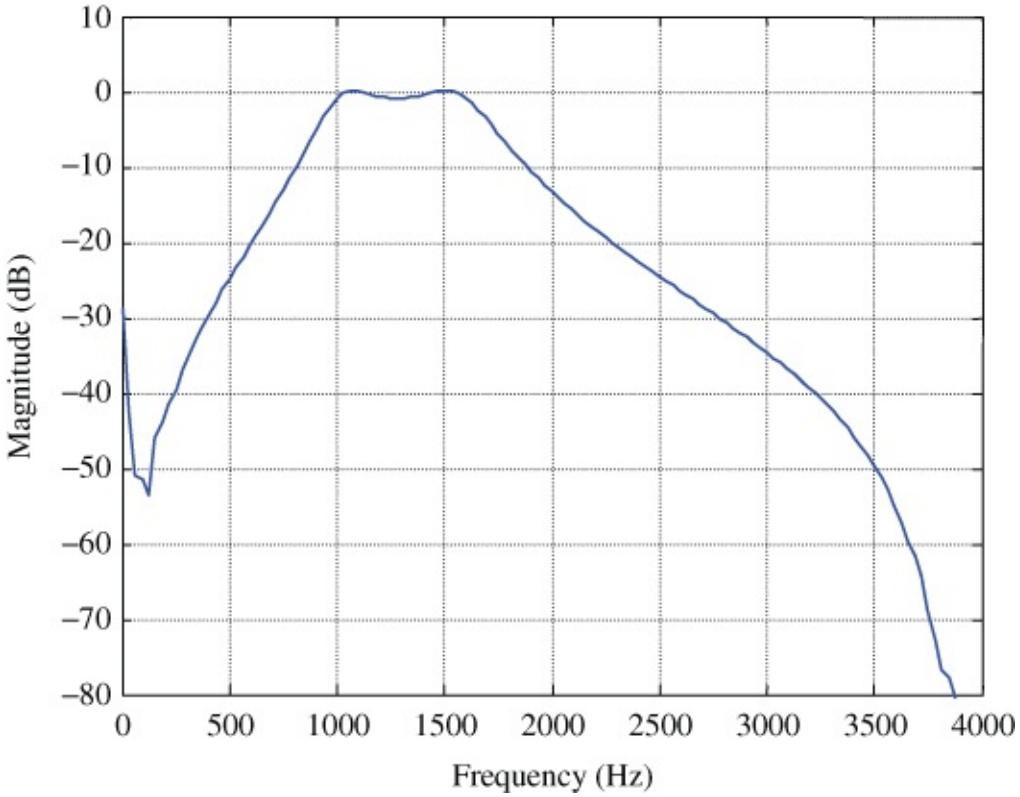


Figure 4.31 Frequency response of signal path through DAC, connecting cable, and ADC shown in Figure 4.27 with biquad filters programmed as a fourth-order elliptic band-pass filter and enabled.

Example 4.9

Generation of a Sine Wave Using a Difference Equation (`stm32f4_sinegenDE_intr.c`).

In Chapter 3, it was shown that the z-transform of a sinusoidal sequence $y(n) = u(n)\sin(n\omega T)$ is given by

$$Y(z) = \frac{z\sin(\omega T)}{z^2 - 2\cos(\omega T)z + 1}. \tag{4.50}$$

Comparing this with the z-transfer function of the second-order filter of Example 4.1

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_1 z}{z^2 + a_1 z + a_2}. \tag{4.51}$$

It is apparent that by appropriate choice of filter coefficients, we can configure the filter to act as a sine wave generator, that is, to have a sinusoidal impulse response. Choosing $a_2 = 1$ 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 (4.51) and setting $x(n) = \delta(n)$, ($X(z) = 1$), and $b_1 = \sin(\omega T)$

$$Y(z) = \frac{X(z)b_1 z}{z^2 + a_1 z + a_2} = \frac{\sin(\omega T)z}{z^2 - 2\cos(\omega T)z + 1}. \quad \textbf{4.52}$$

Equation (4.52) is equivalent to Equation (4.50), implying that the filter impulse response is $y(n) = u(n)\sin(n\omega T)$. Equation (4.51) corresponds to the difference equation

$$y(n) = \sin(\omega T)x(n-1) + 2\cos(\omega T)y(n-1) - y(n-2), \quad \textbf{4.53}$$

which is illustrated in block diagram form in Figure 4.32.

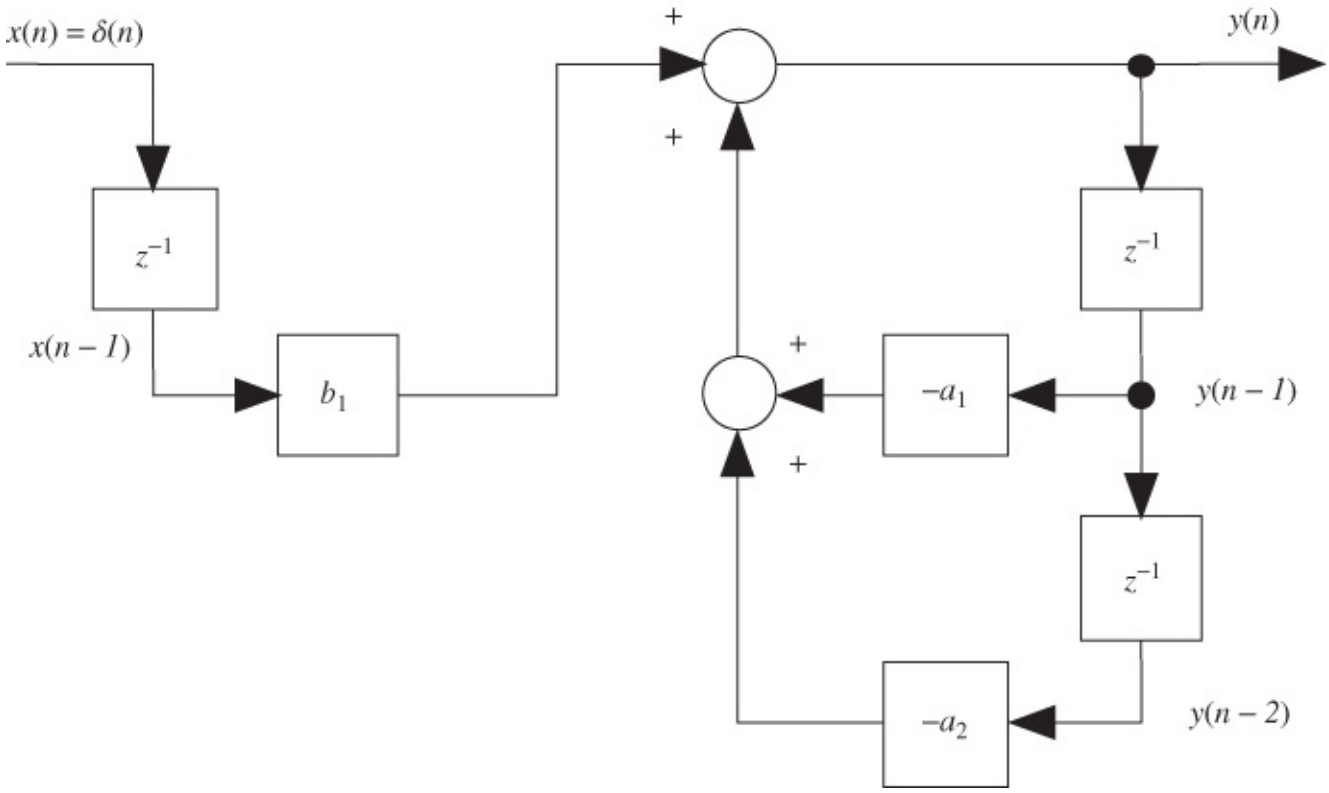


Figure 4.32 Block diagram representation of Equation (4.53).

Since the input, $x(n)$, to the filter is nonzero only at sampling instant $n = 0$, the difference equation is equal to

$$y(n) = 2\cos(\omega T)y(n-1) - y(n-2) \quad \textbf{4.54}$$

for all other n and hence the sine wave generator may be implemented as shown in Figure 4.33, using no input signal but using nonzero initial values for $y(n-1)$ and $y(n-2)$. These initial values determine the amplitude of the sinusoidal output.

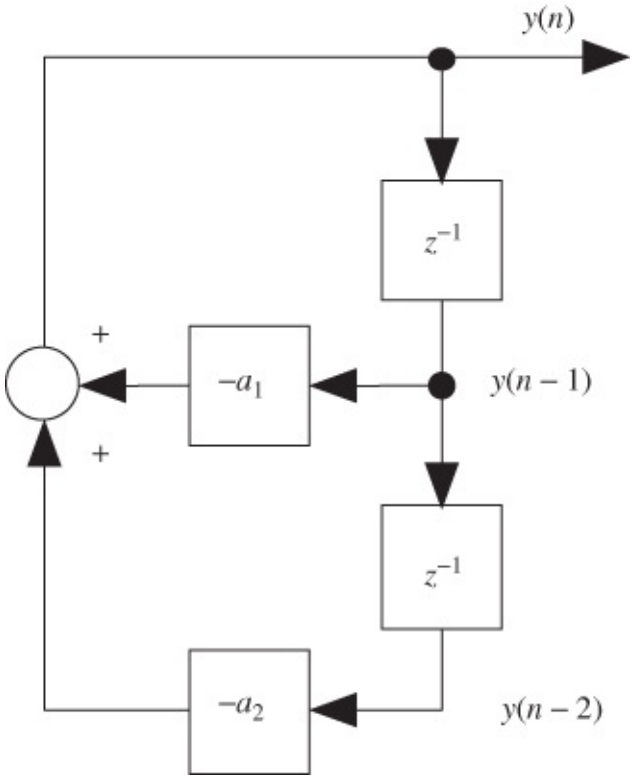


Figure 4.33 Block diagram representation of Equation (4.54).

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, which are one sampling period, or T seconds, apart in time, that is,

$$y(n-1) = A\sin(\omega t + \phi),$$
$$y(n-2) = A\sin(\omega(t+T) + \phi).$$

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 `tm4c123_sinegenDE_intr.c` (Listing 4.9), is

$$y(n-1) = 0$$
$$y(n-2) = A\sin(\omega T)$$

Build and run this program as supplied (`FREQ = 2000`) and verify that the output is a 2000 Hz tone. Change the value of the constant `FREQ`, build and run the program, and verify the generation of a tone of the frequency selected.

Listing 4.9 Program `stm32f4_sinegenDE_intr.c`

```
// stm32f4_sinegenDE_intr.c
#include "stm32f4_wm5102_init.h"
float32_t y[3]; // filter states - previous output values
float32_t a1;   // filter coefficient
const float32_t AMPLITUDE = 8000.0;
const float32_t FREQ = 2000.0;
const float32_t SAMPLING_FREQ = 8000.0;
void SPI2_IRQHandler()
{
  int16_t left_out_sample, left_in_sample;
  int16_t right_out_sample, right_in_sample;
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ⌋ SET)
  {
    left_in_sample = (float32_t)(SPI_I2S_ReceiveData(I2Sx));
    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)
    left_out_sample = (int16_t)(y[0]);
    while (SPI_I2S_GetFlagStatus(I2Sxext,
           SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
  }
  else
  {
    right_in_sample = SPI_I2S_ReceiveData(I2Sx);
    while (SPI_I2S_GetFlagStatus(I2Sxext,
           SPI_I2S_FLAG_TXE ) != SET){}
    right_out_sample = 0;
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
}
int main(void)
{
  y[1] = 0.0;
  y[2] = AMPLITUDE*sin(2.0*PI*FREQ/SAMPLING_FREQ);
  a1 = -2.0*cos(2.0*PI*FREQ/SAMPLING_FREQ);
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
  while(1){}
}
```

Example 4.10

Generation of DTMF Signal Using Difference Equations (`stm32f4_sinegenDTMF_intr.c`).

Program `stm32f4_sinegenDTMF_intr.c`, shown in Listing 4.10, uses the same difference equation method as program `tm4c123_sinegenDE_intr` to generate two sinusoidal signals of different frequencies, which, added together, form a DTMF tone (see also Example 2.12, which used a table lookup method). The program also incorporates a buffer (array `out_buffer`) that is used to store the 256 most recent output samples. Figure 4.34 shows the contents of that buffer in time and frequency domains, plotted using MATLAB function `stm32f4_logfft()` after halting the program and saving the contents of `out_buffer` to a file. Figure 4.35 shows the analog output signal generated by the program captured using an oscilloscope. Unlike the FFT function in the oscilloscope, MATLAB function `stm32f4_logfft()` has not applied a Hamming window to the sample values.

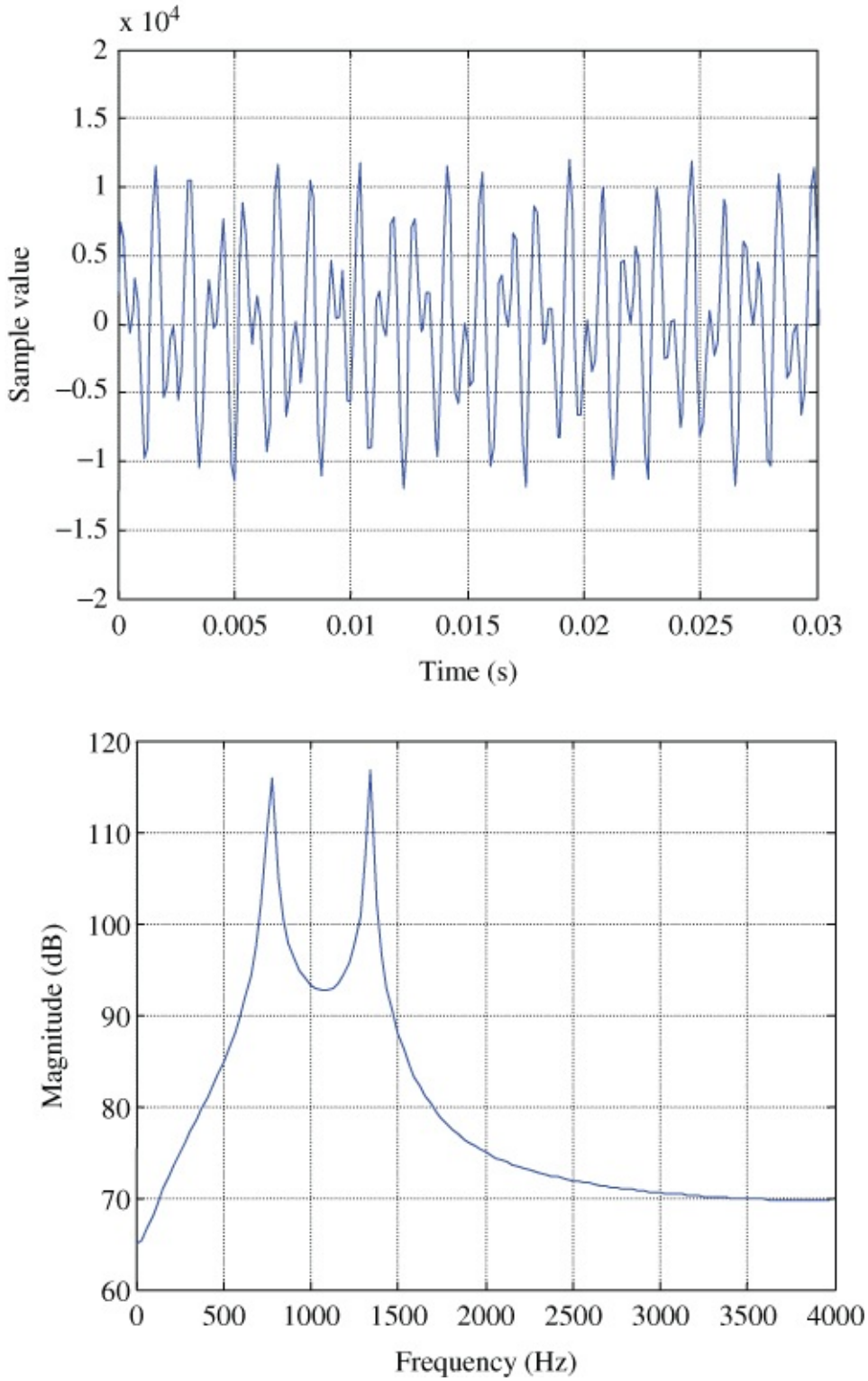


Figure 4.34 Output samples generated by program `stm32f4_sinegenDTMF_intr.c` plotted using MATLAB function `stm32f4_logfft()`.

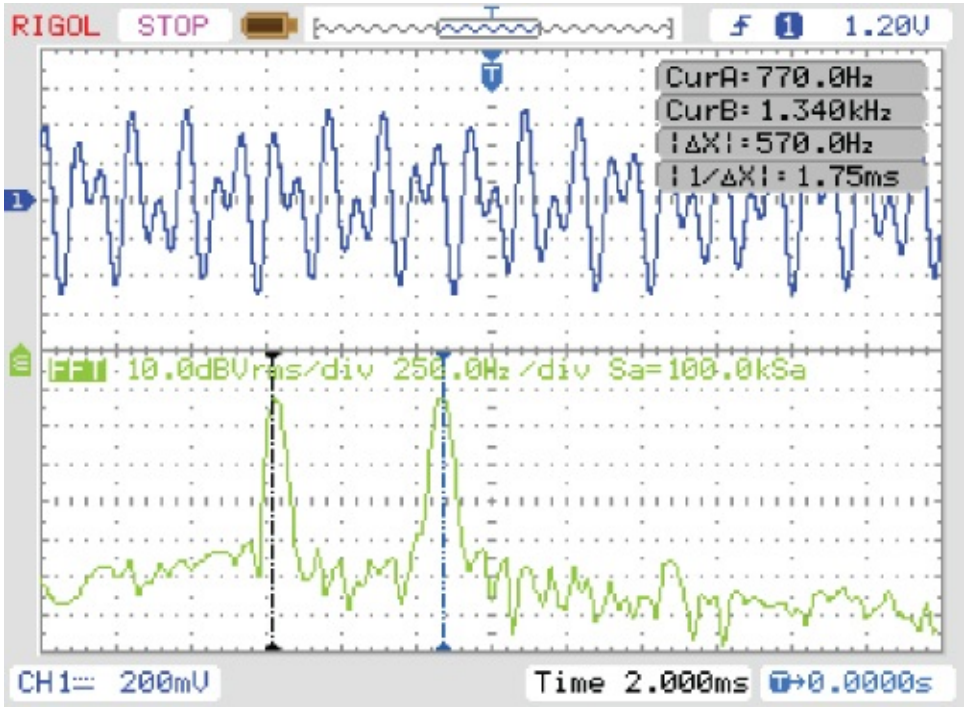


Figure 4.35 Output signal generated by program `stm32f4_sinegenDTMF_intr.c` viewed using a *Rigol DS1052E* oscilloscope.

Listing 4.10 Program `stm32f4_sinegenDTMF_intr.c`

```
// stm32f4_sinegenDTMF_intr.c
#include "stm32f4_wm5102_init.h"
#define FREQLO 770
#define FREQHI 1336
#define SAMPLING_FREQ 8000
#define AMPLITUDE 6000
#define BUFFER_SIZE 256
float ylo[3];
float yhi[3];
float a1lo, a1hi;
float out_buffer[BUFFER_SIZE];
int bufptr = 0;
void SPI2_IRQHandler()
{
  int16_t left_out_sample, left_in_sample;
  int16_t right_out_sample, right_in_sample;
  float32_t output;
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) ⌋ SET)
  {
    GPIO_SetBits(GPIOD, GPIO_Pin_15);
    left_in_sample = SPI_I2S_ReceiveData(I2Sx);
    ylo[0] = -(ylo[1]*a1lo)-ylo[2];
    ylo[2] = ylo[1];                    //update y1(n-2)
    ylo[1] = ylo[0];                    //update y1(n-1)
    yhi[0] = -(yhi[1]*a1hi)-yhi[2];
    yhi[2] = yhi[1];                    //update y1(n-2)
```

```
    yhi[1] = yhi[0];                    //update y1(n-1)
    output = (yhi[0]+ylo[0]);
    out_buffer[bufptr++] = output;
    if (bufptr>= BUFSIZE) bufptr = 0;
    left_out_sample = (int16_t)(output);
    while (SPI_I2S_GetFlagStatus(I2Sxext,
           SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
    GPIO_ResetBits(GPIOD, GPIO_Pin_15);
  }
  else
  {
    right_out_sample = SPI_I2S_ReceiveData(I2Sx);
    while (SPI_I2S_GetFlagStatus(I2Sxext,
           SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
}
int main(void)
{
  ylo[1] = 0.0;
  ylo[2] = AMPLITUDE*sin(2.0*PI*FREQLO/SAMPLING_FREQ);
  a1lo = -2.0*cos(2.0*PI*FREQLO/SAMPLING_FREQ);
  yhi[1] = 0.0;
  yhi[2] = AMPLITUDE*sin(2.0*PI*FREQHI/SAMPLING_FREQ);
  a1hi = -2.0*cos(2.0*PI*FREQHI/SAMPLING_FREQ);
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
  while(1){}
}
```

Example 4.11

Generation of a Swept Sinusoid Using a Difference Equation (`stm32f4_sweepDE_intr.c`).

Listing 4.11 is of program `stm32f4_sweepDE_intr.c`, which generates a sinusoidal signal, sweeping repeatedly from low to high frequency. The program implements the difference equation

$$y(n) = -a_1 y(n-1) - y(n-2), \tag{4.55}$$

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 with 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 `stm32f4_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 shown in Listing 4.11, 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 (`SWEEP_PERIOD/SAMPLING_FREQ)*(MAX_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.

Listing 4.11 Program `stm32f4_sweepDE_intr.c`

```
// stm32f4_sweepDE_intr.c
#include "stm32f4_wm5102_init.h"
#define MIN_FREQ 200
#define MAX_FREQ 3800
#define STEP_FREQ 20
#define SWEEP_PERIOD 400
#define SAMPLING_FREQ 8000.0f
#define AMPLITUDE 4000.0f
#define PI 3.14159265358979f
float32_t y[3] = {0.0, 0.0, 0.0};
float32_t a1;
float32_t freq = MIN_FREQ;
int16_t sweep_count = 0;
;
void coeff_gen(float freq)
{
  float32_t kk;
  kk =  2.0*PI*freq/SAMPLING_FREQ;
  a1 = -2.0*arm_cos_f32(kk);
  y[0] = 0.0;
  y[2] = AMPLITUDE*arm_sin_f32(kk);
  y[1] = 0.0;
  return;
}
void SPI2_IRQHandler()
{
  int16_t left_out_sample = 0;
```

```
  int16_t right_out_sample = 0
  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) != SET)
  {
    GPIO_SetBits(GPIOD, GPIO_Pin_15);
    left_out_sample = SPI_I2S_ReceiveData(I2Sx);
    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];
    y[2] = y[1];                          // update y1(n-2)
    y[1] = y[0];                          // update y1(n-1)
    left_out_sample = (int16_t)(y[0]);
    while (SPI_I2S_GetFlagStatus(I2Sxext,
           SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
    GPIO_ResetBits(GPIOD, GPIO_Pin_15);
  }
  else
  {
    right_out_sample = SPI_I2S_ReceiveData(I2Sx);
    while (SPI_I2S_GetFlagStatus(I2Sxext,
           SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, right_out_sample);
  }
}
int main(void)
{
  coeff_gen(freq);
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_INTR);
  while(1){}
}
```

Example 4.12

Cascaded Second-Order Notch Filters (`tm4c123_iirsos_intr`).

In Chapter 3, two 89-coefficient FIR notch filters were used to remove unwanted tones from a signal. In this example, the use of simple second-order IIR notch filters to achieve a similar result is demonstrated. A second-order IIR notch filter has the form

$$H(z) = \frac{X(z)}{Y(z)} = \frac{1 - 2\cos(\hat{\omega}_n)z^{-1} + z^{-2}}{1 - 2r\cos(\hat{\omega}_n)z^{-1} + r^2 z^{-2}} \quad \textbf{4.56}$$

corresponding, in the z-plane, to two complex conjugate zeros on the unit circle at angles $+/-\hat{\omega}_n$ from the real axis and two complex conjugate poles of magnitude r, at similar angles. This is illustrated in Figure 4.36.

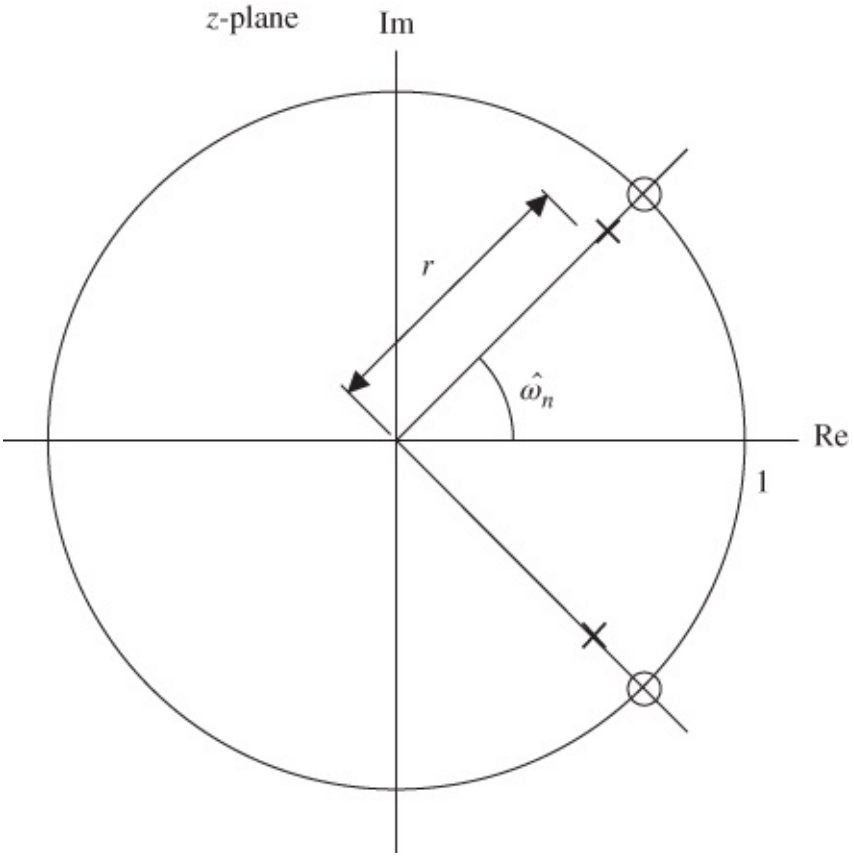


Figure 4.36 Pole-zero map for notch filter described by Equation (4.56) for $r = 0.8$ and $\hat{\omega} = \pi/4$.

The magnitude frequency response of this filter contains a deep notch at frequency $\hat{\omega}_n$ radians with 3 dB cutoff frequencies spaced r radians apart. In other words, the parameter r determines the width of the notch shown in Figure 4.37.

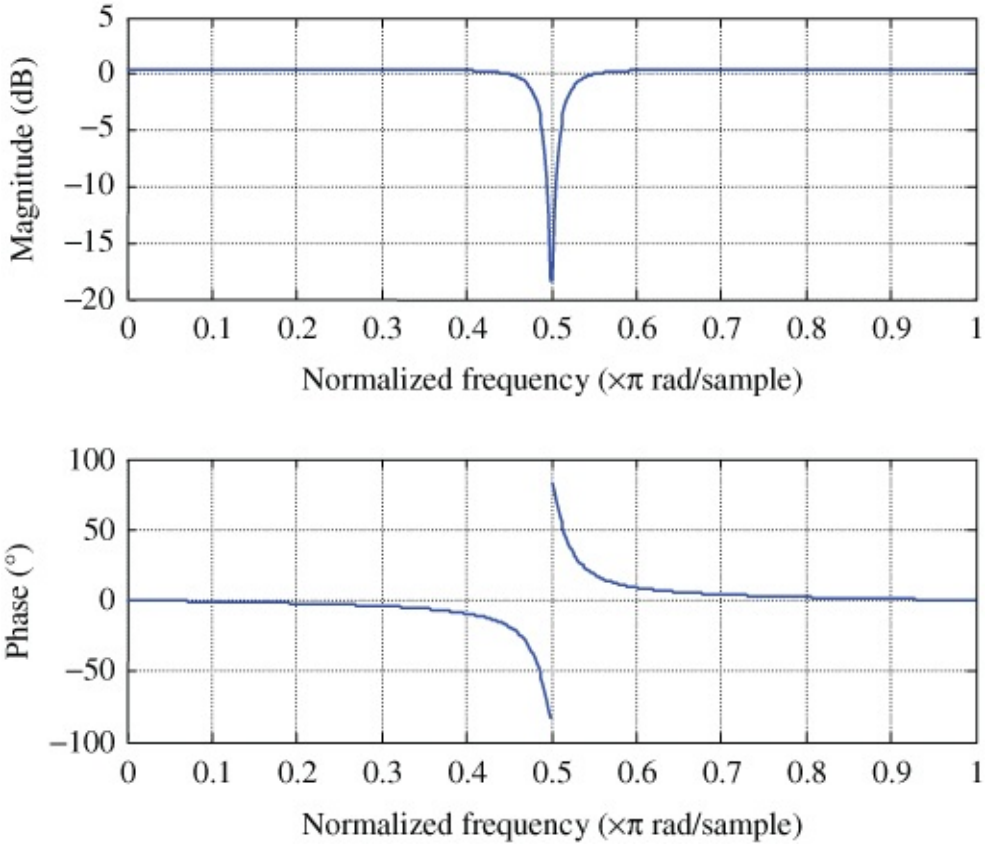


Figure 4.37 Frequency response of notch filter described by Equation (4.56) for $r = 0.8$ and $\hat{\omega} = \pi/4$.

Two cascaded second-order IIR notch filters with notches at 900 and 2700 Hz programs may be implemented using either program `tm4c123_iirsos_intr.c` or `tmc123_iirsos_prbs_intr.c` simply by including the header file `iir_notch_coeffs.h` (shown in Listing 4.12).

Listing 4.12 Coefficient header file `iir_notch_coeffs.h`

```
// iir_notch_coeffs.h
// cascaded second-order notch filters 900 Hz and 2700 Hz
#define NUM_SECTIONS 2
float b[NUM_SECTIONS][3] = {
{1.00000000, 1.04499713, 1.00000000},
{1.00000000, -1.5208, 1.00000000} };
float a[NUM_SECTIONS][3] = {
{1.00000000, 0.94049741, 0.9025},
{1.00000000, -1.44476, 0.9025} };
```

Run program `tmc123_iirsos_intr.c` using that coefficient header file, and test its response to the input signal stored in file `corrupt.wav`. Play the test signal using *Goldwave*, *Windows Media Player*, or similar, and connect the PC sound card output to the (blue) LINE IN

connection on the audio booster card.

