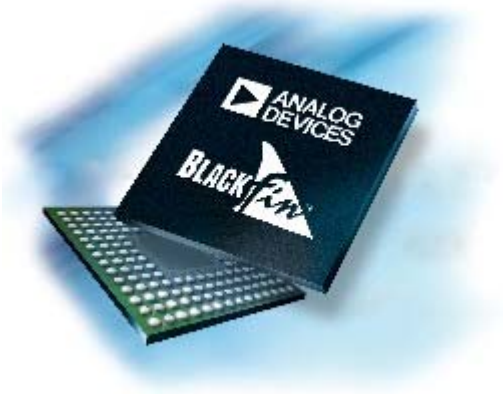
LabVIEW Experiments and Appendix Accompanying

Embedded Signal Processing with the Micro Signal Architecture

By Dr. Woon-Seng S. Gan, Dr. Sen M. Kuo



National Instruments Contributors
By Erik Luther, Jim Cahow, Mark Kaschner, Matthew Pollock, Nitin Thomas

Information on Purchasing This Book Is Available Here

(Printed June 29, 2006)

Contents

1.5 System-Level Design using a Graphical Development Environment 4
1.5.1 Setting up the LabVIEW Embedded Module for ADI Blackfin Processors 4
Hands-on Experiment 1.4: 5
Hands-on Experiment 2.5: 6
Hands-on Experiment 2.6: 7
Hands-on Experiment 3.8: 10
Hands-on Experiment 3.9: 12
Hands-on Experiment 4.6: 15
Hands-on Experiment 4.7: 17
5.6 Implementation of Graphic Equalizer using the LabVIEW Embedded Module for ADI Blackfin 18
Hands-on Experiment 5.10: 18
Hands-on Experiment 5.11: 20
6.7 Implementation of IIR Filter-based Graphic Equalizer using the LabVIEW Embedded Module for ADI Blackfin Processors 22
Hands-on Experiment 6.18: 22
Hands-on Experiment 6.19: 23
7.8 Signal Generation using the LabVIEW Embedded Module for ADI Blackfin Processors 26
Hands-on Experiment 7.11: 27
Hands-on Experiment 7.12: 29
8.10 Sample-Rate Conversion using LabVIEW Embedded Module for ADI Blackfin Processors 30
Hands-on Experiment 8.18: 31
Hands-on Experiment 8.19: 33
9.6 Implementation of MDCT using LabVIEW Embedded Module for ADI Blackfin Processors 35
Hands-on Experiment 9.4: 35
Hands-on Experiment 9.5: 37
10.11 Image Processing using the LabVIEW Embedded Module for ADI Blackfin Processors 39
Hands-on Experiment 10.7: 39
Hands-on Experiment 10.8: 42
Appendix A: An Introduction to Graphical Programming with LabVIEW 44
Contributed by National Instruments 44
A.1 What is LabVIEW? 44
A.1.1 A Picture Is Worth a Thousand Lines of Code 44
A.1.2 Getting Started with LabVIEW 45
A.1.3 Install Software 45
A.1.4 Activating Software 46
A.1.5 Connecting Hardware 46

A.1.6 Running an Example Program 46
A.2 Overview of LabVIEW 47
A.2.1 The LabVIEW Environment 47
A.2.2 Front Panel 48
A.2.3 Block Diagram 48
A.2.4 Debugging in LabVIEW (Windows) 49
A.2.5 Help 50
Hands-on Experiment A.1: 50
Hands-on Experiment A.2: 53
A.3 Introduction to the LabVIEW Embedded Module for ADI Blackfin Processors 55
A.3.1 What is the LabVIEW Embedded Module for ADI Blackfin Processors? 55
A.3.2 Targeting the Blackfin Processor 55
A.3.3 Build Options 57
A.3.4 Target Configuration 59
A.3.6 Debugging in the LabVIEW Embedded Module for ADI Blackfin Processors 60
A.3.7 Non-instrumented Debugging 60
A.3.8 Instrumented Debugging 61
Hands-on Experiment A.3: 61
Hands-on Experiment A.4: 63

1.5 System-Level Design using a Graphical Development Environment

Graphical development environments, such as National Instruments LabVIEW, are effective means to fast prototype and deploy developments from individual algorithms to full system-level designs onto embedded processors. The graphical data-flow paradigm which is used to create LabVIEW programs or virtual instruments (VIs), allows for rapid, intuitive development of embedded code. This is due to its flow-chart-like syntax and inherent ease for implementing parallel tasks.

In this and subsequent sections included at the end of each chapter, we will present a common design cycle that engineers are using to reduce product development time by effectively integrating the software tools they use on the desktop for deployment and testing. This will primarily be done using the LabVIEW Embedded Development Module for the ADI Blackfin BF537 processor developed by National Instruments and ADI, which is an add-on module for LabVIEW to target and deploy to the Blackfin processor. Other LabVIEW add-ons, such as the Digital Filter Design Toolkit, may also be discussed.

Embedded system developers are frequently using computer simulation and design tools such as LabVIEW and the Digital Filter Design Toolkit to quickly develop a system or algorithm for the needs of their project. Next, the developer can leverage their simulated work on the desktop by rapidly deploying that same design using the LabVIEW Embedded Module for ADI Blackfin Processors, and continue to iterate on that design until the design meets the design specifications. Once the design has been completed, many developers will then re-code that design using VisualDSP++ for the most efficient implementation. Therefore, knowledge of the processor architectures and its C/assembly programming is still important for a successful implementation. This book provides balanced coverage of both high-level programming using graphical development environment and conventional C/assembly programming using VisualDSP++.

In the first example using this graphical design cycle, we will demonstrate the implementation and deployment of the dot product algorithm presented in Hands-on Experiment 1.1 using LabVIEW and LabVIEW Embedded Module for ADI Blackfin Processors.

1.5.1 Setting up the LabVIEW Embedded Module for ADI Blackfin Processors

LabVIEW Embedded Module for ADI Blackfin Processors (trial version) can be downloaded from the book companion website. A brief tutorial on using these software tools is included in Appendix A of this book. Once installed, double click on **National Instruments LabVIEW 8.5** under the **All Programs** panel. This will bring up the **LabVIEW 8.5 Getting Started** Window.

The following hands-on experiment provides an introduction of the LabVIEW Embedded Development Module to explore concepts from the previous hands-on experiments.

Hands-on Experiment 1.4:

This exercise introduces the NI LabVIEW Embedded Module for ADI Blackfin Processors and the process for deploying graphical code on the Blackfin processor for rapid prototyping and verification. The dot product application was created using the same vector values used previously in the VisualDSP++ project file, `exp1_1.dpj`. This experiment utilizes arrays, functions, and the Inline C node within LabVIEW.

Open the project file `DotProd – BF537.lvproj` located in the directory `c:\adsp\chap1\exp1_4`. Next, double click on `DotProd_BF.vi` from within the project window to open the front panel. The front panel is the graphical user interface (GUI) which contains the inputs and outputs of the program as shown in Figure 1.18. Select **View → Block Diagram** to switch to the LabVIEW block diagram as shown in Figure 1.19, which contains the source code of the program. The dot product is implemented by passing two vectors (or one-dimensional arrays) to the **Dot Product** function. The result is then displayed in the **Dot Product Result** indicator. The value is also passed to the standard output buffer, because controls and indicators are only available in JTAG debug or instrumented debug modes. The graphical LabVIEW code executes based on the principal of data flow, in this case, from left to right.

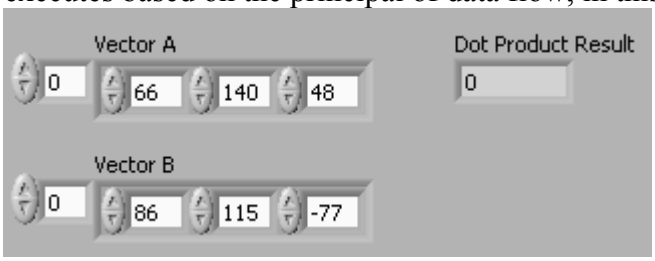


Figure 1.18 Front panel of `DotProd.vi`.

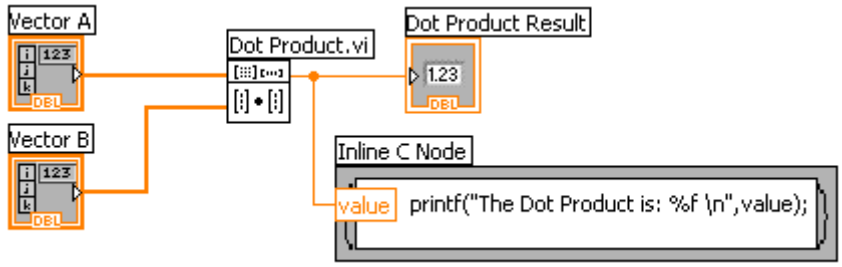


Figure 1.19 Block diagram of `DotProd.vi`.

This graphical approach to programming makes this program simple to implement and self-documenting, which is especially helpful for large scale applications. Also note the use of the Inline C Node, which allows users to test existing C code within the graphical framework of LabVIEW.

Now run the program by clicking on the **Run** arrow  to calculate the dot product of the two vectors. The application will be translated, linked, compiled, and deployed to the Blackfin processor. Open the processor status window and select **Output** to see the numeric result of the dot product operation.

Another feature available for use with LabVIEW Embedded Module for ADI Blackfin Processors is instrumented or non-instrumented debug, which allows users to probe wires on the LabVIEW block diagram and interact with live-updating front panel controls and indicators while the code is actually running on the Blackfin. Instrumented debugging can be accomplished through TCP (Ethernet) or Serial and non-instrumented debugging through JTAG (USB). It is always advisable to use either TCP/IP or serial debugging as they provide faster update rates. To debug the application, right click the **Build Specification** that was created for the application and select **Debug**. Try changing vector elements on the front panel and probing wires on the block diagram. For additional configuration, setups, and debugging information, refer to the Getting Started with the LabVIEW Embedded Module for Analog Devices Blackfin Processors guide found on the textbook companion website.

2.8 Moving-Average Filter in LabVIEW Embedded Module for ADI Blackfin Processors

Digital filters can be easily simulated, prototyped, deployed, and tested in LabVIEW. The following two experiments illustrate the process for simulating and prototyping a moving-average filter for the ADI Blackfin BF537 processor. First, you will use the Digital Filter Design toolkit to develop and simulate the filter on the computer and test its performance using simulated input signals. Next, you will program the Blackfin EZ-KIT using LabVIEW Embedded to perform real-time audio processing that filters an audio signal corrupted by a sine wave.

Hands-on Experiment 2.5:

In this experiment, we will design moving-average filter to remove a 1000 Hz tone that muffles the speech signal. An interactive LabVIEW application has been created to simulate and test filtering results with the actual audio signal.

To begin designing and simulating the filter, open the executable file, `MA_Filter_Sim.exe`, located in the directory `c:\adsp\chap2\exp2_5`. This LabVIEW application was created using LabVIEW for Windows and the LabVIEW Digital Filter Design toolkit. The simulator has been pre-loaded with the three seconds of audio file, `speech_tone_48k.wav`, used in previous VisualDSP++ experiments. If there are loudspeakers attached to the computer, click on **Play** to hear the audio. Do you hear the sine wave obscuring the speech?

Use the **Taps** control to experiment with different numbers of FIR filter taps. Change the **Apply Filter?** control to **Filtered Signal** to see the output signal after the filter is applied. Click on **Play** again to hear the filtered signal. Notice that the 48-tap moving-average filter places a zero precisely on the 1000 Hz tone that distorts the speech

as shown in Figure 2.25. Use the **Zoom** tool  to zoom in on the graph or the **Autoscale** feature to see the entire signal. We can also load a new audio file (`.wav`), and use the same moving-average filter to filter the wave file and listen to the results. Compare the filtered output against the original signal and comment on the results.

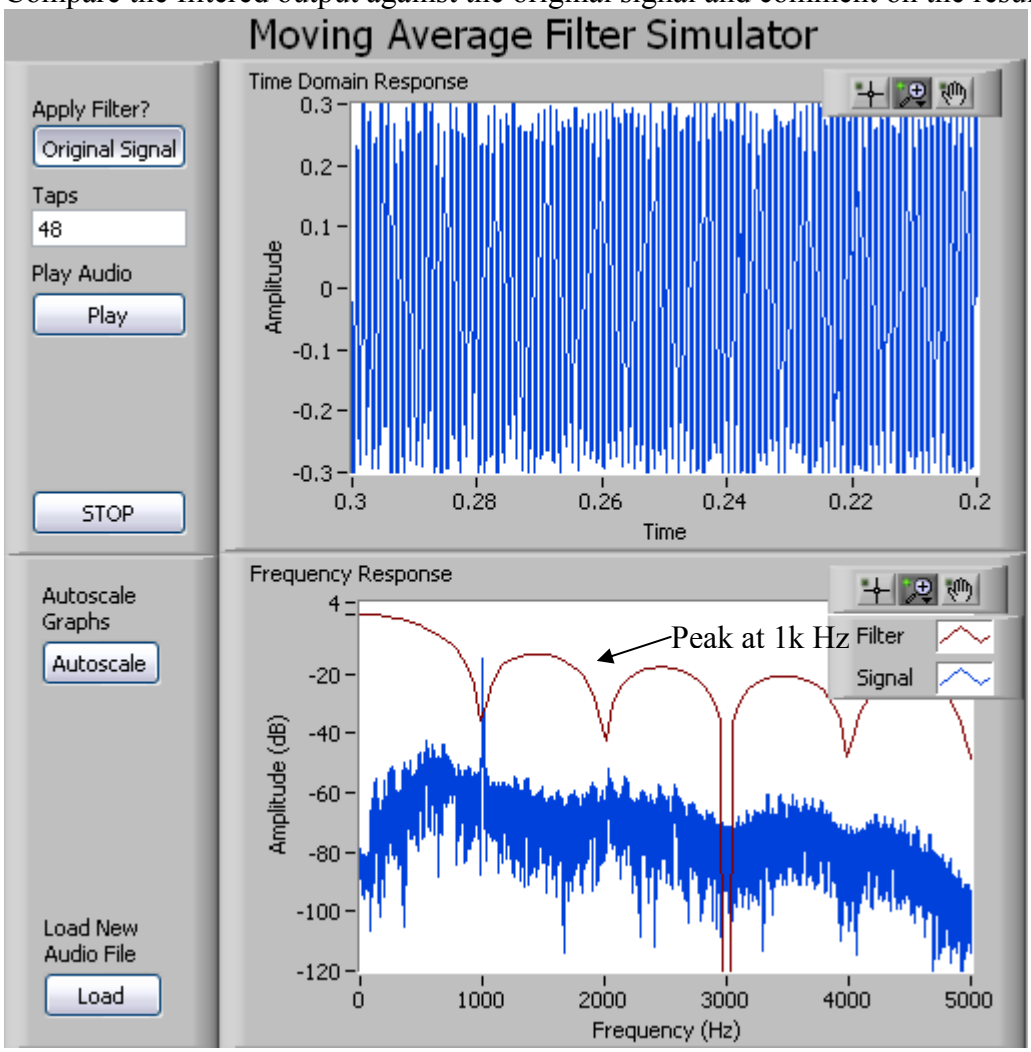


Figure 2.25 Moving-average filter simulator (`MA_Filter_Sim.exe`).

Hands-on Experiment 2.6:

In this experiment, we will use the LabVIEW Embedded Module for ADI Blackfin Processors with the Blackfin EZ-KIT to test the designed moving-average filter. Based on the results from Hands-on Experiment 2.5, we implement a 48-tap moving-average filter on the BF537 EZ-KIT for real-time experiments.

When implementing audio experiments on the Blackfin processor, a good staring point is the Audio Talkthrough example shown in Figure 2.26. The `Audio Talkthrough – BF537.lvproj` for BF537 EZ-KIT project file can be found in the directory `c:\adsp\chap2\exp2_6`, which is useful for testing the audio setup

consisting of computer, cables and loudspeakers (or headphone). On the outside of the **While Loop**, the **Initialize Audio** function creates a circular buffer between the audio CODEC and the Blackfin processor. Inside the While Loop, the **Audio Write–Read** function writes right and left channels of audio data to the output **Audio Buffer,** all zeros on the first iteration of the loop, and then reads in newly acquired audio samples to the right and left channel arrays for processing. This step allows data to be read and processed in one-loop iteration and passed out to the loudspeaker (or headphone) in the next iteration. The process of reading values in the first iteration and then writing them out in the next is called pipelining.

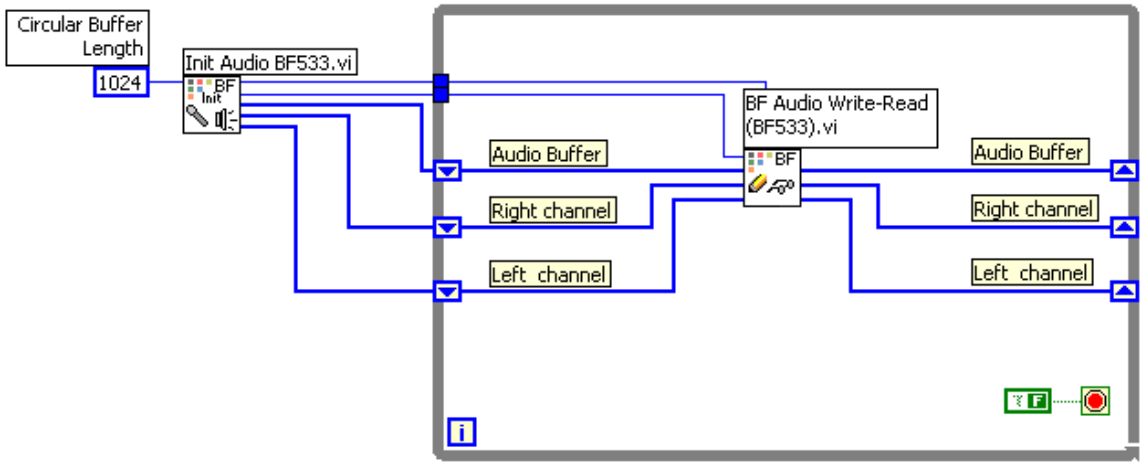


Figure 2.26 Audio talkthrough example.

Once the audio talkthrough has been established, we still have to make two modifications. First, we must wire in an FIR function to filter the input signal; and second, we must initialize that filter with proper parameters. Now open the complete moving-average filter program. The project is called `MA Filter - BF537.lvproj` and it is located in the directory `c:\adsp\chap2\exp2_6`. Figure 2.27 shows the block diagram of the top level VI in this project, `MA Filter - BF537.vi`..

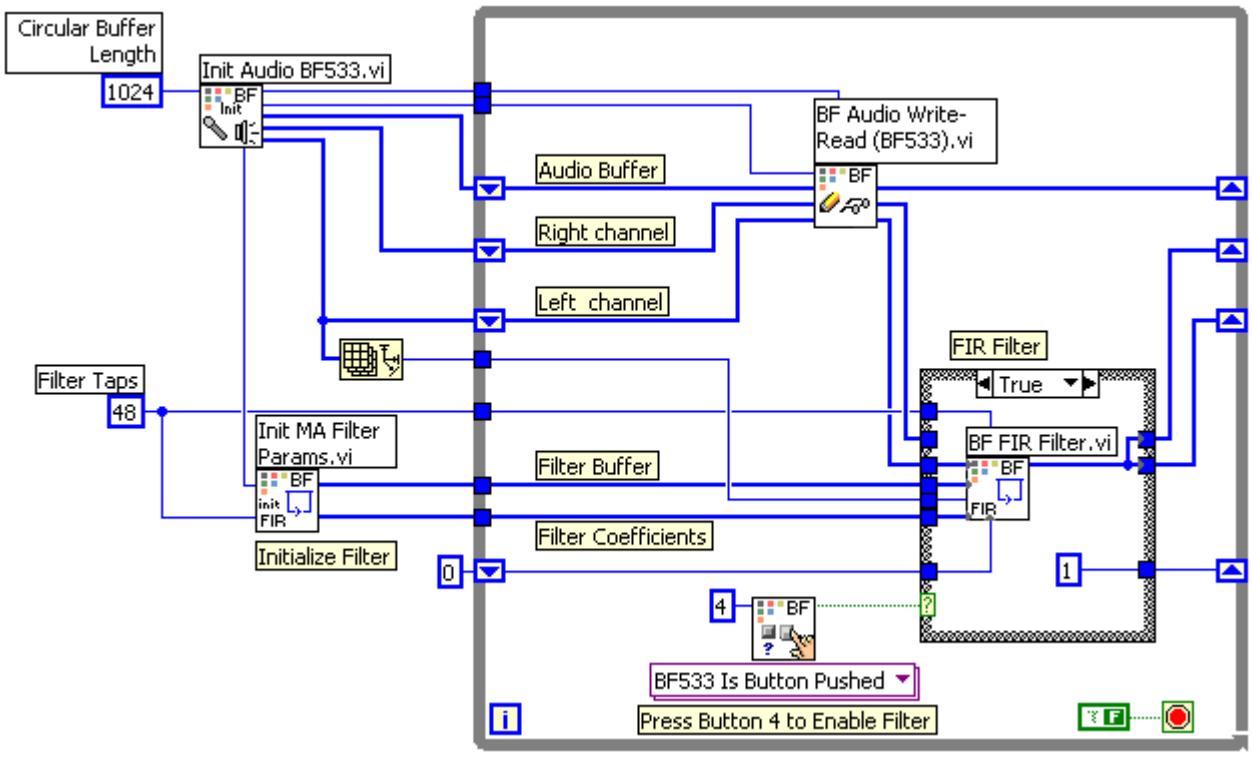


Figure 2.27 Block diagram of `MA Filter.vi`.

In this experiment, we add two functions to complete the filtering exercise. On the outside of the **While Loop**, the **Initialize Filter VI** sets up the moving-average filter designed in Section 2.7 with the specified length (in this case $L = 48$), and outputs the filter coefficients which each have been converted from the decimal 1/48 (or 0.020833) to the fixed-point integer `fract16` representation, a commonly used number format on embedded signal processors.

Inside the **While Loop**, the **BF Audio Write/Read** function writes data to the DAC and then reads values from the ADC through the circular buffer of 1024 samples, 512 for each channel (left and right). The **BF FIR Filter** function, refer to Figure 2.28, filters the input signal with the FIR filter defined by the filter coefficients. This filtering function is placed inside a Case structure so that the signal passes through unchanged unless the SW10/PB4 is pressed on the BF537, allowing us to easily compare the effects of the filter.

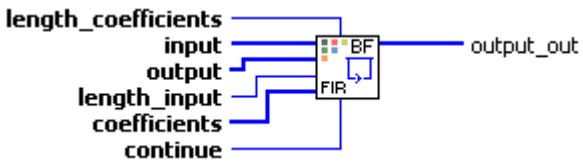


Figure 2.28 Context Help information for the **BF FIR Filter** function.

When this project is compiled and run on the Blackfin target, play the `speech_tone_48k.wav` using `MA_Filter_Sim.exe` from Hands-on Experiment 2.5. Notice how the 1000 Hz tone is filtered out when the filtered switch is pressed. Experiment with different filter lengths and see how it affects the results.

3.7 Frequency Analysis using the LabVIEW Embedded Module for ADI Blackfin Processors

Frequency analysis allows a designer to extract useful signal information in the presence of noise, or detect physical phenomenon like failing bearings in a motor. Accurate time and frequency information is often needed before the embedded design process begins, allowing the designer to choose sampling rates and filters to accurately design and implement the system.

The following examples explore frequency-domain properties in LabVIEW, enable the designer to simulate, prototype, and deploy an application that detects frequency information from time-domain signals. First, we will explore frequency-domain concepts through an interactive simulation. Then, we will implement similar functionality using the FFT through the LabVIEW Embedded Module for ADI Blackfin Processors using the LabVIEW graphical interface to view and interact with signals streaming into the Blackfin EZ-KIT.

Hands-on Experiment 3.8:

This hands-on experiment introduces the effects of windowing on frequency analysis. Understanding the frequency content of signals within a system is a key to develop effective filters and other system components. We will accomplish this by using the FFT algorithm, which is based on having a time-domain signal that continues on to infinity. In real-world applications, finite-length signals will cause leakage in the frequency domain that distorts the result of the FFT. Therefore, windowing is used to minimize the leakage introduced at the beginning and end of a time-domain finite-length signal.

Open the program `Window_FFT_Sim.exe` located in the directory `c:\adsp\chap3\exp3_8`. This LabVIEW application studies different window topologies applied to an input signal, and their affects on the performance of FFT. Figure 3.34 shows the user interface for `Window_FFT_Sim.exe`. Note that the **Time & Frequency** tab shows the **Time Domain** signal and the **Frequency Response**, while the **Window** tab shows the time representation of both the window and the input signal after the window has been applied.

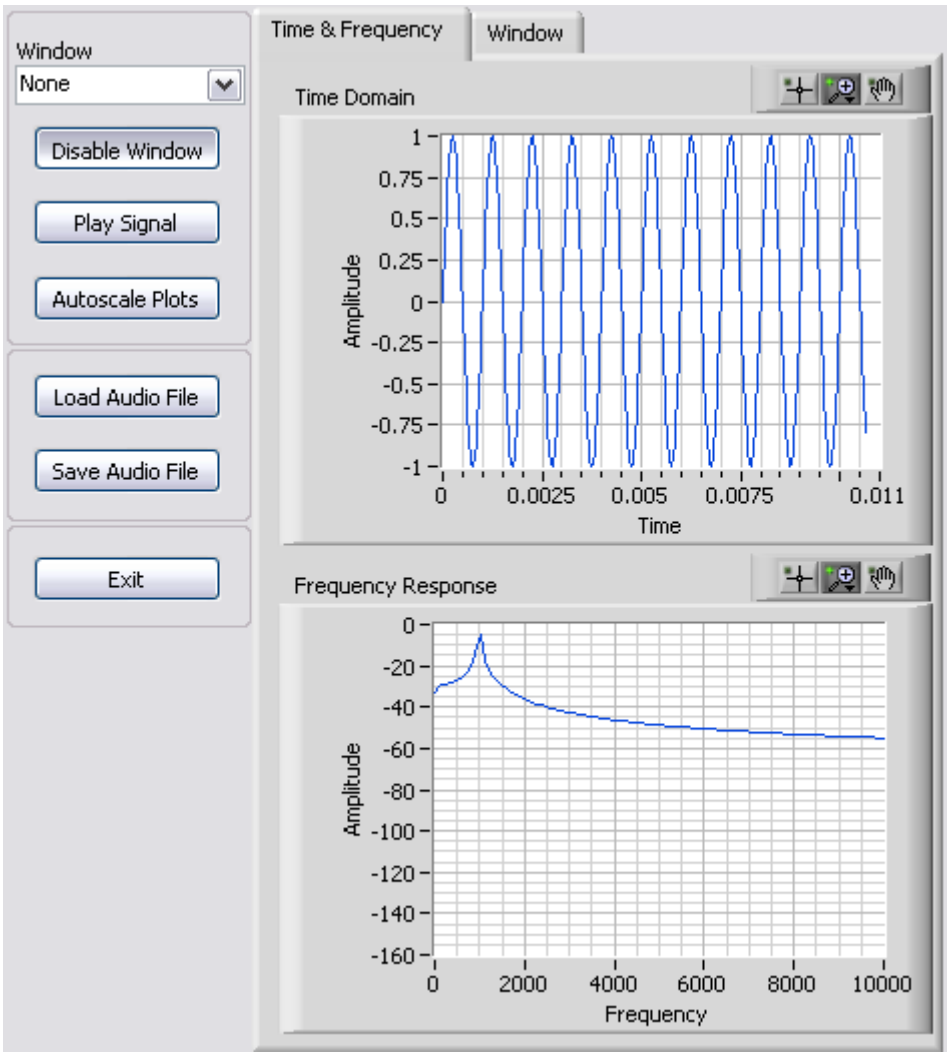


Figure 3.34 Frequency analysis (user interface for `Window_FFT_Sim.exe`).

The default input signal is a 1 kHz sine wave sampled at 48 kHz with a buffer length of 512 samples. When no window is applied, as shown in Figure 3.34, leakage is expected in the frequency domain because the time domain plot shows that the sine wave stops short of completing a full period. The effects of leakage can be seen in the frequency response plot because the power at high frequencies rests above -60 dB. The Window pull-down menu can be used to change the type of windowing to be implemented. Applying a Hamming window provides improved performance as seen in Figure 3.35, where the power at high frequencies drops below –120 dB.

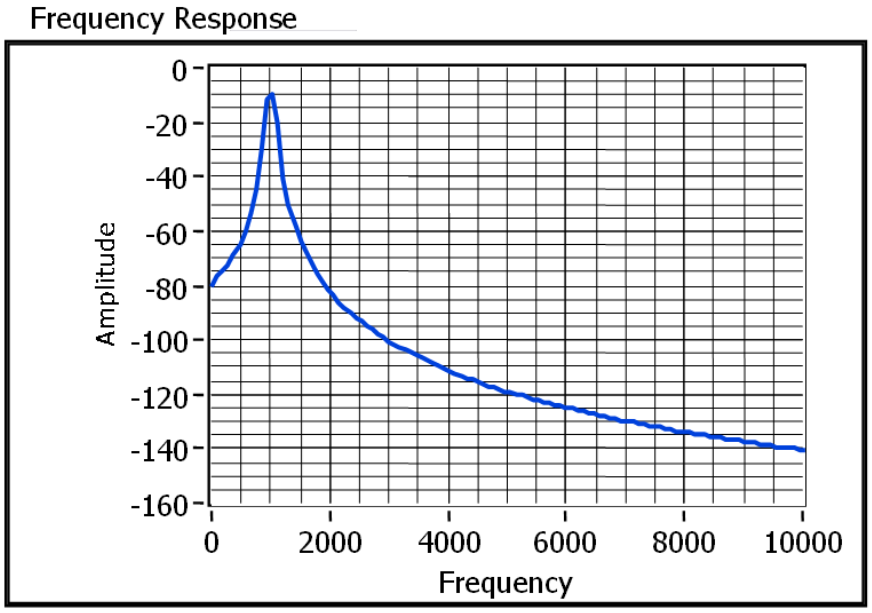


Figure 3.35 Frequency analysis of the input signal with a Hanning window.

Now select the `Bartlett` window. How do the results change? Experiment with the other window types and pay attention to the corresponding frequency response of the sinusoidal input. What tradeoffs are apparent both in the time domain and frequency domains when using the different windows?

Now load and experiment with the audio file, `speech_tone_48k.wav`, which was used in Hands-on Experiment 2.5. Can you see the tonal noise in the frequency-response graph? Experiment with different windows and parameters to see the FFT results. How does windowing affect the non-periodic speech file differently from the periodic sinusoid? How would you expect the results to vary if you were to break up the speech tone into 512-sample buffers? Explain.

Hands-on Experiment 3.9:

This experiment implements the FFT algorithm in the LabVIEW Embedded Module for ADI Blackfin Processors for execution on the Blackfin EZ-KIT to explore the advantages of various windowing implementations. Executing the project in debug mode on the Blackfin processor allows to interact with the LabVIEW front panel and to see the results of the FFT.

Open the `Audio FFT - BF537.lvproj` project appropriate for your Blackfin processor in the directory `c:\adsp\chap3\exp3_9`. Open the block diagram to understand how the FFT is implemented. The block diagram shown in Figure 3.36 is intuitive, allowing the programmer to easily see how to integrate the analog input and processing algorithm. The **Window Type** control is wired to a **Case Structure** allowing the user to specify the window to be applied to the acquired input signal. The windowed time-domain result is plotted to the front panel indicator and passed to the **FFT subVI**. The FFT result is then converted from complex representation to magnitude

response, which is then plotted to a front panel indicator as well. The data type is converted before plotting to limit the amount of data transferred between the code running on the Blackfin target and the front panel interface running on the computer.

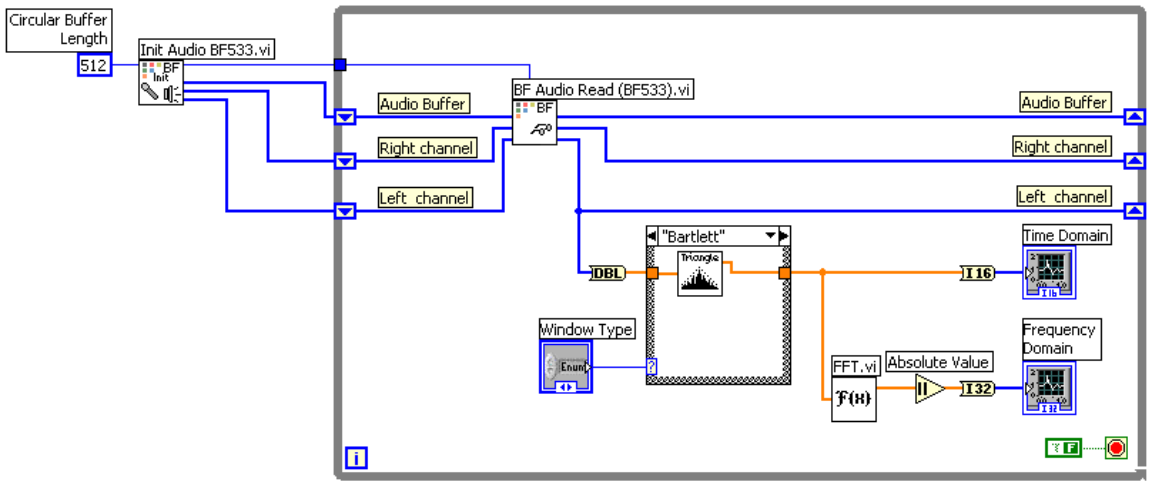


Figure 3.36 Frequency analysis block diagram.

Customize the debugging mode and parameters within the LabVIEW Embedded Module for ADI Blackfin to enhance the experience and value of this exercise. First, modify the target debugging parameters to increase the amount of data to be downloaded to graph indicators. Then, choose the debugging method. Finally, execute the application with debugging support. These steps are described in more detail below.

Modify the target debugging parameters by right clicking the **Target** in the Project Explorer window and selecting **Configure Target → Debug Options**. Change **Max array elements** to 256, and click on **OK**. This is necessary to view all of the data on the graph because the buffer length on the block diagram is specified as 512 samples, which is divided into 256 for each channel. Therefore, the time-domain signal and FFT result will have 256 samples.

The three debugging methods offer different advantages depending on the application and resources. By default, this project is configured to the standard JTAG debugging method using JTAG over USB. JTAG is the standard communication method between the Blackfin processor and VisualDSP++. In this debug mode, data transfers interrupt the processor when transmitting data to and from the PC, which disrupts real-time processing. JTAG debugging is adequate for this and many other applications as long as real-time processing is not necessary. Another option, unique to LabVIEW Embedded Module for ADI Blackfin Processors, is called Instrumented Debugging, which requires the USB cable and an additional cable connection between the Blackfin and PC. The additional cable can be either serial or TCP/IP and allows for quicker update rates, providing a faster, more interactive experience. Instrumented debugging adds extra code to the project to initiate debug data transfers while the LabVIEW code is running on the processor, adding as much as 40% to the size of the embedded executable code. If you are interested in using serial or TCP/IP debug, attach the appropriate cables and change the debug mode for the project by navigating to **Target → Build Specifications**

and change **Debug Mode** to **Instrumented (via serial port)** or **Instrumented (via TCP port)**.

To run the project on the Blackfin processor, close the block diagram window and navigate back to the project window. Right click on the Build Specification and select **Debug** to run the VI in debug mode. Different windows can be applied during runtime by changing the **Window Type** control on the front panel as shown in Figure 3.37. Change the window type to see how it affects the time-domain signal and the performance of the FFT. How does windowing improve the resulting frequency-domain calculation?

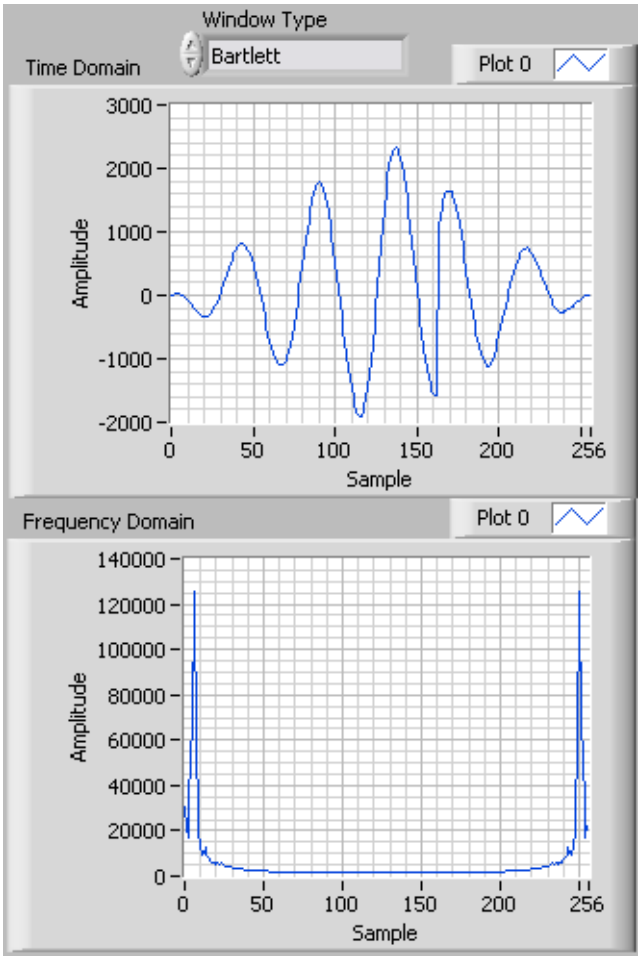


Figure 3.37 Frequency analysis front panel.

Explore the passed values through wires on the block diagram using probes. Notice that the resulting vector from the FFT is complex and the absolute value function converts the complex values to the magnitude response. Modify the experiment to show the real-part of frequency response in one graph and the imaginary part in the other.

4.7 Adaptive Linear Enhancer with the LabVIEW Embedded Module for ADI Blackfin Processors

This section designs and implements ALE using the graphical system design approach. First, we simulate the ALE in LabVIEW to gain an intuitive understanding of its behaviors, and then design and implement the ALE application with the LabVIEW Embedded Module for ADI Blackfin Processors for execution on the Blackfin EZ-KIT.

Hands-on Experiment 4.6:

This experiment focuses on the simulation and implementation of a simple adaptive FIR filter using the LMS algorithm. We will use the simulation created in LabVIEW to explore the different input parameters that affect the behaviors of the LMS algorithm. Simulation is a crucial part of system design because it allows the designer to test and refine parameters before moving to the real embedded target. This can effectively reduce errors, faulty assumptions, and the overall development time.