Figure 4.38 shows pseudorandom noise filtered by program `tmc123_iirsos_prbs_intr.c` using header file `iir_notch_coeffs.h`.

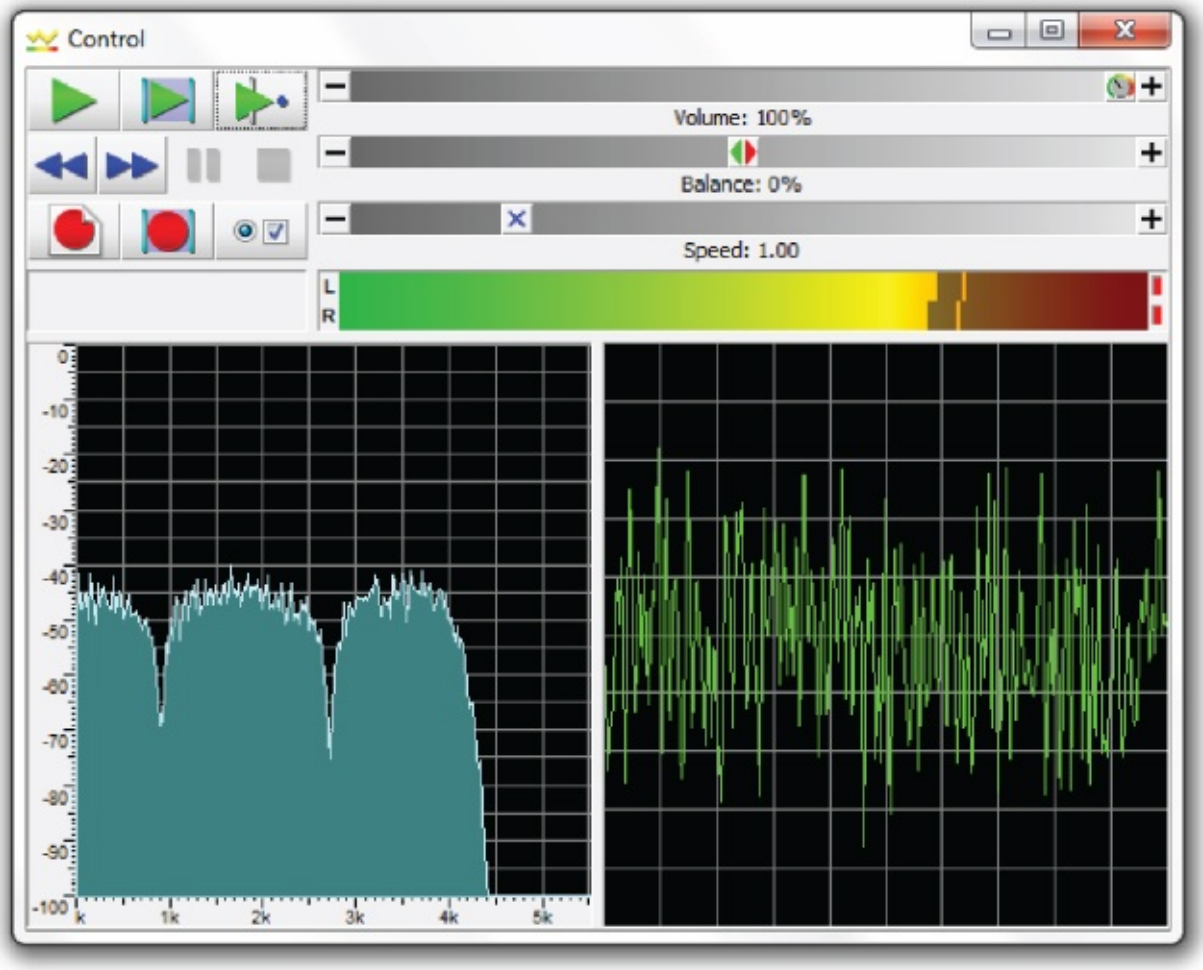


Figure 4.38 Pseudorandom noise filtered by program `tmc123_iirsos_prbs_intr.c` using header file `iir_notch_coeffs.h`.

Reference

1. Texas Instruments, Inc., “TLV320AIC3104 Low-Power Stereo Audio Codec for Portable Audio and Telephony”, Literature no. SLAS510D, 2014.

Chapter 5
Fast Fourier Transform

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), Fourier Series (FS), Discrete-time Fourier transform (DTFT), and 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, and if that signal is discrete, then one cycle comprises a finite number of samples. The DFT transforms N complex time-domain samples into N complex frequency-domain values. Hence, both forward and inverse DFTs are described by finite summations as opposed to either 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 (FFT) 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 lengths of the sample sequences involved. The FFT makes use of the periodic nature, and of symmetry, in the *twiddle factors* used in the DFT. 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 given by

$$X_N(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}. \qquad 5.1$$

The constants W are referred to as *twiddle constants* or *twiddle factors*, where

$$W_N = e^{-j(2\pi/N)}. \quad \textbf{5.2}$$

Computing all N values of $X_N(k)$ ($0 \le k < N$) involves the evaluation of N^2 product terms of the form $x(n)W_N^{kn}$, each of which (aside from the requirement of computing W_N^{kn}) requires a complex multiplication. For larger N, the computational requirements (N^2 complex multiplications) of the DFT can be very great. The FFT algorithm takes advantage of the periodicity

$$W_N^{(k+N)} = W_N^k \quad \textbf{5.3}$$

and symmetry

$$W_N^{(k+N/2)} = -W_N^k \quad \textbf{5.4}$$

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 $N^2 = 64$ different combinations of n and k used in evaluation of Equation (5.1) result in only $N = 8$ 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.

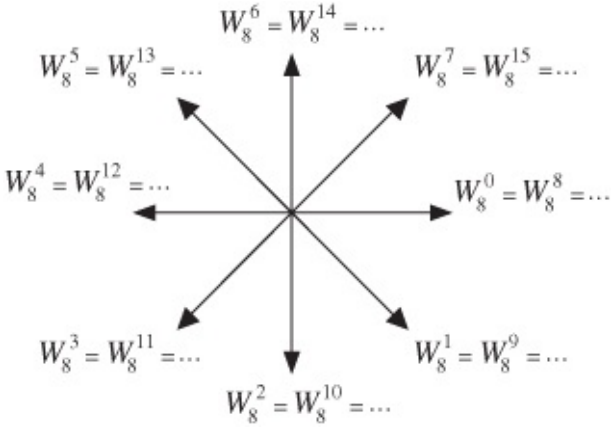


Figure 5.1 Twiddle factors W_N^{kn} for $N = 8$ represented as vectors in the complex plane.

A second, important, way in which the radix-2 FFT saves computational effort is by decomposing an N-point DFT into a combination 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 $X_2(1) = x(0) - x(1)$.

5.3 Decimation-in-Frequency FFT Algorithm with RADIX-2

Consider the N-point DFT

$$X_N(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \tag{5.5}$$

where

$$W_N = e^{-j(2\pi/N)}, \tag{5.6}$$

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 $x(0), x(1), \cdots, x(N/2-1)$ and the other containing the remaining $N/2$ values $x(N/2), x(N/2+1), \cdots, x(N-1)$. Splitting the DFT summation into two parts

$$\begin{aligned} X_N(k) &= \sum_{n=0}^{(N/2)-1} x(n) W_N^{nk} + \sum_{n=0}^{(N/2)-1} x\left(n+\frac{N}{2}\right) W_N^{(n+(N/2))k} \\ &= \sum_{n=0}^{(N/2)-1} x(n) W_N^{kn} + W_N^{(N/2)k} \sum_{n=0}^{(N/2)-1} x\left(n+\frac{N}{2}\right) W_N^{nk}. \end{aligned} \tag{5.7}$$

The term $W_N^{(N/2)k}$ is placed outside the second summation since it is not a function of n. Making use of

$$W_N^{(N/2)k} = e^{-j\pi k} = (-1)^k. \tag{5.8}$$

Equation (5.7) becomes

$$X_N(k) = \sum_{n=0}^{(N/2)-1} \left[x(n) + (-1)^k x\left(n+\frac{N}{2}\right)\right] W_N^{nk}. \tag{5.9}$$

Because $(-1)^k = 1$ for even k and $(-1)^k = -1$ for odd k, Equation (5.9) can be split into two parts. For even k,

$$X_N(k) = \sum_{n=0}^{(N/2)-1} \left[x(n) + x\left(n+\frac{N}{2}\right)\right] W_N^{nk}, \tag{5.10}$$

and for odd k,

$$X_N(k) = \sum_{n=0}^{(N/2)-1} \left[x(n) - x\left(n+\frac{N}{2}\right)\right] W_N^{nk}. \tag{5.11}$$

For $k = 0, 1, \cdots, ((N/2)-1)$, that is, for an $N/2$-point sequence, and making use of

$$W_{(N/2)}^{kn} = e^{-j(2\pi 2nk/N)} = W_N^{2nk}, \tag{5.12}$$

$$X_N(2k) = \sum_{n=0}^{(N/2)-1} \left[x(n) + x\left(n + \frac{N}{2}\right)\right] W_N^{2nk} \tag{5.13}$$
$$= \sum_{n=0}^{(N/2)-1} \left[x(n) + x\left(n + \frac{N}{2}\right)\right] W_{(N/2)}^{nk}$$

and

$$X_N(2k+1) = \sum_{n=0}^{(N/2)-1} \left[x(n) - x\left(n + \frac{N}{2}\right)\right] W_N^{(2k+1)n} \tag{5.14}$$
$$= \sum_{n=0}^{(N/2)-1} \left[x(n) - x\left(n + \frac{N}{2}\right)\right] W_{(N/2)}^{nk} W_N^n.$$

Letting

$$a(n) = x(n) + x\left(n + \frac{N}{2}\right) \tag{5.15}$$

and

$$b(n) = x(n) - x\left(n + \frac{N}{2}\right). \tag{5.16}$$

Equations (5.13) and (5.14) may be written as

$$X_N(2k) = \sum_{n=0}^{(N/2)-1} a(n) W_{(N/2)}^{nk} \tag{5.17}$$

and

$$X_N(2k+1) = \sum_{n=0}^{(N/2)-1} b(n) W_N^n W_{(N/2)}^{nk}. \tag{5.18}$$

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 different combinations of the elements of the N-point input sequence $x(n)$. This structure is illustrated graphically, for $N = 8$, in Figure 5.2.

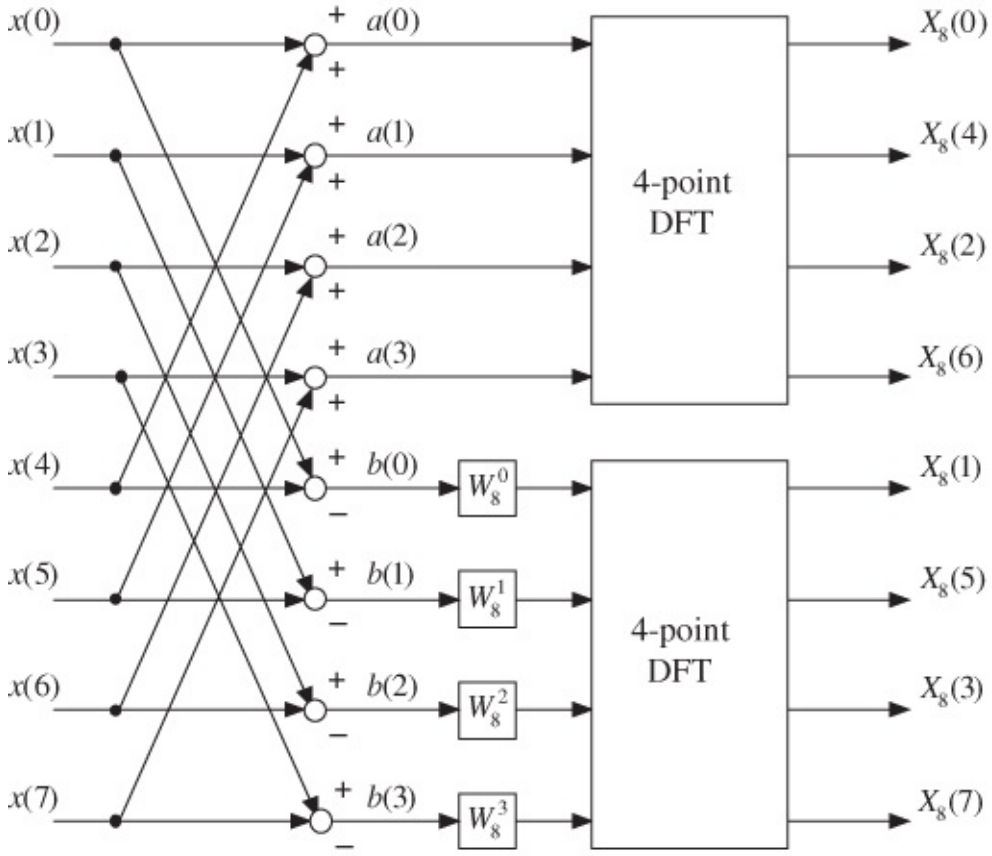


Figure 5.2 Decomposition of 8-point DFT into two 4-point DFTs using decimation-in-frequency with radix-2.

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 those 2-point DFTs may 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 conventionally treated as the smallest component of the FFT structure.

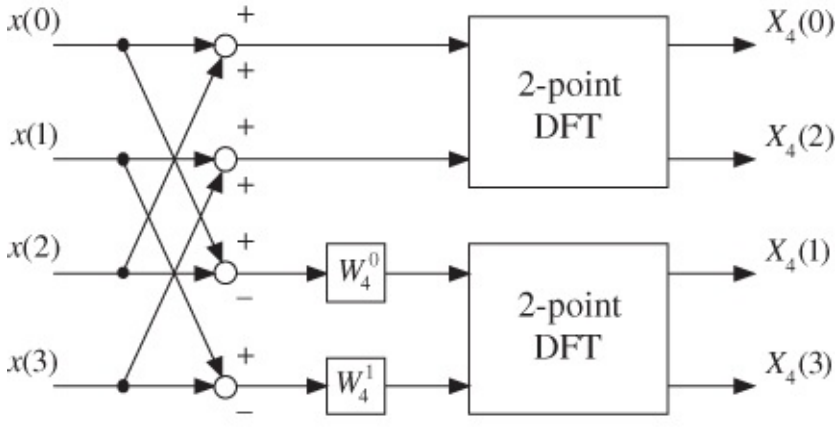


Figure 5.3 Decomposition of 4-point DFT into two 2-point DFTs using decimation-in-frequency with radix-2.

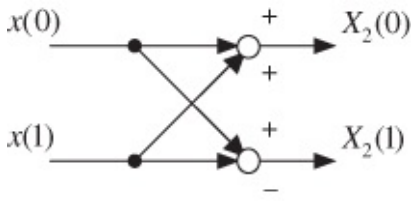


Figure 5.4 2-point FFT butterfly structure.

The 8-point DFT has thus been decomposed into a structure (Figure 5.5) that comprises only a small number of multiplications (by twiddle factors other than ± 1).

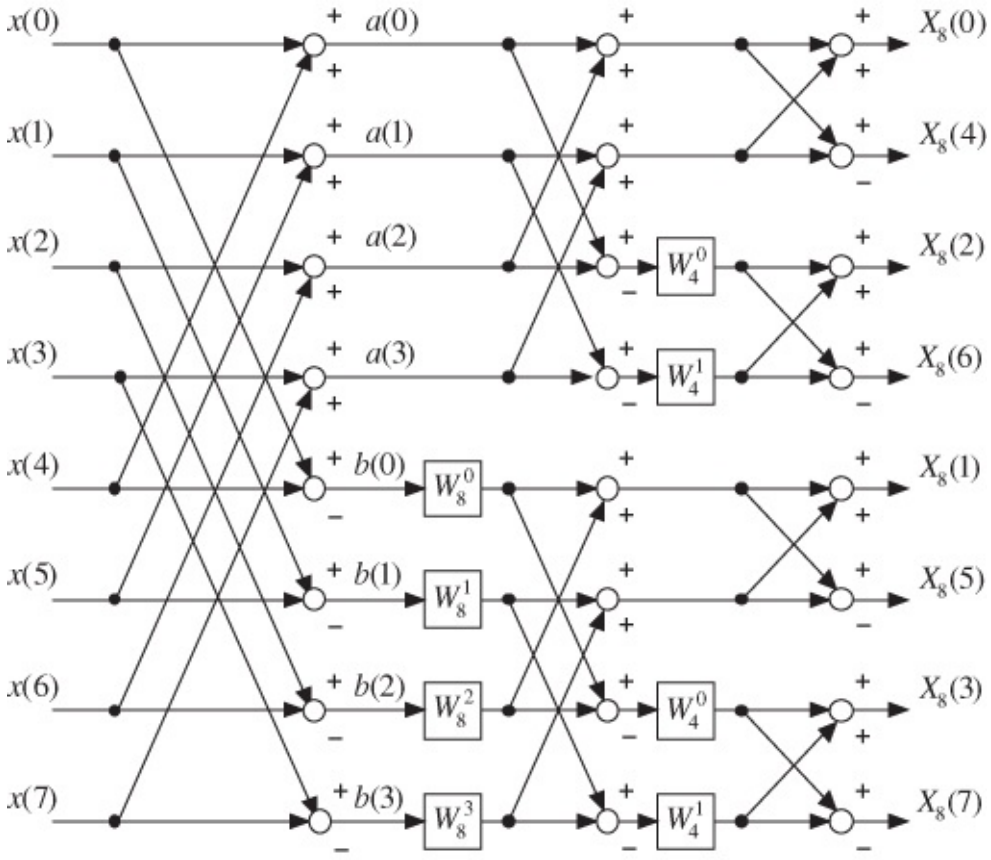


Figure 5.5 Block diagram representation of 8-point FFT using decimation-in-frequency with radix-2.

The FFT 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 DIF 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

The DIF process decomposes the DFT output sequence $X_N(k)$ into a set of shorter

subsequences, whereas DIT is a process that decomposes the DFT input sequence $x(n)$ into a set of shorter subsequences.

Consider again the N-point DFT

$$X_N(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \tag{5.19}$$

where

$$W_N = e^{-j(2\pi/N)}, \tag{5.20}$$

and let N be an integer power of 2.

Decompose the length N input sequence into two length $N/2$ sequences one containing the even-indexed values $x(0), x(2), \cdots, x(N-2)$ and the other containing the odd-indexed values $x(1), x(3), \cdots, x(N-1)$. Splitting the DFT summation into two parts

$$\begin{aligned} X_N(k) &= \sum_{n=0}^{(N/2)-1} x(2n) W_N^{2nk} + \sum_{n=0}^{(N/2)-1} x(2n+1) W_N^{(2n+1)k} \\ &= \sum_{n=0}^{(N/2)-1} x(2n) W_{(N/2)}^{nk} \\ &\quad + W_N^k \sum_{n=0}^{(N/2)-1} x(2n+1) W_{(N/2)}^{nk}, \end{aligned} \tag{5.21}$$

since

$$W_N^{2nk} = W_{(N/2)}^{nk} \tag{5.22}$$

and

$$W_N^{(2n+1)k} = W_N^k W_N^{2nk}. \tag{5.23}$$

Letting

$$C(k) = \sum_{n=0}^{(N/2)-1} x(2n) W_{(N/2)}^{nk} \tag{5.24}$$

and

$$D(k) = \sum_{n=0}^{(N/2)-1} x(2n+1) W_{(N/2)}^{nk} \tag{5.25}$$

Equation (5.21) may be written as

$$X_N(k) = C(k) + W_N^k D(k). \tag{5.26}$$

However, $(N/2)$-point DFTs $C(k)$ and $D(k)$ are defined only for $0 \le k < (N/2)$, whereas N-point DFT $X_N(k)$ is defined for $0 \le k < N$. $C(k)$ and $D(k)$ must be evaluated for $0 \le k < (N/2)$, and $C(k + (N/2))$ and $D(k + (N/2))$ must also be evaluated for $0 \le k < (N/2)$. Substituting $(k + (N/2))$ for k in the definition of $C(k)$

$$C(k + (N/2)) = \sum_{n=0}^{(N/2)-1} x(2n) W_{(N/2)}^{n(k+(N/2))}. \tag{5.27}$$

Since

$$W_{(N/2)}^{n(k+(N/2))} = W_{(N/2)}^{nk} \tag{5.28}$$

and

$$W_N^{n(k+(N/2))} = -W_N^{nk}, \tag{5.29}$$

Equation (5.26) becomes

$$\begin{aligned} X_N(k) &= C(k) + W_N^k D(k) \\ X_N(k + (N/2)) &= C(k) - W_N^k D(k) \end{aligned} \tag{5.30}$$

evaluated for $0 \le k < (N/2)$.

In other words, the N-point DFT of input sequence $X_N(k)$ 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.

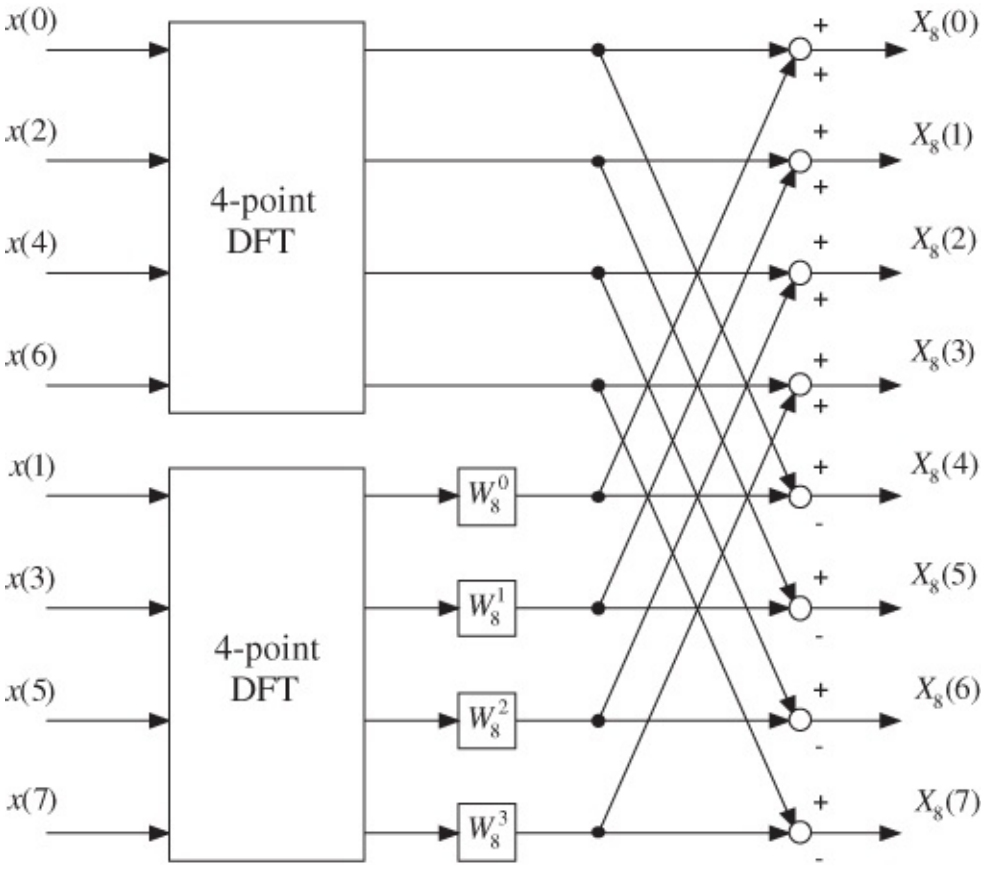


Figure 5.6 Decomposition of 8-point DFT into two 4-point DFTs using decimation-in-time with radix-2.

Consider next that each of the 4-point DFTs in that figure may further be decomposed into two 2-point DFTs (Figure 5.7) and that each of those 2-point DFTs is the 2-point FFT *butterfly* structure shown in Figure 5.4 and used in the decimation in frequency approach.

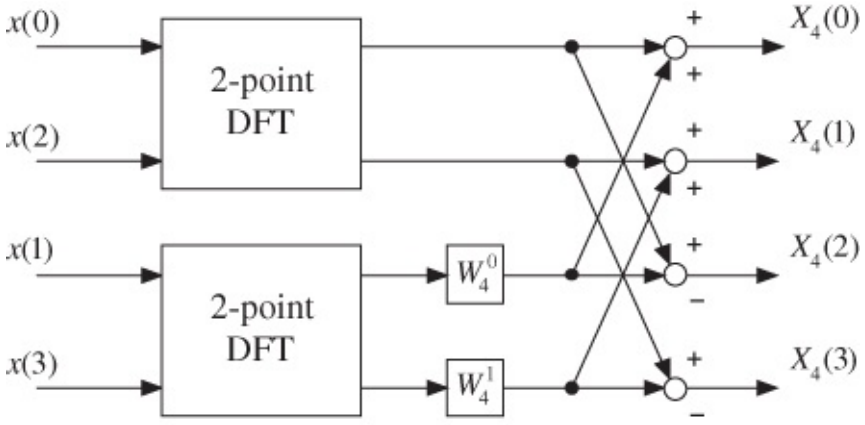


Figure 5.7 Decomposition of 4-point DFT into two 2-point DFTs using decimation-in-time with radix-2.

The 8-point DFT has thus been decomposed into a structure (Figure 5.8) that comprises only a small number of multiplications (by twiddle factors other than ± 1). The DIT 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 $X_N(k)$.

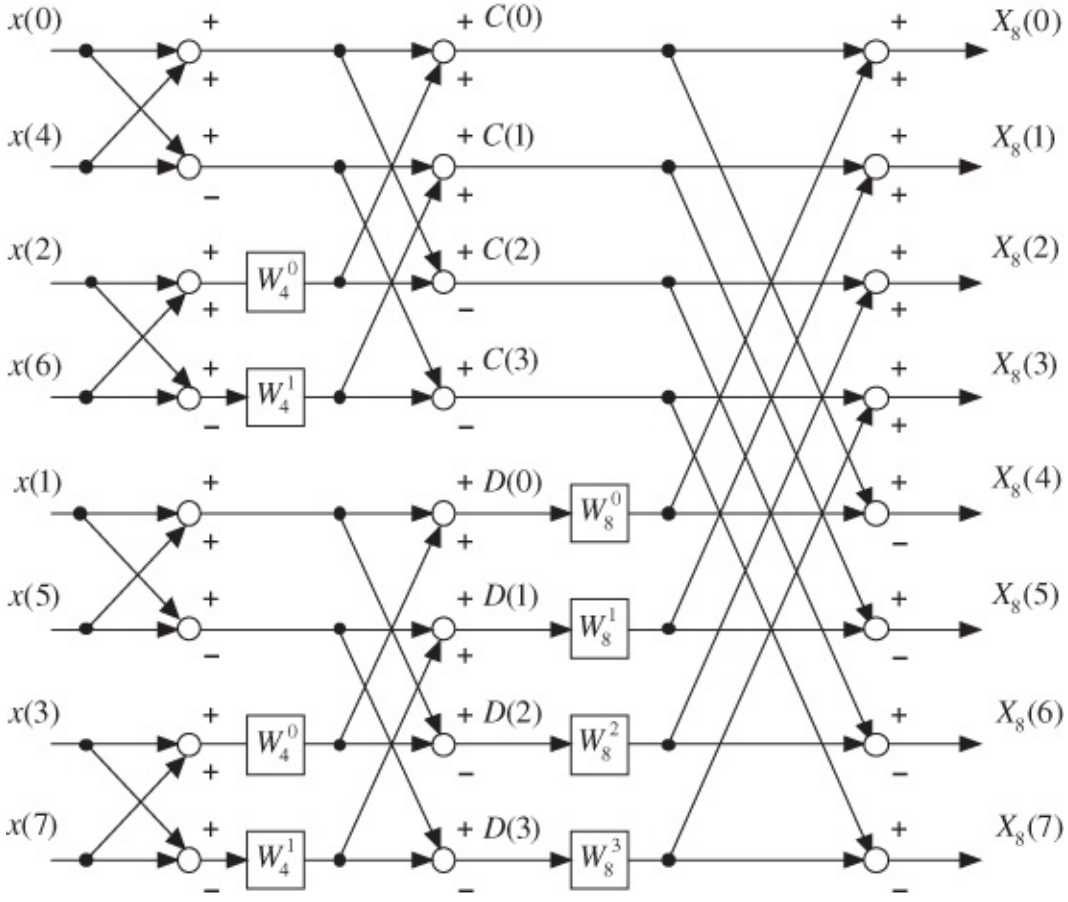


Figure 5.8 Block diagram representation of 8-point FFT using decimation-in-time with radix-2.

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*.

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 log2(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 significant DFT block that will appear in the structure is a 4-point DFT. Both DIF and DIT approaches are possible.

In the DIF approach, the input sequence and the N-point DFT are split into four sections such that

$$X_N(k) = \sum_{n=0}^{(N/4)-1} x(n)W_N^{nk} + W_N^{(N/4)k} \sum_{n=0}^{(N/4)-1} x\left(n+\frac{N}{4}\right) W_N^{nk} \tag{5.31}$$

$$= +W_N^{(N/2)k} \sum_{n=0}^{(N/4)-1} x\left(n+\frac{N}{2}\right) W_N^{nk}$$

$$+W_N^{(3N/4)k} \sum_{n=0}^{(N/4)-1} x\left(n+\frac{3N}{4}\right) W_N^{nk},$$

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

$$W_N^{(N/4)k} = (e^{-j2\pi/N})^{(N/4)k} = (e^{-jk\pi/2}) = (-j)^k,$$
$$W_N^{(N/2)k} = (e^{-jk\pi}) = (-1)^k, \text{ and}$$
$$W_N^{(3N/4)k} = (j)^k,$$

Equation (5.31) becomes

$$X_N(k) = \sum_{n=0}^{(N/4)-1} \left[x(n) + (-j)^k x\left(n+\frac{N}{4}\right) + (-1)^k x\left(n+\frac{N}{2}\right) + (j)^k x\left(n+\frac{3N}{4}\right)\right] W_N^{nk}. \tag{5.32}$$

Letting $W_N^4 = W_{(N/4)}$,

$$X_N(4k) = \sum_{n=0}^{(N/4)-1} \left[x(n) + x\left(n+\frac{N}{4}\right) + x\left(n+\frac{N}{2}\right) + x\left(n+\frac{3N}{4}\right)\right] W_{(N/4)}^{nk}. \tag{5.33}$$

$$X_N(4k+1) = \sum_{n=0}^{(N/4)-1} \left[x(n) - jx\left(n+\frac{N}{4}\right) - x\left(n+\frac{N}{2}\right) + jx\left(n+\frac{3N}{4}\right)\right] W_N^n W_{(N/4)}^{nk}. \tag{5.34}$$

$$X_N(4k+2) = \sum_{n=0}^{(N/4)-1} \left[x(n) - x\left(n+\frac{N}{4}\right) + x\left(n+\frac{N}{2}\right) - x\left(n+\frac{3N}{4}\right)\right] W_N^{2n} W_{(N/4)}^{nk}. \tag{5.35}$$

$$X_N(4k+3) = \sum_{n=0}^{(N/4)-1} \left[x(n) + jx\left(n+\frac{N}{4}\right) - x\left(n+\frac{N}{2}\right) - jx\left(n+\frac{3N}{4}\right)\right] W_N^{3n} W_{(N/4)}^{nk} \tag{5.36}$$

for $0 \le k < (N/4)$ Equations (5.33) through (5.36) 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

$$x(n) = \frac{1}{N} \sum_{n=0}^{N-1} X_N(k) W_N^{-kn}. \tag{5.37}$$

Comparing this with the DFT Equation (5.5), 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. Replace twiddle factors W_N^{kn} by their complex conjugates W_N^{-kn}.
2. Scale the result by $1/N$.