Open the program `LMS_Fitler_Sim.exe` in the directory `c:\adsp\chap4\exp4_6`. The user interface for this application is shown in Figure 4.31. Some input parameters can be altered to study the effects on the LMS algorithm and its outputs. There are two tabs: **Time Domain** and **Frequency Domain**. The **Time Domain** plots show the noisy sine wave, filtered output signal, filter coefficients, and the error signal. Click the **Frequency Domain** tab to evaluate the magnitude and phase responses for the FIR filter coefficients as they adapt to reduce the noise.

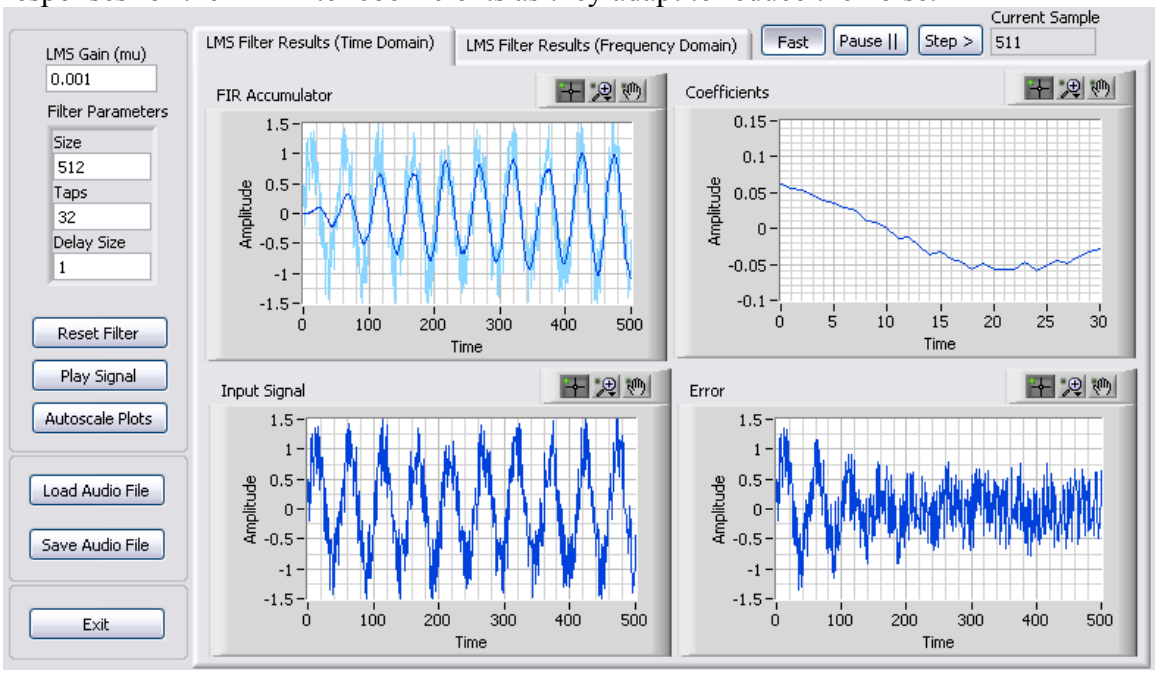


Figure 4.31 Time-domain plots of the signals and coefficients of the filter using the LMS algorithm (`LMS_Filter_Sim.exe`).

The most interesting part of the simulation is to see how the LMS algorithm learns and adapts its coefficients. Click the **Reset Filter** button to reset the filter coefficients to zero and restart this learning process. Additional features have been added to the top-right corner for pausing and single stepping through the algorithm. We can single step through the calculation while viewing either the time-domain or frequency-domain plots. These graphical features greatly aid the visualization and understanding of adaptive filtering, which is an iterative algorithm. Select the **LMS Filter Results (Frequency Domain)** tab to see how the FIR filter adapts to suppress the noise while preserving the desired input 1 kHz signal as shown in Figure 4.32.

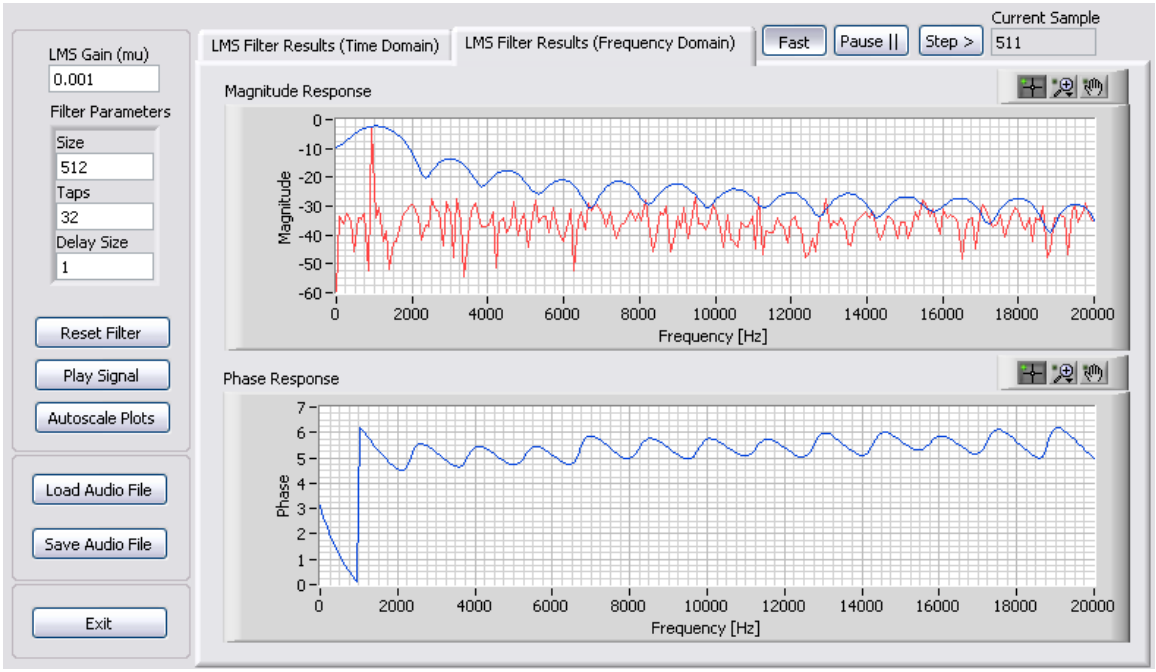


Figure 4.32 Magnitude and phase responses of the adaptive filter designed by the LMS algorithm (`LMS_Filter_Sim.exe`).

The input parameters for the LMS algorithm that can be changed include the **LMS Gain** (step size μ), the number of FIR filter taps (length L), and the delay size Δ. As discussed earlier, the step size determines the amount of correction to be applied to the filter coefficients as the filter adapts. The **Taps** parameter determines the length of the FIR filter. **Delay Size** indicates the distance in samples between the current acquired sample and the newest sample used by the filter.

As the filter designer, optimize the LMS algorithm by fine tuning the step size for fast learning, while minimizing coefficient fluctuations once the filter is converged in steady state. How the input parameters effect the resulting filter? Can the LMS algorithm lose stability? Explain.

Hands-on Experiment 4.7:

This experiment implements the LMS algorithm with the LabVIEW Embedded Module for ADI Blackfin Processors using C programming. Algorithms written in C can be quickly prototyped in the LabVIEW Embedded, allowing them to be implemented as a subcomponent of the graphical system. Graphical system implementation allows other parts of the system to be abstracted, thus allowing the programmer to focus on the algorithm.

Open the `LMS Adaptive Filter - BF537.lvproj` project in the directory `c:\adsp\chap2\exp4_7`. When viewing the block diagram of the `LMS Adaptive Filter.vi`, notice that it is similar to the audio talk-through example discussed in Chapter 2, which has been modified to include buffer initializations, specification of taps and delay size, and an Inline C Node as seen in Figure 4.33. The parameters `Taps` and `Delay Size` can be customized to change the behavior of the LMS algorithm. The Inline C Node contains nearly the same LMS algorithm found in the C implementation in earlier experiments with an alternate FIR filter implementation. In this experiment, the FIR filter is implemented with a multiply-accumulate operation within a For Loop to reduce the amount of C programming needed. The original implementation using the `fir_fr16` library function required additional programming for initialization, which linked global buffers, coefficients, and state variables to the filter. Push button input was also added allowing the user to choose which component of the algorithm to output.

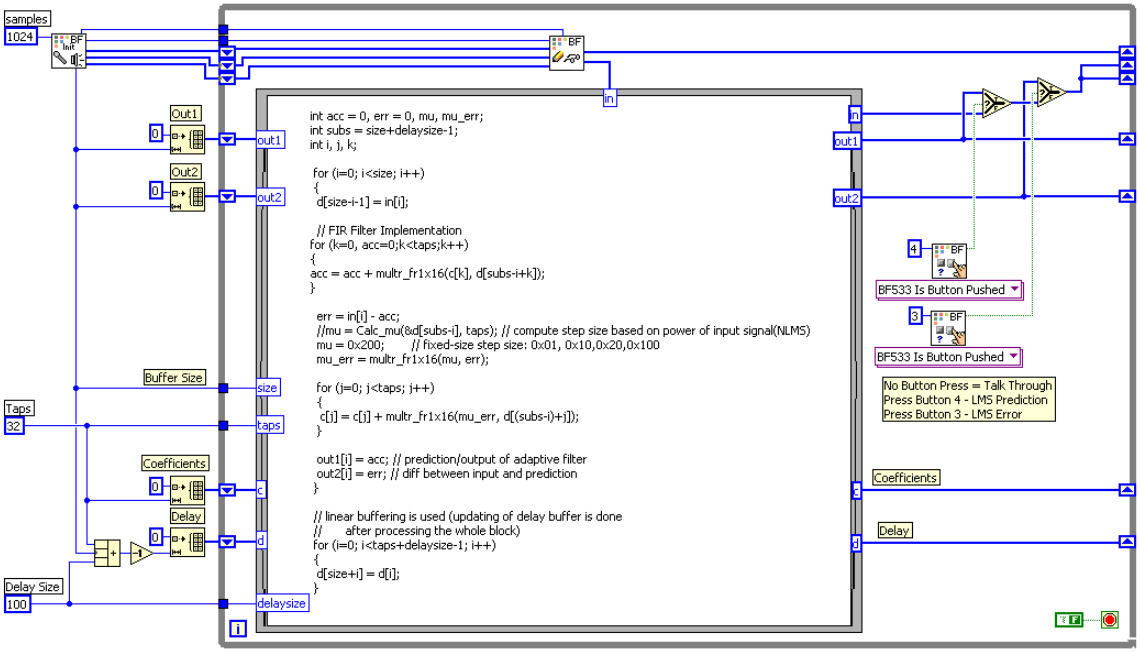


Figure 4.33 The adaptive LMS filter block diagram.

The LMS algorithm is computationally intensive, making it necessary to configure Blackfin optimizations to run the LMS algorithm as fast as possible. These settings optimize the C code generated from LabVIEW and are found in the **Build**

Specifications Properties menu by right clicking the Build Specification and choosing **Properties** in the Project Explorer window. First, the **Build Configuration** is changed from **Debug** to **Release**. This option removes all extra debugging code from the generated C. **Stack Variables** and **Disable Parallel Execution** are both enabled to optimize the way memory is managed and parallel programming structures are handled. These optimizations are ideal for implementing fast, serialized algorithms that perform many memory operations, such as the LMS algorithm that performs an FIR filtering and coefficient updating for each of the 512 samples in the acquired audio buffer.
Run the project on the Blackfin processor. Play the `speech_tone_48k.wav` that was used in the Hands-on Experiment 2.4, and feed the wave output to the audio input of the BF537 EZ-KIT. When the SW10/PB4 of the BF537 is pressed, the LMS algorithm adapts to suppress the speech. When the SW11/PB3 of the BF537 is pressed, the error signal is heard. What do you notice about the error signal? Why can the error signal be used as the desired filter output?

5.6 Implementation of Graphic Equalizer using the LabVIEW Embedded Module for ADI Blackfin

Graphic equalization is a very scalable technology allowing a designated number of audio frequency bands to be extracted, amplified, and reassembled to improve or process audio signals. Commercial graphic equalizers use a wide variety of analog and digital technology to achieve similar functionality. Most have a standard user interface using slider bars allowing each band to be individually amplified or attenuated. The algorithm for equalizing audio is computationally intensive in its theoretical form, but can be simplified and implemented in LabVIEW using the same principles discussed in the moving-average filter application from Chapter 2.

In the following exercises, you will simulate, prototype, and deploy a graphic equalizer using the FIR filter coefficients derived in previous examples. The 8-band graphic equalizer simulation will allow you to load custom coefficients and an audio signal. This gives you the ability to modify and listen to the effects of different equalizer gains. The equalizer will then be run on the Blackfin EZ-KIT, demonstrating a real-time filtering application created with graphical programming.

Hands-on Experiment 5.10:

In this experiment, an 8-band graphic equalizer is implemented using FIR filtering to modify the frequency content of audio signals. The simulation will allow you to hear the effects of the equalizer and see its results in both the time and frequency domains. Custom filter coefficients can also be loaded to test the results of your own equalizer designs. You can easily modify the gain of each frequency band and observe the differences in filter characteristics.

Navigate to the program called `FIR_EQ_Sim.exe` located in the directory `c:\adsp\chap5\exp5_10` to begin exploring graphic equalization. In Figure 5.38, we see the user interface for `FIR_EQ_Sim.exe`. Separate tabs show various plots of the audio signal and allow you to customize the gain applied to each band. The **Time Signal** tab shows the time-domain input signal. Click the **Enable Filter** button to turn on equalization. The **Frequency Signal** tab shows both the input signal and its frequency content, and the graph can be viewed with either a linear or logarithmic (dB) scale. The **Frequency Bands** tab shows the magnitude response of each of the eight filters individually. Notice that while each of the individual filters is not ideal, the overall passband is relatively flat when they are combined. The **Load Bands** tab gives us the ability to open new sets of coefficient data files and change the gains applied by the equalizer slider positions. The set of eight slider bars at the bottom of the user interface corresponds to the gain applied to each of eight frequency bands.

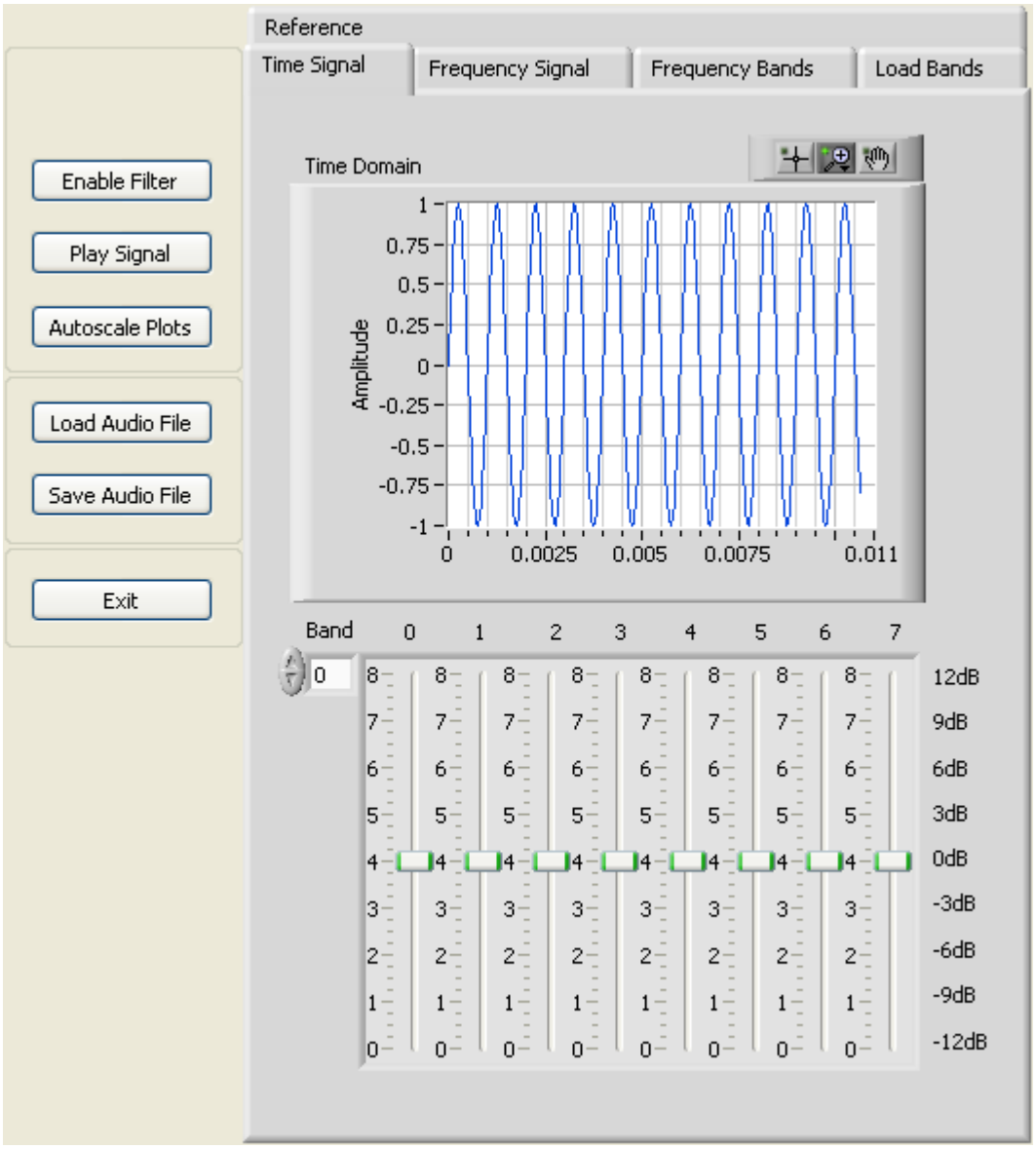


Figure 5.38 8-Band graphic equalizer (user interface for `FIR_EQ_Sim.exe`).

The default input signal is a 1 kHz sine wave sampled at 48 kHz with a buffer length of 512 samples, as seen in Figure 5.38. Different wave files can be loaded for additional filter testing. Custom coefficients can be loaded from data files in `fract16`, a common delimited format for custom equalizer simulations.

Click the **Frequency Bands** tab and adjust the values of the band #1 slider bar. As you change the selected gain for that frequency band, note how the magnitude response for that band changes in the graph as well.

Now, load and experiment with the audio file `speech_tone_48k.wav` that we used in Hands-on Experiment 2.5. Recall that this audio file contains tonal noise that degrades the overall quality of the audio. Can you identify the frequency band that contains the tonal noise? Experiment with the equalizer gain settings to attenuate the noise as much as possible while still retaining the rest of the signal. How would these settings need to change if the tonal noise had a different frequency? How would you redesign the filter if you need finer resolution control over high frequencies and less resolution control over low frequencies while keeping just 8 bands?

Hands-on Experiment 5.11:

This experiment implements 8-band FIR filtering on the Blackfin processor using LabVIEW Embedded Module for ADI Blackfin Processors. For this project, we will run in **Release** mode to take advantage of the excellent speed optimizations when the various debugging features are not necessary. There are two key processing steps in this experiment. First, the single FIR filter Blackfin library function is the only processing VI that modifies the original signal. The computationally intensive portion of this VI is the calculation of the FIR filter coefficients. This filter coefficient calculation is packaged into a subVI for code modularity and is used as a single processing block.

Begin by opening the `FIR Equalizer - BF537.lvproj` project appropriate for your Blackfin hardware in the directory `c:\adsp\chap5\exp5_11`. You will find that the project is composed of three files, one for each of the main processing blocks in the project. Double-click on the top-level VI, `FIR Equalizer - BF537.vi`, and open the block diagram to understand how the equalizer is implemented. Notice that the processing algorithm is divided into distinct sections for buffer initialization, coefficient calculation, and filtering, where each of these operations has its own subVI. This application streams a real-time audio signal from the Blackfin audio input, processes it, and generates an output signal from the Blackfin EZ-KIT audio out port. When SW13/PB1 of BF537 is pressed, the selected FIR filter is enabled and applied to the signal. Pay special attention to the block diagram of the Init Eq Coeffs.vi subVI (shown in Figure 5.40), which implements the calculation of the filter coefficients.

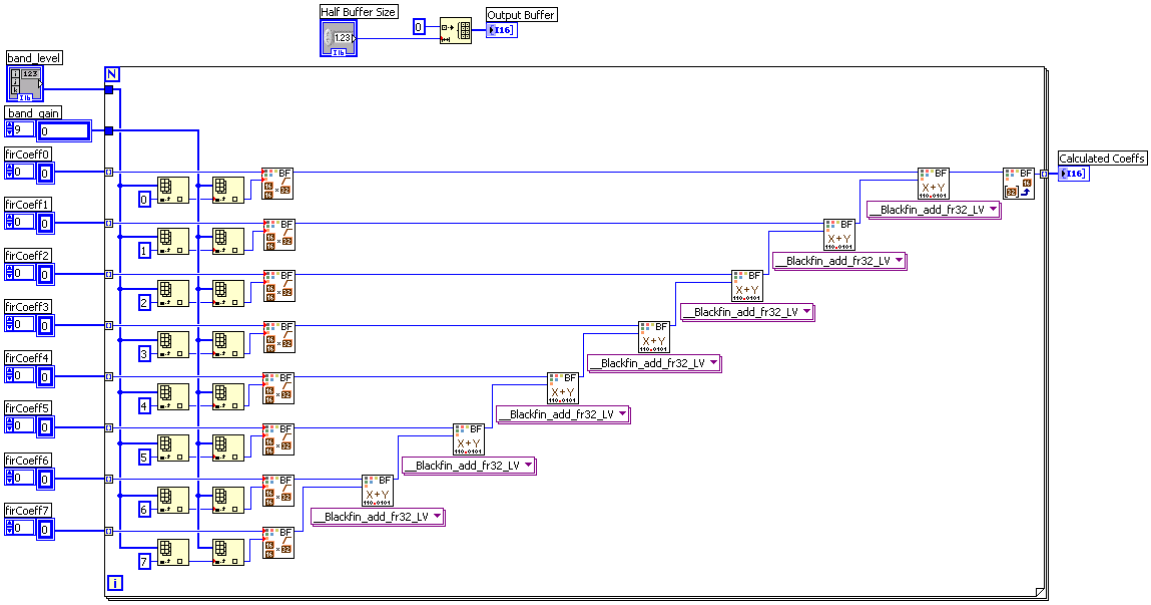


Figure 5.40 Coefficient calculation block diagram.

This block diagram shows how to calculate the coefficients of the FIR filter. Graphical representation of the algorithm allows the parallelism of the code to be easily seen. Parallelism is valuable in understanding complex algorithms with recurring symmetry. For each of the eight bands, the input coefficients are multiplied by the gain selected for that band. We then sum these products to calculate a single set of coefficients. Once the coefficients are calculated by the Init Eq Coeffs subVI, they are passed to the FIR filter function block along with the audio buffer for real-time processing.

Connect the computer audio output to the input of the Blackfin processor and the output of the Blackfin processor to loudspeakers or headphones. On the computer, play the audio signal that you would like to equalize. Compile and run the project on the Blackfin EZ-KIT. To enable the equalizer, press and hold SW13/PB1 of BF537.

The application can be customized with different graphic equalizer gain settings, or by changing the filter coefficients for each band. Customize the gains for different bands by changing the selected preset. Use the same settings simulated in the Hands-on Experiment 5.10 to attenuate the 1 kHz sine wave from the `speech_tone_48k.wav` audio file and evaluate its performance with the Blackfin processor implementation. Does it behave the same way? Recompile the project and run it. Can you hear the difference in the audio? Experiment with the various gain presets and try creating your own.

6.7 Implementation of IIR Filter-based Graphic Equalizer using the LabVIEW Embedded Module for ADI Blackfin Processors

The IIR filter is commonly used in equalizer designs and can be implemented using many of the same principles from Chapter 5. The following experiments explore an IIR filter-based graphic equalizer. The first experiment uses LabVIEW for Windows to simulate an 8-band equalizer, and the second experiment employs the LabVIEW Embedded Module for ADI Blackfin Processors to create a 4-band equalizer on the Blackfin processor. Interrupts are introduced in the LabVIEW Embedded Module for ADI Blackfin Processors example for more efficient and functional user interaction.

Hands-on Experiment 6.18:

This experiment explores the implementation and behavior of a multi-band IIR Equalizer by means of a LabVIEW for Windows simulation. Navigate to the compiled simulation `IIR_EQ_Sim.exe` located in the directory `c:\adsp\chap6\exp6_18` and launch the executable code. The user interface, shown in Figure 6.34, for this application is nearly identical to that of `FIR_EQ_Sim.exe` given in the last chapter. Run the application and switch to the **Frequency Bands** tab. Note the shape of each frequency band.

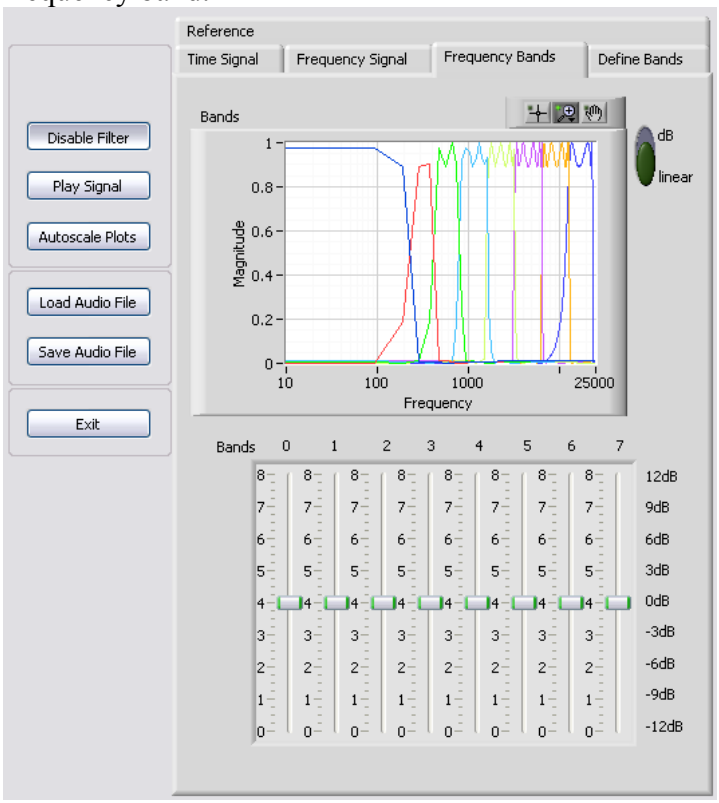


Figure 6.34 Frequency bands for IIR-filter based graphic equalizer.

Compare the frequency response of each band with the response seen in the previous chapter. What do you notice about the comparative sharpness of the IIR and FIR filter methods? What might account for this difference? Explain.

The IIR filter equalizer cannot be computed using the same method that was used in the FIR equalizer case, because impulse responses of IIR filters cannot be determined from their coefficients alone. Therefore, the IIR band equalizer is computed by applying several IIR filter to the input signal in parallel, one for each band. The selected gain is then applied to each band and then the results are summed to determine the overall response. Switch to the **Frequency Signal** tab of the simulation and choose the **Linear Display** option. The combined response of the eight filters is shown along with that of the input signal. Notice that the spikes occur in the plot corresponding to the transitions between each frequency band. Experiment with the slider values for each frequency band to see the relationship between the individual bands and the combined response. Note how greater amounts of overlap in the transition bands on the **Frequency Bands** tab correspond to larger spikes in the frequency response.

Each of the eight filters is defined by a set of filter parameters accessible on the **Define Bands** tab of the simulation. The **Filter Band Specifications** array contains specification parameters consisting of high and low cutoff frequencies, the desired order of the filter, the passband ripple, and stopband attenuation parameters (in dB). The first filter, at index 0, is a lowpass filter, and all others are bandpass filters. The parameters for each band are preloaded to match the filter specifications used in Section 5.3.

Load an audio file to hear the effects of the equalizer. The `speech_tone_48k.wav` file used in previous experiments is ideal for this purpose, as it contains noise centered at 1 kHz. Experiment with the various filter options to attenuate the noise while retaining as much of the voice signal as possible. What settings meet these criteria? How do the audio results differ from those heard in the 8-band FIR filter simulation?

Hands-on Experiment 6.19:

In this experiment, we will combine many of the concepts discussed previously, including audio input and output, the use of subVIs, and the Inline C Node, to create a 4-band IIR equalizer within the LabVIEW Embedded Module for ADI Blackfin Processors. This experiment introduces interrupt handling, which allows us to use the Blackfin EZ-KIT push-buttons for updating the band gains during execution on the Blackfin processor.

Open the `IIR_4_Band_Equalizer - BF537.lvproj` LabVIEW embedded project which is available in the directory `c:\adsp\chap6\exp6_19`. Open the block diagram for `IIR_4_Band_Equalizer.vi`, as shown in Figure 6.35, which contains the top-level processing structure for the IIR equalizer.

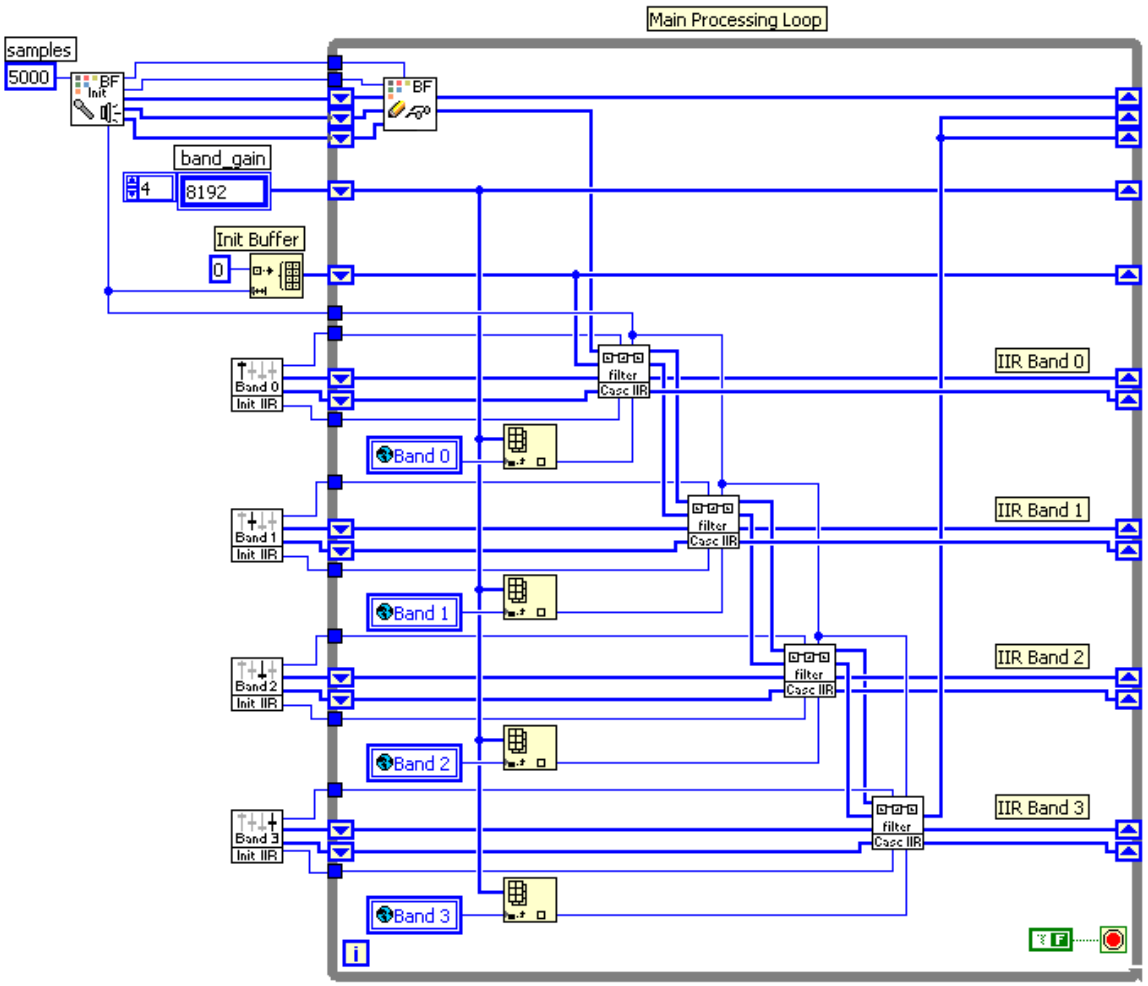


Figure 6.35 Block diagram of IIR filter-based equalizer (`IIR_4_Band_Equalizer.vi`).

The graphical code makes it easy to recognize the parallelism of band implementations. Each band is individually initialized with coefficients which are passed to the cascaded IIR filter block. Each filter block passes both the original input parameters and its result to the next stage. When all four stages have been completed, the output is passed back to the Audio Write/Read VI.

The filter used within the cascaded IIR filter block is nearly identical to that introduced in the previous VisualDSP++ experiment. The Inline C Node is used to combine the textual implementation of the filter into the graphical code. The output of each stage is accumulated with the previous stages rather than creating an intermediate array for each band and subsequently applying band gains. This structure allows for a pipelined approach where the output of the *n*th band's IIR filter subVI is the sum of the weighted IIR filter outputs of the first *n* bands. This method is advantageous because additional frequency bands can be added without modifying any of the underlying processing code. To add another frequency band, simply add another stage to the pipeline. This removes the use of global variables and structures to make the code more modular. When working with embedded applications, small improvements to repetitious code can often contribute to large performance enhancements.

Connect the Blackfin EZ-KIT audio input to the audio output jack of your PC and connect loudspeakers to the Blackfin EZ-KIT audio output. Click on the **Run** button to execute the IIR equalizer project on the Blackfin processor. With an audio file playing on the computer, adjust the equalizer band gains by pressing buttons 1 through 4 on the Blackfin EZ-KIT. Each button corresponds to one of the four frequency bands and their levels. In the BF537, SW13/PB1 to SW10/PB4 correspond to band 0 to band 3, respectively. When a button corresponding to a band is pressed, the gain applied to that band will be incremented until the gain reaches the maximum value, at which point it will reset to the smallest value. You can keep track of the current level using the LED display, which shows a binary representation of the gain level of the last modified band.

Unlike the FIR filter-based equalizer in the previous chapter, the IIR filter-based equalizer uses interrupts to determine when a button has been pressed. In past experiments, the button values were checked every iteration of the loop, which is a polling method. Handling interrupts with the LabVIEW Embedded Module for ADI Blackfin Processors is very straightforward and makes your applications significantly more interactive and responsive. On the block diagram of `IIR_4_Band_Equalizer.vi`, scroll down to the second loop as shown in Figure 6.36. This loop structure handles all of the processing necessary for the button-pressing functionality and only executes when a button-based interrupt is received by the Blackfin processor.

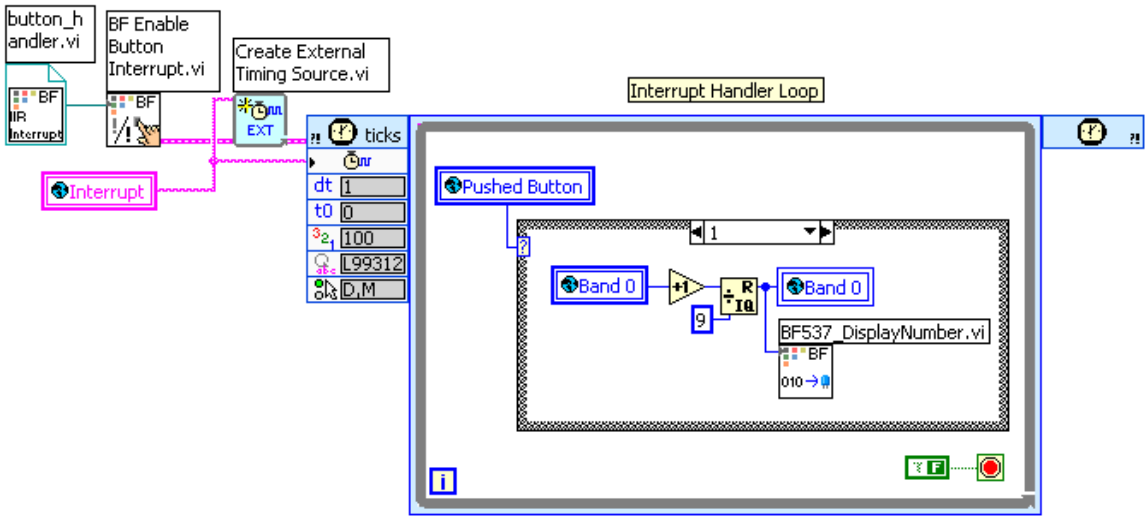


Figure 6.36 Interrupt handler loop in `IIR_4_Band_Equalizer.vi`.

When button interrupts are enabled using the `BF Enable Button Interrupt.vi`, an interrupt flag causes the timed loop to iterate one time. In this case, the number of the button pressed corresponds to a specific frequency band. The gain for that band is incremented unless it was already at the maximum level, in which case it will reset to the lowest value. The binary representation of the gain level, a value from 0-7, is then output to the LED display. The numeric result is shared with the main processing loop through a global variable, as seen in Figure 6.36.

Recall how the IIR filtering occurs in a pipelined fashion, with each frequency band calculated sequentially and accumulated with the rest of the filtered signal. How

would you implement an 8-band equalizer with this architecture? What code would need to be added to the existing VI, and what code could be re-used?

7.8 Signal Generation using the LabVIEW Embedded Module for ADI Blackfin Processors

In previous chapters, we have acquired input signals with the ADC, processed the sampled signals, and output them with the DAC. In this section, we create signals on the Blackfin EZ-KIT that can be used as test signals for other systems. Test signals are typically generated to contain specific frequency information or simulate real-world signals. These user-defined signals allow engineers to test systems for specific performance or real-world behavior while still in the lab. For instance, communication signals can be generated based on input data and adhere to a common standard agreed upon by both the transmitter and receiver. A third use of signal generation is the creation of control signals. For example, a pulse-width modulated (PWM) square wave can be used to control motor speed or heater voltage (or temperature) in an experiment.

In the following experiments, we will focus on dual-tone multi-frequency (DTMF) communication signals. DTMF is the standard protocol used for routing numbers pressed on a touch-tone telephone. A typical DTMF keypad is divided into rows and columns, where each row and column contains its own unique frequency as shown in Figure 7.32. When a key is pressed, two single-frequency signals are generated and added together to create the touch tone.

	column 1 1209 Hz	column 2 1336 Hz	column 3 1477 Hz	column 4 1633 Hz
row 1 697 Hz	1	2	3	A
row 2 770 Hz	4	5	6	B
row 3 852 Hz	7	8	9	C
row 4 941 Hz	*	0	#	D

Figure 7.32 Touch-tone phone DTMF mapping.