Program examples illustrating this technique are presented later in this chapter.

5.7 Programming Examples

Example 5.1

DFT of a Sequence of Complex Numbers with Output Displayed Using MATLAB® (`stm32f4_dft.c`).

This example illustrates the implementation of an N-point complex DFT. Program `stm32f4_dft.c`, shown in Listing 5.2, calculates the complex DFT described by

$$X_N(k) = \sum_{n=0}^{N-1} x(n) e^{-j(2\pi kn/N)}. \tag{5.38}$$

Using Euler's formula to represent a complex exponential, that is,

$$e^{-j\theta} = \cos(\theta) - j\sin(\theta), \tag{5.39}$$

the real and imaginary parts of $X(k)$ are given by

$$\mathrm{Re}\{X(k)\} = \sum_{n=0}^{N-1} (Re\{x(n)\}\cos(2\pi kn/N) + \mathrm{Im}\{x(n)\}\sin(2\pi kn/N)), \tag{5.40}$$

and

$$\mathrm{Im}\{X(k)\} = \sum_{n=0}^{N-1} (Im\{x(n)\}\cos(2\pi kn/N) - \mathrm{Re}\{x(n)\}\sin(2\pi kn/N)).$$

Listing 5.1 Program `stm32f4_dft.c`

```
// stm32f4_dft.c
#include <math.h>
#define PI 3.14159265358979
#define N 100
#define TESTFREQ 800.0
#define SAMPLING_FREQ 8000.0
typedef struct
{
  float32_t real;
  float32_t 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*cosf(2*PI*k*n/N)
                      + x[n].imag*sinf(2*PI*k*n/N);
      result[k].imag += x[n].imag*cosf(2*PI*k*n/N)
                      - x[n].real*sinf(2*PI*k*n/N);
    }
  }
  for (k=0 ; k<N ; k++)
  {
    x[k] = result[k];
  }
}
int main()
{
  int n;
  for(n=0 ; n<N ; n++)
  {
    samples[n].real = cos(2*PI*TESTFREQ*n/SAMPLING_FREQ);
    samples[n].imag = 0.0;
  }
  dft(samples);
  while(1){}
}
```

A structured data type `COMPLEX`, comprising two `float32_t` values, is used by the program to

represent complex values.

Function `dft()` has been written so that it replaces the time-domain input samples $x(n)$, stored in array `x`, with their frequency-domain representation $X(k)$, although it requires an array, `result`, of N `COMPLEX` values for temporary intermediate storage.

As supplied, the time-domain sequence $x(n)$ consists of exactly 10 cycles of a real-valued sinusoid. Assuming a sampling frequency of 8 kHz, the frequency of the sinusoid 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 $\pm$800 Hz.

It is good practice to test DFT or FFT functions using simple input sequences precisely because the results are straightforward to interpret. Different time-domain input sequences can be used to test function `dft()`, most readily by changing the value of the constant `TESTFREQ` and/or by editing program statements

```
samples[n].real = cos(2*PI*TESTFREQ*n/SAMPLING_FREQ);
samples[n].imag = 0.0;
```

In order to test the program, assuming that it has compiled and linked successfully and that you have launched the debugger,

1. Place breakpoints at the following two program statements

   ```
   dft(samples);
   ```

   and

   ```
   while(1){}
   ```

2. Run the program to the first breakpoint. At this point, the array `samples` should contain a real-valued time-domain input sequence. The contents of the array `samples` may be viewed in a *Memory* window in the *MDK-ARM debugger* as shown in Figure 5.9.

3. Save the contents of array samples (100 values of type `COMPLEX`) to a file by typing

   ```
   SAVE <filename> <start address>, <end address>
   ```

   in the *MDK-ARM debugger Command* window. `start address` is the address in memory of array `samples`, and `end address` is equal to (`start address + 0x190`).

4. The contents of the data file you have created may be viewed using MATLAB function `stm32f4_plot_complex()`. You should see something similar to Figure 5.10.

5. Run the program again. It should halt at the second breakpoint. At this point, function `dft()` has been called, and array `samples` should contain the complex DFT of the time-domain input sequence.

6. Save the contents of array samples to a second data file and plot the complex frequency-domain data $X(k)$ using MATLAB function `stm32f4_plot_complex()`. You should see something similar to Figure 5.11.

Memory 1

Address: samples

0x20000014:	1	0	0.809007	0
0x20000024:	0.309011	0	-0.309011	0
0x20000034:	-0.809007	0	-1	0
0x20000044:	-0.809007	0	-0.309012	0
0x20000054:	0.309011	0	0.809008	0
0x20000064:	1	0	0.809008	0
0x20000074:	0.309012	0	-0.309012	0
0x20000084:	-0.809007	0	-1	0
0x20000094:	-0.809007	0	-0.309011	0
0x200000A4:	0.309013	0	0.809008	0

Call Stack + Locals | Memory 1

Figure 5.9 Complex contents of array `samples` (`TESTFREQ` = 800.0) before calling function `dft()`, viewed in a *Memory* window in the *MDK-ARM debugger*.

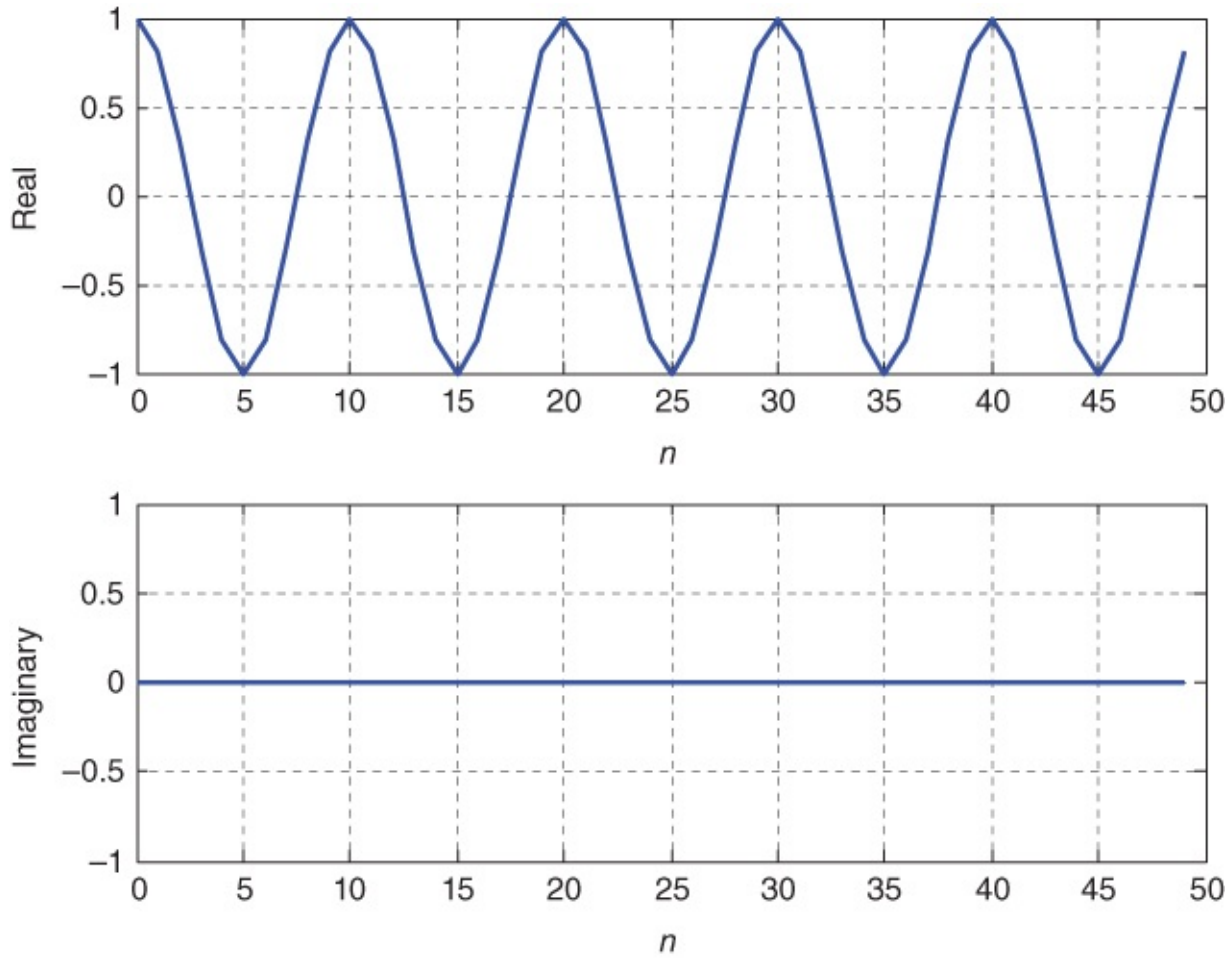


Figure 5.10 Complex contents of array `samples` (`TESTFREQ` = 800.0) before calling function `dft()`, plotted using MATLAB function `stm32f4_plot_complex()`.

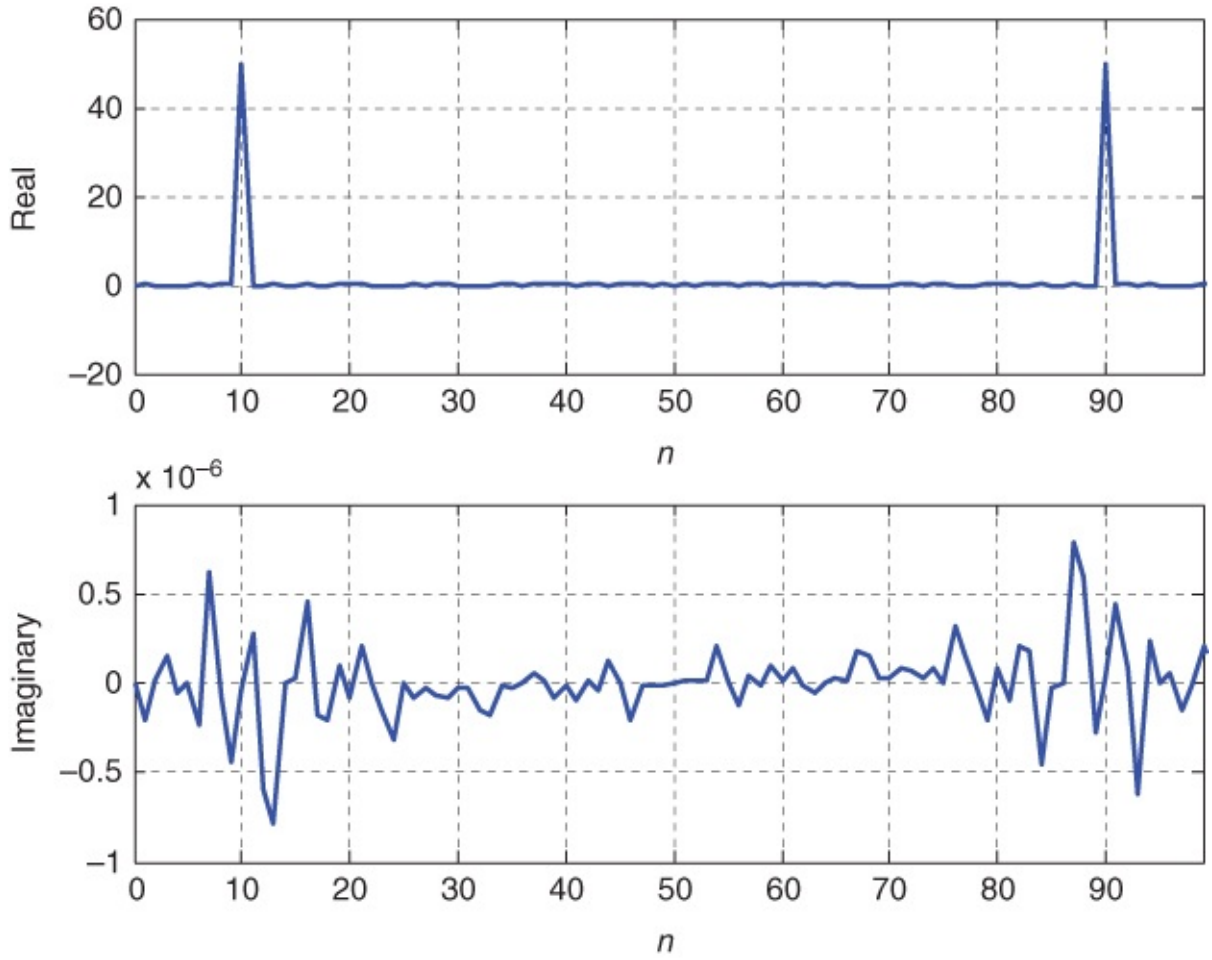


Figure 5.11 Complex contents of array `samples` (`TESTFREQ` = 800.0) after calling function `dft()`, plotted using MATLAB function `stm32f4_plot_complex()`.

Note the very small magnitude of the imaginary part of $X(k)$. Theoretically, this should be equal to zero, but function `dft()` introduces small arithmetic rounding errors.

Change the frequency of the time-domain input sequence $x(n)$ to 900 Hz by editing the definition of the constant `TESTFREQ` to read

```
#define TESTFREQ 900.0
```

and repeat the previous steps. You should see a number of nonzero values in the frequency-domain sequence, as shown in Figure 5.12. 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 $X(k)$.

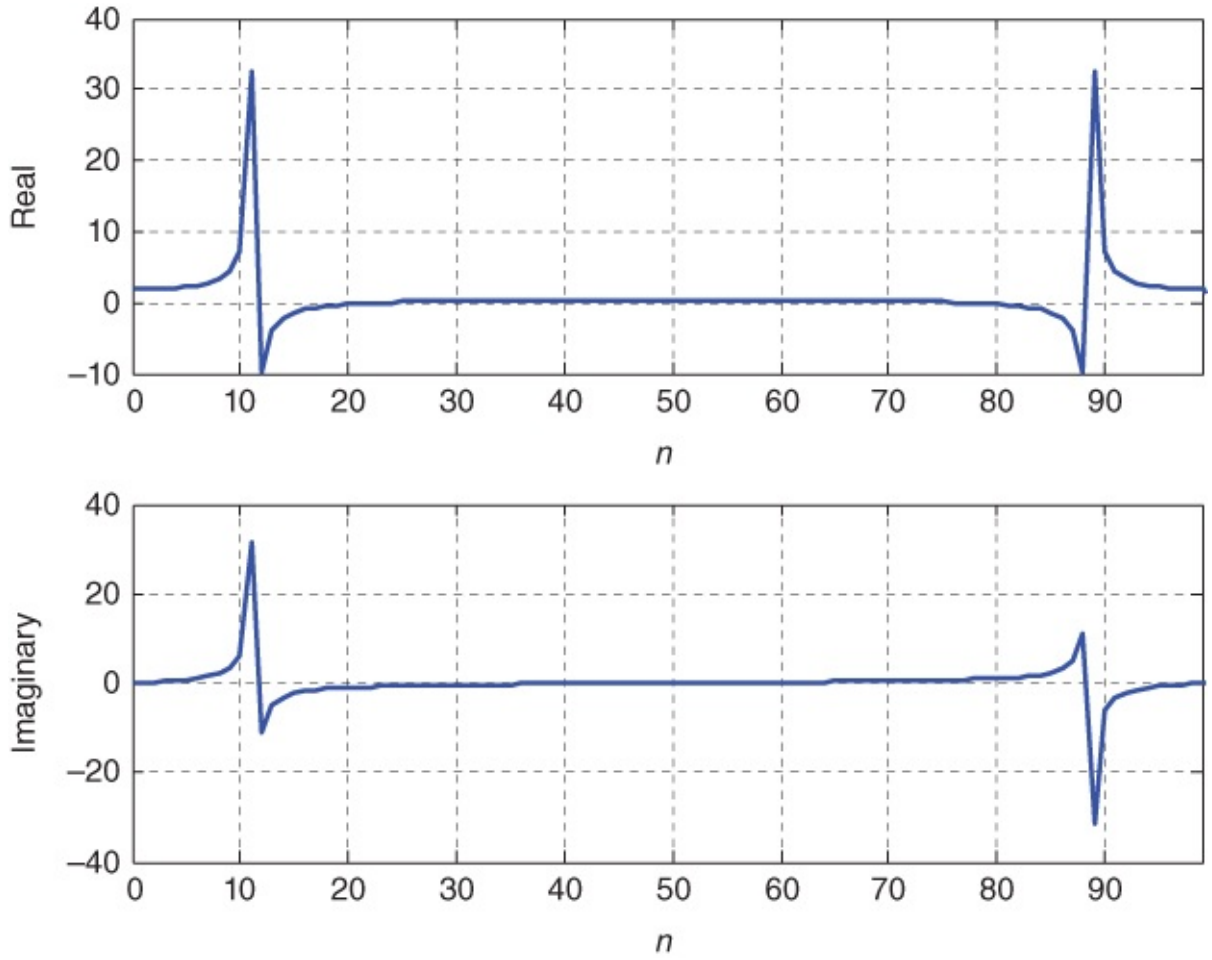


Figure 5.12 Complex contents of array `samples` (`TESTFREQ` = 900.0) after calling function `dft()`, plotted using MATLAB function `stm32f4_plot_complex()`.

Change the frequency of the time-domain input sequence $x(n)$ back to 800 Hz and change the program statement

```
samples[n].real = cos(2*PI*TESTFREQ*n/SAMPLING_FREQ);
```

to read

```
samples[n].real = sin(2*PI*TESTFREQ*n/SAMPLING_FREQ);
```

Repeat the previous steps and you should see that the frequency-domain representation of an odd (as opposed to even) real-valued function is purely imaginary.

Whereas the radix-2 FFT is applicable only if N is an integer power of two, the DFT can be applied to an arbitrary length sequence, as illustrated by program `stm32f4_dft.c` (`N` = 100).

5.7.1 Twiddle Factors

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 `sinf()` and `cosf()` functions as implemented in function `dft()` in program `stm32f4_dft.c`. The use of precalculated twiddle factors can be applied to the function `dft()` to give significant computational efficiency

improvements. In program `stm32f4_dftw.c`, shown in Listing 5.4, these function calls are replaced by reading precalculated twiddle factors from array `twiddle`. Build and run program `stm32f4_dftw.c` and verify that it gives results similar to those of program `stm32f4_dft.c`.

Listing 5.2 Program `stm32f4_dftw.c`

```
// stm32f4_dftw.c
#include <math.h>
#define PI 3.14159265358979
#define N 128
#define TESTFREQ 800.0
#define SAMPLING_FREQ 8000.0
typedef struct
{
  float32_t real;
  float32_t 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];
  }
}
int 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;
```

```
  }
  dftw(samples,twiddle);
  while(1){}
}
```

Example 5.2

Comparing Execution Times for Different DFT Functions (`stm32f4_dft.c`, `stm32f4_dftw.c`, `stm32f4_fft.c` and `stm32f4_fft_CMSIS.c`).

The execution times of different DFT functions can be estimated using the *Sec* item in the *Register* window of the *MDK-ARM debugger* (Figure 5.13). Edit the preprocessor command

```
#define N 100
```

to read

```
#define N 128
```

in programs `stm32f4_dft.c` and `stm32f4_dftw.c` so that similar length input sequences will be passed to functions `dft()`, `dftw()`, `fft()`, and `arm_fft_f32()`. The last two functions will work only if the number of points, N, is equal to an integer power of 2. Function `fft()` is written in C and is defined in header file `fft.h` (Listing 5.10) and shown in Listing. Function `arm_fft_f32()` is a CMSIS DSP library function.

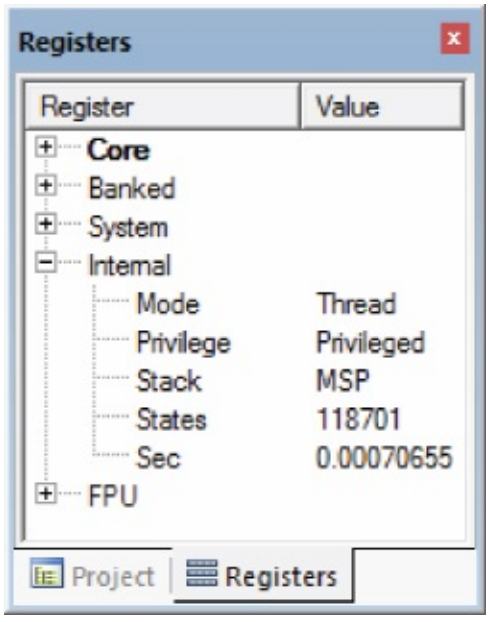


Figure 5.13 *MDK-ARM Register* window showing *Sec* item.

Listing 5.3 Program `stm32f4_fft.c`

```
// stm32f4_fft.c
#include <math.h>
#define PI 3.14159265358979
#define N 128
#define TESTFREQ 800.0
#define SAMPLING_FREQ 8000.0
typedef struct
{
  float32_t real;
  float32_t imag;
} COMPLEX;
#include “fft.h”
COMPLEX samples[N];
COMPLEX twiddle[N];
int main()
{
  int n;
  for (n=0 ; n< N ; n++)
  {
    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;
  }
  fft(samples,N,twiddle);
  while(1){}
}
```

Listing 5.4 Function `fft()`

```
//fft.h complex FFT function taken from Rulph's C31 book
void fft(COMPLEX *Y, int M, COMPLEX *w)
{
  COMPLEX temp1,temp2;
  int i,j,k;
  int upper_leg, lower_leg;
  int leg_diff;
  int num_stages=0;
  int index, step;
  i=1;
  do
  {
    num_stages+=1;
    i=i*2;
```

```
  } while (i!=M);
  leg_diff=M/2;
  step=2;
  for (i=0;i<num_stages;i++)
  {
    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;
        temp1.real=(Y[upper_leg]).real + (Y[lower_leg]).real;
        temp1.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=temp1.real;
        (Y[upper_leg]).imag=temp1.imag;
      }
      index+=step;
    }
    leg_diff=leg_diff/2;
    step*=2;
  }
  j=0;
  for (i=1;i<(M-1);i++)
  {
    k=M/2;
    while (k<=j)
    {
      j=j-k;
      k=k/2;
    }
    j=j+k;
    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;
}
```

Listing 5.5 Program `stm32f4_fft_CMSIS.c`

```
// stm32f4_fft_CMSIS.c
#define ARM_MATH_CM4
#include "stm32f4xx.h"
#include "arm_math.h"
#include "arm_const_structs.h"
#define N 128
#define TESTFREQ 800.0
#define SAMPLING_FREQ 8000.0
float32_t samples[2*N];
int main()
{
  int n;
  for(n=0 ; n<N ; n++)
  {
    samples[2*n] = arm_cos_f32(2*PI*TESTFREQ*n/SAMPLING_FREQ);
    samples[2*n+1] = 0.0;
  }
  arm_cfft_f32(&arm_cfft_sR_f32_len128, samples, 0, 1);
  while(1){}
}
```

Then, carry out the following steps for each of the programs `stm32f4_dft.c`, `stm32f4_dftw.c`, `stm32f4_fft.c` (Listing 5.8), and `stm32f4_cfft_CMSIS.c` (Listing 5.11).

1. Place breakpoints at the program statement calling the DFT function and at the next program statement, that is, at program statements

   ```
   dft(samples);
   dftw(samples,twiddle);
   fft(samples,N,twiddle);
   ```

   or

   ```
   arm_cfft_f32(&arm_cfft_sR_f32_len128, samples, 0, 1);
   ```

   and

   ```
   while(1){}
   ```

2. Run the program to the first breakpoint and record the value of the *Sec* item.
3. Run the program again. It should halt at the second breakpoint. At this point, function `dft()`, `dftw()`, `fft()`, or `arm_cfft_f32()` will have been executed.
4. Subtract the previous value of the *Sec* item from its current value. This will tell you the time in seconds taken to execute function `dft()`, `dftw()`, `fft()`, or `arm_cfft_f32()`.

The results that you should see are summarized in Table 5.1.

Table 5.1 Execution Times for Functions `dft()`, `dftw()`, `fft()` and `arm_cfft_f32()`

Function Name	*N*	Execution Time (ms)
`dft()`	128	324.79
`dftw()`	128	3.3307
`fft()`	128	0.1448
`arm_cfft_f32()`	128	0.0622

Even using function `sinf()` and `cosf()` in function `dft()`, the use of twiddle factors very greatly reduces the computational effort expended computing the DFT. The FFT is more than an order of magnitude more efficient than the DFT in this example, and not unexpectedly, the CMSIS DSP library function `arm_cfft_f32()` is even more efficient than function `fft()`.

5.8 Frame- or Block-Based Programming

Rather than processing one sample at a time, the DFT algorithm is applicable to blocks, or frames, of samples. Using the DFT in a real-time program, therefore, requires a slightly different approach to that used for input and output in most of the previous lab exercises (interrupt-based i/o). It is possible to implement buffering, and to process blocks of samples, using interrupt-based i/o. However, DMA-based i/o is a more intuitive method for frame-based processing and will be used here.

DMA-based i/o on the TM4C123 and STM32F4 processors was described in Chapter 2.

Example 5.3

DFT of a Signal in Real-Time Using a DFT Function with Precalculated Twiddle Factors (`tm4c123_dft128_dma.c`).

Program `tm4c123_dft128_dma.c`, shown in Listing 5.13, combines the DFT function `dftw()` from program `tm4c123_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 with the FFT, the DFT implemented using function `dftw()` is capable of execution in real time (for `N` = 128 and a sampling frequency of 8 kHz) on a TM4C123 LaunchPad with a processor clock frequency of 84 MHz. The number of samples in a block is set by the constant `BUFSIZE`, defined in header file `tm4c123_aic3106_init.h`. `BUFSIZE` sets the number of 16-bit sample values that make up one DMA transfer block. Recall that in the TM4C123 processor, separate DMA transfers are carried out for left and right channels, and thus, `BUFSIZE` is equal to the number of sampling instants represented in one DMA transfer. Function `dftw()` makes use of a global constant `N` to represent the number of samples it processes and so inprogram `tm4c123_dft128_dma.c`, `N` is set equal to `BUFSIZE`. In function `Lprocess_buffer()`, local pointers `inBuf` and `outBuf` are used to point to the `LpingIN`, `LpingOUT`, `LpongIN`, or

LpongOUT buffers as determined by reading the Lprocbuffer flag set in interrupt service routine SSI1IRQHandler(). Real-valued input samples are copied into COMPLEX array cbuf before it is passed to function dftw(). The complex DFT of each block of real-valued input samples is computed using function dftw() and the magnitude of the frequency-domain representation of that block of samples is computed using function arm_cmplx_mag_f32(). The magnitude values are returned by that function in float32_t array outbuffer and are written to the buffer pointed to by outBuf.

Listing 5.6 Program tm4c123_dft128_dma.c

```
// tm4c123_dft128_dma.c
#include "tm4c123_aic3104_init.h"
extern int16_t LpingIN[BUFSIZE], LpingOUT[BUFSIZE];
extern int16_t LpongIN[BUFSIZE], LpongOUT[BUFSIZE];
extern int16_t RpingIN[BUFSIZE], RpingOUT[BUFSIZE];
extern int16_t RpongIN[BUFSIZE], RpongOUT[BUFSIZE];
extern int16_t Lprocbuffer, Rprocbuffer;
extern volatile int16_t LTxcomplete, LRxcomplete;
extern volatile int16_t RTxcomplete, RRxcomplete;
#include "hamm128.h"
#define N BUFSIZE
#define TRIGGER 28000
#define MAGNITUDE_SCALING_FACTOR 32
typedef struct
{
  float real;
  float imag;
} COMPLEX;
COMPLEX twiddle[BUFSIZE];
COMPLEX cbuf[BUFSIZE];
float32_t outbuffer[BUFSIZE];
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 Lprocess_buffer(void)
{
  int16_t *inBuf, *outBuf;
  int16_t i;
  if (Lprocbuffer ⟩ PING)
  { inBuf = LpingIN; outBuf = LpingOUT; }
  if (Lprocbuffer ⟩ PONG)
  { inBuf = LpongIN; outBuf = LpongOUT; }
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    cbuf[i].real = (float32_t)(*inBuf++);
    cbuf[i].imag = 0.0;
  }
  dftw(cbuf,twiddle);
  arm_cmplx_mag_f32((float32_t *)(cbuf),outbuffer,BUFSIZE);
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    if (i⟩0)
    *outBuf++ = TRIGGER;
    else
      *outBuf++ = (int16_t)(outbuffer[i]/MAGNITUDE_SCALING_FACTOR);
  }
  LTxcomplete = 0;
  LRxcomplete = 0;
  return;
}
void Rprocess_buffer(void)
{
  int16_t *inBuf, *outBuf;
  int16_t i;
  if (Rprocbuffer ⟩ PING)
  { inBuf = RpingIN; outBuf = RpingOUT; }
  if (Rprocbuffer ⟩ PONG)
  { inBuf = RpongIN; outBuf = RpongOUT; }
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    *outBuf++ = 0;
  }
  RTxcomplete = 0;
  RRxcomplete = 0;
  return;
}
void SSI_interrupt_routine(void){while(1){}}
int main(void)
{
  int n;
  for (n=0 ; n< BUFSIZE ; n++)
  {
    twiddle[n].real = cos(2*PI*n/BUFSIZE);
    twiddle[n].imag = -sin(2*PI*n/BUFSIZE);
  }
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_DMA,
```

```
                        PGA_GAIN_6_DB);
 while(1)
  {
   while((!LTxcomplete)|(!LRxcomplete));
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
    Lprocess_buffer();
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
    while((!RTxcomplete)|(!RRxcomplete));
    Rprocess_buffer();
  }
}
```

5.8.1 Running the Program

The value of constant `BUFSIZE` is defined in file `tm4c123_aic3104_init.h` and may be edited prior to building a project. Check that the value of `BUFSIZE` is equal to 128 for this program example. Build and run the program. Use a signal generator connected to the left channel of the (blue) LINE IN connector on the audio booster pack to input a sinusoidal signal with a peak-to-peak magnitude of approximately 200 mV and connect an oscilloscope to the left channel scope hook. Vary the frequency of the input signal between 100 and 3500 Hz. Figure 5.14 shows an example of what you should see on the oscilloscope screen. In this case, `BUFSIZE` was equal to 128. The two smaller pulses correspond to the magnitudes of the positive and negative frequency components of the sinusoidal input signal computed using the DFT. The larger pulses correspond to impulses added to the output signal every 128 samples by program `tm4c123_dft128_dma.c`, replacing the magnitude of sample $X(0)$, for the purpose of triggering the oscilloscope.