In this experiment, LabVIEW is used to simulate an ideal DTMF generator (or encoder) and a DTMF receiver (or decoder). Each signal consists of two specific tonal frequencies that uniquely identify it. The LabVIEW Embedded Module for ADI Blackfin Processors is then used to prototype and deploy a DTMF encoder on the Blackfin EZ-

KIT. The interactive LabVIEW graphical interface is used for live debugging and interaction with the DTMF generator running on the Blackfin processor.

Hands-on Experiment 7.11:

The purpose of this exercise is to gain an intuitive understanding of how a DTMF signal generator works and an understanding of their time and frequency domain characteristics. LabVIEW and the LabVIEW Embedded Module for Blackfin are capable of generating many types of signals in addition to these sine waves and DTMF signals.

Open the program `DTMF_Encoder_Sim.exe` located in the directory `c:\adsp\chap7\exp7_11` to see the user interface shown in Figure 7.33. The purpose of this LabVIEW-built application is to gain an intuitive understanding of how DTMF signals are generated and an understanding of their time and frequency domain characteristics. The array of 16 buttons is arranged in a 4 by 4 matrix as shown in Figure 7.32. Each row and column has a unique frequency assignment giving it a distinct dual-tone combination. Two tonal signals are added together to create a DTMF signal. The column and row tonal frequencies can be customized, although they may not be interpreted properly by most phone receivers. Note that we use different frequencies in Fig. 7.33 instead of standard DTMF frequencies defined in Figure 7.32.

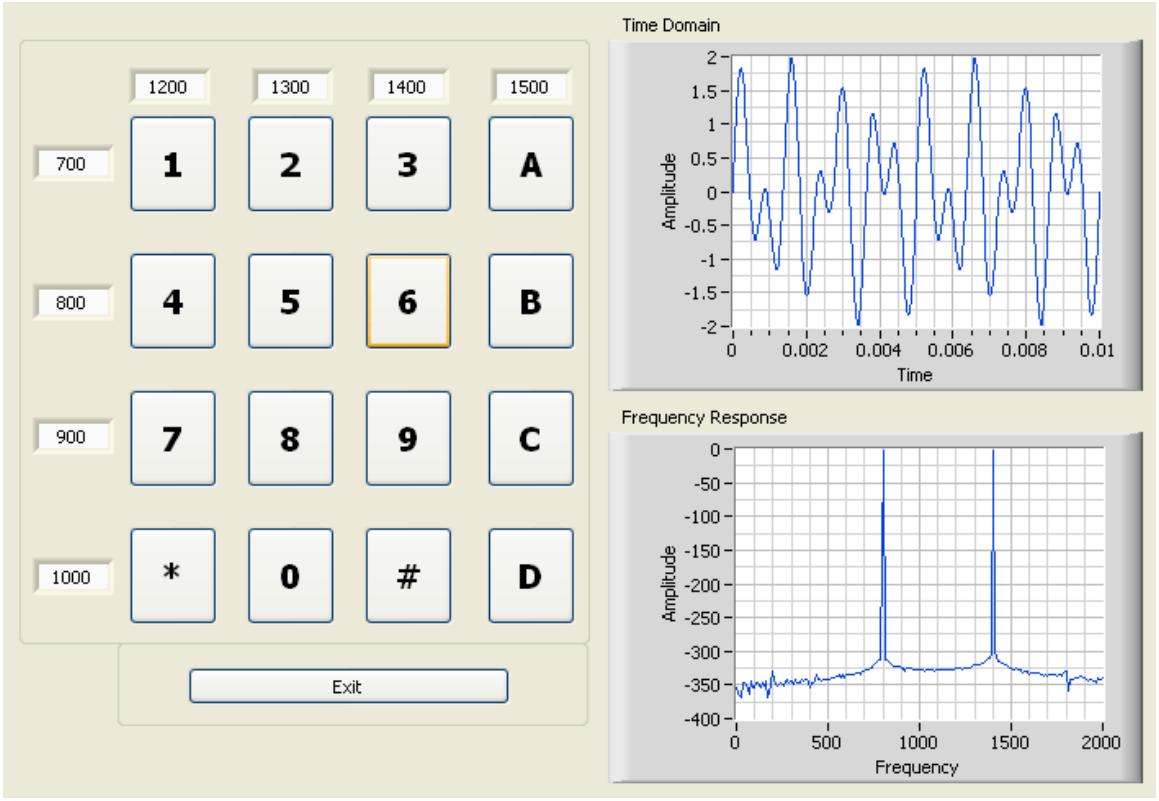


Figure 7.33 DTMF encoder (User interface for `DTMF_Encoder_Sim.exe`).

In Figure 7.33, the button "6" is pressed causing sine waves of frequencies 800 Hz and 1400 Hz to be combined into a single signal. The resulting time-domain and

frequency-domain responses can be seen in their respective graphs in the right side of Figure 7.33. Test other buttons to verify that the time-domain signal contains the correct corresponding frequencies. Attach loudspeakers or headphone to the audio output of sound card to hear the resulting DTMF tones.

Now open the `DTMF_Decoder_Sim.exe` application located in the directory `c:\adsp\chap7\exp7_11`. The interface shown in Figure 7.34 decodes DTMF signals according to the frequencies specified in Figure 7.33.

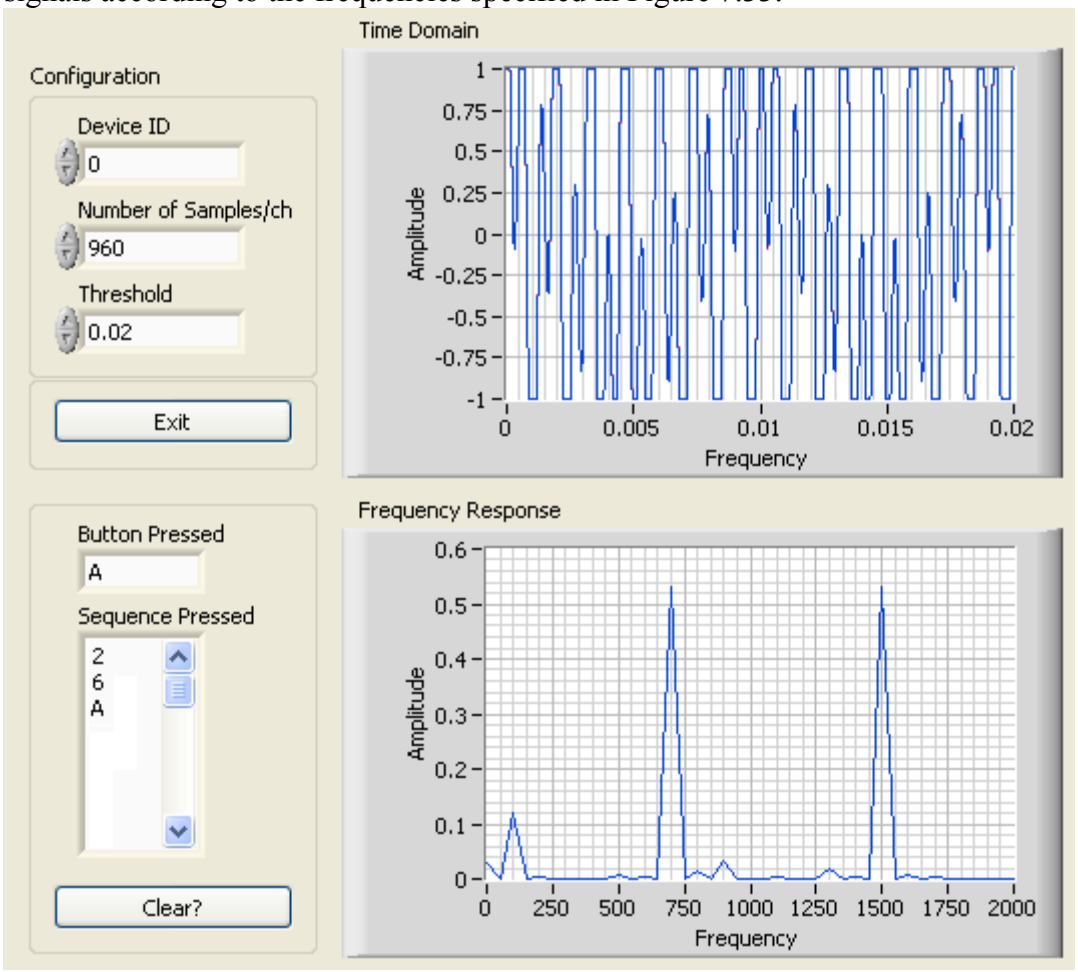


Figure 7.34 DTMF decoder (User interface for `DTMF_Decoder_Sim.exe`).

The sound card output can be connected to the computer audio/microphone input to create a loop-back for the DTMF communications. This allows generating and decoding signals when both simulations are running concurrently. The DTMF decoder interprets the detected frequencies as a row and column in the format of Figure 7.33, and displays the corresponding button value. Both the time and frequency domain signals are shown on their respective graph. The cursors show the values of the last acquired frequency spikes. **Device ID** specifies which sound card to use if more than one is detected in your machine. **Number of Samples/ch** can be varied to detect different pulse durations. Finally, **Threshold** is used for noise immunity and can be adjusted to reduce false readings.

Notice that the acquired signals are not necessarily identical to those generated in each row and column combination. The values are near the expected value but are rounded off to the nearest 100 Hz value to eliminate discontinuities between signal buffers. Why is this necessary or beneficial? Experiment with different values for the DTMF encoder frequencies and listen to the tones using loudspeakers (or headphone). Are these results expected? Can the two frequencies for a button be placed too close together such that they are not distinguishable by the decoder? Explain.

Hands-on Experiment 7.12:

This experiment evaluates the DTMF encoder on the Blackfin EZ-KIT using the LabVIEW Embedded Module for ADI Blackfin Processors. Running the project in debug mode on the target gives us the ability to interact with the graphical front panel interface and dictate which tones are generated by the Blackfin processor. We can then use the DTMF decoder application from Hands-on Experiment 7.11 to receive and decode the resulting touch-tone signals.

Open the `DTMF - BF537.lvproj` project located in the directory `c:\adsp\chap7\exp7_12`. Notice that there are four VIs in the project and the top level VI is `DTMF - BF537.vi`. Double-click this VI to open its front panel as shown in Figure 7.35.

Figure 7.35 Blackfin DTMF encoder (Front panel for `DTMF – BF537.vi`).

Debug mode allows users to press one of the 16 Boolean buttons within the Boolean array input and generate the corresponding dual-tone signal. Wires on the block diagram can also be probed to view the data being passed along wires. Open the block diagram for `DTMF - BF537.vi` to see how the DTMF encoder is implemented in the graphical LabVIEW code, as shown in Figure 7.36.

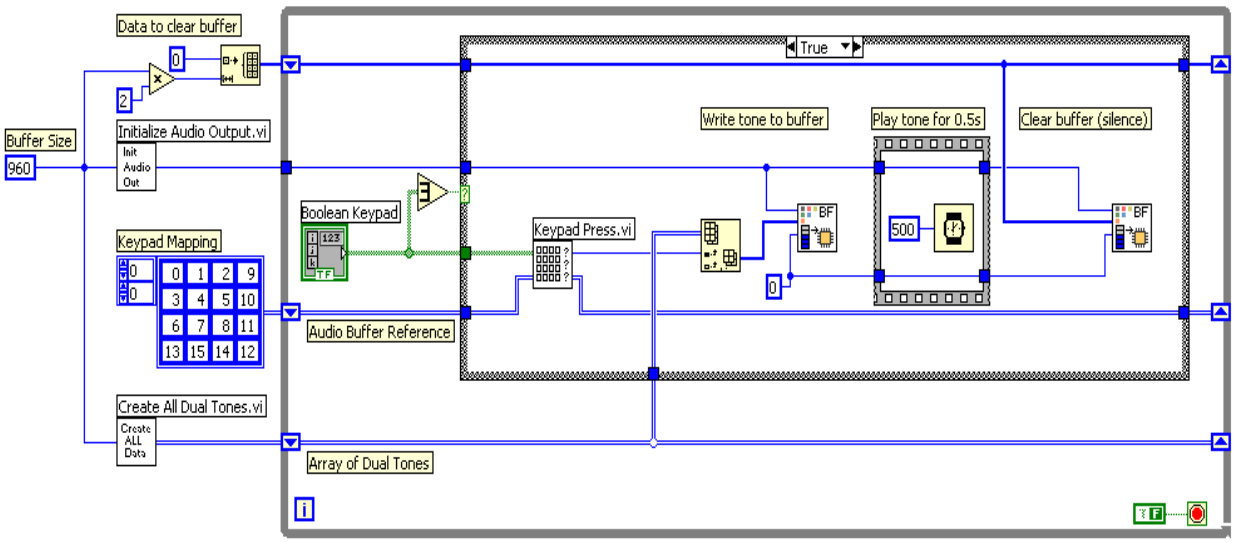


Figure 7.36 Blackfin DTMF encoder (Block diagram for `DTMF – BF537.vi`).

The data flow determines the execution order which is generally from left to right. The process for outputting a key pressed on the keypad on the front panel follows several distinct steps. First, the program evaluates if a key has been pressed by calculating a Boolean logical OR of every element in the array. If the result is true, the key press is detected and the resulting signal is sent to the DAC output buffer. The buffer is allowed to output for the 500 msec, and then the output buffer is set to zero. In this instance, data flow is used to implement a 500 ms delay between the two output buffer write VIs by preventing the program from continuing until the timer within the sequence structure is complete. In this main VI, open all three subVIs one by one and explore their block diagrams to see how their functionality contributes to the overall DTMF encoder.

Execute the application on the Blackfin EZ-KIT in debug mode, and make sure that the audio output is wired to speakers (or headphones). Press buttons on the front panel to hear the tones created by the Blackfin processor. Next, wire the audio out from the Blackfin processor to the audio/microphone input on the computer. Open the `DTMF_Decoder_Sim.exe` that was used in the previous experiment to view and decode the signals generated by the Blackfin processor.

8.10 Sample-Rate Conversion using LabVIEW Embedded Module for ADI Blackfin Processors

In practical engineering applications, it is often necessary to change the sampling rate of signals within a system. For example, the Blackfin EZ-KIT operates with an audio CODEC that digitizes signals at 48 kHz, which is a common rate for capturing audio signals. It is often desirable to process signals using a lower sampling rate, such as 8000 Hz, which minimizes the amount of computation necessary by reducing the number of samples. Sample rate conversion is commonly used in industrial applications for

analyzing and manipulating communication signals, mixing signals from multiple sources, and noise cancellation.

The following experiments use LabVIEW to illustrate the concepts of sample-rate conversion. We will control the sampling rates of signals and observe what design considerations are affected by re-sampling. The interactive simulation allows you to see and hear how decimation and interpolation filters can be used to manipulate and process audio signals. This understanding will further reinforce these concepts while using the Blackfin EZ-KIT target.

Hands-on Experiment 8.18:

This experiment reinforces the concepts of sample-rate conversion. This conversion allows us to combine and process signals sampled at different rates within the same system. This experiment begins with an input signal sampled at a frequency of 48 kHz and then analyzes the results of applying decimation and interpolation filters to decrease and increase the sampling rate, respectively. Remember from Section 8.8 that the downsampling process first requires lowpass filtering of the input signal to prevent aliasing by ensuring that the signal satisfies the Nyquist rate.

Open the compiled LabVIEW simulation `Resample_Sim.exe` located in the directory `c:\adsp\chap8\exp8_18` to explore sample-rate conversion. The graphical user interface is similar to those in earlier chapters and is shown in Figure 8.15. The interface shares many controls, indicators, and functionality used in previous hands-on LabVIEW experiments.

The tabs on the **Tab control** are unique, allowing the user to choose which signals to view, including input, downsampled, and upsampled signals in both the time and frequency domain. The **Upsampled Output** tab enables the **Upsample To** rate selector, which allows each of the upsampled rates to be seen and heard. The frequency slider allows the frequency of the default test sine wave to be manipulated. The sine wave input signal is sampled at 48 kHz to simulate that of the Blackfin EZ-KIT, downsampled by a factor of six to 8 kHz, and then upsampled to 16 kHz, 32 kHz, or 48 kHz. The 32 kHz and 48 kHz sampling rates are achieved by first upsampling by a factor of two to 16 kHz, and then upsampling the resulting signal by factors of two or three, respectively.

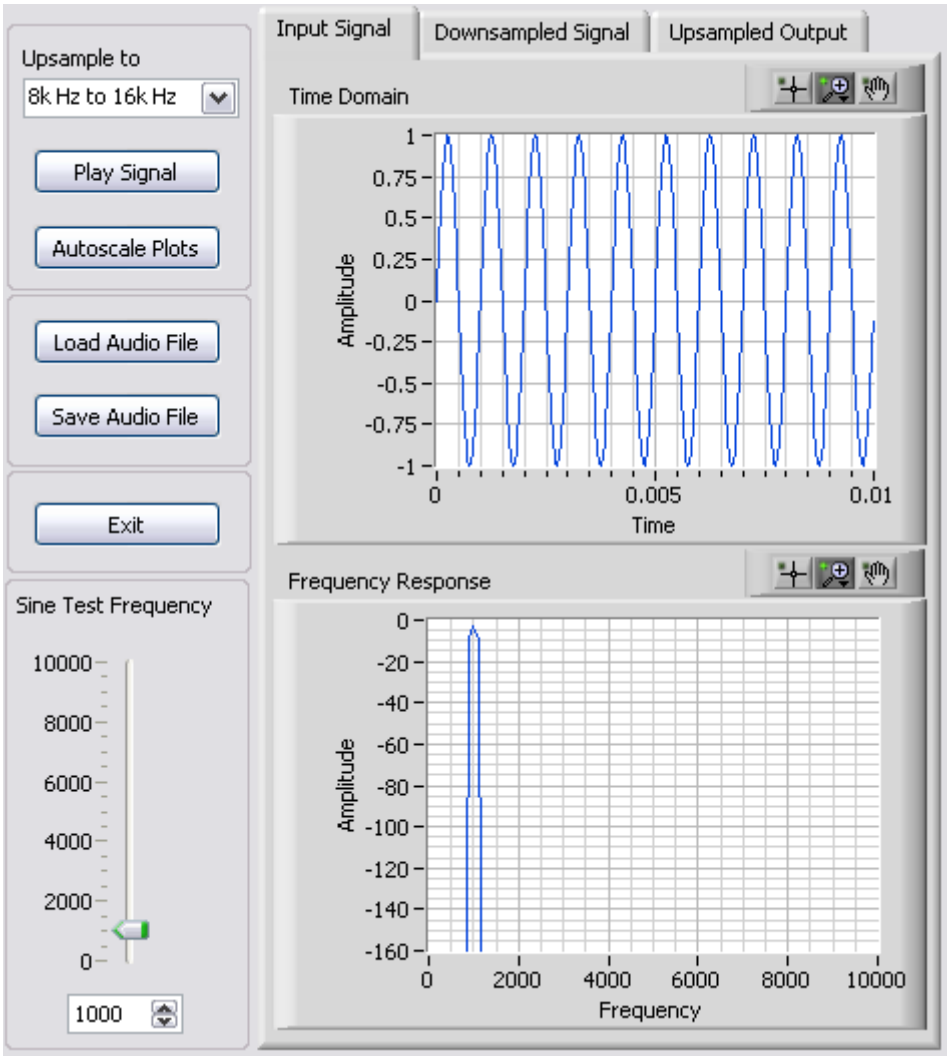


Figure 8.15 User interface for `Resample_Sim.exe`.

Upon opening the simulation, a default sine wave signal with 1 kHz frequency is used as the input. Explore the different tabs to gain an understanding of how the signal changes when decimation and interpolation filters are applied. Also, manipulate the signal frequency while observing its time and frequency domain data. Pay attention to the effects of sample-rate conversion on audio bandwidth. What observations can you make about the time-domain plots of the input signal after its sampling rate has been altered? Can you hear these effects when playing back the altered signals? Experiment with different input sine wave frequencies by changing the value of the **Sine Test Frequency** slider control.

Now load an audio file to experiment with the effects of decimation and interpolation. You can use the `speech_tone_48k.wav` file from Hands-on Experiment 2.5. Play the signal at different sampling rates to determine how sample-rate conversion affects the audio quality. What observations can you make? How might consumer electronics, digital audio media, or the music recording industry utilize these concepts?

Hands-on Experiment 8.19:

This experiment examines a LabVIEW Embedded Module for ADI Blackfin Processors project that employs decimation and interpolation filters to mix signals of different sampling rates. LabVIEW programming structure and code is reused from previous exercises to maintain a similar program structure and more efficiently develop a sample-rate conversion application that can be deployed to the Blackfin EZ-KIT.

Open the `Resample_Ex - BF537.lvproj` project located in the directory `c:\adsp\chap8\exp8_19`. Open the top level VI named `Resample_Ex – BF537.vi` and view its block diagram shown in Figure 8.16. This VI provides the ability to acquire live audio at 48 kHz, down-sample it by a factor of six to 8 kHz, mix the 8 kHz signal with a 1 kHz sine wave sampled at 8 kHz, and then up-sample back to 48 kHz for audio output. Which portions of the block diagram look familiar?

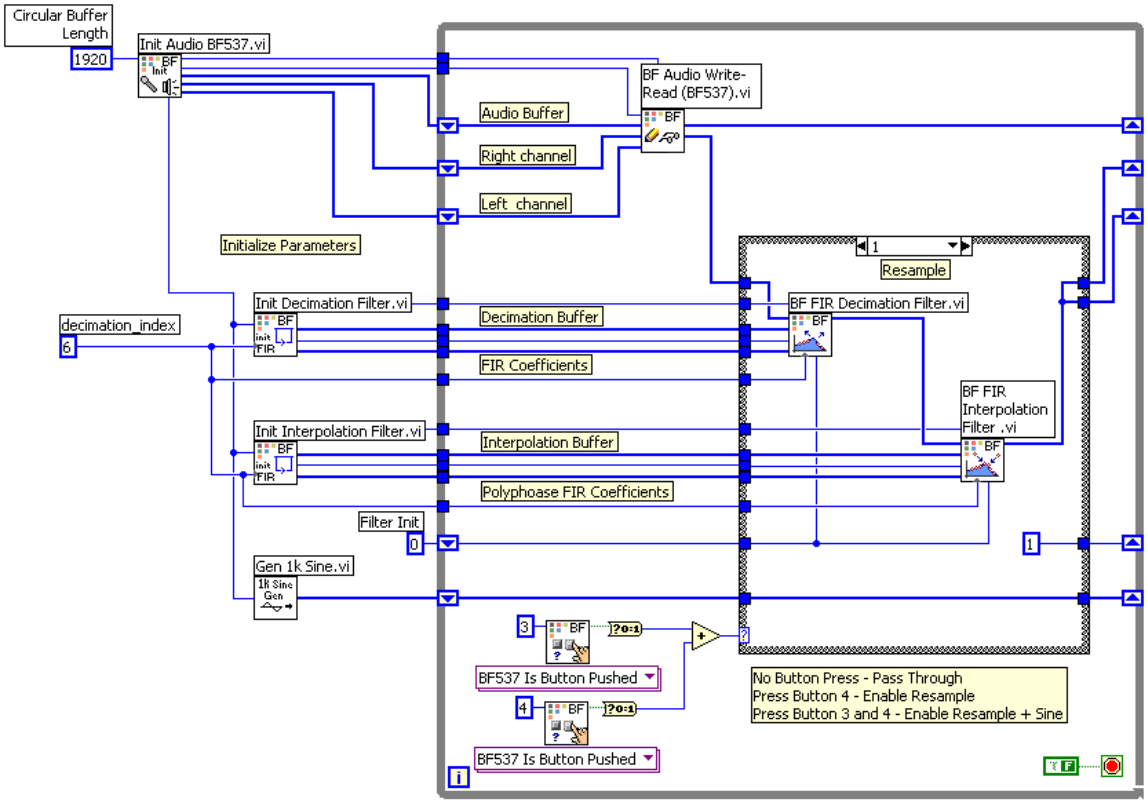


Figure 8.16 Block diagram of `Resample_Ex – BF537.vi`.

Note that the structure for writing and reading audio is common to most previous LabVIEW Embedded Module for ADI Blackfin Processors audio exercises. In particular, the `Init Audio BF537.vi` and the `BF Audio Write-Read (BF537).vi` set the audio buffer sizes, generate and acquire the audio signals. After acquiring a buffer of data, the sample rate is converted, mixed, and output in the next loop iteration. As in previous experiments, the graphical program is logically organized using subVIs that allows modular pieces of the program to be reused, thus, reducing development time.

The three custom subVIs, found in the lower left of the block diagram, initialize the filters and sine wave generation before the while loop begins. The `Init`

`Decimation Filter.vi` and `Init Interpolation Filter.vi` both store filter coefficients and create the necessary data buffers to perform the FIR decimation and interpolation filtering inside the processing loop. A 1 kHz sine wave sampled at 8 kHz is generated from an 8-point lookup table by the `Gen 1k Sine.vi`, and is mixed with the downsampled live audio when the correct case is executed. These subVI functions are part of the project and can be seen in the Project Explorer Manager window below the main VI.

The case structure in the processing loop determines which of the three processing modes is used. The three modes include audio talk-through, resample, and resample with mix. The button selections for the EZ-KIT hardware are shown in Table 8.21.

Table 8.21 Switch selections for different modes.

Mode	**Switch on BF537 EZ-KIT**
(1) Pass-through	None
(2) Downsample to 8 kHz→ Mixing at 8 kHz→ Upsample to 48 kHz	SW10/PB4 and SW11/PB3 pressed together
(3) Downsample to 8 kHz→ Upsample to 48 kHz (without mixing)	SW10/PB4

The case structure in the center of the VI contains the three cases seen in Figure 8.17. One of the three cases is executed each iteration of the loop depending on the button combination being pressed. Several wires are required to configure the decimation and interpolation filters, but the general flow of the input signal is easily followed.

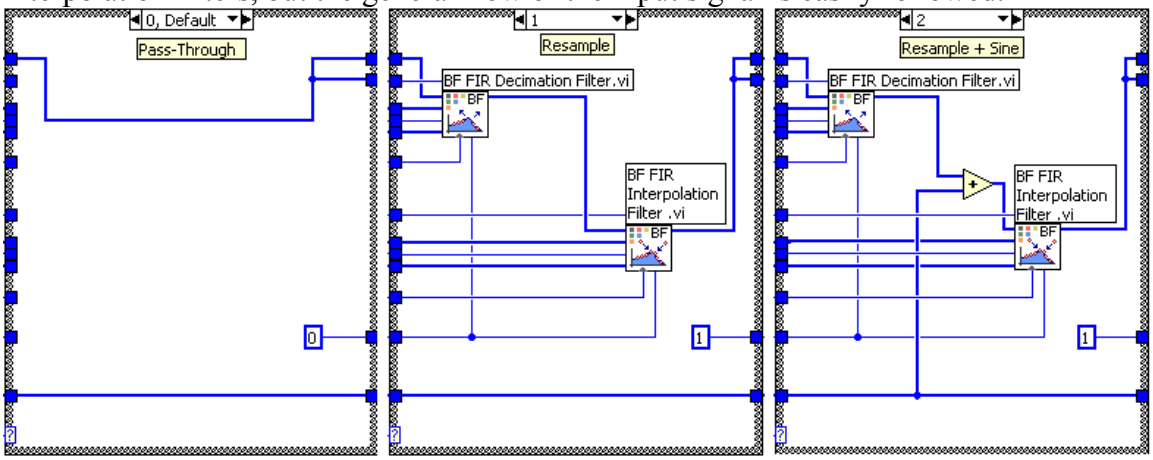


Figure 8.17 Cases in the `Resample_Ex – BF537.vi` graphical code.

Connect the audio input of the Blackfin EZ-KIT to the PC sound card or CD player. Then, compile and run the LabVIEW Embedded project on the Blackfin processor. Press different switch combinations to hear each of the three audio modes. What differences are heard between audio pass-through and the resampled audio signals? Explain. Can you hear the mixed 1 kHz tone?

9.6 Implementation of MDCT using LabVIEW Embedded Module for ADI Blackfin Processors

The MDCT, defined earlier in section 9.2.3, divides the audio signal into overlapping frequency bands and later recombines them using the inverse MDCT (IMDCT), defined in section 9.2.4, to reproduce the signal with minimal distortion. In this experiment, we will focus on the IMDCT which is used in the Ogg Vorbis decoder. First, we will analyze two methods of implementing the IMDCT in LabVIEW: the direct implementation of the IMDCT and the faster implementation using FFT. Next, the algorithms will be benchmarked with both LabVIEW for Windows and the Blackfin EZ-KIT. Benchmarking will be done by isolating the IMDCT and placing it in a simple structure that can be reused to study the performance of any function within LabVIEW or the LabVIEW Embedded Module for ADI Blackfin Processors.

Hands-on Experiment 9.4:

The IMDCT is an overlapping algorithm that generates an output data set that is twice the size of the input signal. The resulting signal is then overlapped and combined with the last 50% of the previous packet and the first 50% of the next packet to reconstruct the signal.

Launch LabVIEW and open the `Test_IMDCT_by_Eqn.llb` located in the directory `c:\adsp\chap9\exp9_4` to begin exploring the IMDCT. The Test_MDCT_Eqn VI, inside the .llb library, allows you to test and verify that the implementation of the IMDCT is correct. The input signal is a sine wave as shown in Figure 9.16. This test signal is unique in that the past, present, and next data packets are identical.

The input signal is passed through a window function to remove the beginning and ending discontinuities and then transformed using the MDCT, defined in section 9.2.3. Next, the IMDCT, defined in 9.2.4, is used to transform the DCT coefficients back to the time domain. The windowing function is applied again to combine the past, present, and next signal packets to complete the reconstruction of the original input signal.

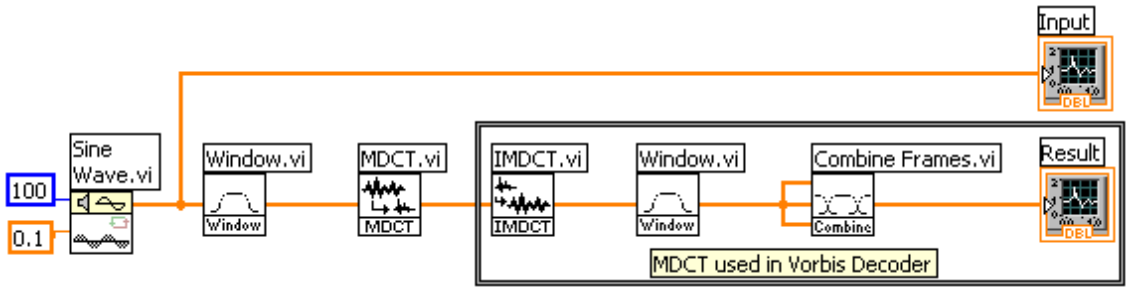


Figure 9.16 Direct implementation of MDCT and IMDCT, `MDCT by Eqn.vi`.

Run the VI and verify that the input and output signals are identical. Open the IMDCT subVI shown in Figure 9.17 and analyze the graphical code used to implement the algorithm.

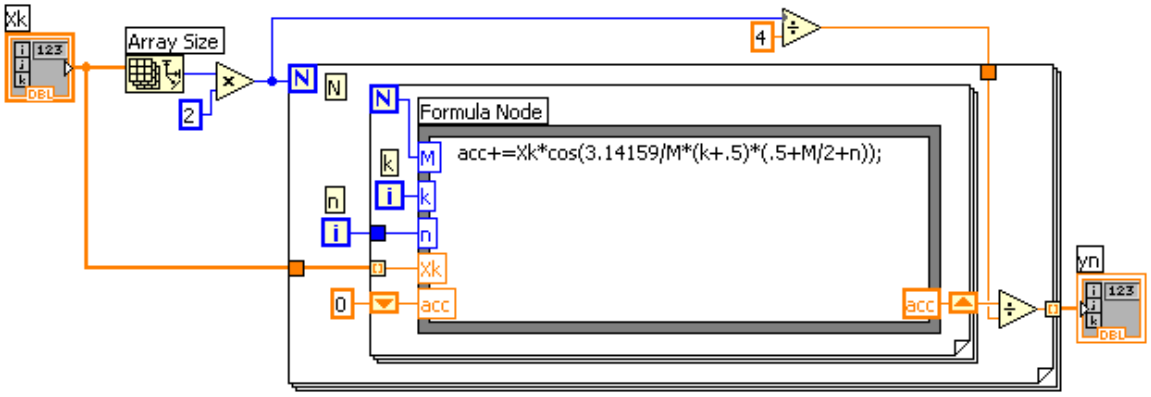


Figure 9.17 Direct implementation of IMDCT, `IMDCT.vi`.

As shown in Figure 9.17, the outer For Loop iterates $2N$ times, where N is the size of the input signal, to generate each point of the output signal. The Formula Node inside the inner For Loop calculates and accumulates all of the values according to the IMDCT equation. Notice that the output of the algorithm has twice the number of points as the input signal.

The properties of the FFT can be used to improve the speed of execution for computing the IMDCT. Ideally, when packets are created with a length that is a power of two, the radix-2 FFT algorithm can be used. The equations for calculating the IMDCT using the inverse FFT is derived in the following three steps. The detailed explanation can be found in reference [52]. Some additional processing is necessary, as shown in the LabVIEW implementation in Figure 9.18.

$$X'(k) = X(k) * e^{j\frac{2\pi}{N}kn_0}, \text{ where } n_0 = \left(\frac{N}{4} + \frac{1}{2}\right), \tag{9.6.1}$$

$$x(n) = IFFT\left[X'(k)\right], \tag{9.6.2}$$

$$y(n) = x(n) * e^{j\frac{2\pi}{N}(n+n_0)}, \tag{9.6.3}$$

where N represents the number of samples to be recovered, k is the frequency index, and n is the time index.

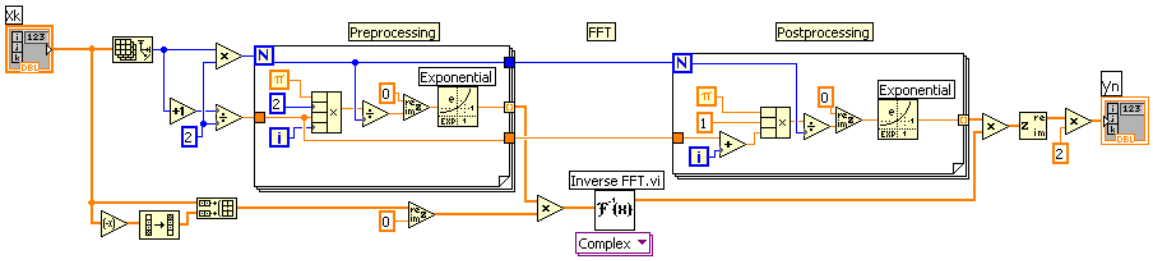


Figure 9.18 FFT implementation of IMDCT, `IMDCT_FFT.vi`.

Modify the MDCT by Eqn VI by replacing both `MDCT.vi` and `IMDCT.vi` with `MDCT_FFT.vi` and `IMDCT_FFT.vi`, respectively. To do this, right click on `MDCT.vi`, select **Replace → All Palettes → Select A VI…** and navigate to `MDCT_FFT.vi` found in the directory `c:\adsp\chap9\exp9_4\Test_IMDCT_by_FFT.llb`. Do the same for the `IMDCT_FFT.vi`. Run the exercise again to verify that this FFT implementation of IMDCT generates the same results as the direct implementation of the IMDCT.

Hands-on Experiment 9.5:

In this experiment, we benchmark the performance of the IMDCT using a common architecture that can be used to benchmark any function or algorithm in LabVIEW or the LabVIEW Embedded Module for ADI Blackfin Processors. DSP algorithms typically execute very fast, with sub-millisecond execution times. For this reason, we derive the average execution time by running the algorithm several times while measuring the total execution time. When operating on the embedded target, the final execution time is output to the standard output port using the One Button Dialog VI.

Open the `IMDCT_Benchmark.vi` located in `Bench_IMDCT.llb` in the directory `c:\adsp\chap9\exp9_5`. This VI provides a common benchmarking architecture for testing any application. Figure 9.19 shows that within the benchmarking VI, the data set **Xk** simulates a 128-point packet of DCT input data. The IMDCT has been placed in the For Loop so that it can execute several iterations in order to calculate an average execution time per iteration.

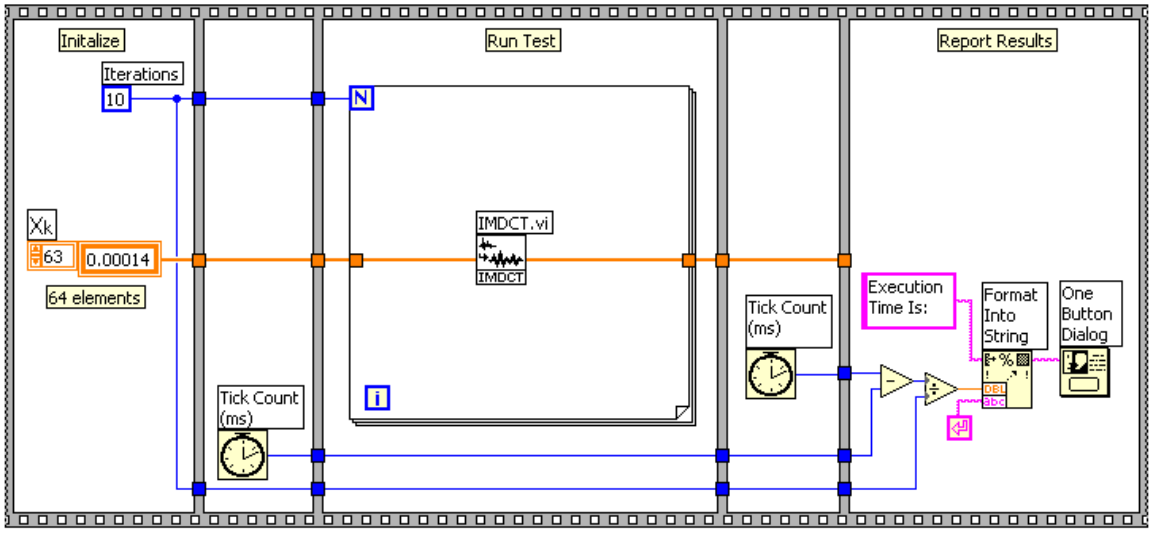


Figure 9.19 A benchmarking structure, `IMDCT_Benchmark.vi`.

Use Table 9.4 to record your execution results. Fill out the benchmarking table by running the VI shown in Figure 9.19 targeted to Windows. Record the resulting execution

time in the top-left cell of Table 9.4. Then, right-click the `IMDCT.vi` and replace it with `IMDCT_FFT.vi`. Run the VI again and record the result.

Table 9.4 Execution time for different platforms.