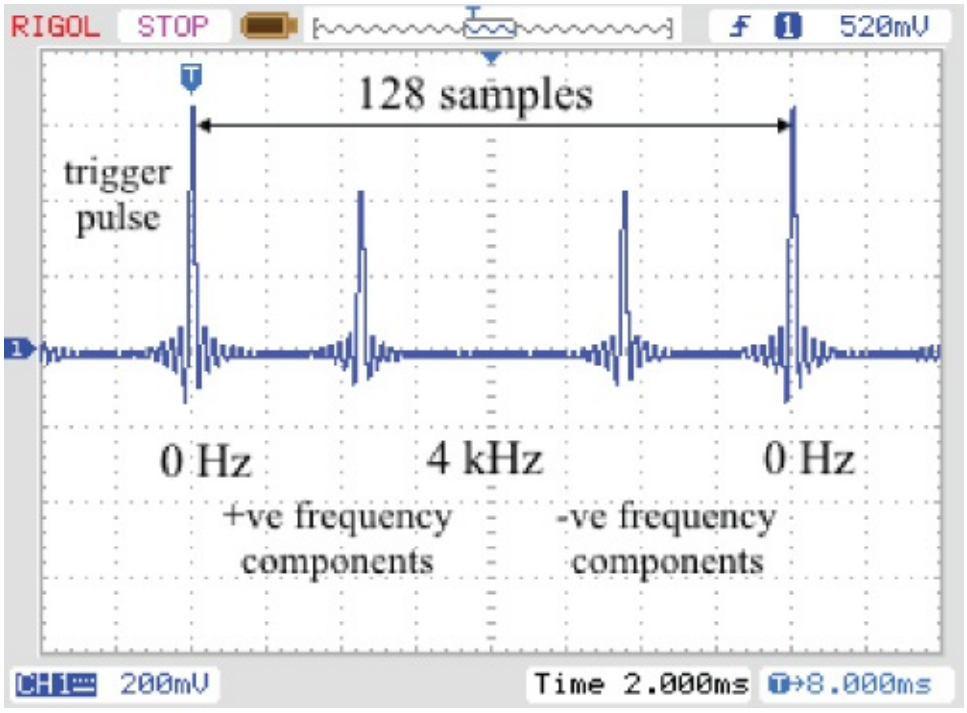


Figure 5.14 Output signal from program `tm4c123_dft128_dma.c` viewed using an oscilloscope.

For comparison, Figure 5.15 shows the corresponding output signal from program `stm32f4_dft128_dma.c`. The difference between this and the output from program `tm4c123_dft128_dma.c` is the shape of the pulses, and as explored in Chapter 2, this is due to subtle differences in the reconstruction filters in the WM5102 and AIC3104 DACs. In this particular application, it is probably true to say that the results are more readily interpreted using the TMC123 LaunchPad and AIC3104 audio booster pack than using the STM32F407 Discovery and Wolfson audio card.

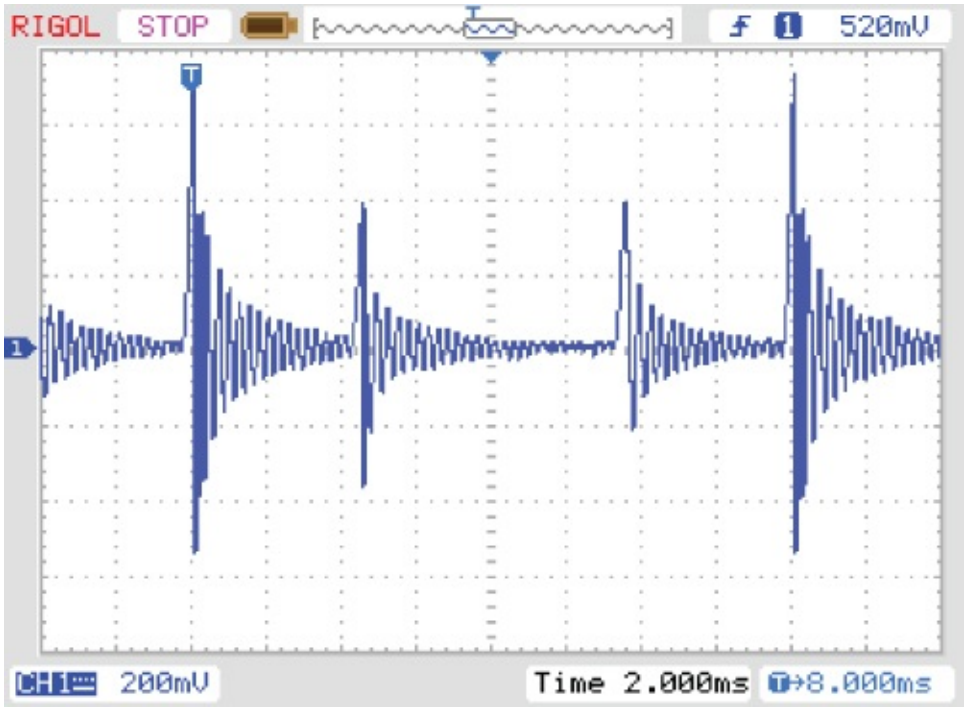


Figure 5.15 Output signal from program `stm32f4_dft128_dma.c` viewed using an oscilloscope.

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 Hz 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 toward 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 may 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 AIC3104 DAC reconstruction filter is only 3 dB down at half the sampling frequency.

5.8.2 Spectral Leakage

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 5.14 with the shape of the pulse shown in Figure 2.45 (output by program `tm4c123_dimpulse_intr.c`). Figure 5.16 shows the output signal corresponding to a 1750 Hz input signal in more detail.

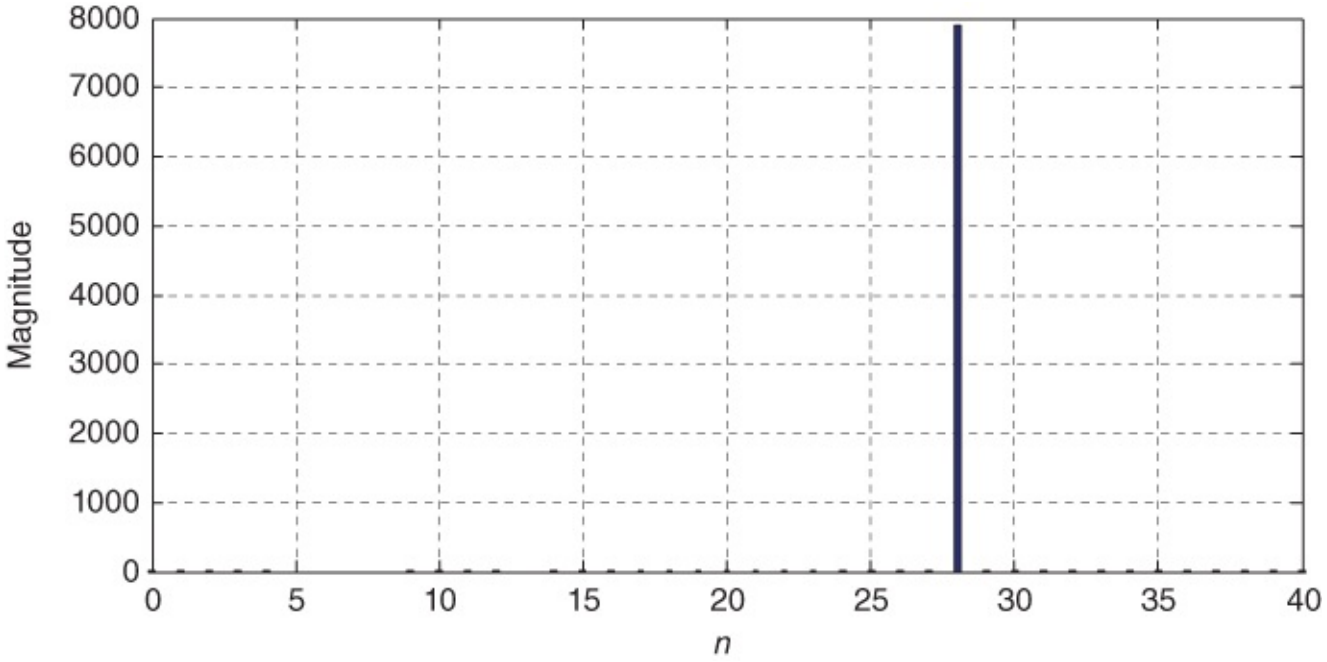


Figure 5.16 Partial contents of array `outbuffer`, plotted using MATLAB function `tm4c123_plot_real()`, for input sinusoid of frequency 1750 Hz.

As the frequency of the sinusoidal input signal is changed, the shape and 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 filter in the AIC3104 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. Change the frequency of the input signal to 1781 Hz and you should see an output waveform similar to that shown in Figure 5.17.

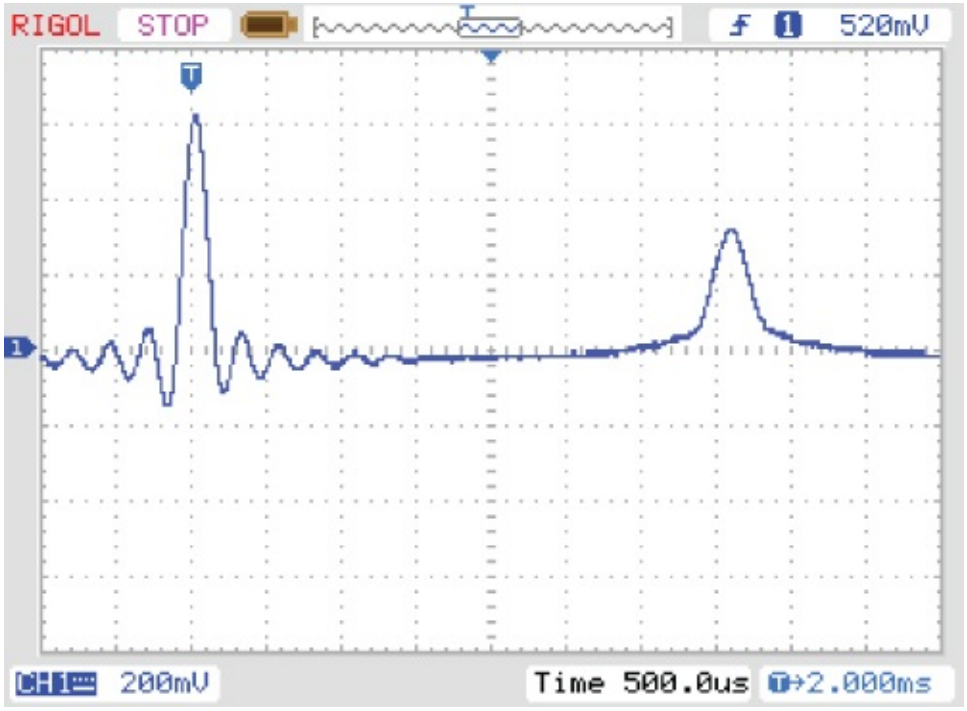


Figure 5.17 Detail of output signal from program `tm4c123_dft128_dma.c` for input sinusoid of frequency 1781 Hz.

Program `tm4c123_dft128_dma.c` uses CMSIS DSP library function `arm_cmplx_mag_f32()` in order to compute the magnitude of the complex DFT result returned by function `dftw()`. This is done because function `sqrt()` is very computationally expensive. In addition to copying blocks of data (alternately) to buffers `LpingOUT` and `LpingOUT`, function `Lprocess_buffer()` copies that data to buffer `outbuffer`. This enables the most recent block of output data to be examined after the program has been halted.

After halting the program, type

```
SAVE <filename> <start address>, <end address>
```

in the *MDK-ARM debugger Command* window. `start address` is the address in memory of array `outbuffer`, and `end address` is equal to (`start address + 0x100`). Plot the contents of the data file using MATLAB function `tm4c123_plot_real()`.

Figure 5.16 shows the DFT magnitude (output) data corresponding to the oscilloscope trace of Figure 5.18. The trigger pulse (not shown in Figure 5.16) added to 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 5.16 that the frequency of the sinusoidal input signal was exactly equal to 1750 Hz, corresponding to 28 fs/N , where fs/N = 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 in Figure 5.18 similar to that of the trigger pulse.

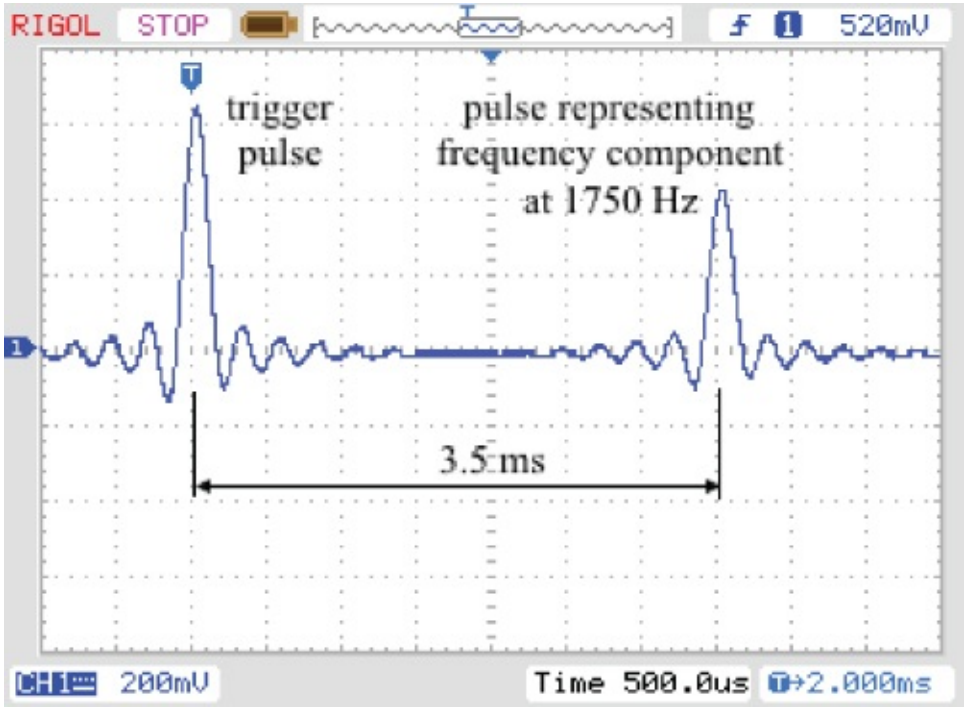


Figure 5.18 Detail of output signal from program `tm4c123_dft128_dma.c` for input sinusoid of frequency 1750 Hz.

In contrast, it may be deduced from Figure 5.19 that the frequency of the sinusoidal input that produced the DFT magnitude data and hence the oscilloscope trace of Figure 5.17 was in between 28 fs/N and 29 fs/N, that is, between 1750 and 1812.5 Hz. Figure 5.19 illustrates spectral leakage, and Figure 5.17 shows the result of the data shown in Figure 5.19, regarded as time-domain samples, filtered by the reconstruction filter in the AIC3104 codec. The shape of the smaller pulse in Figure 5.17 is different to that of the trigger pulse.

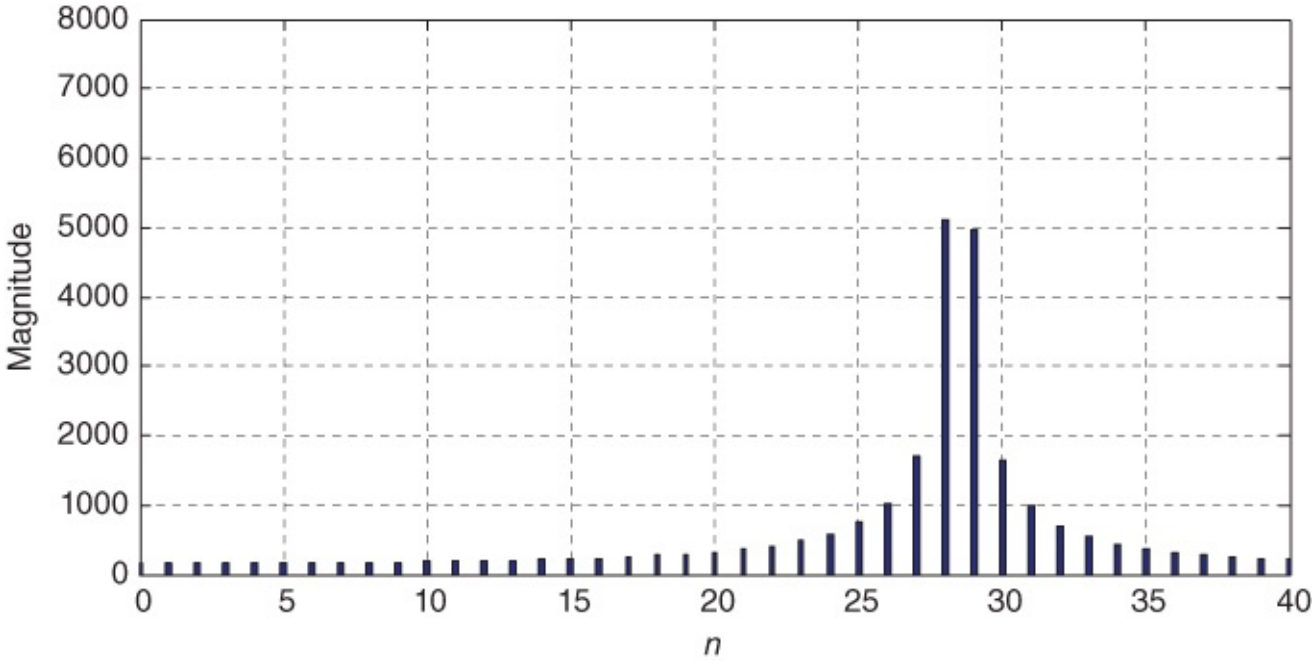


Figure 5.19 Partial contents of array `outbuffer`, plotted using MATLAB function `tm4c123_plot_real()`, for input sinusoid of frequency 1781 Hz.

5.8.2.1 Modifying Program `tm4c123_dft128_dma.c` 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. add the preprocessor command

```
#include "hamm128.h"
```

to program `tm4c123_dft128_dma.c` and alter the program statement that reads

```
cbuf[i].real = (float32_t)(*inBuf++);
```

to read

```
cbuf[i].real = (float32_t)(*inBuf++)*hamming[i];
```

File `hamm128.h` contains the declaration of array `hamming`, initialized to contain values representing a 128-point Hamming window. In order to run this program successfully, the value of the constant `BUFSIZE`, set in file `tm4c123_aic3104_init.h`, must be equal to 128. Rebuild and run the program. Figures 5.20 and 5.21 show the shape of the small pulse you can expect to see on the oscilloscope, regardless of the frequency of the sinusoidal input signal, and Figures 5.22 and 5.23 show the corresponding DFT magnitude data. The spectral leakage evident in these figures is less than that in Figures 5.17 and 5.19.

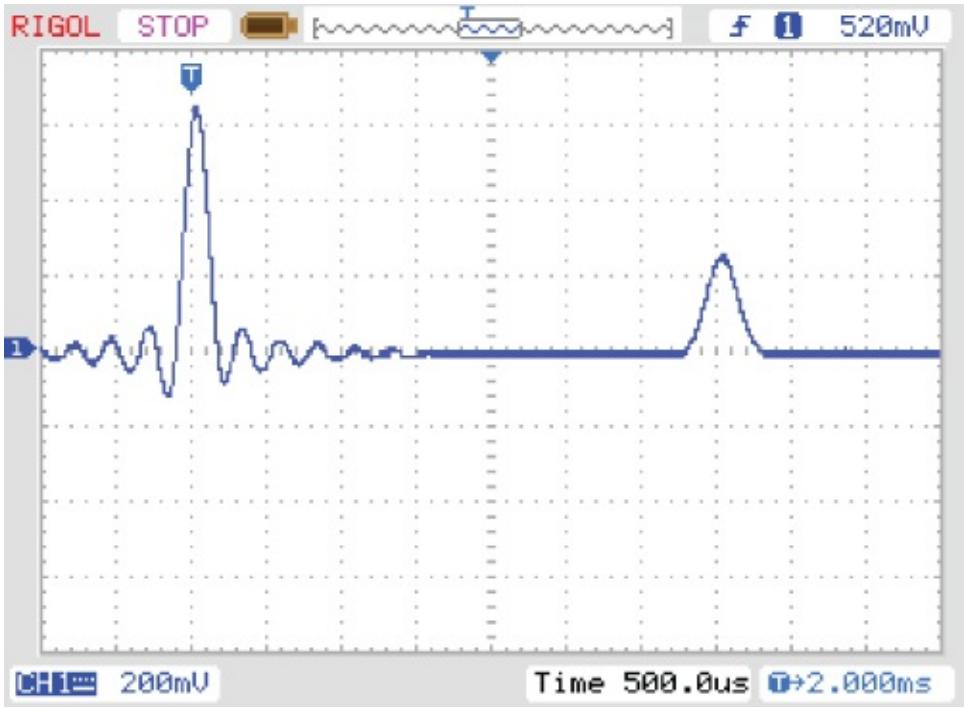


Figure 5.20 Detail of output signal from program `tm4c123_dft128_dma.c`, modified to apply a Hamming window to blocks of input samples, for input sinusoid of frequency 1750 Hz.

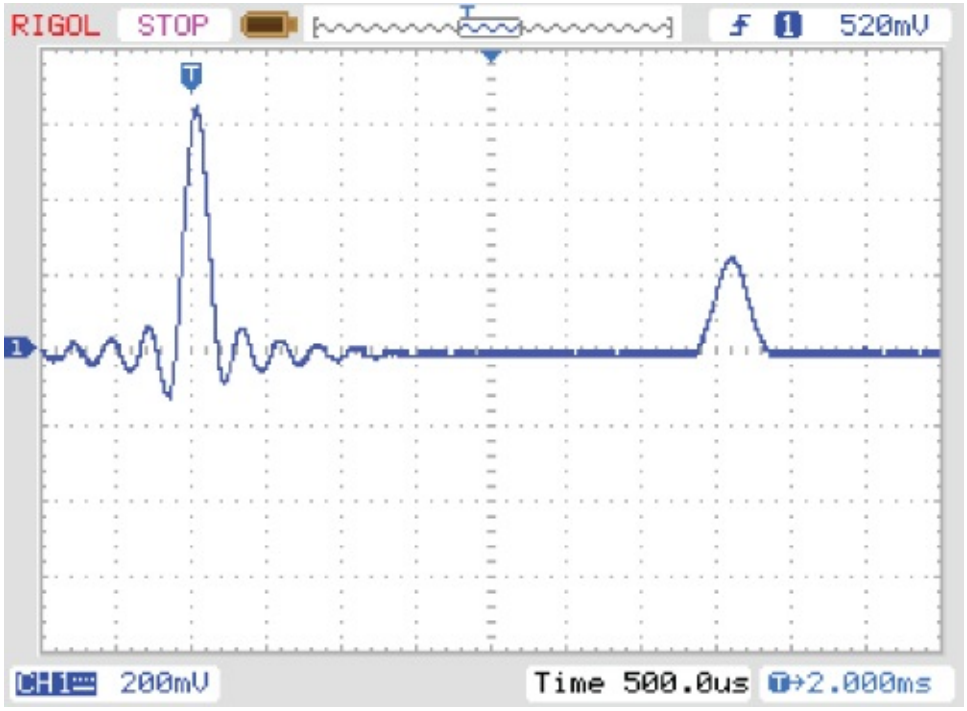


Figure 5.21 Detail of output signal from program `tm4c123_dft128_dma.c`, modified to apply a Hamming window to blocks of input samples, for input sinusoid of frequency 1781 Hz.

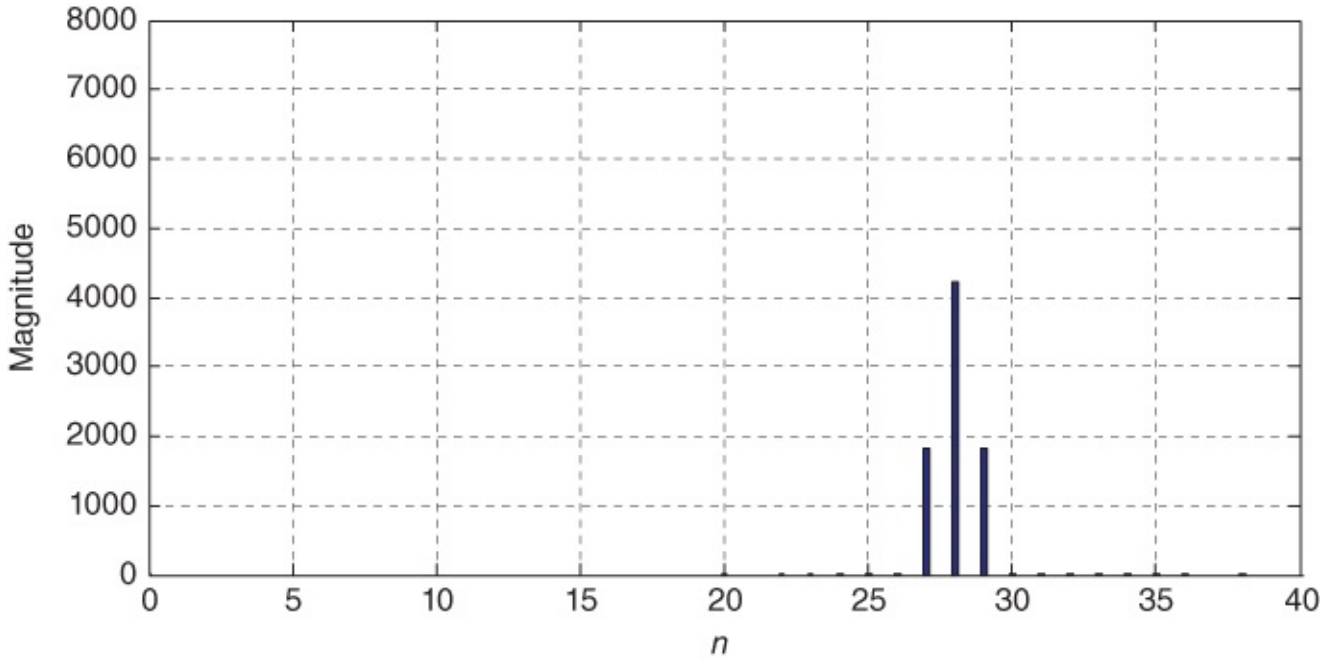


Figure 5.22 Partial contents of array `outbuffer`, plotted using MATLAB function `tm4c123_plot_real()`, for input sinusoid of frequency 1750 Hz. (Hamming window applied to blocks of input samples.)

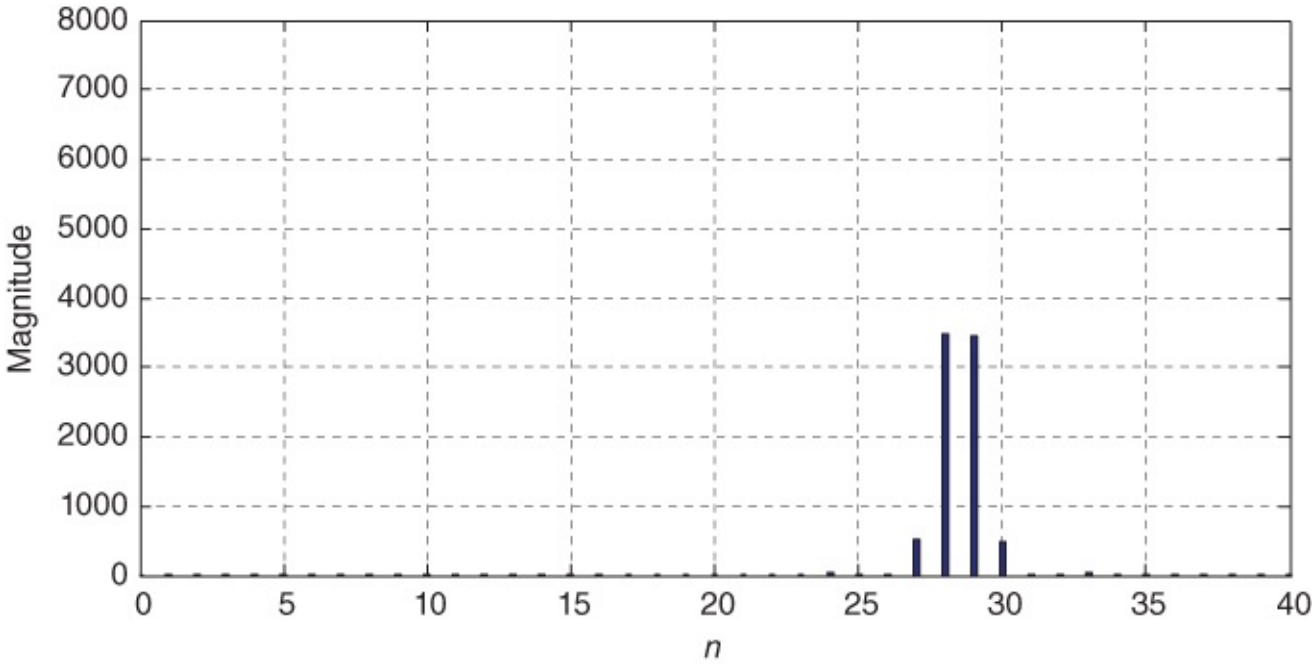


Figure 5.23 Partial contents of array `outbuffer`, plotted using MATLAB function `tm4c123_plot_real()`, for input sinusoid of frequency 1781 Hz. (Hamming window applied to blocks of input samples.)

Example 5.4

FFT of a Real-Time Input Signal Using an FFT Function in C (`tm4c123_fft128_dma.c`).

Program `tm4c123_fft128_dma.c` implements a 128-point FFT in real time using an external input signal. It calls an FFT function `fft()` written in C. That function is defined in the separate header file `fft.h` (Listing 5.10). The function was written originally for use with the Texas Instruments C31 DSK and is described in [1].

Program `tm4c123_fft128_dma.c` is similar to program `tm4c123_dft 128_dma.c` in all respects other than its use of function `fft()` in place of the less computationally efficient function `dftw()` and the slightly different computation of twiddle factors. Build and run this program. Repeat the experiments carried out in Example 5.10 and verify that the results obtained are similar.

Example 5.5

FFT of a Real-Time Input Signal Using CMSIS DSP function `arm_cfft_f32()` (`tm4c123_fft128_CMSIS_dma.c`).

Program `tm4c123_fft128_CMSIS_dma.c`, shown in Listing 5.14 uses the computationally

efficient CMSIS DSP library function `arm_cfft_f32()` in order to calculate the FFT of a block of samples. Other than that, it is similar to the previous two examples and should produce similar results. The twiddle factors used by function `arm_cfft_f32()` are provided by a structure defined in header file `arm_const_structs.h` and the parameter passed to the function must match the value of the constant `BUFSIZE` defined in file `tm4c123_aic3104_init.h`. For example, if the value of `BUFSIZE` is equal to 256, then the address of the structure `arm_cfft_sR_f32_len256` must be passed to function `arm_cfft_f32()`. Similarly, the header file defining the Hamming window used by the program must match the value of the constant `BUFSIZE`. For example, if the value of `BUFSIZE` is equal to 64, then the file `hamm64.h` must be included.

Listing 5.7 Program `tm4c123_fft128_CMSIS_dma.c`

```
// tm4c123_fft128_CMSIS_dma.c
#include "tm4c123_aic3104_init.h"
#include "arm_const_structs.h"
// the following #include must match BUFSIZE
// defined in file tm4c123_aic3104_init.h
//#include "hamm64.h"
#include "hamm128.h"
//#include "hamm256.h"
extern int16_t LpingIN[BUFSIZE], LpingOUT[BUFSIZE];
extern int16_t LpongIN[BUFSIZE], LpongOUT[BUFSIZE];
extern int16_t RpingIN[BUFSIZE], RpingOUT[BUFSIZE];
extern int16_t RpongIN[BUFSIZE], RpongOUT[BUFSIZE];
extern int16_t Lprocbuffer, Rprocbuffer;
extern volatile int16_t LTxcomplete, LRxcomplete;
extern volatile int16_t RTxcomplete, RRxcomplete;
#define TRIGGER 28000
#define MAGNITUDE_SCALING_FACTOR 32
float32_t cbuf[2*BUFSIZE];
float32_t outbuffer[BUFSIZE];
void Lprocess_buffer(void)
{
  int16_t *inBuf, *outBuf;
  float32_t *cbufptr;
  int16_t i;
  if (Lprocbuffer ⌋ PING)
  { inBuf = LpingIN; outBuf = LpingOUT; }
  if (Lprocbuffer ⌋ PONG)
  { inBuf = LpongIN; outBuf = LpongOUT; }
  cbufptr = cbuf;
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    *cbufptr++ = (float32_t)(*inBuf++)*hamming[i];
    *cbufptr++ = 0.0f;
  }
// the following function call must match BUFSIZE
// defined in file tm4c123_aic3104_init.h
//  arm_cfft_f32(&arm_cfft_sR_f32_len64, (float32_t *)(cbuf), 0, 1);
```

```
    arm_cfft_f32(&arm_cfft_sR_f32_len128, (float32_t *)(cbuf), 0, 1);
//  arm_cfft_f32(&arm_cfft_sR_f32_len256, (float32_t *)(cbuf), 0, 1);
  arm_cmplx_mag_f32((float32_t *)(cbuf),outbuffer,BUFSIZE);
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    if (i⌋0)
      *outBuf++ = TRIGGER;
    else
      *outBuf++ = (int16_t)(outbuffer[i]/MAGNITUDE_SCALING_FACTOR);
  }
  LTxcomplete = 0;
  LRxcomplete = 0;
  return;
}
void Rprocess_buffer(void)
{
  int16_t *inBuf, *outBuf;
  int16_t i;
  if (Rprocbuffer ⌋ PING)
  { inBuf = RpingIN; outBuf = RpingOUT; }
  if (Rprocbuffer ⌋ PONG)
  { inBuf = RpongIN; outBuf = RpongOUT; }
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    *outBuf++ = 0;
  }
  RTxcomplete = 0;
  RRxcomplete = 0;
  return;
}
void SSI_interrupt_routine(void){while(1){}}
int main(void)
{
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_DMA,
                       PGA_GAIN_6_DB);
 while(1)
  {
    while((!LTxcomplete)|(!LRxcomplete));
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
    Lprocess_buffer();
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
    while((!RTxcomplete)|(!RRxcomplete));
    Rprocess_buffer();
  }
}
```

Example 5.6

Real-Time FFT of a Sinusoidal Signal from a Lookup Table (`tm4c123_fft128_sinetable_dma.c`).

This example program adapts program `tm4c123_fft128_dma.c` 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 AIC3104 codec, allowing both input and output signals to be viewed together on an oscilloscope as shown in Figure 5.24. Pressing SW1 on the TM4C123 LaunchPad steps the frequency of the input signal through a range of different values.

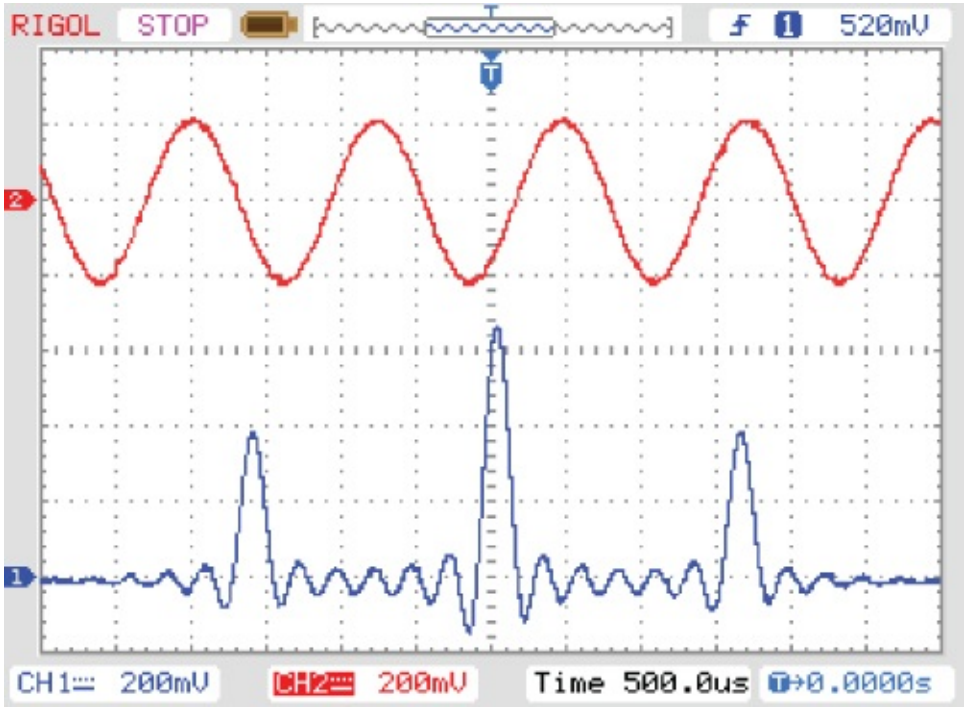


Figure 5.24 Output signal generated by program `tm4c123_fft128_sinetable_dma.c`, displayed using a *Rigol DS1052E* oscilloscope.

5.9 Fast Convolution

Fast convolution is a technique whereby two sequences of (time-domain) samples are convolved not by direct implementation of the convolution sum but by multiplying together their frequency-domain representations. Computation of the convolution sum is computationally expensive for large N, 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:

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 by 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 M filter coefficients will be $(N + M - 1)$ 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 blocks of input samples and zero-padding both these samples and the filter coefficients used to length $2N$ before using a $2N$ -point FFT. This requires that the number of filter coefficients is less than or equal to $(N + 1)$.
3. Overlapping and adding the length $2N$ blocks of output samples obtained using a $2N$ -point inverse FFT of the result of multiplying frequency-domain representations of input samples and filter coefficients.

Example 5.7

Real-Time Fast Convolution (`tm4c123_fastconv_dma.c`).

Fast convolution is implemented by program `tm4c123_fastconv_dma.c`, shown in Listing 5.16.

As described earlier, because multiplication of two signals represented in the frequency domain corresponds to circular, and not linear, convolution in the time domain, it is necessary to zero-pad both the blocks of `BUFSIZE` input samples and the `N` FIR filter coefficients to a length greater than `(BUFSIZE + N -1)` before transforming into the frequency domain. In this

program example, a (BUFSIZE*2)-point FFT is used and hence the maximum number of FIR filter coefficients possible is (BUFSIZE+1). Inverse Fourier transformation (DFT) of the result of the multiplication of two (BUFSIZE*2)-point signal representations in the frequency domain yields (BUFSIZE*2) time-domain samples. Yet, these are the result of processing just BUFSIZE time-domain input samples. Blocks of (BUFSIZE*2) time-domain output samples are overlapped (by BUFSIZE samples) and added together in order to obtain the overall output of the filter in blocks of BUFSIZE samples.

Listing 5.8 Program `tm4c123_fastconv_dma.c`

```
// tm4c123_fastconv_dma.c
#include "tm4c123_aic3104_init.h"
#include "lp55.h"
#include "fft.h"
extern int16_t LpingIN[BUFSIZE], LpingOUT[BUFSIZE];
extern int16_t LpongIN[BUFSIZE], LpongOUT[BUFSIZE];
extern int16_t RpingIN[BUFSIZE], RpingOUT[BUFSIZE];
extern int16_t RpongIN[BUFSIZE], RpongOUT[BUFSIZE];
extern int16_t Lprocbuffer, Rprocbuffer;
extern volatile int16_t LTxcomplete, LRxcomplete;
extern volatile int16_t RTxcomplete, RRxcomplete;
COMPLEX procbuf[2*BUFSIZE],coeffs[2*BUFSIZE];
COMPLEX twiddle[2*BUFSIZE];
float overlap[BUFSIZE];
float a,b;
void Lprocess_buffer(void)
{
  int16_t *inBuf, *outBuf;
  int16_t i;
  if (Lprocbuffer == PING)
  { inBuf = LpingIN; outBuf = LpingOUT; }
  if (Lprocbuffer == PONG)
  { inBuf = LpongIN; outBuf = LpongOUT; }
  for (i = 0; i < (2*BUFSIZE) ; i++)
  {
    procbuf[i].real = 0.0;
    procbuf[i].imag = 0.0;
  }
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    procbuf[i].real = (float32_t)(*inBuf++);
  }
  fft(procbuf,2*BUFSIZE,twiddle);
  for (i=0 ; i<(2*BUFSIZE) ; i++)
  {
    a = procbuf[i].real;
    b = procbuf[i].imag;
    procbuf[i].real = coeffs[i].real*a
                    - coeffs[i].imag*b;
    procbuf[i].imag = -(coeffs[i].real*b
                      + coeffs[i].imag*a);
```

```
  }
  fft(procbuf,2*BUFSIZE,twiddle);
  for (i=0 ; i<(2*BUFSIZE) ; i++)
  {
    procbuf[i].real /= (2*BUFSIZE);
  }
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    *outBuf++ = (int16_t)(procbuf[i].real + overlap[i]);
    overlap[i] = procbuf[i+BUFSIZE].real;
  }
  LTxcomplete = 0;
  LRxcomplete = 0;
  return;
}
void Rprocess_buffer(void)
{
  int16_t *inBuf, *outBuf;
  int16_t i;
  if (Rprocbuffer ⌋ PING)
  { inBuf = RpingIN; outBuf = RpingOUT; }
  if (Rprocbuffer ⌋ PONG)
  { inBuf = RpongIN; outBuf = RpongOUT; }
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    *outBuf++ = 0x0000;
  }
  RTxcomplete = 0;
  RRxcomplete = 0;
  return;
}
void SSI_interrupt_routine(void){while(1){}}
int main(void)
{
  int i;
  for (i=0 ; i< (2*BUFSIZE) ; i++)
  {
    twiddle[i].real = cos(PI*i/(2*BUFSIZE));
    twiddle[i].imag = -sin(PI*i/(2*BUFSIZE));
  }
  for(i=0 ; i<((2*BUFSIZE)) ; i++)
  { coeffs[i].real = 0.0; coeffs[i].imag = 0.0;}
  for(i=0 ; i<N ; i++) coeffs[i].real = h[i];
  fft(coeffs,(2*BUFSIZE),twiddle);
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_DMA,
                       PGA_GAIN_6_DB);
 while(1)
  {
    while((!RTxcomplete)|(!RRxcomplete));
    Rprocess_buffer();
    while((!LTxcomplete)|(!LRxcomplete));
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
    Lprocess_buffer();
```

```
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
  }
}
```

5.9.1 Running the Program

Build and run the program and verify that it implements a low-pass filter, as determined by the use of filter coefficient header file `lp55.h`. Functionally, the program is equivalent to program `tm4c123_fir_dma.c`, described in Chapter 3, and similarly introduces a delay of `BUFSIZE*2` sampling instants to an analog signal passing through the system. The program has been written so that, just as in the case of program `tm4c123_fir_dma.c`, the FIR filter coefficients are read from a separate header file specified by a preprocessor command, for example,

```
#include “lp55.h”
```

The maximum possible value of `N` (the number of filter coefficients, defined in the header file), is `(BUFSIZE+1)`.

5.9.2 Execution Time of Fast Convolution Method of FIR Filter Implementation

The execution time of the fast convolution algorithm implemented by program may be estimated by observing the pulse output on GPIO pin PE2 using an oscilloscope.

Example 5.8

Graphic Equalizer (`tm4c123_graphicEQ_dma.c`).

Listing 5.17 is of program `tm4c123_graphicEQ_dma.c`, which implements a three-band graphic equalizer. It uses CMSIS DSP library function `arm_cfft_f32()` in order implement the complex FFT calculations.

Listing 5.9 Program `tm4c123_graphicEQ_dma.c`)

```
// tm4c123_graphicEQ_dma.c
#include "tm4c123_aic3104_init.h"
#include "GraphicEQcoeff.h"
#include "fft.h"
extern int16_t LpingIN[BUFSIZE], LpingOUT[BUFSIZE];
extern int16_t LpongIN[BUFSIZE], LpongOUT[BUFSIZE];
extern int16_t RpingIN[BUFSIZE], RpingOUT[BUFSIZE];
extern int16_t RpongIN[BUFSIZE], RpongOUT[BUFSIZE];
extern int16_t Lprocbuffer, Rprocbuffer;
extern volatile int16_t LTxcomplete, LRxcomplete;
```

```
extern volatile int16_t RTxcomplete, RRxcomplete;
COMPLEX xL[2*BUFSIZE],coeffs[2*BUFSIZE];
COMPLEX twiddle[2*BUFSIZE];
COMPLEX treble[2*BUFSIZE],bass[2*BUFSIZE];
COMPLEX mid[2*BUFSIZE];
float overlap[BUFSIZE];
float a,b;
float32_t bass_gain = 0.1;
float32_t mid_gain = 1.0;
float32_t treble_gain = 0.25;
int16_t NUMCOEFFS = sizeof(lpcoeff)/sizeof(float);
void Lprocess_buffer(void)
{
  int16_t *inBuf, *outBuf;
  int16_t i;
  if (Lprocbuffer == PING)
  { inBuf = LpingIN; outBuf = LpingOUT; }
  if (Lprocbuffer == PONG)
  { inBuf = LpongIN; outBuf = LpongOUT; }
  for (i = 0; i < (2*BUFSIZE) ; i++)
  {
    xL[i].real = 0.0;
    xL[i].imag = 0.0;
  }
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    xL[i].real = (float32_t)(*inBuf++);
  }
  fft(xL,2*BUFSIZE,twiddle);
  for (i=0 ; i<BUFSIZE ; i++)
  {
    coeffs[i].real = bass[i].real*bass_gain
                   + mid[i].real*mid_gain
                   + treble[i].real*treble_gain;
    coeffs[i].imag = bass[i].imag*bass_gain
                   + mid[i].imag*mid_gain
                   + treble[i].imag*treble_gain;
  }
  for (i=0 ; i<(2*BUFSIZE) ; i++)
  {
    a = xL[i].real;
    b = xL[i].imag;
    xL[i].real = coeffs[i].real*a - coeffs[i].imag*b;
    xL[i].imag = -(coeffs[i].real*b + coeffs[i].imag*a);
  }
  fft(xL,2*BUFSIZE,twiddle);
  for (i=0 ; i<(2*BUFSIZE) ; i++)
  {
    xL[i].real /= (2*BUFSIZE);
  }
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    *outBuf++ = (int16_t)(xL[i].real + overlap[i]);
    overlap[i] = xL[i+BUFSIZE].real;
  }
```

```
  LTxcomplete = 0;
  LRxcomplete = 0;
  return;
}
void Rprocess_buffer(void)
{
  int16_t *inBuf, *outBuf;
  int16_t i;
  if (Rprocbuffer ) PING)
  { inBuf = RpingIN; outBuf = RpingOUT; }
  if (Rprocbuffer ) PONG)
  { inBuf = RpongIN; outBuf = RpongOUT; }
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    *outBuf++ = 0x0000;
  }
  RTxcomplete = 0;
  RRxcomplete = 0;
  return;
}
void SSI_interrupt_routine(void){while(1){}}
int main(void)
{
  int i;
  for (i=0 ; i< (2*BUFSIZE) ; i++)
  {
    twiddle[i].real = cos(PI*i/(2*BUFSIZE));
    twiddle[i].imag = -sin(PI*i/(2*BUFSIZE));
  }
  for(i=0 ; i<(BUFSIZE*2) ; i++)
  {
    coeffs[i].real = 0.0;
    coeffs[i].imag = 0.0;
    bass[i].real = 0.0;
    mid[i].real = 0.0 ;
    treble[i].real = 0.0;
    bass[i].imag = 0.0;
    mid[i].imag = 0.0 ;
    treble[i].imag = 0.0;
  }
  for(i=0 ; i<(BUFSIZE) ; i++)
  {
    overlap[i] = 0.0;
  }
  for(i=0 ; i<NUMCOEFFS ; i++)
  {
    bass[i].real = lpcoeff[i];
    mid[i].real = bpcoeff[i];
    treble[i].real = hpcoeff[i];
  }
  fft(bass,(2*BUFSIZE),twiddle);
  fft(mid,(2*BUFSIZE),twiddle);
  fft(treble,(2*BUFSIZE),twiddle);
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
```

```
                        IO_METHOD_DMA,
                        PGA_GAIN_6_DB);
 while(1)
  {
    while((!RTxcomplete)|(!RRxcomplete));
    Rprocess_buffer();
    while((!LTxcomplete)|(!LRxcomplete));
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
    Lprocess_buffer();
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
  }
}
```

The coefficient header file `graphicEQcoeff.h` contains three sets of coefficients: a low-pass filter with a cutoff frequency of 1.3 kHz, a band-pass filter with cutoff frequencies at 1.3 and 2.6 kHz, and a high-pass filter with a cutoff frequency of 2.6 kHz. These filters were designed using the MATLAB function `fir1()`. The three sets of filter coefficients are transformed into the frequency domain just once, and subsequently, linear, weighted sums of their frequency domain representations are used in the fast convolution process. A similar overlap-add scheme to that used in the fast convolution example is employed. The gains in the three frequency bands are set by the program statements

```
float32_t bass_gain = 0.1;
float32_t mid_gain = 1.0;
float32_t treble_gain = 0.25;
```

As provided, the program would allow for the values of these gains to be changed while it is running. The weighted sum of the frequency-domain representations of the three filters is computed for each block of samples by the following program statements.

```
  for (i=0 ; i<BUFSIZE ; i++)
  {
    coeffs[i].real = bass[i].real*bass_gain
                   + mid[i].real*mid_gain
                   + treble[i].real*treble_gain;
    coeffs[i].imag = bass[i].imag*bass_gain
                   + mid[i].imag*mid_gain
                   + treble[i].imag*treble_gain;
  }
```

However, as it stands, the program does not include a mechanism for altering the gains.

The magnitude frequency response of the graphic equalizer may be observed using the FFT function on an oscilloscope or using *Goldwave* and either by replacing the programs statement

```
    xL[i].real = (float32_t)(*inBuf++);
```

with the program statement

```
    xL[i].real = (float32_t)(prbs(8000));
```

or by using a microphone as an input device and blowing gently on the microphone.

Figure 5.25 shows the frequency content of pseudorandom noise that has been filtered by the graphic equalizer with the gain settings `bass_gain = 0.1`, `mid_gain = 0.1`, and `treble_gain = 0.25`.

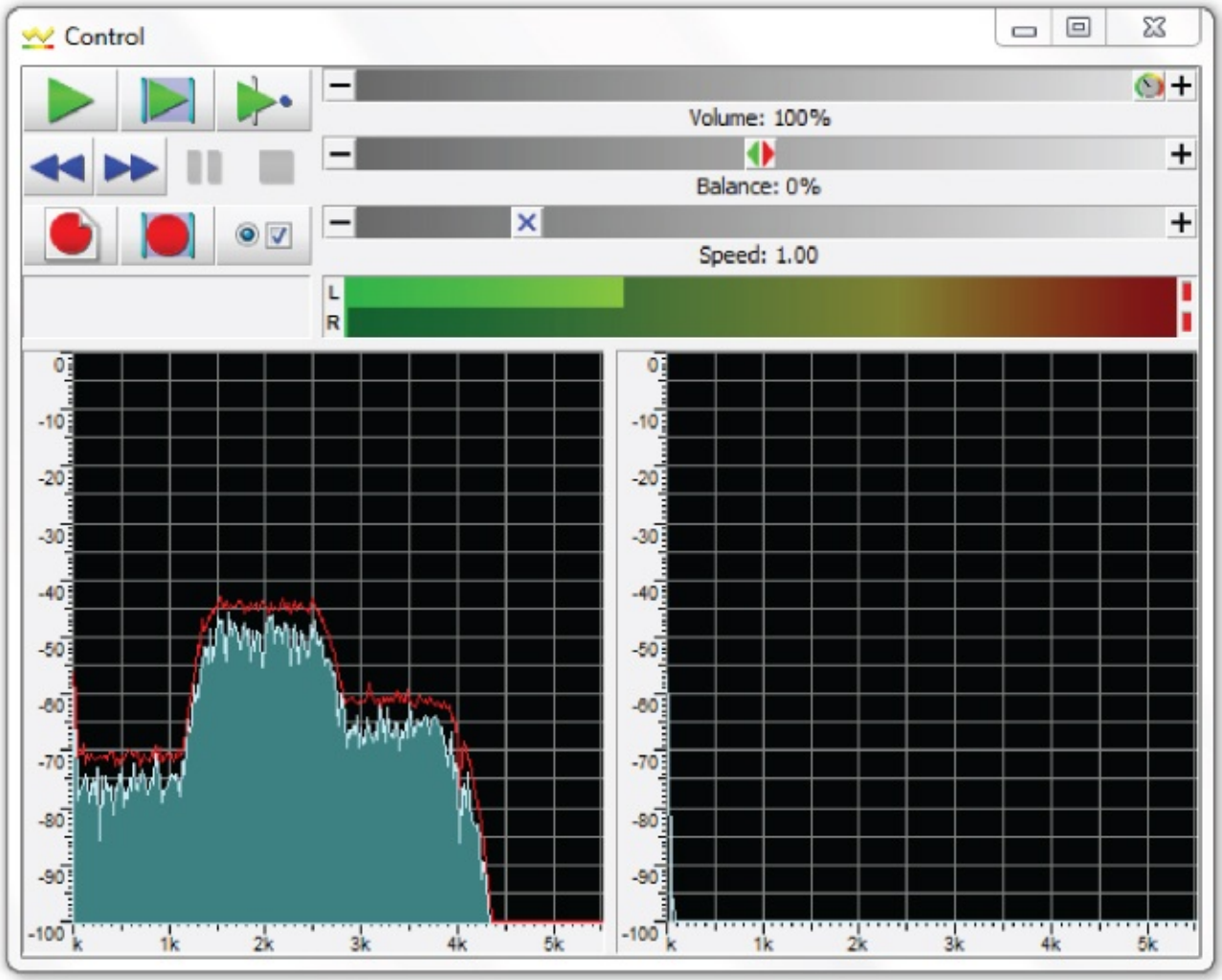


Figure 5.25 Output signal from program `tm4c123_graphicEQ_CMSIS_dma.c`, displayed using *Goldwave*, for a pseudorandom noise input signal. `bass_gain = 0.1`, `mid_gain = 0.1`, `treble_gain = 0.25`.

Reference

1. Chassaing, R., "*Digital Signal Processing Laboratory Experiments with C31*", John Wiley & Sons, Inc., New York, 1999.

Chapter 6
Adaptive Filters

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.

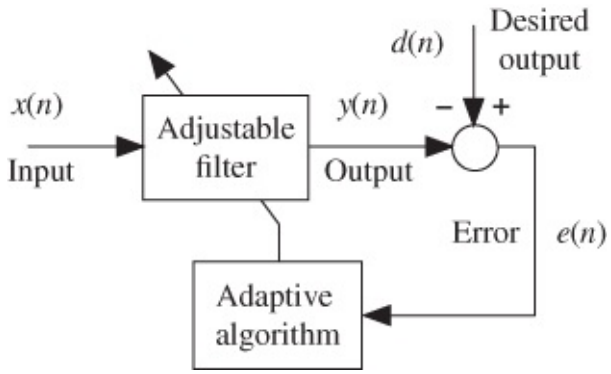


Figure 6.1 Basic adaptive filter structure.

Its component parts are an adjustable filter, a mechanism for performance measurement (in this case, a comparator to measure the instantaneous error between adaptive filter output and desired output) 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

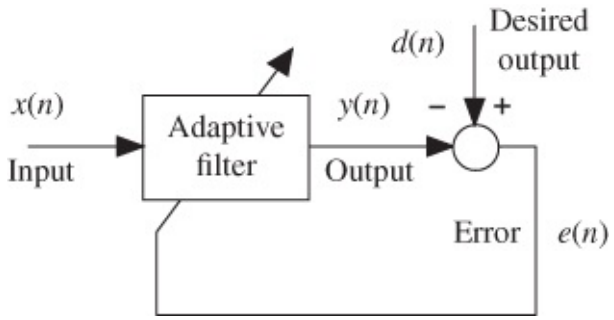


Figure 6.2 Simplified block diagram of basic adaptive filter structure.

It is conventional to refer to the coefficients of an adaptive FIR filter as weights, and the filter coefficients of the adaptive filters in several of the program examples in this chapter are stored in arrays using the identifier `w` rather than `h` as tended to be 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,

$$\xi(n) = E[e^2(n)]. \qquad \textbf{6.1}$$

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 desired signal $s(n)$. In doing this, the filter learns something about the characteristics of the signal $s(n)$ and/or of the process that generated it. Adaptive prediction is used widely in signal encoding and noise reduction.

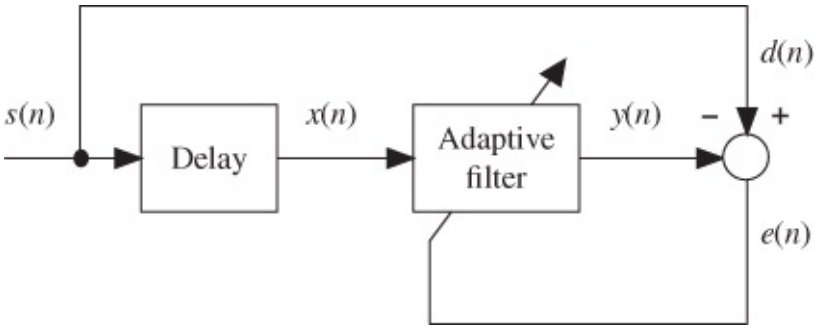


Figure 6.3 Basic adaptive filter structure configured for prediction.

6.2.2 System Identification or Direct Modeling

In this configuration (Figure 6.4), broadband noise $s(n)$ is input both to the adaptive filter and to an unknown plant or system. If adaptation is successful and the mean squared error is minimized (to zero in an idealized situation), 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 an audio 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.

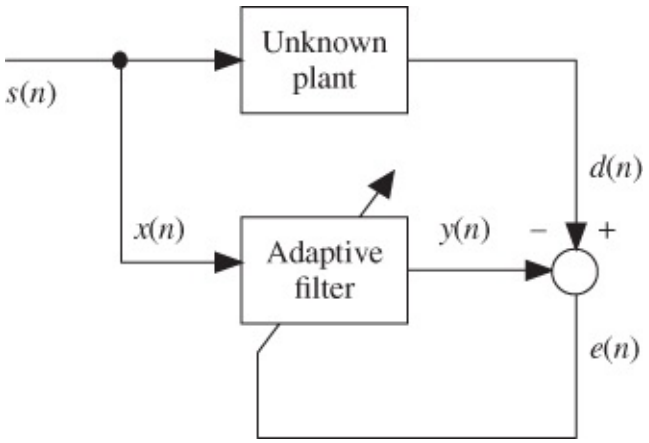


Figure 6.4 Basic adaptive filter structure configured for system identification.

6.2.3 Noise Cancellation

This configuration differs from the previous two in that while the mean squared error is minimized, it is not minimized to zero, even in the ideal case, 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 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. It is assumed here also 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, leading to a reduction in the performance of the noise cancellation system.

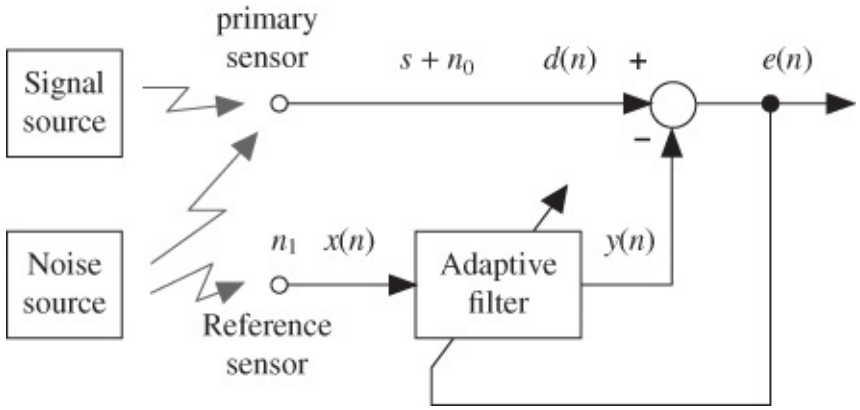


Figure 6.5 Basic adaptive filter structure configured for noise cancellation.