	Execution time in milliseconds		
	Windows	**Blackfin default**	**Blackfin optimized**
`IMDCT.vi`			
`IMDCT_FFT.vi`			

After you have completed testing in Windows, change the execution target to the Blackfin EZ-KIT, by opening the project, `IMDCT_Benchmark_Default_BF537.lvproj,` located in the `c:\adsp\chap9\exp9_5` directory. Execute the VI on the embedded target. The resulting time should be sent back to the PC through the standard output port and can be seen on the **Output** tab of the **Processor Status** window. Once you have tested `IMDCT.vi`, again replace it with the `IMDCT_FFT.vi`.

Finally, close the current project and open the new project `IMDCT_Benchmark_Optimized_BF537.lvproj` located in the directory `c:\adsp\chap9\exp9_5`. Execute the VI on the embedded target and record the execution results. Again replace the `IMDCT.vi` with `IMDCT_FFT.vi` to test execution time of both IMDCT computation methods. Examine the configured Blackfin Build Options from within the Project Explorer to see which optimizations were enabled.

Once the table is complete, study the results. Which compiling optimizations effectively reduce the execution time? What methods could be used to further increase the execution speed of IMDCT? Explain. Also, consider data types and the analysis library used.

Note: Do not try to run the Windows benchmarking test when a Blackfin project is open. The VIs will try to run on the Blackfin EZ-KIT as the project refers to the same library. Close the embedded project and then run the Windows benchmarking tests.

10.11 Image Processing using the LabVIEW Embedded Module for ADI Blackfin Processors

Images are commonly processed by the same signal processing principles we have introduced in previous chapters. Image processing can be performed in real time, for example, to compress live video to store it to disk, or as a post-processing step such as manually touching up a digital photo before printing.

The following experiments utilize graphical system design to prototype and implement signal processing algorithms in Windows and then on the Blackfin EZ-KIT. First, we will open the application and execute it with LabVIEW targeted to Windows for conceptual analysis, and development. The execution target will then be changed to the Blackfin processor, allowing the execution of the same code on the EZ-KIT. This ability to program and develop algorithms independent of the intended hardware target is the basis for true graphical system design and code-sharing across platforms.

Hands-on Experiment 10.7:

The extensive libraries of analysis functions in LabVIEW can be applied to both one- and multi-dimensional data. In this experiment, we will explore the effects of basic mathematical operations on a 2D image. The subtraction operation is used to invert the image. Division allows us to reduce the number of quantized values present in the image. Multiplication affects the contrast. We will also explore thresholding, which converts the grayscale image to black and white. These types of operations are common when preparing an image for additional processing or to highlight specific details. For example, inverting an image can often make dark or faint features easier to see with the human eye.

Begin by opening the `Basic Image Processing.vi` located in the `c:\adsp\chap10\exp10_7` directory. Open the block diagram and examine the graphical code. Notice that the original image, represented by an array, is 100 × 96 pixels with values ranging from 0 to 255. The value of the tab control is used to specify which processing operation in the case structure is executed. Manually change the visible case of the **Case** structure to view the **Invert** case on the block diagram. Image inversion is calculated with equation 10.11.1 and implemented in Figure 10.23.

Inversed Image = 255 – Input Image (10.11.1)

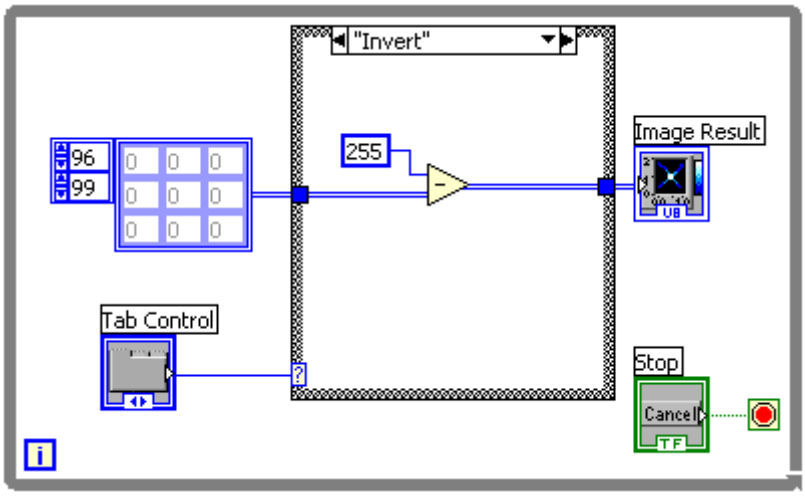


Figure 10.23 Block diagram for `Basic Image Processing.vi`.

Polymorphism allows the Subtract function to receive inputs of different representation and dimensionality and handle them appropriately. The polymorphic Subtraction function in LabVIEW allows the original 2D image to be subtracted from the scalar value by simply wiring the two values to its inputs. The result is then displayed on the **Image Result** indicator. The other processing cases contain algorithms that require some additional function blocks. The **Multiply** case, for example, needs additional code to protect against numerical overflow. The **Threshold** case uses the **Greater Than or Equal To** function to evaluate whether the pixel value is above or below the specified threshold. Each of these cases produces a resulting image that is displayed on the same **Image Result** intensity graph.

Run the VI from the Front Panel. Move through the tabs one at a time and examine the effects of each processing algorithm. Are the resulting images what you expected?

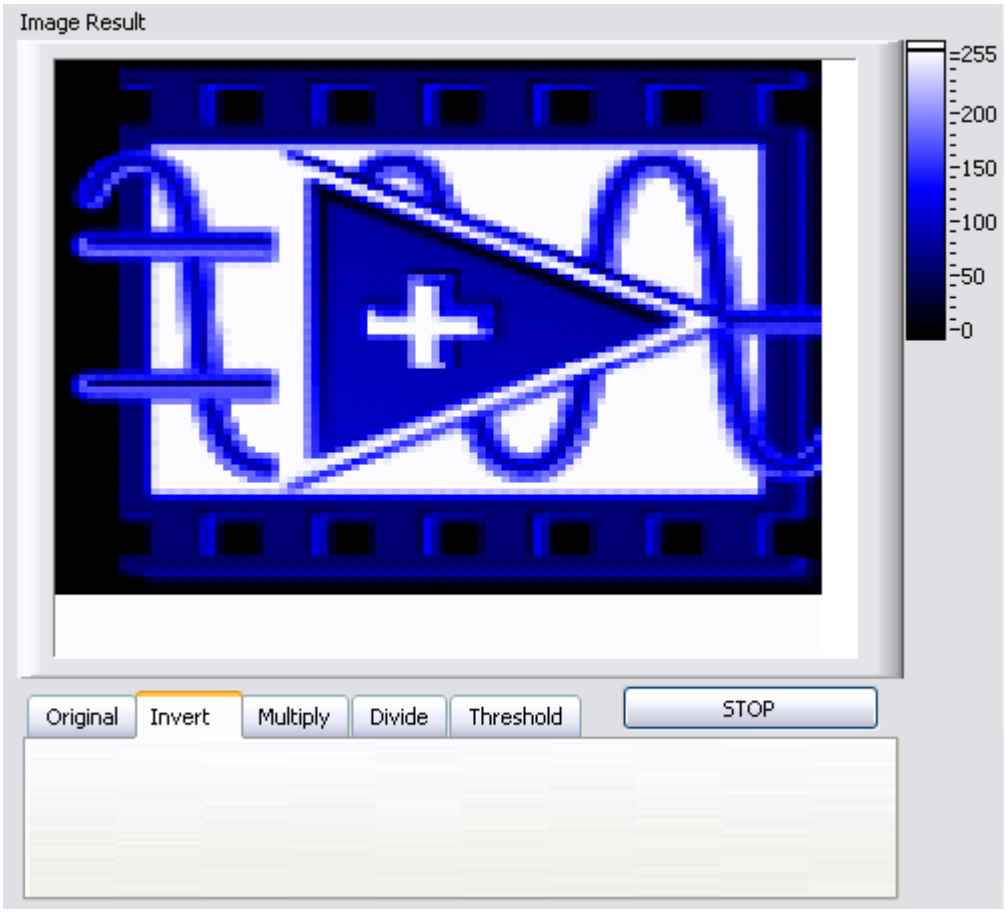


Figure 10.24 Front panel for `Basic Image Processing.vi`.

A common form of image optimization is reducing the number of colors used to represent the image. Using the **Divide** tab, what value for Y results in an image with four colors? Does it correspond to the algebraic equation 255 / Y = 4, when you solve for Y?

To target the BF537 Blackfin Processor, open the `Basic Image Processing.lvproj` located in the `c:\adsp\chap10\lv_exp10_7` directory.

As we discussed earlier, the input is an 8-bit grayscale image containing 100 × 96 pixels. However, the default LabVIEW Embedded Module for ADI Blackfin Processors debugging mechanism only reports the first 128 data points for any array. This number must be increased to greater than 9600 to properly see the results of processed image. To change the default debugging array size, navigate to **Target → Configure Target → Debugging Options** from the Project Explorer window. Increase the **Max Array Elements** to `10000`. This will allow the entire image to be retrieved during debug execution. Increasing the **Debug update period** to `1000` ms (1 second) will also improve performance by causing fewer interrupts to the processor during execution. Click **OK** and return to the Project Explorer window.

Run the VI in debug mode to download, run, and link the processor to the LabVIEW front panel. Click through the tabs to verify that the same results are achieved on the Blackfin processor as in Windows. This is an excellent example demonstrating how LabVIEW abstracts the hardware target from design, allowing you to focus on the algorithms and concepts. Click the **Threshold** tab. Can you calculate how much more

data it takes to represent a 100 × 100 pixel, 8-bit, 256 color level image in memory compared to a 1-bit, 2 color level, image?

Hands-on Experiment 10.8:

This experiment implements a 2D image filter that consists of a 3 × 3 kernel matrix. We utilize the concepts of graphical system design to implement and test the filter in LabVIEW targeted to Windows and then port the design to the Blackfin processor.

Open the `2D Convolution Image Processing.llb` project provided in the directory `c:\adsp\chap10\lv_exp10_8`. As in Hands-on Experiment 10.7, this VI can be executed with the LabVIEW Embedded Module for ADI Blackfin Processors when targeted to Windows or to the Blackfin processor. When analyzing the block diagram, we see that the image filter kernels are the same as those designed earlier in the Example 10.7.

The lowpass filter implementation is shown in Figure 10.25. The 2D convolution subVI performs the point-by-point convolution of the image and the 3 × 3 filter kernel, and then divides by the scalar, in this case 9. The scalar is used to reduce the overall multiplied effect of the filter kernel to 1. Open the 2D Convolution subVI to see how the graphical program performs the 2D convolution.

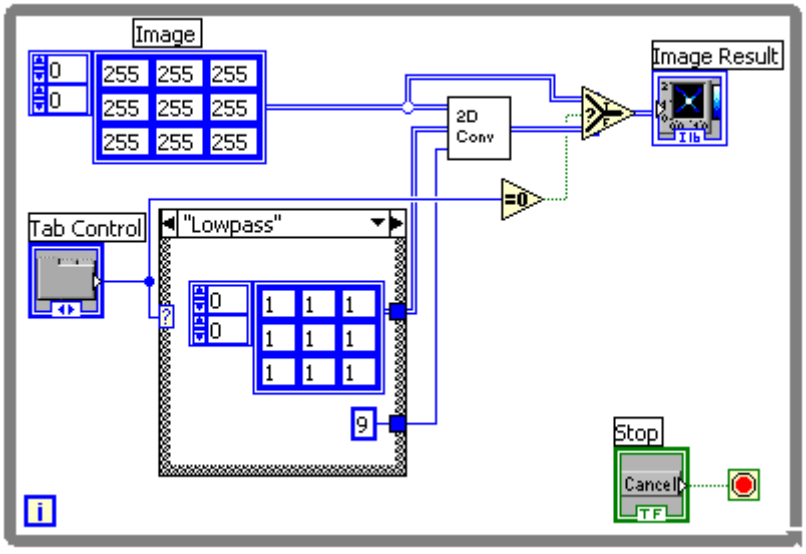


Figure 10.25 Block diagram for `2D Convolution Image Processing.vi`.

Open the front panel and run the VI. Click the **Custom** tab shown in Figure 10.26. The Laplacian filter is performed by default. Change the filter to the Sobel using the parameters found in Example 10.7. Notice that there are two Sobel filter implementations, which are not symmetric about both axes in contrast to the delta and lowpass filters. How do the resulting images differ when the orientation of the Sobel filter is changed?

To run the project on the Blackfin processor, open the `2D Convolution Image Processing.lvproj` project file in the directory `c:\adsp\chap10\lv_exp10_8`.

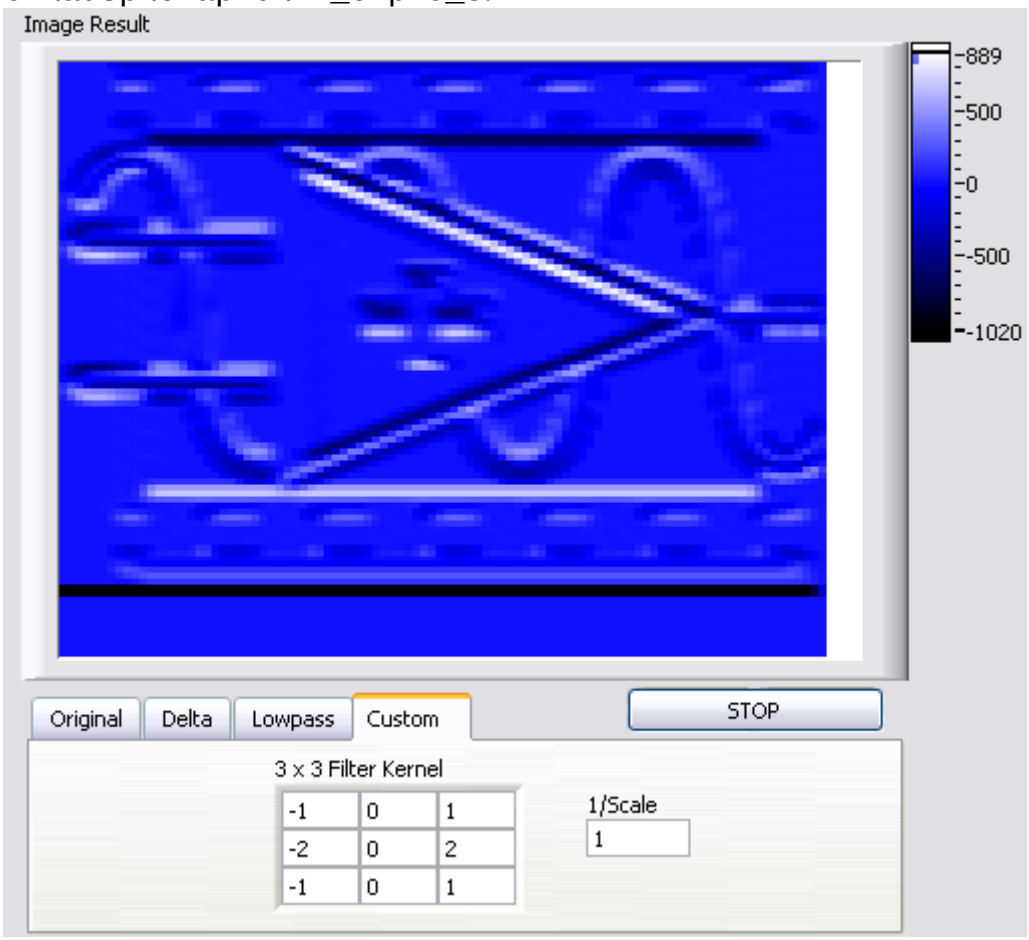


Figure 10.26 Front panel for `2D Convolution Image Processing.vi`.

As discussed earlier, the input is an 8-bit grayscale image containing 100 × 96 pixels. Again, the default LabVIEW Embedded Module for ADI Blackfin Processors debugging parameters must be changed properly to see the results of processed image. In the project **Debugging Options**, increase the **Max Array Elements** to `10000` and the **Debug update period** to `1000` ms.

Run the VI on the Blackfin by clicking the **Debug** execution button. As in Hands-on Experiment 10.7, be sure that the behavior of the algorithms on the Blackfin processor behaves the same as it did in LabVIEW targeted to Windows.

Explore the values being passed through wires on the block diagram using probes. The custom tab is useful for testing common 3 × 3 image filter implementations as well as your own designs. What 3 × 3 filter values allow you to invert the image as shown in Figure 10.24?

Appendix A: An Introduction to Graphical Programming with LabVIEW

Contributed by National Instruments

This appendix provides an overview of National Instruments LabVIEW and the LabVIEW Embedded Module for ADI Blackfin Processors software.

A.1 What is LabVIEW?

LabVIEW is a full-featured graphical programming language and development environment for embedded system design. LabVIEW provides a single graphical design tool for algorithm development, embedded system design, prototyping, and interfacing with real-world hardware. Additional modules have been designed to expand the core functionalities of LabVIEW to real-time operating systems, DSP, and FPGA programming making LabVIEW an ideal platform for signal processing algorithm design and implementation.

This section briefly introduces the LabVIEW development environment from installation and basic programming to system development. This material provides a high-level overview of the concepts necessary to be successful with LabVIEW and the LabVIEW Embedded Module for ADI Blackfin Processors. Additional references to supplemental manuals and resources are provided for more in-depth information.

A.1.1 A Picture Is Worth a Thousand Lines of Code

Graphical system design and prototyping opens the door to the masses in embedded development. The industry is confirming that higher levels of abstraction in embedded tools are needed, suggesting electronic system-level (ESL) design is an answer and citing concepts such as hardware/software co-design, predictive analysis, transaction-level modeling, and others. Does this sound like simplifying the problem or just making it more complex?

Fundamentally, the problem is that the domain experts – scientists, researchers, and engineers developing new algorithms – are at the mercy of the relatively few embedded system developers who are experienced in dealing with today's complex tools. The industry requires a fundamental change to empower the thousands of domain experts to experiment and prototype algorithms for embedded systems to prove success early before they are overtaken with the complexity of implementation.

The test and measurement industry saw this same phenomenon. Using a graphical modeling and programming approach, engineers can more quickly represent algorithms and solve their problems through block diagrams. This is especially true for embedded systems development. With a graphical programming language, engineers succinctly embody the code that is actually running on the hardware. For example, they can represent parallel executing loops simply by placing two graphical loop structures next to

each other on the diagram. How many lines of code would it take to represent that in today's tools? How many domain experts could look at text code and quickly interpret exactly what is happening? Only a truly innovative user experience – like Windows for PCs or the spreadsheet for financial analysis – can truly empower the masses to move into a new area.

Domain experts also need a simple and integrated platform for testing algorithms with real-world I/O. The reason these embedded systems fail so often is because there are so many unknowns during development that are not discovered until the end of the development cycle.

After seeing this shift in the test and measurement industry from traditional text-based programming approaches like BASIC and C to LabVIEW, it seems more obvious than ever. Graphical system design is essential to open embedded systems to more people.

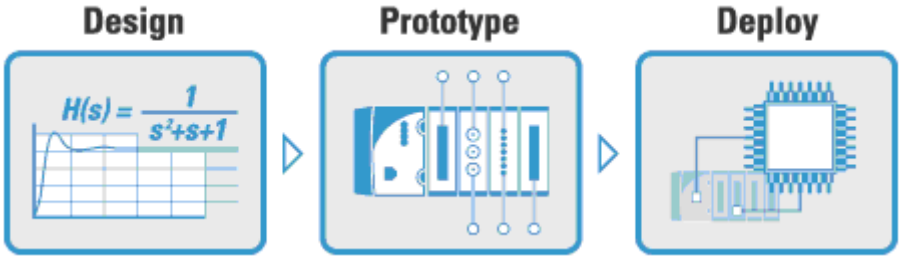


Figure A.1 Graphical system design in LabVIEW.