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 earlier, 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 shown in 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.

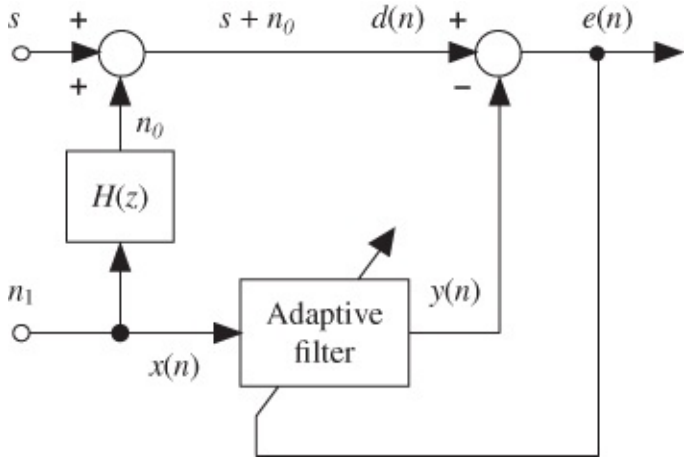


Figure 6.6 Alternative representation of basic adaptive filter structure configured for noise cancellation emphasizing the difference $H(z)$ in paths from a single noise source to primary and reference sensors.

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 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 natural 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 and using a copy of this stored at the receiver as the desired signal. In most cases, a pseudorandom, broadband signal is used for this purpose.

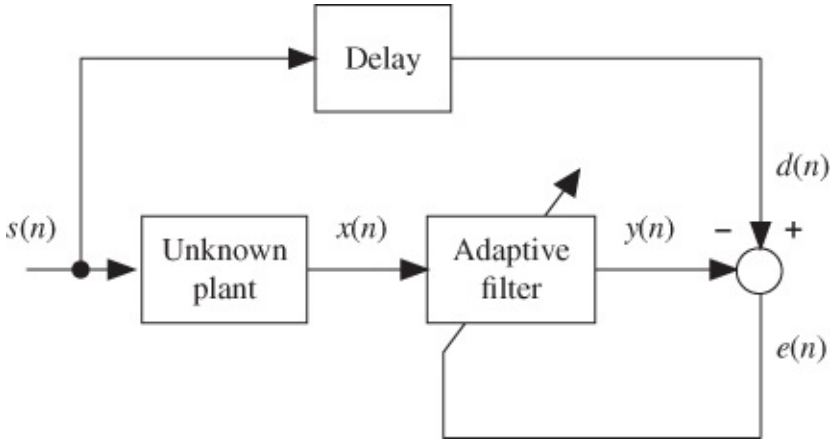


Figure 6.7 Basic adaptive filter structure configured for equalization.

6.3 Performance Function

Consider the block diagram representation of an FIR filter introduced in Chapter 3 and shown again in Figure 6.8.

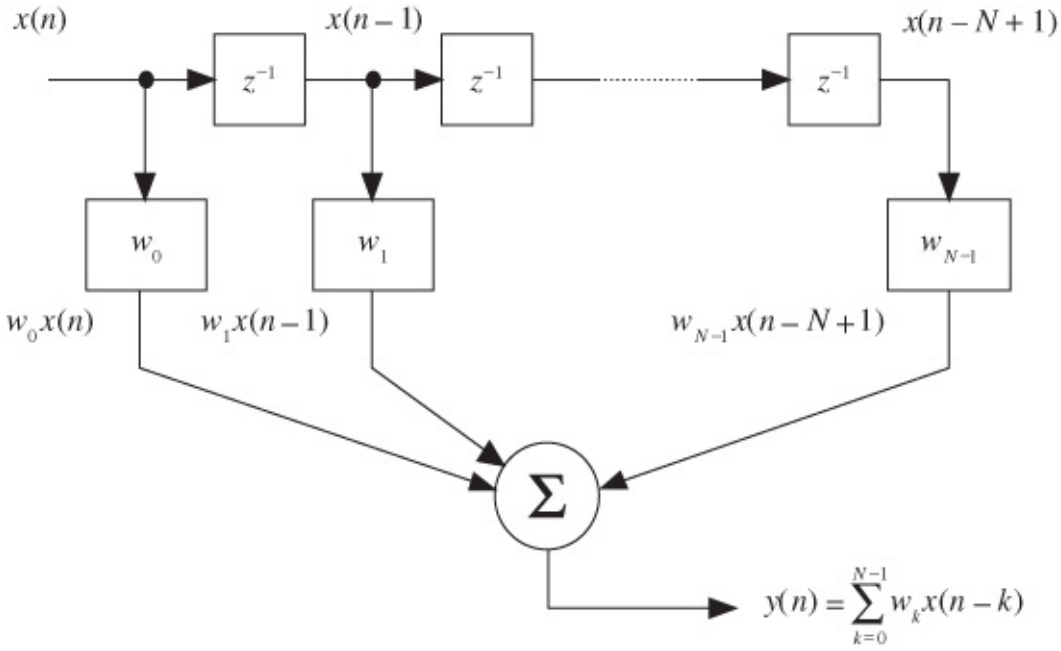


Figure 6.8 Block diagram representation of FIR filter.

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 $\boldsymbol{w}(n)$ and $\boldsymbol{x}(n)$, respectively, where

$$\boldsymbol{w}(n) = [w_0(n), w_1(n), \cdots, w_{N-1}(n)]^T \tag{6.2}$$

and

$$\boldsymbol{x}(n) = [x(n), x(n-1), \cdots, x(n-N+1)]^T. \tag{6.3}$$

Hence, using vector notation, the filter output at the n th sample instant is given by

$$y(n) = \boldsymbol{w}^T(n)\boldsymbol{x}(n) = \boldsymbol{x}^T(n)\boldsymbol{w}(n). \tag{6.4}$$

Instantaneous error is given by

$$\begin{aligned} e(n) &= d(n) - y(n) \\ &= d(n) - \boldsymbol{x}^T(n)\boldsymbol{w}(n) \end{aligned} \tag{6.5}$$

and instantaneous squared error by

$$e^2(n) = d^2(n) - 2d(n)\boldsymbol{x}^T(n)\boldsymbol{w}(n) + \boldsymbol{w}^T(n)\boldsymbol{x}(n)\boldsymbol{x}^T(n)\boldsymbol{w}(n). \tag{6.6}$$

Mean squared error (expected value of squared error) is therefore given by

$$\xi(n) = E[e^2(n)] \qquad \textbf{6.7}$$
$$= E[d^2(n) - 2d(n)\boldsymbol{x}^T(n)\boldsymbol{w}(n) + \boldsymbol{w}^T(n)\boldsymbol{x}(n)\boldsymbol{x}^T(n)\boldsymbol{w}(n)].$$

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 $d(n)\boldsymbol{x}(n)$ and $\boldsymbol{x}(n)\boldsymbol{x}^T(n)$ are constants and hence

$$\xi(n) = E[d^2(n)] + \boldsymbol{w}^T(n)E[\boldsymbol{x}^T(n)\boldsymbol{x}(n)]\boldsymbol{w}(n) - 2E[d(n)\boldsymbol{x}^T(n)]\boldsymbol{w}(n) \qquad \textbf{6.8}$$
$$= E[d^2(n)] + \boldsymbol{w}^T(n)\boldsymbol{R}\boldsymbol{w}(n) - 2\boldsymbol{p}^T\boldsymbol{w}(n),$$

where the vector $\boldsymbol{p}$ of cross-correlation between input and desired output is defined as

$$\boldsymbol{p} = E[d(n)\boldsymbol{x}(n)] \qquad \textbf{6.9}$$

and the input autocorrelation matrix $\boldsymbol{R}$ is defined as

$$\boldsymbol{R} = E[\boldsymbol{x}(n)\boldsymbol{x}^T(n)]. \qquad \textbf{6.10}$$

The performance function, or surface, $\xi(n)$ is a quadratic function of $\boldsymbol{w}(n)$ and as such is referred to in the following equations as $\xi(\boldsymbol{w})$. Since it is a quadratic function of $\boldsymbol{w}(n)$, it has one global minimum corresponding to $\boldsymbol{w}(n) = \boldsymbol{w}_{\text{opt}}$. The optimum value of the weights, $\boldsymbol{w}_{\text{opt}}$, may be found by equating the gradient of the performance surface to zero, that is,

$$\frac{\partial \xi(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial}{\partial \boldsymbol{w}}[E[d^2(n)] + \boldsymbol{w}^T(n)\boldsymbol{R}\boldsymbol{w}(n) - 2\boldsymbol{p}^T\boldsymbol{w}(n)] \qquad \textbf{6.11}$$
$$= 2\boldsymbol{R}\boldsymbol{w} - 2\boldsymbol{p}.$$

and hence, in terms of the statistical quantities just described

$$2\boldsymbol{R}\boldsymbol{w}_{\text{opt}} - 2\boldsymbol{p} = 0 \qquad \textbf{6.12}$$

and hence,

$$\boldsymbol{w}_{\text{opt}} = \boldsymbol{R}^{-1}\boldsymbol{p}. \qquad \textbf{6.13}$$

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 $\boldsymbol{R}$.

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 (i.e., difficult to visualize or to represent in a

figure). 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.

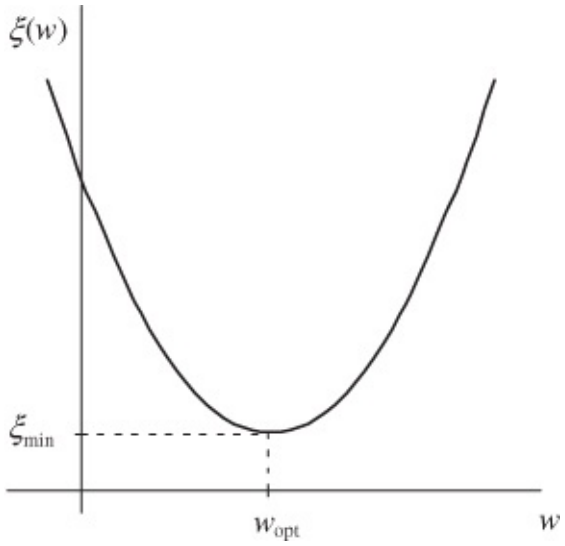


Figure 6.9 Performance function for single weight case.

6.4 Searching for the Minimum

An alternative to solving Equation (6.13) by matrix inversion is to *search* the performance function for $\boldsymbol{w}_{opt}$, 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 weights are adjusted in the direction of the negative gradient of the performance function and by an amount proportional to the magnitude of the gradient, that is,

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) - \beta \frac{\partial \xi(\boldsymbol{w})}{\partial \boldsymbol{w}}\Big|_{\boldsymbol{w}=\boldsymbol{w}(n)}, \tag{6.14}$$

where $\boldsymbol{w}(n)$ 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. One iteration of the steepest descent algorithm described by Equation (6.14) is illustrated for the case of a single weight in Figure 6.10.

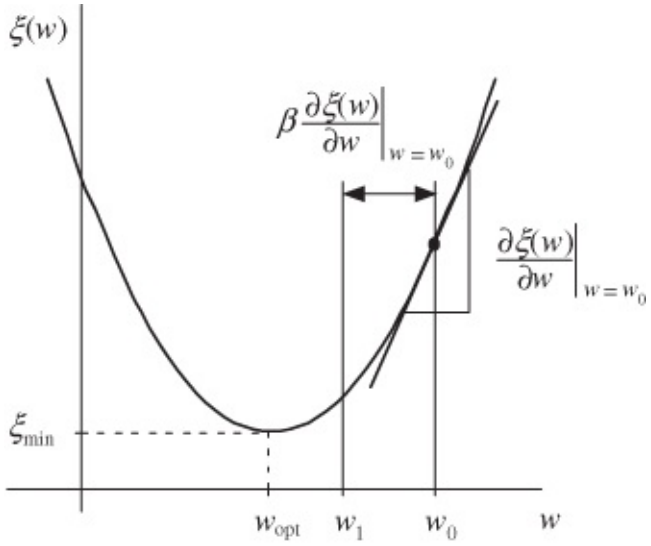


Figure 6.10 Steepest descent algorithm illustrated for single weight case.

6.5 Least Mean Squares Algorithm

The steepest descent algorithm requires an estimate of the gradient $\nabla(n)$ of the performance surface $\xi(\boldsymbol{w})$ 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 (LMS) algorithm uses instantaneous error squared $e^2(n)$ as an estimate of mean squared error $E[e^2(n)]$ and yields an estimated gradient

$$\hat{\nabla}(n) = \begin{bmatrix} \frac{\partial \hat{\xi}(n)}{\partial w_0(n)} \\ \frac{\partial \hat{\xi}(n)}{\partial w_1(n)} \\ \vdots \\ \frac{\partial \hat{\xi}(n)}{\partial w_{N-1}(n)} \end{bmatrix} = \begin{bmatrix} \frac{\partial e^2(n)}{\partial w_0(n)} \\ \frac{\partial e^2(n)}{\partial w_1(n)} \\ \vdots \\ \frac{\partial e^2(n)}{\partial w_{N-1}(n)} \end{bmatrix} \tag{6.15}$$

or

$$\hat{\nabla}(n) = \frac{\partial \hat{\xi}(n)}{\partial \boldsymbol{w}(n)} = \frac{\partial e^2(n)}{\partial \boldsymbol{w}(n)}. \tag{6.16}$$

Equation (6.6) gave an expression for instantaneous squared error

$$e^2(n) = d^2(n) - 2d(n)\boldsymbol{x}^T(n)\boldsymbol{w}(n) + \boldsymbol{w}^T(n)\boldsymbol{x}(n)\boldsymbol{x}^T(n)\boldsymbol{w}(n).$$

Differentiating this with respect to $\boldsymbol{w}(n)$,

$$\begin{aligned} \frac{\partial e^2(n)}{\partial \boldsymbol{w}(n)} &= 2\boldsymbol{x}(n)\boldsymbol{x}^T(n)\boldsymbol{w}(n) - 2d(n)\boldsymbol{x}(n) \\ &= 2\boldsymbol{x}(n)(\boldsymbol{x}^T(n)\boldsymbol{w}(n) - d(n)) \\ &= -2e(n)\boldsymbol{x}(n). \end{aligned} \tag{6.17}$$

Hence, the steepest descent algorithm, using this gradient estimate, is

$$\begin{aligned} \boldsymbol{w}(n+1) &= \boldsymbol{w}(n) - \beta\hat{\nabla}(n) \\ &= \boldsymbol{w}(n) + 2\beta e(n)\boldsymbol{x}(n). \end{aligned} \tag{6.18}$$

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 multiply capability than the straightforward LMS algorithm.

6.5.1 LMS Variants

For the sign-error LMS algorithm, Equation (6.18) becomes

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) + \beta\ \mathrm{sgn}(e(n))\boldsymbol{x}(n), \tag{6.19}$$

where $\mathrm{sgn}()$ is the signum function

$$\mathrm{sgn}(u) = \begin{cases} 1, & \text{if} \quad u \geq 0 \\ -1, & \text{if} \quad u < 0. \end{cases} \tag{6.20}$$

For the sign-data LMS algorithm, Equation (6.18) becomes

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) + \beta\mathrm{sgn}(\boldsymbol{x}(n))e(n). \tag{6.21}$$

For the sign-sign LMS algorithm, Equation (6.18) becomes

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) + \beta\mathrm{sgn}(e(n))\mathrm{sgn}(\boldsymbol{x}(n)), \tag{6.22}$$

which reduces to

$$\boldsymbol{w}(n+1) = \begin{cases} \boldsymbol{w}(n) + \beta, & \text{if } \mathrm{sgn}(e(n)) = \mathrm{sgn}(\boldsymbol{x}(n)) \\ \boldsymbol{w}(n) - \beta, & \text{otherwise} \end{cases} \tag{6.23}$$

and which involves no multiplications.

6.5.2 Normalized LMS Algorithm

The rate of adaptation of the standard LMS algorithm is sensitive to the magnitude of the input signal $\boldsymbol{x}(n)$. This problem may be ameliorated by using the normalized LMS algorithm in which

the adaptation rate β in the LMS algorithm is replaced by

$$\beta_{\text{NLMS}} = \frac{1}{\sum_{n=0}^{(N-1)} x^2(n)}. \qquad \textbf{6.24}$$

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 (`stm32f4_adaptive.c`).

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 $d(n)$ and the reference input to the adaptive filter $x(n)$.
2. Calculate the adaptive FIR filter output $y(n)$, applying Equation (6.4).
3. Calculate the instantaneous error $e(n)$ by applying Equation (6.5).
4. Update the coefficient (weight) vector 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 at the next sampling instant.

Program `stm32f4_adaptive.c` is shown in Listing 6.2. The desired signal is chosen to be $d(n) = 2\cos()2\pi n/8$, and the input to the adaptive filter is chosen to be $x(n) = \sin()2\pi n/8$. The adaptation rate β, filter order N, and number of samples processed in the program are equal to 0.01, 21, and 60, respectively. The overall output is the adaptive filter output $y(n)$, which converges to the desired cosine signal $d(n)$.

Listing 6.1 Program `stm32f4_adaptive.c`.

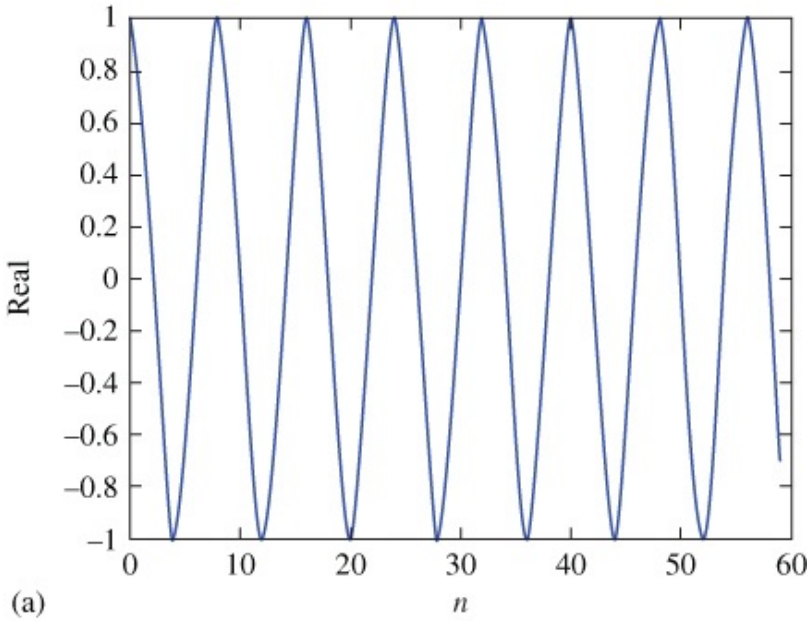
```
// stm32f4_adaptive.c
#include "stm32f4_wm5102_init.h"
#define BETA 0.01f                              // learning rate
#define N  21                                        // number of
filter coeffs
#define NUM_ITERS  60                  // number of iterations
float32_t desired[NUM_ITERS]; // storage for results
float32_t y_out[NUM_ITERS];
float32_t error[NUM_ITERS];
float32_t w[N] = {0.0};       // adaptive filter weights
float32_t x[N] = {0.0};       // adaptive filter delay line
int i, t;
float32_t d, y, e;
int main()
{
  for (t = 0; t < NUM_ITERS; t++)
  {
    x[0] = sin(2*PI*t/8);     // get new input sample
    d = 2*cos(2*PI*t/8);      // get new desired output
    y = 0;                    // compute filter output
    for (i = 0; i <= N; i++)
      y += (w[i]*x[i]);
    e = d - y;                // compute error
    for (i = N; i >= 0; i--)
    {
      w[i] += (BETA*e*x[i]);  // update filter weights
      if (i != 0)
        x[i] = x[i-1];        // shift data in delay line
    }
    desired[t] = d;           // store results
    y_out[t] = y;
    error[t] = e;
  }
  while(1){}
}
```

Because the program does not use any real-time input or output, it is not necessary for it to call function `stm32f4_wm5102_init()`. Figure 6.11 shows plots of the desired output `desired`, adaptive filter output `y_out`, and error `error`, plotted using MATLAB® function `stm32f4_plot_real()` after the contents of those three arrays have been saved to files by typing

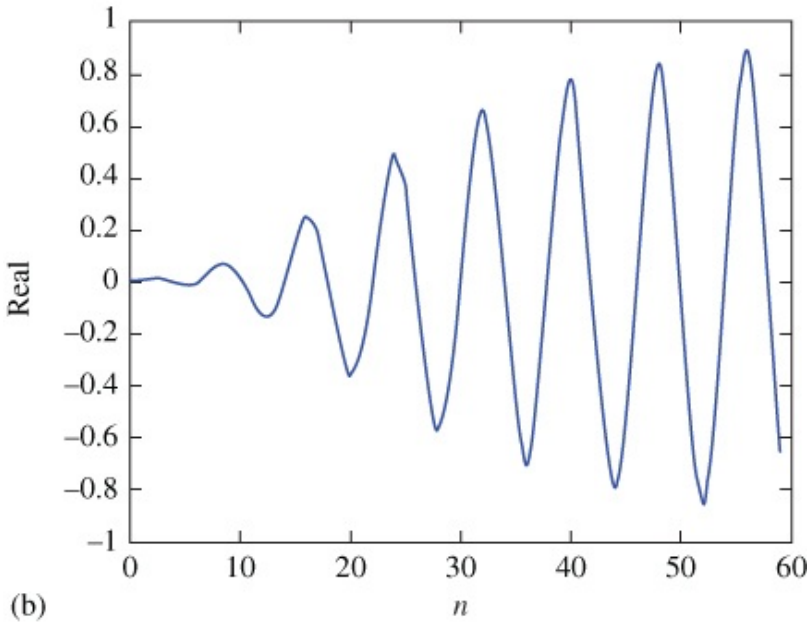
```
SAVE desired.dat <start address>, <start address + 0xF0>
SAVE y_out.dat <start address>, <start address + 0xF0>
SAVE error.dat <start address>, <start address + 0xF0>
```

in the *Command* window in the *MDK-ARM debugger*, where `start address` is the address of arrays `desired`, `y_out`, and `error`. Within 60 sampling instants, the filter output effectively

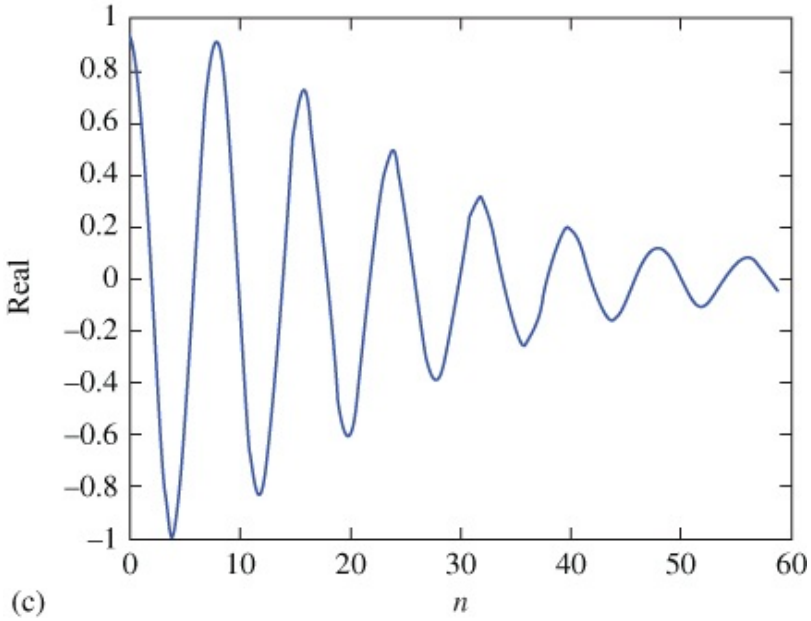
converges to the desired cosine signal. Change the adaptation or convergence rate BETA to 0.02 and verify a faster rate of adaptation and convergence.


Real
n
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−1
0
10
20
30
40
50
60

(a)


Real
n
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−1
0
10
20
30
40
50
60

(b)


Real
n
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−1
0
10
20
30
40
50
60

(c)

Figure 6.11 Plots of (a) desired output, (b) adaptive filter output, and (c) error generated using program `stm32f4_adaptive.c` and displayed using MATLAB function `stm32f4_plot_real()`.

Example 6.2

Adaptive Filter for Noise Cancellation Using Sinusoids as Inputs (`tm4c123_adaptnoise_intr.c`).

This example illustrates the use of the LMS algorithm to cancel an undesired sinusoidal noise signal. Listing 6.4 shows program `tm4c123_adaptnoise_intr.c`, which implements an adaptive FIR filter using the structure shown in Figure 6.5.

Listing 6.2 Program `tm4c123_adaptnoise_intr.c`.

```
// tm4c123_adaptnoise_intr.c
#include "tm4c123_aic3104_init.h"
#define SAMPLING_FREQ 8000
#define NOISE_FREQ 1200.0
#define SIGNAL_FREQ 2500.0
#define NOISE_AMPLITUDE 8000.0
#define SIGNAL_AMPLITUDE 8000.0
#define BETA 4E-12f                    // adaptive learning rate
#define N 10                                 // number of weights
float32_t w[N];              // adaptive filter weights
float32_t x[N];              // adaptive filter delay line
float32_t theta_increment_noise;
float32_t theta_noise = 0.0;
float32_t theta_increment_signal;
float32_t theta_signal = 0.0;
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  float32_t inputL, inputR;
  int16_t i;
  float32_t yn, error, signal, signoise, refnoise, dummy;
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  inputL = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  inputR = (float32_t)(sample_data.bit16[0]);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  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*arm_cos_f32(theta_noise));
  signoise = (NOISE_AMPLITUDE*arm_sin_f32(theta_noise));
  signal = (SIGNAL_AMPLITUDE*arm_sin_f32(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
    dummy = BETA*error;
    dummy = dummy*x[i];
    w[i] = w[i] + dummy;
    x[i] = x[i-1];
  }
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
  sample_data.bit32 = ((int16_t)(error));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  sample_data.bit32 = ((int16_t)(signal + signoise));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
}
int main()
{
  int16_t i;
  for (i=0 ; i<N ; i++)
  {
    w[i] = 0.0;
    x[i] = 0.0;
  }
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1){}
}
```

A desired sinusoid `signal`, of frequency `SIGNAL_FREQ` (2500 Hz), with an added (undesired) sinusoid `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 6.12. A sinusoid `refnoise`, with a frequency of `NOISE_FREQ` (1200 Hz), represents the reference noise signal in Figure 6.12 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 AIC3104 codec on right and left channels, respectively.

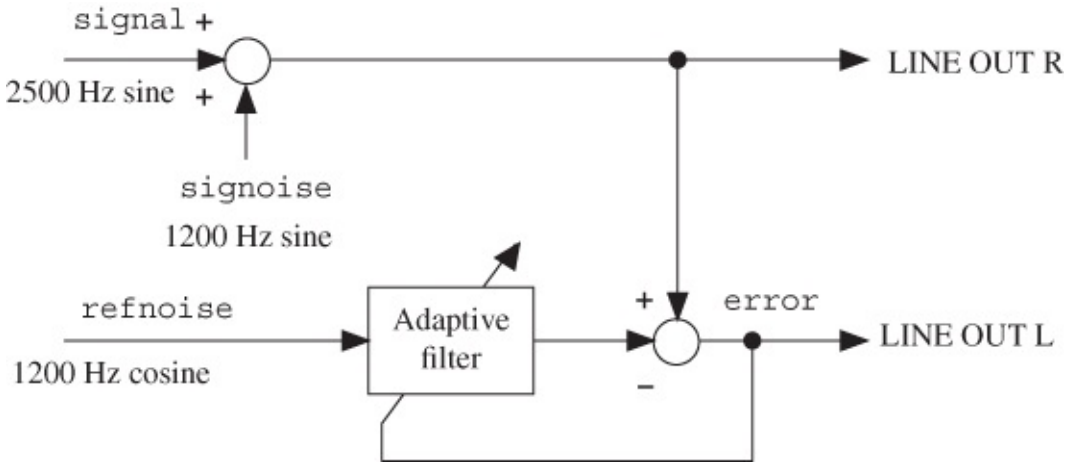


Figure 6.12 Block diagram representation of program `tm4c213_adaptnoise_intr.c`.

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. Program `tm4c213_adaptnoise_intr.c` demonstrates real-time adjustments to the coefficients of an FIR filter. The program makes use of CMSIS DSP library functions `arm_sin_f32()` and `arm_cos_f32()` in order to compute signal and noise signal values. Standard functions `sin()` and `cos()` would be computationally too expensive to use in real time. The adaptive FIR filter is implemented straightforwardly using program statements

```
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
  dummy = BETA*error;
  dummy = dummy*x[i];
  w[i] = w[i] + dummy;
  x[i] = x[i-1];
}
```

in which the entire contents of the filter delay line `x` are shifted at each sampling instant. Program `tm4c123_adaptnoise_CMSIS_intr.c` shown in Listing 6.5 makes use of the more computationally efficient CMSIS DSP library function `arm_lms_f32()`.

Listing 6.3 Program `tm4c123_adaptnoise_CMSIS_intr.c).`

```
// tm4c123_adaptnoise_CMSIS_intr.c
#include "tm4c123_aic3104_init.h"
```

```
#define SAMPLING_FREQ 8000
#define NOISE_FREQ 1200.0f
#define SIGNAL_FREQ 2500.0f
#define NOISE_AMPLITUDE 8000.0f
#define SIGNAL_AMPLITUDE 8000.0f
#define BETA 1E-13
#define NUM_TAPS 10
#define BLOCK_SIZE 1
float32_t theta_increment_noise;
float32_t theta_noise = 0.0;
float32_t theta_increment_signal;
float32_t theta_signal = 0.0;
float32_t firStateF32[BLOCK_SIZE + NUM_TAPS -1];
float32_t firCoeffs32[NUM_TAPS] = {0.0};
arm_lms_instance_f32 S;
void SSI_interrupt_routine(void)
}
  AIC3104_data_type sample_data;
  float32_t inputl, inputr;
  float32_t yout, error, signal, signoise
  float32_t refnoise, sigplusnoise;
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  inputl = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  inputr = (float32_t)(sample_data.bit16[0]);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  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*arm_cos_f32(theta_noise));
  signoise = (NOISE_AMPLITUDE*arm_sin_f32(theta_noise));
  signal = (SIGNAL_AMPLITUDE*arm_sin_f32(theta_signal));
  sigplusnoise = signoise+signal;
  arm_lms_f32(&S, &refnoise, &sigplusnoise,
              &yout, &error, 1);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
  sample_data.bit32 = ((int16_t)(error));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  sample_data.bit32 = ((int16_t)(sigplusnoise));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
}
int main()
{
  arm_lms_init_f32(&S, NUM_TAPS, firCoeffs32,
                   firStateF32, BETA, 1);
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1);
}
```

6.6.1 Using CMSIS DSP Function `arm_lms_f32()`

In order to implement an adaptive FIR filter using CMSIS DSP library function `arm_lms_f32()`, as in the previous example, the following variables must be declared.

1. A structure of type `arm_lms_instance_f32`
2. A floating-point state variable array `firStateF32`
3. A floating-point array of filter coefficients `firCoeffs32`.

The `arm_lms_instance_f32` structure comprises an integer value equal to the number of coefficients, `N` used by the filter, a floating-point value equal to the learning rate `BETA`, and pointers to the array of `N` floating-point filter coefficients and to the `N + BLOCKSIZE - 1` floating-point state variable array. The state variable array contains current and previous values of the input to the filter. Before calling function `arm_lms_f32()`, the structure is initialized using function `arm_lms_init_f32()`. This assigns values to the elements of the structure and initializes the contents of the state variable array to zero. Function `arm_lms_init_f32()` does not allocate memory for the filter coefficient or state variable arrays. They must be declared separately. Function `arm_lms_init_f32()` does not initialize or alter the contents of the filter coefficient array.

Subsequently, function `arm_lms_f32()` may be called, passing to it pointers to the `arm_lms_instance_f32` structure and to floating-point arrays of input samples, output samples, desired output samples, and error samples. Each of these arrays contains `BLOCKSIZE` samples. Each call to function `arm_lms_f32()` processes `BLOCKSIZE` samples, and although it is possible to set `BLOCKSIZE` to one, the function is optimized according to the architecture of the ARM Cortex-M4 to operate on at least four samples per call. More details of function `arm_lms_f32()` can be found in the CMSIS documentation [1].

Example 6.3

Adaptive FIR Filter for Noise Cancellation Using External Inputs (`stm32f4_noise_cancellation_CMSIS_dma.c`).

This example extends the previous one to cancel an undesired noise signal using external inputs. Program `stm32f4_noise_cancellation_CMSIS_dma.c`, shown in Listing 6.7, requires two external inputs, a desired signal, and a reference noise signal to be input to left and right channels of LINE IN, 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 in file `speechnoise.wav`. This may be played through a PC sound card and input to the audio card 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.

Listing 6.4 Program
stm32f4_noise_cancellation_CMSIS_dma.c

```
// stm32f4_noise_cancellation_CMSIS_dma.c
#include "stm32f4_wm5102_init.h"
#include "bilinear.h"
extern uint16_t pingIN[BUFSIZE], pingOUT[BUFSIZE]
extern uint16_t pongIN[BUFSIZE], pongOUT[BUFSIZE];
int rx_proc_buffer, tx_proc_buffer;
volatile int RX_buffer_full = 0;
volatile int TX_buffer_empty = 0;
#define BLOCK_SIZE 1
#define NUM_TAPS 32
float32_t beta = 1e-11;
float32_t firStateF32[BLOCK_SIZE + NUM_TAPS -1];
float32_t firCoeffs32[NUM_TAPS] = {0.0};
arm_lms_instance_f32 S;
float32_t input, signoise, wn, yn, yout, error;
float w[NUM_SECTIONS][2] = {0.0f, 0.0f}; // IIR coeffs
void DMA1_Stream3_IRQHandler()
{
  if(DMA_GetITStatus(DMA1_Stream3,DMA_IT_TCIF3))
  {
    DMA_ClearITPendingBit(DMA1_Stream3,DMA_IT_TCIF3);
    if(DMA_GetCurrentMemoryTarget(DMA1_Stream3))
      rx_proc_buffer = PING;
    else
      rx_proc_buffer = PONG;
    RX_buffer_full = 1;
  }
}
void DMA1_Stream4_IRQHandler()
{
  if(DMA_GetITStatus(DMA1_Stream4,DMA_IT_TCIF4))
  {
    DMA_ClearITPendingBit(DMA1_Stream4,DMA_IT_TCIF4);
    if(DMA_GetCurrentMemoryTarget(DMA1_Stream4))
      tx_proc_buffer = PING;
    else
      tx_proc_buffer = PONG;
    TX_buffer_empty = 1;
  }
}
void process_buffer()
{
  int i;
  uint16_t *rxbuf, *txbuf;
  float32_t refnoise, signal;
  int16_t left_in_sample, right_in_sample;
  int section;
  // determine which buffers to use
  if (rx_proc_buffer ⟩ PING) rxbuf = pingIN;
```

```
  else rxbuf = pongIN;
  if (tx_proc_buffer == PING) txbuf = pingOUT;
  else txbuf = pongOUT;
  for (i=0 ; i<(BUFSIZE/2) ; i++)
  {
    right_in_sample = *rxbuf++;
    left_in_sample = *rxbuf++;
    refnoise = (float32_t)(right_in_sample);
    signal = (float32_t)(left_in_sample);
    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;
    arm_lms_f32(&S, &refnoise, &signoise,
                &yout, &error, 1);
    *txbuf++ = (int16_t)(signoise);
    *txbuf++ = (int16_t)(error);
  }
  TX_buffer_empty = 0;
  RX_buffer_full = 0;
}
int main()
{
  arm_lms_init_f32(&S, NUM_TAPS, firCoeffs32,
                   firStateF32, beta, 1);
  stm32_wm5102_init(FS_8000_HZ,
                    WM5102_LINE_IN,
                    IO_METHOD_DMA);
  while(1)
  {
    while (!(RX_buffer_full && TX_buffer_empty)){}
    GPIO_SetBits(GPIOD, GPIO_Pin_15);
    process_buffer();
    GPIO_ResetBits(GPIOD, GPIO_Pin_15);
  }
}
```

Figure 6.13 shows the program in block diagram. Within the program, a primary noise signal, correlated to 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.

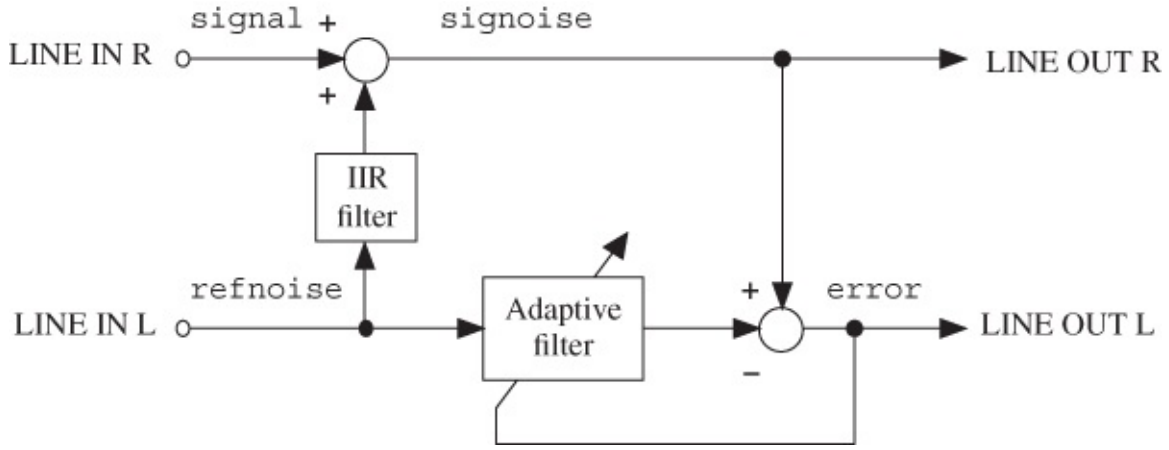


Figure 6.13 Block diagram representation of program `tm4c123_noise_cancellation_intr.c`.

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 adaptation may be very slow. This is an example of the disadvantage of the LMS algorithm versus the NLMS algorithm. After adaptation has taken place, the 32 coefficients of the adaptive FIR filter, `firCoeffs32`, may be saved to a data file by typing

```
SAVE <filename> <start address>, <end address>
```

in the *Command* window of the *MDK-ARM debugger*, where `start address` is the address of array `firCoeffs32` and `end address` is equal to `start address + 0x80`, and plotted using MATLAB function `stm32f4_logfft()`. This should reveal the time- and frequency-domain characteristics of the IIR filter implemented by the program, as identified by the adaptive filter and as shown in Figure 6.14.

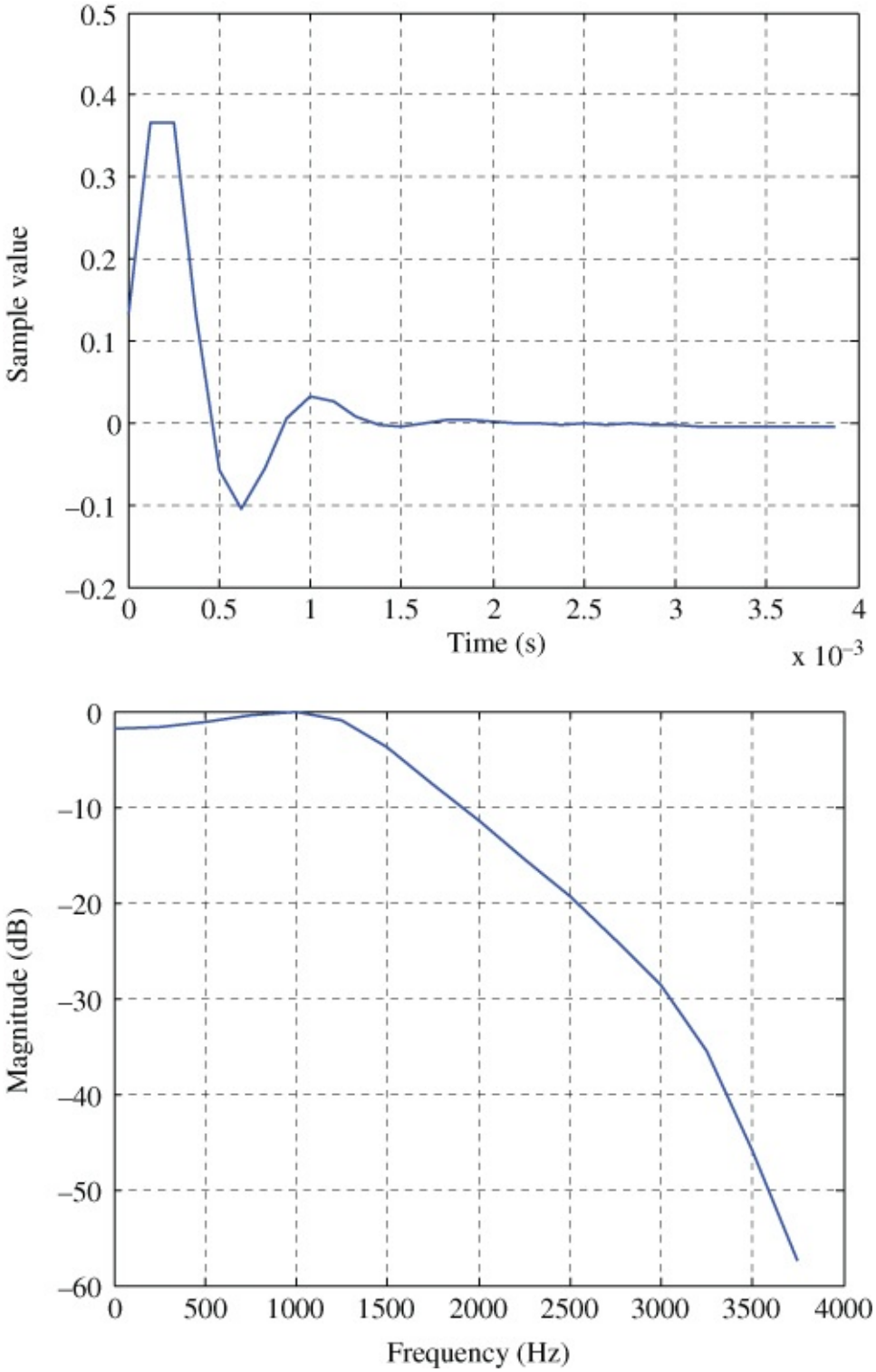


Figure 6.14 Impulse response and magnitude frequency response of IIR filter identified by the adaptive filter in program `tm4c123_noise_cancellation_intr.c` and plotted using MATLAB function `tm4c123_logfft()`.

Example 6.4

Adaptive FIR Filter for System Identification of a Fixed FIR Filter as the Unknown System `tm4c123_adaptIDFIR_CMSIS_intr.c`).

Listing 6.9 shows program `tm4c123_adaptIDFIR_CMSIS_intr.c`, 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 6.15. 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 header file `bp55.h`, previously used in Example 3.23. A 60-coefficient adaptive FIR filter is used to identify the fixed (unknown) FIR band-pass filter.

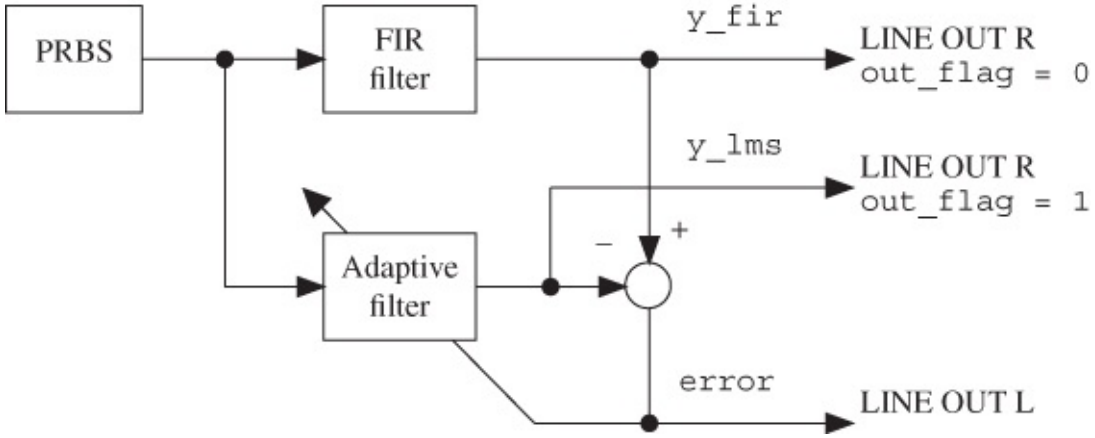


Figure 6.15 Block diagram representation of program `tm4c123_adaptIDFIR_CMSIS_intr.c`.

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.

Listing 6.5 Program `tm4c123_adaptIDFIR_CMSIS_intr.c`.

```
// tm4c123_adaptIDFIR_CMSIS_intr.c
#include "tm4c123_aic3104_init.h"
#include "bp55.h"
#define BETA 5E-13              // adaptive learning rate
#define NUM_COEFFS 60
#define BLOCK_SIZE 1
float32_t firStateF32[BLOCK_SIZE + NUM_COEFFS -1];
float32_t firCoeffs32[NUM_COEFFS] = {0.0};
arm_lms_instance_f32 S_lms;
```

```
float32_t state[N];
arm_fir_instance_f32 S_fir;
volatile int16_t out_flag = 0; // determines output
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  float32_t x_fir, y_fir;
  float32_t inputl, inputr;
  float32_t x_lms, y_lms, error;
  SSIDataGet(SSI0_BASE,&sample_data.bit32);   // input RIGHT
  inputl = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);   // input LEFT
  inputr = (float32_t)(sample_data.bit16[0]);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4); // PE2 high
  x_fir = (float32_t)(prbs(8000));
  x_lms = x_fir;
  arm_fir_f32(&S_fir, &x_fir, &y_fir, 1);
  arm_lms_f32(&S_lms, &x_lms, &y_fir, &y_lms, &error, 1);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0); // PE2 low
  if (out_flag ) 0)
    sample_data.bit32 = ((int16_t)(y_lms));
  else
    sample_data.bit32 = ((int16_t)(y_fir));
  SSIDataPut(SSI0_BASE,sample_data.bit32); // output RIGHT
  sample_data.bit32 = ((int16_t)(error));
  SSIDataPut(SSI1_BASE,sample_data.bit32); // output LEFT
  SSIIntClear(SSI0_BASE,SSI_RXFF);   // clear interrupt flag
}
int main()
{
  arm_fir_init_f32(&S_fir, N, h, state, 1);
  arm_lms_init_f32(&S_lms, NUM_COEFFS, firCoeffs32, firStateF32,
                   BETA, 1);
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1)
  {
    ROM_SysCtlDelay(10000);
    // test SWI1 closed/pressed
    if (!GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_4))
    {
      ROM_SysCtlDelay(10000);
      out_flag = (out_flag+1)%2; // toggle out_flag
      // wait until SWI not closed/pressed
      while (!GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_4)){}
    }
  }
}
```

Build, load, and run the program. The blue user pushbutton on the Discovery may be used to toggle the output between `y_fir` (the output from the fixed (unknown) FIR filter) and `y_lms`

(the output from the adaptive FIR filter) as the signal written to the right channel of LINE OUT on the audio card. `error` (the error signal) is always written to the left channel of LINE OUT. Verify that the output of the adaptive FIR filter (`y_lms`) converges to bandlimited noise similar in frequency content to the output of the fixed FIR filter (`y_fir`) and that the variance of the error signal (`error`) gradually diminishes as adaptation takes place.

Edit the program to include the coefficient file `bs55.h` (in place of `bp55.h`), which implements a 55-coefficient FIR band-stop filter centered at 2 kHz. Rebuild and run the program and verify that, after adaptation has taken place, the output of the adaptive FIR filter is almost identical to that of the FIR band-stop filter. Figure 6.16 shows the output of the program while adaptation is taking place. The upper time-domain trace shows the output of the adaptive FIR filter, the lower time-domain trace shows the error signal, and the magnitude of the FFT of the output of the adaptive FIR filter is shown below them. Increase (or decrease) the value of `beta` by a factor of 10 to observe a faster (or slower) rate of convergence. Change the number of weights (coefficients) from 60 to 40 and verify a slight degradation in the identification process. You can examine the adaptive filter coefficients stored in array `firCoeffs32` in this example by saving them to data file and using the MATLAB function `stm32f4_logfft()` to plot them in the time and frequency domains.

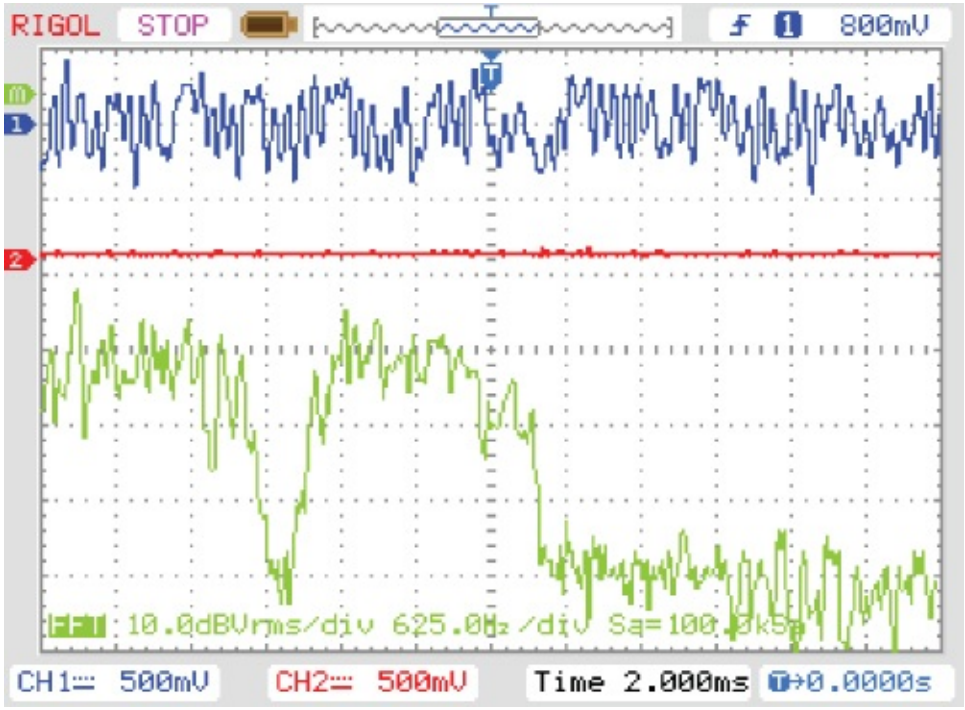
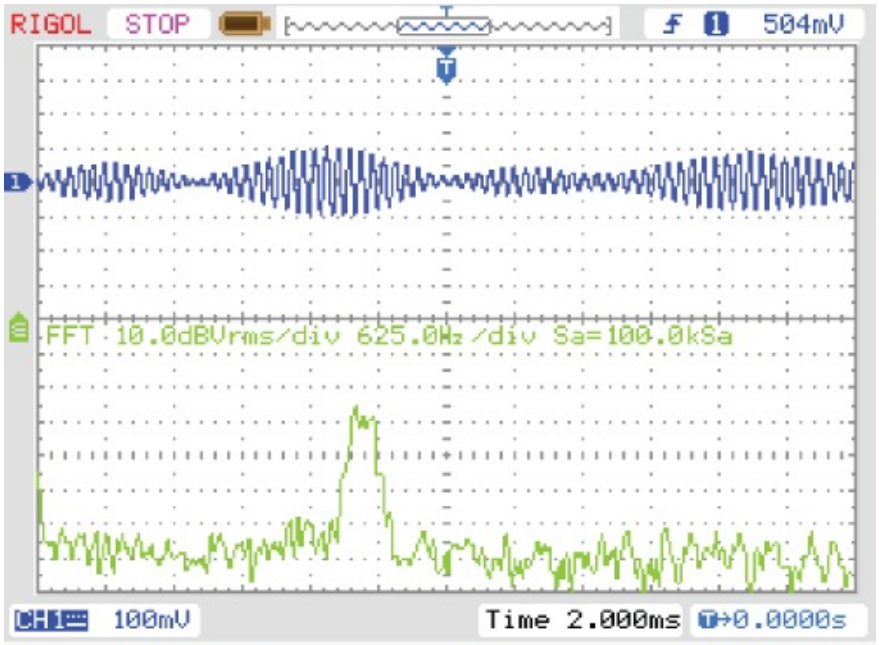


Figure 6.16 Output from program `stm32f4_adaptIDFIR_CMSIS_intr.c` using coefficient header file `bs55.h` viewed using *Rigol DS1052E* oscilloscope.

Example 6.5

Adaptive FIR Filter for System ID of a Fixed FIR as an Unknown System with Adaptive Filter Initialized as a Band-Pass Filter (`stm32f4_adaptIDFIR_CMSIS_init_intr.c`).

In this example, program `stm32f4_adaptIDFIR_CMSIS_intr.c` has been modified slightly in order to create program `stm32f4_adaptIDFIR_init_intr.c`. This program initializes the weights, `firCoeffs32`, 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 `adaptIDFIR_CMSIS_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 on frequency 2 kHz. The adaptation process is illustrated in Figure 6.17, which shows the frequency content of the output of the adaptive filter at different stages in the adaptation process.


RIGOL
STOP
504mV
FFT 10.0dBVrms/div 625.0Hz/div Sa=100.0kSa
CH1 100mV
Time 2.000ms
T→0.0000s


RIGOL
STOP
504mV
FFT 10.0dBVrms/div 625.0Hz/div Sa=100.0kSa
CH1 100mV
Time 2.000ms
T→0.0000s

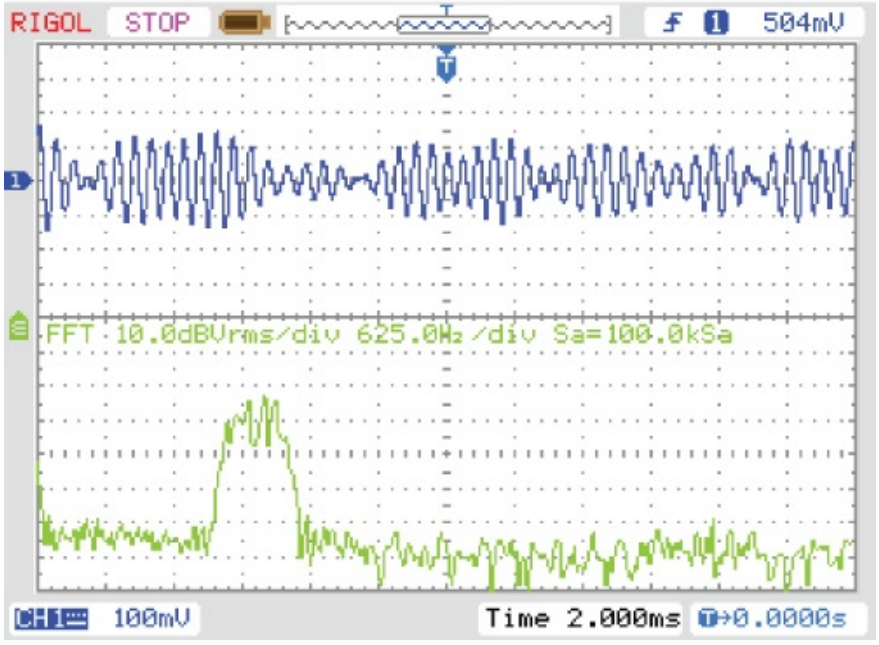

RIGOL
STOP
504mV
FFT 10.0dBVrms/div 625.0Hz/div Sa=100.0kSa
CH1 100mV
Time 2.000ms
T→0.0000s

Figure 6.17 Output from adaptive filter in program `tm4c123_adaptIDFIR_CMSIS_init_intr.c`.

As in most of the example programs in this chapter, the rate of adaptation has been set very low.

Example 6.6

Adaptive FIR for System ID of Fixed IIR as an Unknown System (`tm4c123_iirsosadapt_CMSIS_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 `tm4c123_iirsosadapt_CMSIS_intr.c`, shown in Listing 6.12 combines programs `tm4c123_iirsos_intr.c` (Example 4.2) and `tm4c123_adaptIDFIR_CMSIS_intr.c` in order to illustrate this (Figure 6.18). The IIR filter coefficients used are those of a fourth-order low-pass elliptic filter (see Example 4.9) and are read from file `elliptic.h`. 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 on the audio booster pack gives an indication of the progress of the adaptation process. Figures 6.19 and 6.20 show the output of the adaptive filter (displayed using the FFT function of a *Rigol DS1052E* oscilloscope) and the magnitude FFT of the coefficients (weights) of the adaptive FIR filter saved to a data file and displayed using MATLAB function `tm4c123_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 using program `tm4c123_iirsosdelta_intr.c` in Example 4.10.

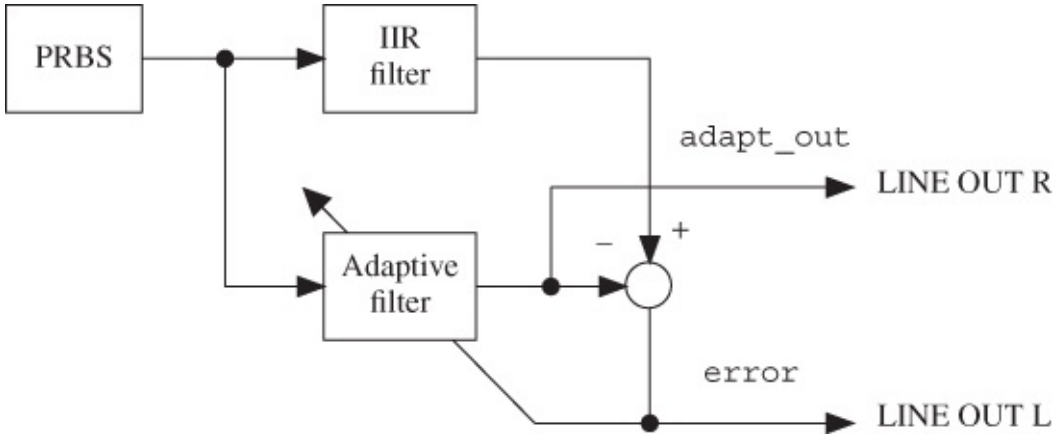


Figure 6.18 Block diagram representation of program `tm4c123_iirsosadapt_CMSIS_intr.c`.

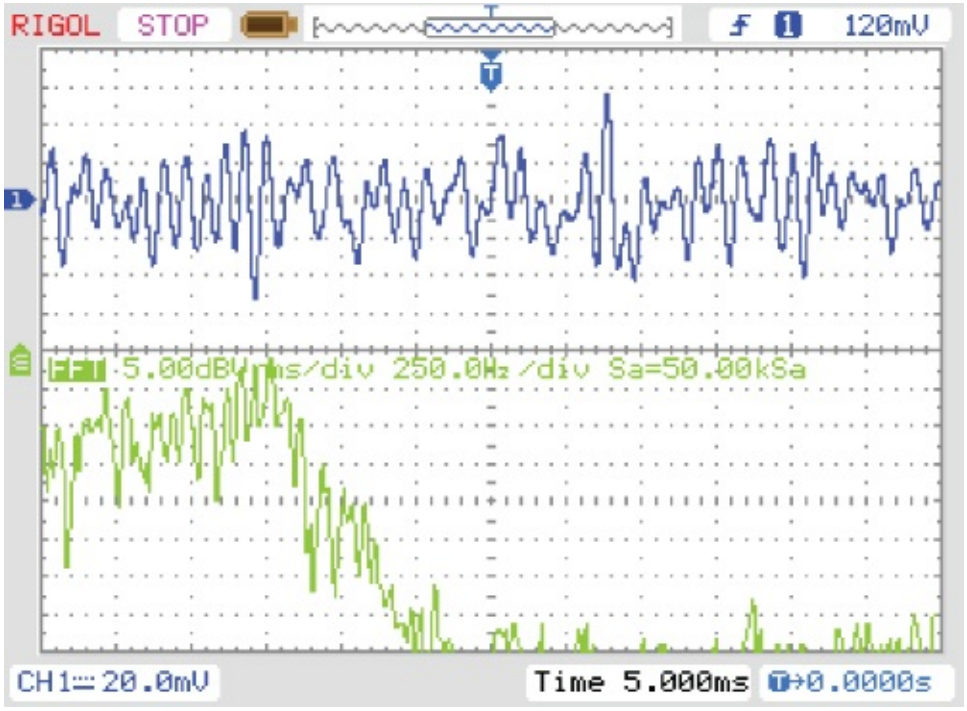


Figure 6.19 Output from adaptive filter in program `tm4c123_iirsosadapt_CMSIS_intr.c` viewed using a *Rigol DS1052E* oscilloscope.

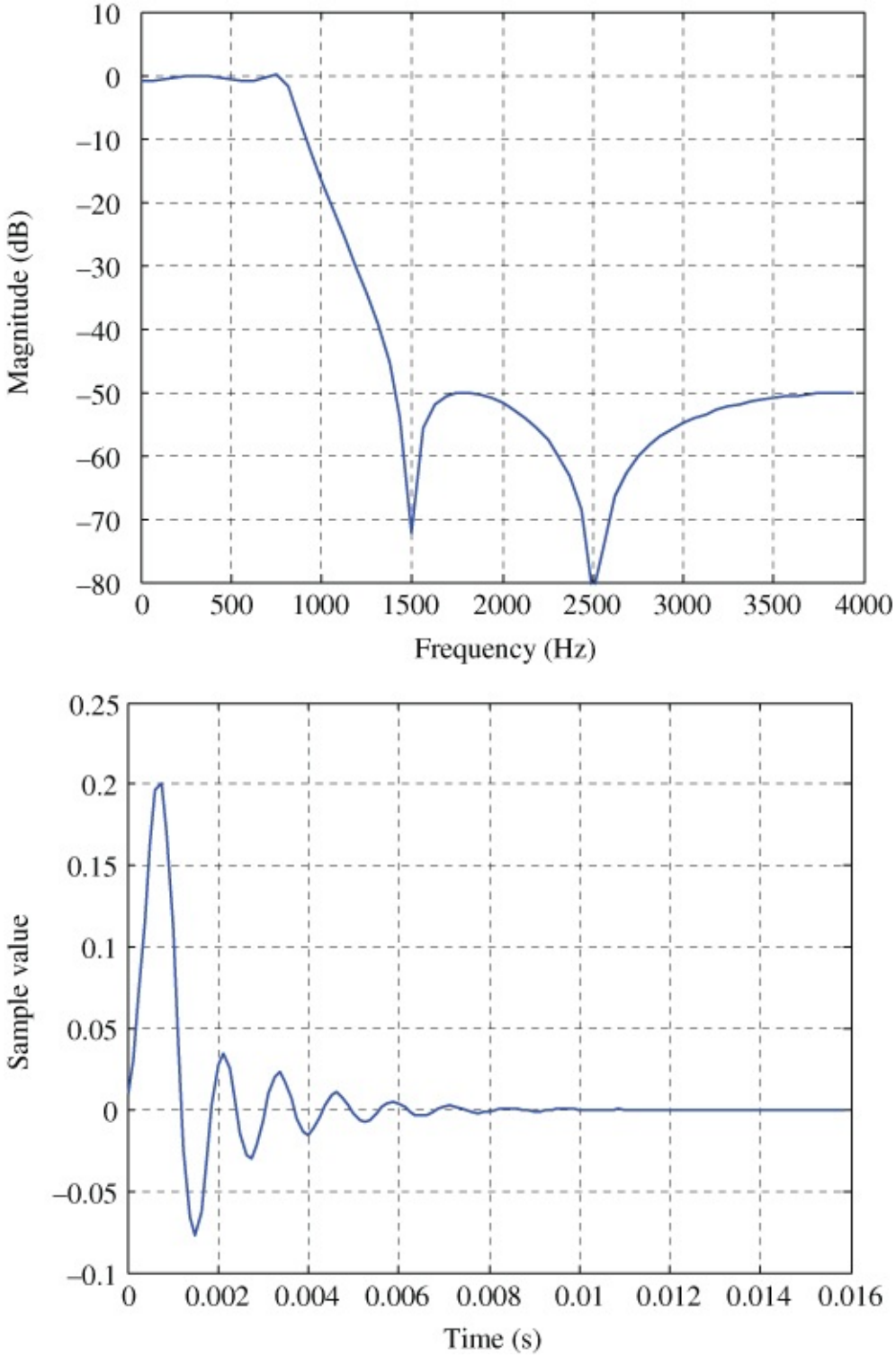


Figure 6.20 Adaptive filter coefficients from program `tm4c123_iirsosadapt_CMSIS_intr.c` plotted using MATLAB function `tm4c123_logfft()`.

Listing 6.6 Program `tm4c123_iirsosadapt_CMSIS_intr.c`.

```
// tm4c123_iirsosadapt_CMSIS_intr.c
```

```
#include "tm4c123_aic3104_init.h"
#include "elliptic.h"
#define BETA 1E-12
#define NUM_TAPS 128
#define BLOCK_SIZE 1
float32_t firStateF32[BLOCK_SIZE + NUM_TAPS -1];
float32_t firCoeffs32[NUM_TAPS] = {0.0};
arm_lms_instance_f32 S;
float32_t w[NUM_SECTIONS][2] = {0.0f, 0.0f};
void SSI_interrupt_routine(void)
{
  AIC3104_data_type sample_data;
  float32_t adapt_in, adapt_out, error;
  float32_t iir_in, wn, iir_out, input;
  int16_t section;
  SSIDataGet(SSI0_BASE,&sample_data.bit32);
  input = (float32_t)(sample_data.bit16[0]);
  SSIDataGet(SSI1_BASE,&sample_data.bit32);
  input = (float32_t)(sample_data.bit16[0]);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
  iir_in = (float32_t)(prbs(8000));
          adapt_in = iir_in;
          for (section=0 ; section<NUM_SECTIONS ; section++)
  {
    wn = iir_in - a[section][1]*w[section][0]
                - a[section][2]*w[section][1];
    iir_out = 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;
    iir_in = iir_out;
  }
  arm_lms_f32(&S, &adapt_in, &iir_out, &adapt_out,
              &error, 1);
  GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
  sample_data.bit32 = ((int16_t)(adapt_out));
  SSIDataPut(SSI0_BASE,sample_data.bit32);
  sample_data.bit32 = ((int16_t)(error));
  SSIDataPut(SSI1_BASE,sample_data.bit32);
  SSIIntClear(SSI0_BASE,SSI_RXFF);
}
int main()
{
  arm_lms_init_f32(&S, NUM_TAPS, firCoeffs32,
                   firStateF32, BETA, 1);
  tm4c123_aic3104_init(FS_8000_HZ,
                       AIC3104_LINE_IN,
                       IO_METHOD_INTR,
                       PGA_GAIN_6_DB);
  while(1);
}
```

Example 6.7

Adaptive FIR Filter for System Identification of an External System (`tm4c123_sysid_CMSIS_intr.c`).

Program `tm4c123_sysid_CMSIS_intr.c`, introduced in Chapter 2, extends the previous examples to allow the identification of an external system, connected between the LINE OUT and LINE IN sockets of the audio booster pack. In Example 3.17, program `tm4c123_sysid_CMSIS_intr.c` was used to identify the characteristics of a moving average filter implemented using a second TM4C123 LaunchPad and audio booster pack. Alternatively, a purely analog system or a filter implemented using different DSP hardware can be connected between LINE OUT and LINE IN and its characteristics identified. Connect two systems as shown in Figure 6.21. Load and run `tm4c123_iirsos_intr.c`, including coefficient header file `elliptic.h` on the first. Run program `tm4c123_sysid_CMSIS_intr.c` on the second. Halt program `tm4c123_sysid_CMSIS_intr.c` after a few seconds and save the 256 adaptive filter coefficients `firCoeffs32` to a data file. You can plot the saved coefficients using MATLAB function `tm4c123_logfft()`. Figure 6.22 shows typical results.

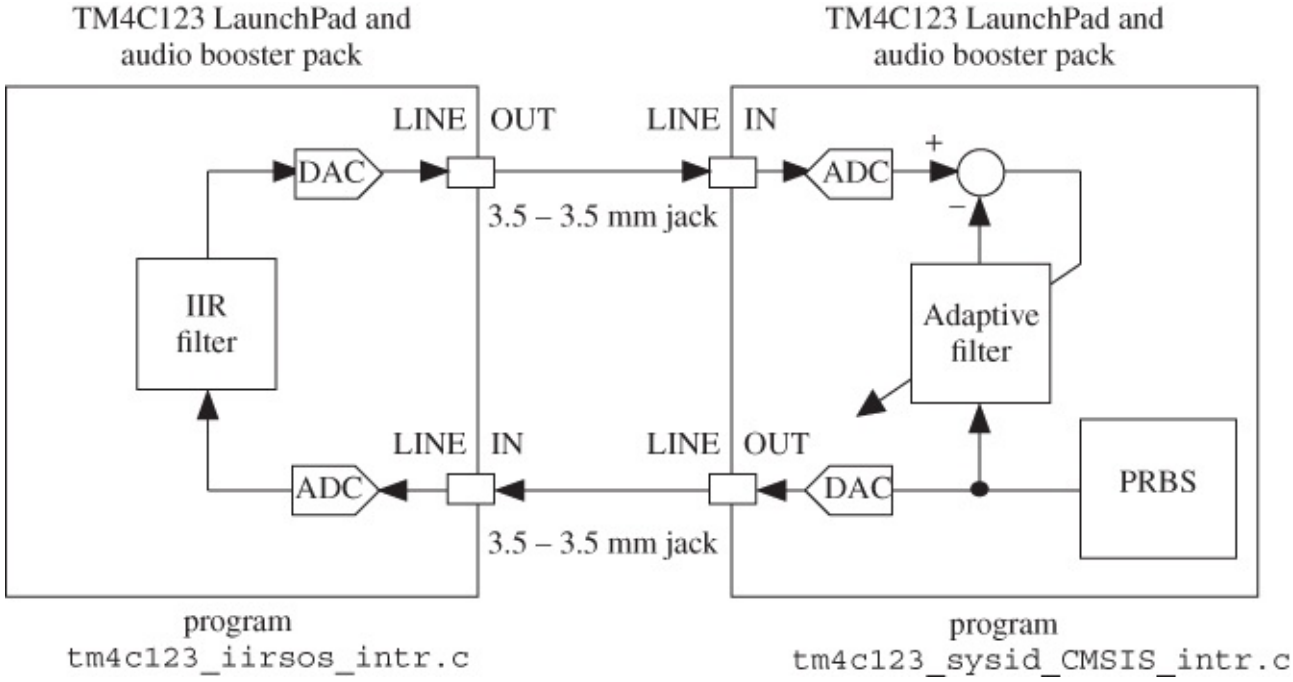


Figure 6.21 Connection diagram for program `tm4c123_sysid_CMSIS_intr.c` in Example 6.14.

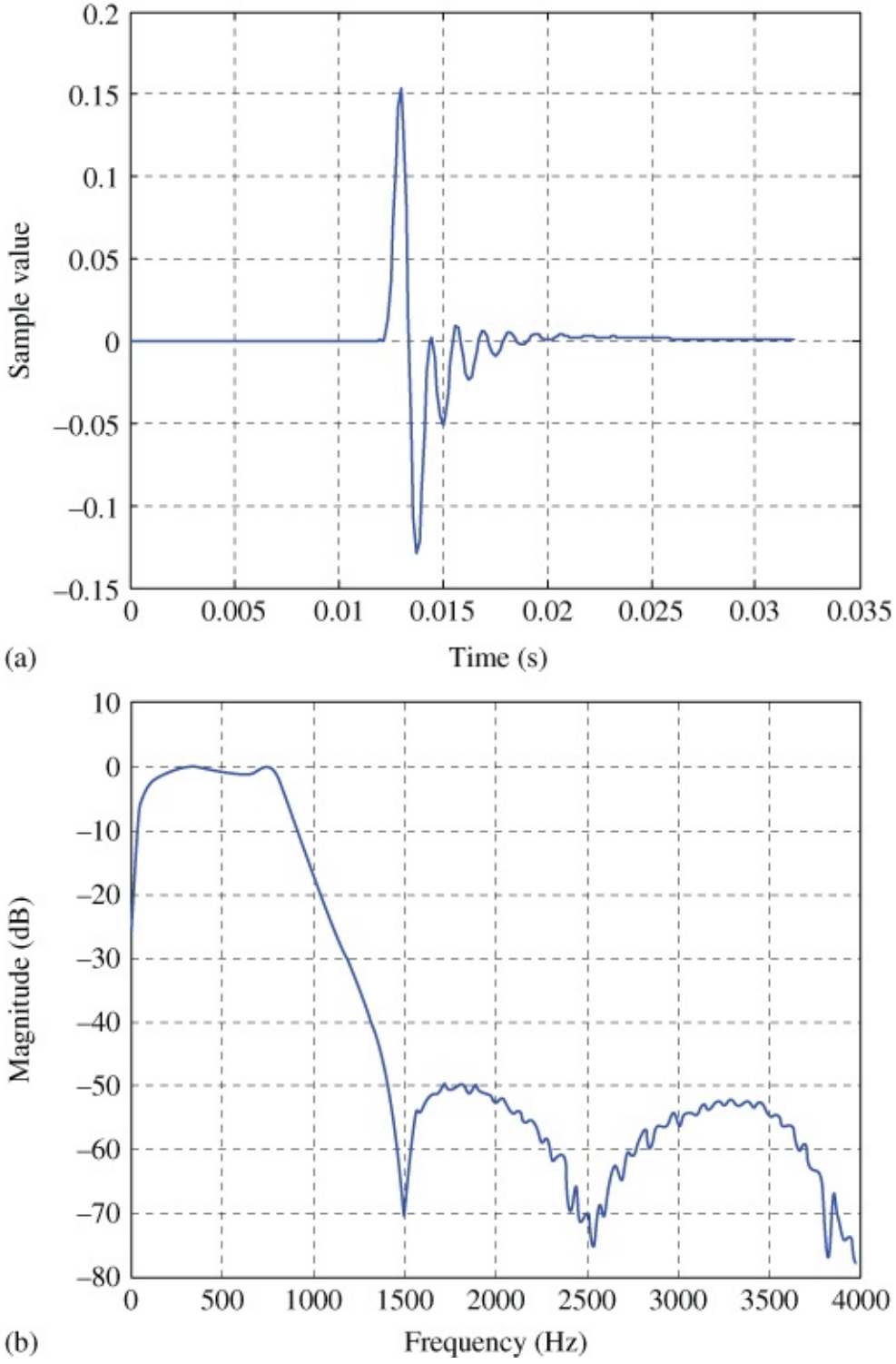


Figure 6.22 Adaptive filter coefficients from program `tm4c123_sysid_CMSIS_intr.c` plotted using MATLAB function `tm4c123_logfft()`

A number of features of the plots shown in Figure 6.22 are worthy of comment. Compare the magnitude frequency response in Figure 6.22(b) with that in Figure 6.20(b). The characteristics of 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 `tm4c123_sysid_CMSIS_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 6.20(b). Compare the impulse response in Figure 6.22(b) with that in Figure 6.20(b). Apart from its slightly different form, corresponding to the roll-off in its magnitude frequency response, there is an additional delay of approximately 12 ms.

Example 6.8

Adaptive FIR Filter for System Identification of an External System Using DMA-Based I/O (`tm4c123_sysid_CMSIS_dma.c`).

Functionally, program `tm4c123_sysid_CMSIS_dma.c` (Listing 6.15) is similar to program `tm4c123_sysid_CMSIS_intr.c`. Both programs use an adaptive FIR filter, implemented using CMSIS DSP library function `arm_lms_f32()`, to identify the impulse response of a system connected between LINE OUT and LINE IN connections to the audio booster pack. However, program `tm4c123_sysid_CMSIS_dma.c` is more computationally efficient and can run at higher sampling rates. Because function `arm_lms_f32()` is optimized to process 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 path identified, and this is evident in the examples of successfully adapted weights shown in Figure 6.23. In each case, the adaptive filter used 256 weights. The program has been run successfully using up to 512 weights at a sampling rate of 8 kHz on the TM4C123 LaunchPad with a processor clock rate of 84 MHz.

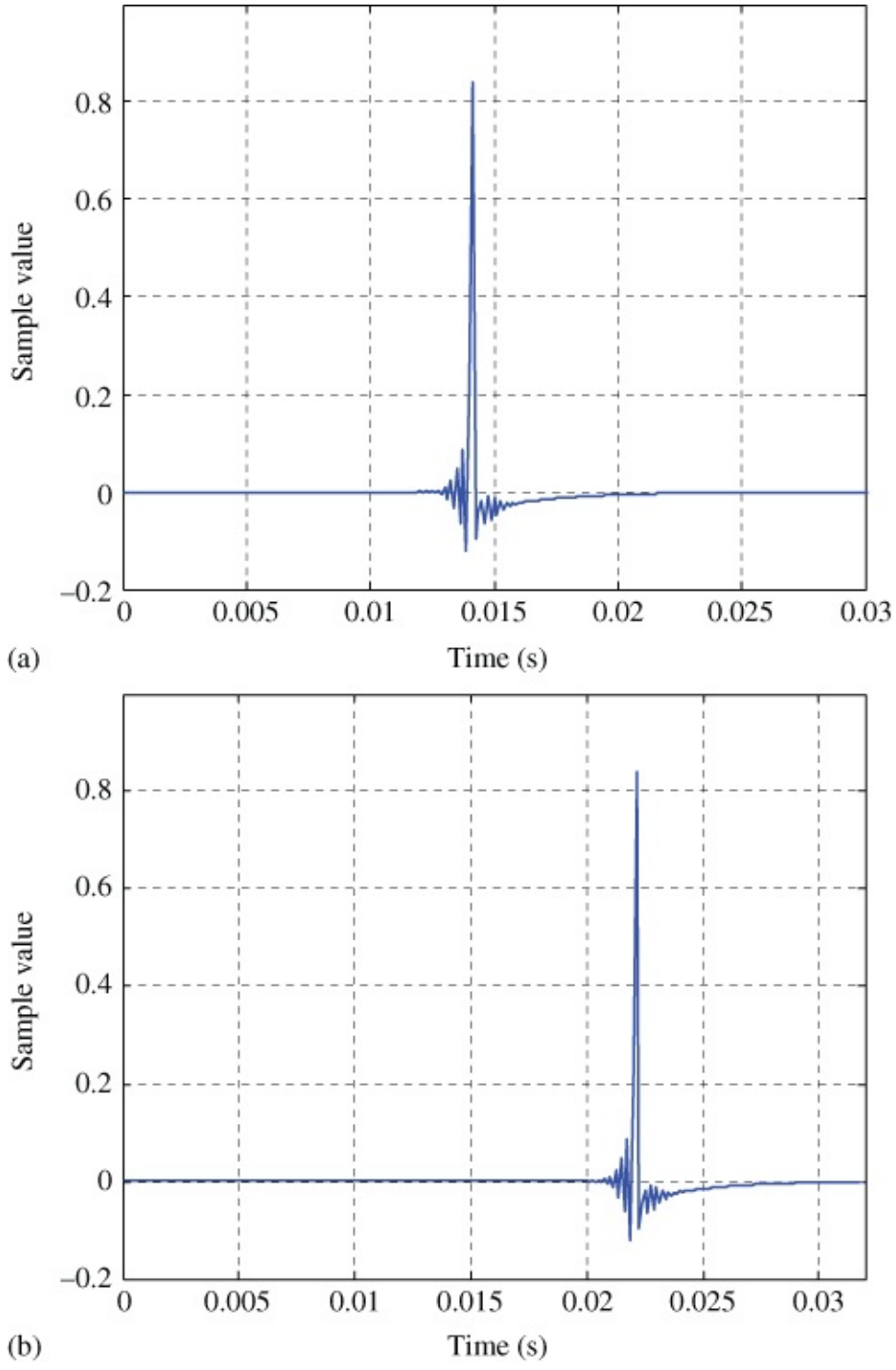


Figure 6.23 Adaptive filter coefficients from program `tm4c123_sysid_CMSIS_dma.c` plotted using MATLAB function `tm4c123_logfft()`. (a) `BUFSIZE` = 32 (b) `BUFSIZE` = 64.

Listing 6.7 Program `tm4c123_sysid_CMSIS_dma.c`.

```
// tm4c123_sysid_CMSIS_dma.c
#include "tm4c123_aic3104_init.h"
extern int16_t LpingIN[BUFSIZE], LpingOUT[BUFSIZE];
extern int16_t LpongIN[BUFSIZE], LpongOUT[BUFSIZE];
```

```
extern int16_t RpingIN[BUFSIZE], RpingOUT[BUFSIZE];
extern int16_t RpongIN[BUFSIZE], RpongOUT[BUFSIZE];
extern int16_t Lprocbuffer, Rprocbuffer;
extern volatile int16_t Lbuffer_full, Rbuffer_full;
extern volatile int16_t LTxcomplete, LRxcomplete,
extern volatile int16_t RTxcomplete, RRxcomplete;
#define BETA 1E-12         // adaptive learning rate
#define NUM_TAPS 256
float32_t firStateF32[BUFSIZE + NUM_TAPS -1];
float32_t firCoeffs32[NUM_TAPS] = {0.0};
arm_lms_instance_f32 S;
void Lprocess_buffer(void)
{
  float32_t adapt_in[BUFSIZE], adapt_out[BUFSIZE]
  float32_t desired[BUFSIZE], error[BUFSIZE];
  int16_t *inBuf, *outBuf;    // temp buffer pointers
  int i;
  if (Lprocbuffer ⌡ PING)    // use ping or pong buffers
  { inBuf = LpingIN; outBuf = LpingOUT; }
  if (Lprocbuffer ⌡ PONG)
  { inBuf = LpongIN; outBuf = LpongOUT;}
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    adapt_in[i] = (float32_t)(prbs(8000));
    desired[i] = (float32_t)(*inBuf++);
  }
    arm_lms_f32(&S, adapt_in, desired,
                adapt_out, error, BUFSIZE);
  for (i = 0; i < (BUFSIZE) ; i++)
  {
    *outBuf++ = (int16_t)(adapt_in[i]);
  }
  LTxcomplete = 0;
  LRxcomplete = 0;
  return;
}
void Rprocess_buffer(void)
{
  int16_t *inBuf, *outBuf;    //  temp buffer pointers
  int i;
  if (Rprocbuffer ⌡ PING)    // use ping or pong buffers
  { inBuf = RpingIN; outBuf = RpingOUT; }
  if (Rprocbuffer ⌡ PONG)
  { inBuf = RpongIN; outBuf = RpongOUT;}
  for (i = 0; i < (BUFSIZE) ; i++) *outBuf++ = (int16_t)(0);
  RTxcomplete = 0;
  RRxcomplete = 0;
  return;
}
void SSI_interrupt_routine(void){while(1){}}
int main(void)
{
  arm_lms_init_f32(&S, NUM_TAPS, firCoeffs32, firStateF32,
                   BETA, BUFSIZE);
  tm4c123_aic3104_init(FS_8000_HZ,
```

```
                        AIC3104_LINE_IN,
                        IO_METHOD_DMA,
                        PGA_GAIN_6_DB);
  while (1)
  {
    while((!RTxcomplete)|(!RRxcomplete));
    Rprocess_buffer();
    while((!LTxcomplete)|(!LRxcomplete));
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 4);
    Lprocess_buffer();
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
  }
}
```

Index

stm32f4_dftw.c

stm32f4_dimpulse_DAC12_intr.c

stm32f4_dimpulse_intr.c

stm32f4_fft.c

stm32f4_fft_CMSIS.c

stm32f4_fft128_dma.c

stm32f4_fir_coeffs.m

stm32f4_fir_dma.c

stm32f4_fir_intr.c

stm32f4_fir_prbs_buf_intr.c

stm32f4_fir_prbs_CMSIS_dma.c

stm32f4_iirsos_intr.c

stm32f4_iirsostr_intr.c

stm32f4_logfft.m

stm32f4_loop_buf_intr.c

stm32f4_loop_dma.c

stm32f4_loop_intr.c

stm32f4_loop_poll.c

stm32f4_noise_cancellation_ CMSIS_dma.c

stm32f4_plot_complex.m

stm32f4_plot_real.m

stm32f4_prbs_DAC12_intr.c

stm32f4_sine_intr.c

stm32f4_sine48_intr.c

stm32f4_sine8_intr.c

stm32f4_sine8_DAC12_intr.c

stm32f4_sinegenDE_intr.c

stm32f4_sinegenDTMF_intr.c

stm32f4_square_DAC12_intr.c

stm32f4_square_intr.c

stm32f4_sweep_intr.c

stm32f4_sweepDE_intr.c

System identification

of codec antialiasing and reconstruction filters

of FIR filter

of IIR filter

of moving average filter

tm4c123_adaptIDFIR_CMSIS_intr.c

tm4c123_adaptnoise_CMSIS_intr.c

tm4c123_adaptnoise_intr.c

tm4c123_aic3104_biquad.m

tm4c123_aliasing_intr.c

tm4c123_AM_poll.c,

tm4c123_delay_intr.c

tm4c123_dft128_dma.c

tm4c123_dimpulse_intr.c

tm4c123_echo_intr.c

tm4c123_fastconv_dma.c

tm4c123_fft128_CMSIS_dma.c

tm4c123_fft128_sinetable_dma.c

tm4c123_fir_dma.c

tm4c123_fir_intr.c

tm4c123_fir_prbs_CMSIS_dma.c

tm4c123_fir_prbs_intr.c

tm4c123_fir3lp_intr.c

tm4c123_fir3ways_intr.c

tm4c123_fir4types_intr.c

tm4c123_fir_coeffs.m

tm4c123_flanger_dimpulse_intr.c

tm4c123_flanger_intr.c

tm4c123_graphicEQ_dma.c

tm4c123_iirsos_CMSIS_intr.c

tm4c123_iirsos_coeffs.m

tm4c123_iirsos_delta_intr.c

tm4c123_iirsos_intr.c

tm4c123_iirsos_prbs_intr.c

tm4c123_iirsosadapt_CMSIS_intr.c

tm4c123_logfft.m

tm4c123_loop_buf_intr.c

tm4c123_loop_dma.c

tm4c123_loop_intr.c

tm4c123_loop_poll.c

tm4c123_notch2_intr.c

tm4c123_prandom_intr.c

tm4c123_prbs_biquad_intr.c

tm4c123_prbs_deemph_intr.c

tm4c123_prbs_prbs_intr.c

tm4c123_prbs_intr.c

tm4c123_ramp_intr.c

tm4c123_scrambler_intr.c

tm4c123_sine48_intr.c

tm4c123_sine48_loop_intr.c

tm4c123_sineDTMF_intr.c

tm4c123_square_1kHz_intr.c

tm4c123_square_intr.c

tm4c123_sysid_average_CMSIS_ intr.c

tm4c123_sysid_biquad_intr.c

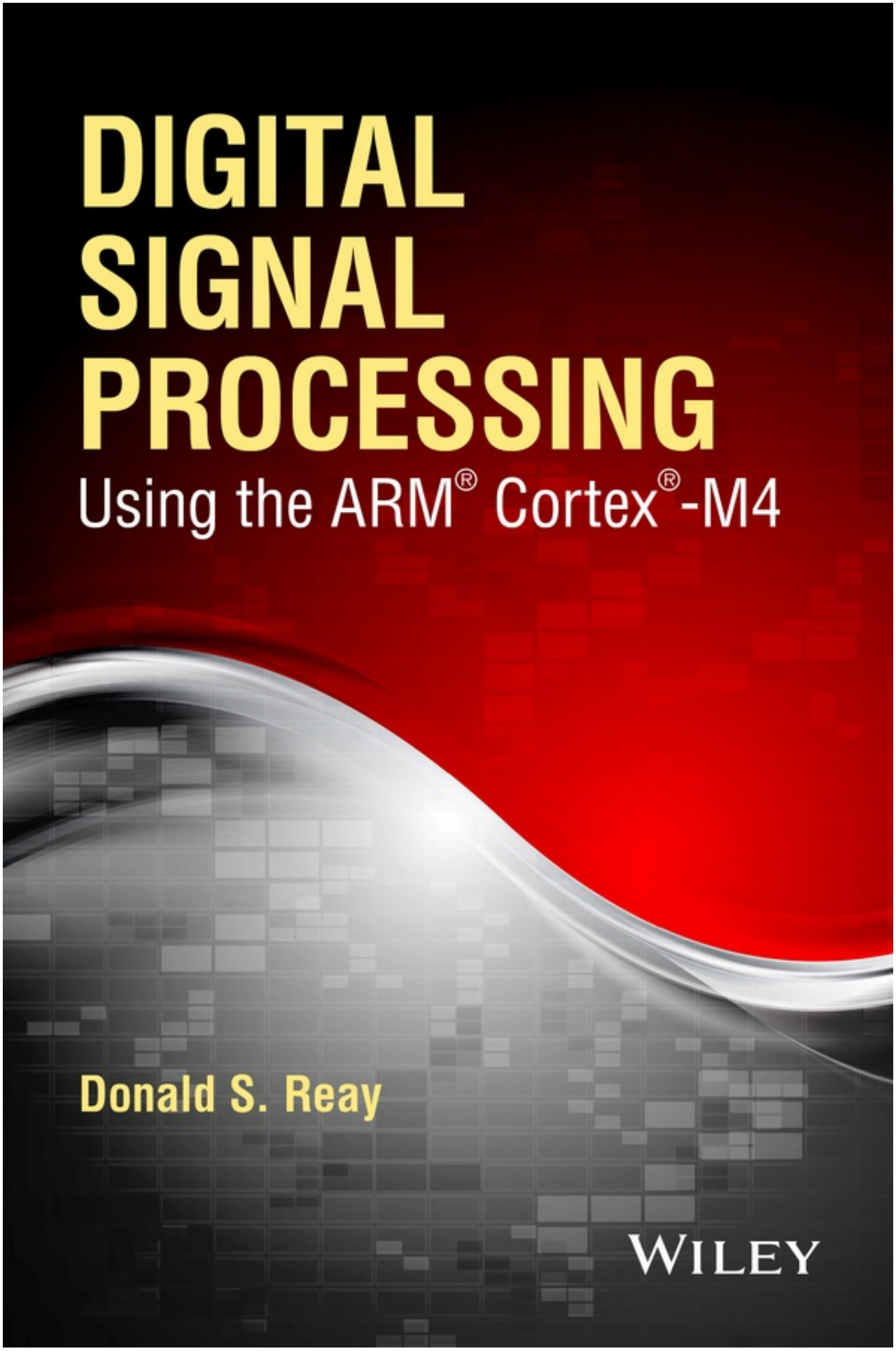
tm4c123_sysid_CMSIS_dma.c

tm4c123_sysid_CMSIS_intr.c

tm4c123_sysid_deemph_CMSIS_intr.c

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.


DIGITAL
SIGNAL
PROCESSING
Using the ARM® Cortex®-M4
Donald S. Reay
WILEY