A.1.2 Getting Started with LabVIEW

The LabVIEW development system is the core software with which other modules, toolkits, and drivers can be installed to expand functionality and capability. For the purposes of this book, the LabVIEW Embedded Module for Analog Devices Blackfin Processors should also be installed, adding the ability to generate Blackfin-compatible C code that can be targeted to the Blackfin EZ-KIT using the Analog Devices VisualDSP++ tool chain under the hood.

A.1.3 Install Software

The installation order is as follows:

(1) Install VisualDSP++ 5.0 for LabVIEW
(2) Install the LabVIEW Embedded Module for ADI Blackfin

A.1.4 Activating Software

The software will default to a 60 day evaluation version. If you have purchased a copy of the software and have received an activation code, follow these steps to activate both the LabVIEW Embedded Module for ADI Blackfin and the Visual DSP++ 5.0 at once.

(1) Navigate to the NI License Manager Start by selecting **Start → Programs → National Instruments → NI License Manager**. The NI License Manager will open, showing all NI software installed on the computer that can be activated.

(2) Expand the LabVIEW category to expose the LabVIEW Embedded Module.

(3) Right click on the **Embedded Module for Blackfin Processors** and select **Activate**. Follow the steps provided in the **NI Activation Wizard** to activate the software.

A.1.5 Connecting Hardware

After the appropriate software has been installed, attach the power connector to the Blackfin EZ-KIT and then connect the EZ-KIT to the computer using the USB cable. Windows will detect the EZ-KIT and install the appropriate driver.

A.1.6 Running an Example Program

All the LabVIEW Blackfin examples can be found in the `C:\Program Files\NationalInstruments\LabVIEW8.5\examples\lvemb\Blackfin` directory. Another way to get to the examples would be to use the **Example Finder** which can be accessed through **Help → Find Examples**. Select the **Search** tab and search for **Blackfin**.
After opening an example, click the **Run** button to compile, link, and download the application to the Blackfin EZ-KIT.

A.2 Overview of LabVIEW

A.2.1 The LabVIEW Environment

The LabVIEW development environment allows you to develop graphical programs and graphical user interfaces easily and effectively. A program in LabVIEW is called a Virtual Instrument (VI), and acts as an individual function similar to a function defined in the C programming language. The main executable VI is called the top-level VI and VIs used as modular subroutines are called subVIs. Every VI is made of two key components, the *front panel* and *block diagram*, shown in Figure A.2. The front panel is the graphical user interface and the block diagram contains the graphical code that implements the functionality of the VI. Once you create and save a VI, it can be used as a subVI in the block diagram of another VI, much like a subroutine in a text based programming language.

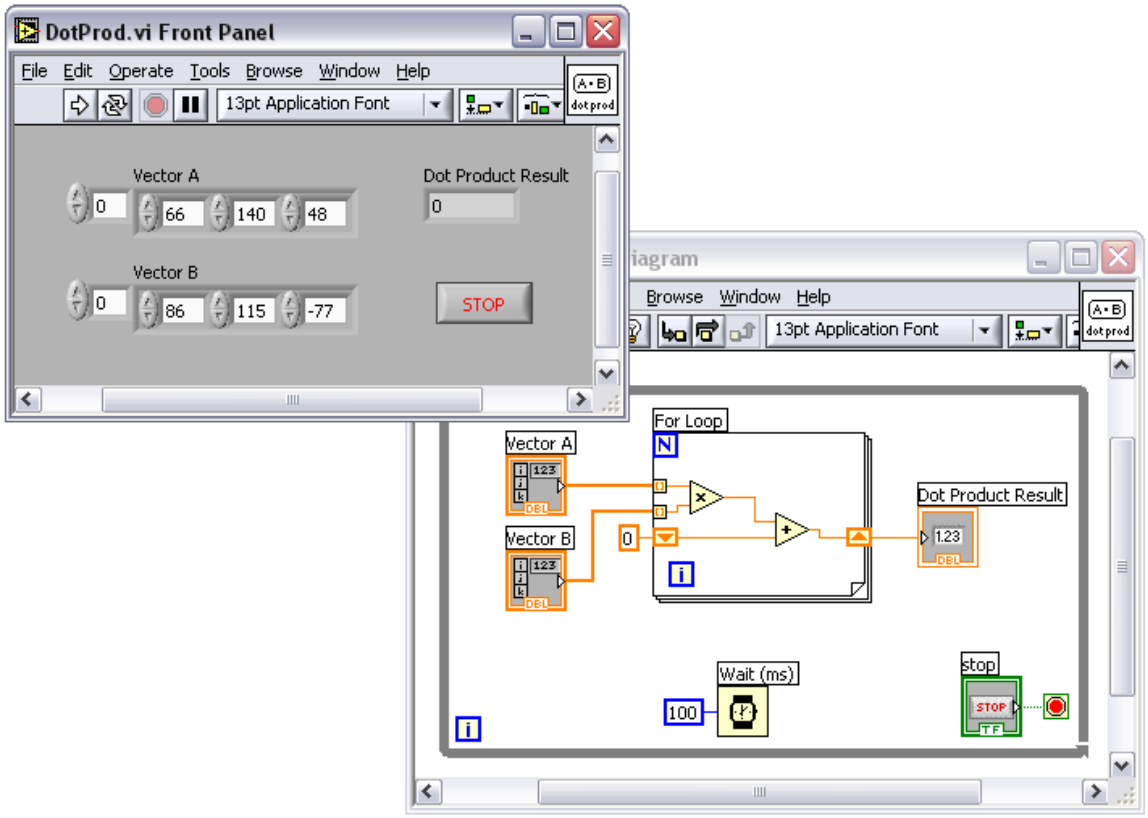
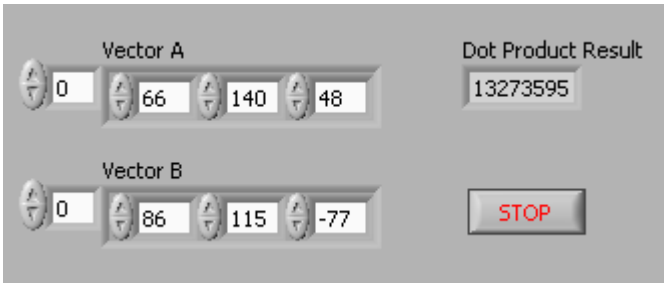


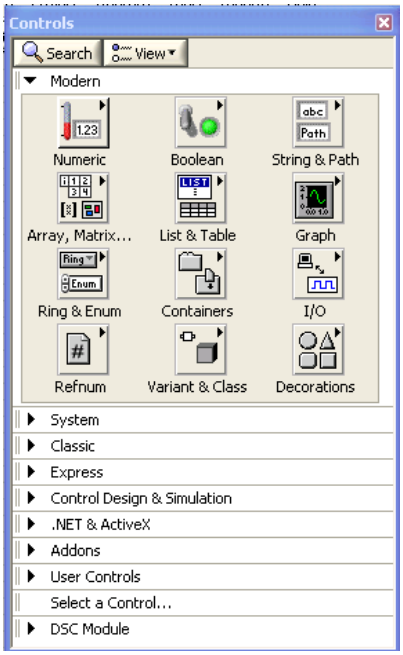
Figure A.2 LabVIEW Front Panel and Block Diagram.

A.2.2 Front Panel

The front panel contains objects known as controls and indicators. Controls are the inputs that provide data to the block diagram as discussed in A.2.3, while output data from the block diagram is viewed and reported on front panel indicators. In Figure A.3 (a), two numeric array controls, Vector A and Vector B, allow the user to input two numeric vectors. The result of the dot product is displayed as a single scalar value in the **Dot Product Result** numeric indicator. There is also a Boolean control button that will stop the VI.

The **Controls** palette in Figure A.3 (b) contains all the front panel objects that you can use to build or modify the user interface. To open the **Controls** palette, right click the free space within the front panel. The **Controls** palette can remain visible while you work by clicking the pushpin icon  in the upper left corner.





(a) (b)

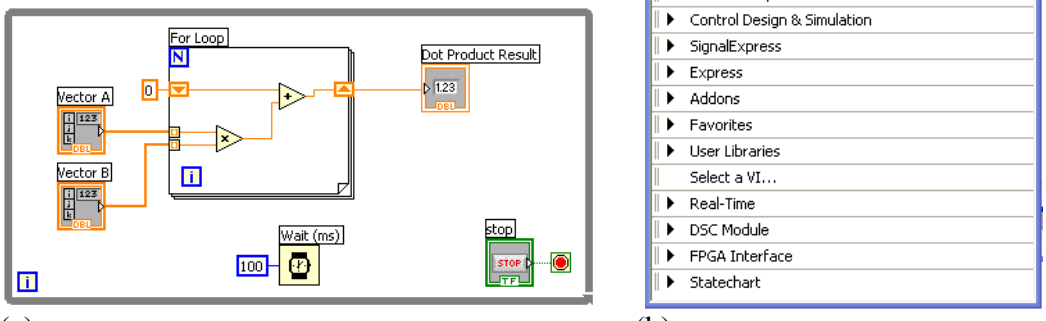
Figure A.3 (a) Front Panel and (b) Controls Palette.

A.2.3 Block Diagram

The block diagram contains the graphical code that defines how the VI will operate. Front panel controls and indicators have a corresponding terminal block on the block diagram. The corresponding terminals for array controls Vector A, Vector B and the Boolean stop button provide input to the block diagram, while the Dot Product Result numeric indicator terminal allows the result of the algorithm to be displayed.

Wires define how data will flow between terminals and other nodes on the block diagram. VIs execute based on the concept of data flow, meaning that a node, subVI or structure, will not execute until data is available at all its inputs. Data wire color and thickness are used to show data type (floating point number, integer number, Boolean, string, etc…) and dimensionality (scalar or array). For example, in Figure A.4 (a), the input vectors both enter the left side of the For Loop with a thick orange wire representing double precision numeric 1-D *arrays*. The wire exiting the For Loop is a thin orange wire representing a double precision numeric *scalar*.

When creating or modifying a block diagram, the **Functions** palette provides access to all functions, structures, and subVIs available in LabVIEW. To access the **Functions** palette, right-click the empty space on the block diagram, as shown in Figure A.4 (b). Again, the pushpin can be used to keep the palette from disappearing.



(a) (b)

Figure A.4 Functions palette for block diagram objects.

A.2.4 Debugging in LabVIEW (Windows)

Many debugging features in LabVIEW make it ideal for designing and testing algorithms before moving to a hardware target like the Blackfin EZ-KIT. LabVIEW continuously

compiles code while it is being developed, allowing errors to be detected and displayed before the application is executed. This method of error checking provides immediate feedback by ensuring that subVIs, and structures have been wired properly during development. Programming errors are shown in the form of broken or dashed wires or a broken **Run** arrow toolbar icon. When the **Run** arrow toolbar icon appears broken and grey in color, click it to see the list of errors preventing the VI from executing, shown in Figure A.5. Use the **Show Error** button or double-click the error description to highlight the problematic code on the block diagram or the object on the front panel that is causing the error.

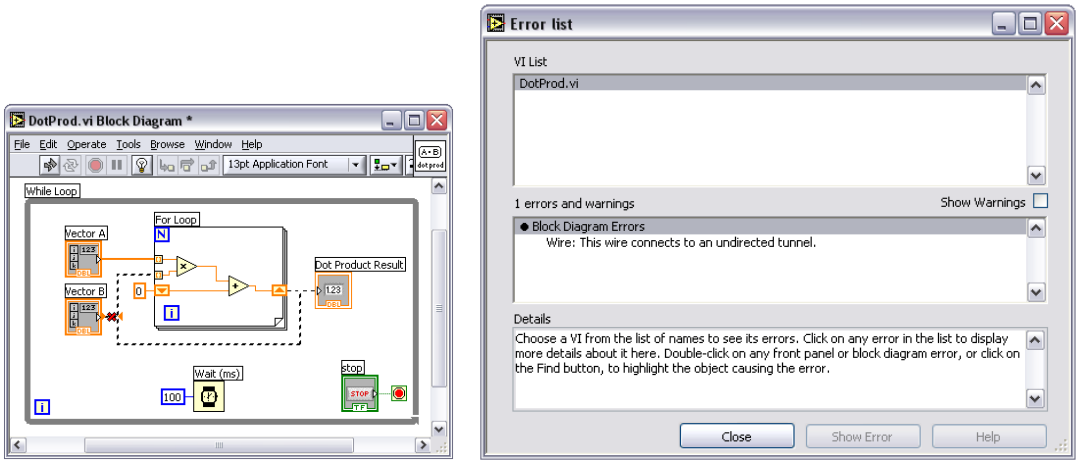


Figure A.5 Debugging broken wires and using the Error List.

There are additional debugging features unique to LabVIEW that help you understand the operation of a VI during runtime. The block diagram debugging features include the use of breakpoints to pause code during execution, adding probes to see the data value present on wires, and highlight execution mode, a debugging feature that slows execution and shows the actual flow of data along wires and through structures. These tools assist in understanding and confirming program logic and program flow.

A.2.5 Help

Several help options exist for programming and function reference. These features can be accessed through the **Help** menu on both the front panel and block diagram tool bar. Context Help can be activated by selecting the **Help → Show Context Help** or by using Ctrl + H. Context Help displays a floating help window that changes based on the current block diagram or front panel element under the cursor. Launch the comprehensive *LabVIEW Help* by selecting **Help → Search the LabVIEW Help**.
The National Instruments support web site, www.ni.com, is also a great resource for finding answers to questions. Thousands of KnowledgeBase entries and searchable forums on the NI web site offer interactive up-to-date help resources.

Hands-on Experiment A.1:

In this experiment, we will create the VI shown in Figure A.2 that performs the dot product of two input vectors. This simple VI will introduce the concepts of dataflow programming in LabVIEW and graphical constructs such as arrays, looping structures, and shift registers. Start by opening LabVIEW and creating a new VI. Notice that the front panel user interface has a gray workspace and when the block diagram is opened, it has a white workspace. We will begin by first creating the user interface on the front panel and then coding the dot product algorithm on the block diagram.

Begin on the front panel by right-clicking the free space to display the **Controls** palette. Click on the **Pushpin** to give the palette its own window. First, we will create **Vector A** by placing an **Array** shell on the front panel. You will quickly become comfortable with the location of controls and indicators in their respective palettes. For now, take advantage of the palette search feature to find functions and controls if you are not sure where they are located. Find the **Array** shell by opening the **Controls** palette and clicking **Search,** located at the top of the palette. Type "`array`" in the search field and note that as you type, the list box populates with your search results. Locate the **Array** control in the list and double-click the entry. The window will change to the appropriate subpalette and the **Array** shell will be highlighted. Click the **Array** shell icon and drag your cursor where you want to place it on the front panel. Arrays are created empty, that is, without an associated data type. Define the array data type by dropping a numeric, Boolean, or string control within the array shell. From the **Numeric** subpalette on the **Controls** palette, select a numeric control and drop it into the **Array** shell, as shown in Figure A.6. The color indicates the data type. In this case, orange indicates that the array holds double precision numbers. Double-click the **Array** label to change its text to **Vector A**. The array can be expanded vertically or horizontally to show multiple elements at once. Expand **Vector A** horizontally to show three elements.

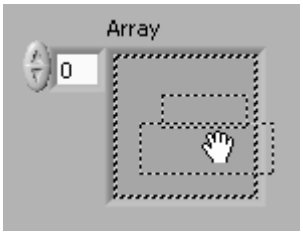


Figure A.6 Creating an Array control.

Create **Vector B** by creating a copy of Vector A. You can click the edge **Vector A** to select it and then select **Edit → Copy** and then **Edit → Paste** to create a copy. Place the copy below the original (see Figure A.7). Change the label for this control to **Vector B**.

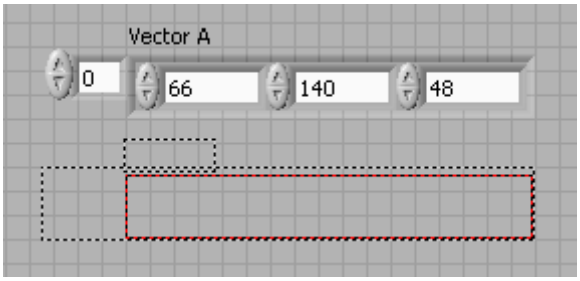


Figure A.7 Copying objects holding CTRL while dragging the object.

Now that the controls, or inputs, to the VI have been created, we need an indicator to show the output of the dot product operation. On the **Controls** palette, select the **Numeric Indicator**, available by selecting **Controls → Numeric → Numeric Indicator**, and place it on the front panel. Name this indicator **Dot Product Result**. Finally, place a **Stop** button on the front panel so that the user can terminate the application. This control is found on the **Controls → Boolean** subpalette.

After the front panel graphical user interface has been completed, we turn our attention to completing the dot product functionality on the block diagram. If the block diagram is not already open, select **Window → Show Block Diagram** (or Ctrl + E). Block diagram code executes based on the principle of dataflow. The rule of dataflow states that functions execute only when all of their required inputs are populated with data. This programming architecture allows both sequential and parallel execution to be easily defined graphically. When the block diagram for the dot product is complete, as shown in Figure A.4 (a), data passes from the inputs on the left to the outputs on the right and follows the dataflow imposed by the wires connecting the block diagram objects. The next step is to create the block diagram by first placing the correct structures and functions and then connecting them together with the appropriate wiring.

Place a **For Loop** available on the **Structures** palette in the center of the block diagram. It may be necessary to move the controls and indicators out of the center to do so. A **For Loop** in LabVIEW behaves like a for loop in most text based programming environments, iterating an integer number of times as specified by the input terminal *N*,  .

After the **For Loop** has been placed and sized, the next step is to place both the Multiply  and the Add  functions within the loop. Move **Vector A** and **Vector B** to the left of the loop and begin wiring **Vector A** through the **For Loop** and into the top input of the **multiply** function. The mouse cursor automatically changes to the wiring tool when it is placed above the input or output of an icon or another wire. Use the wiring tool to connect inputs and outputs by clicking and drawing wires between them. Now, wire **Vector B** to the bottom input of the Multiply function, and the output of the Multiply function to the bottom input of the Add function. The product of each pair of multiplied values will be cumulatively added by means of a shift register every time the loop iterates. Now create a shift register on the left side of the **For Loop** by right-clicking and selecting **Add Shift Register**. Shift registers will be added to each side of the loop automatically. A shift register allows values from previous loop iterations to be used in the next iteration of the loop. Wire the output of the **add** operation to the **right shift register**. The shift register will then change to the color of the wire. Now right-click on the **left shift register** and create a constant, wire the **left shift register** to the top of the **add**, and wire the **right shift register** to the **Dot Product Result**. Notice in Figure A.4 (a), the shift register input is initialized with a value of zero, and during each iteration of the **For Loop**, the product of the elements from **Vector A** and **B** is summed and stored in the shift register . This stored value will appear at the shift register input during the next iteration of the loop. When the For Loop has iterated through all of the vector elements, the final result will be passed from the right shift register to the **Dot Product Result**. Can you now see how the graphical code within this For Loop is performing the dot product of the two input arrays?

Finish the program by adding a While Loop, available on the **Functions → Structures** subpalette that encloses all of the existing code on the block diagram. This structure will allow the code inside to run continuously until the stop condition is met. For our VI, wire the **stop** button to the stop condition so that it can be used to stop the loop from the front panel thus, ending execution of the entire VI. Finally, we must add functionality in the While Loop to keep it from running at the maximum speed of the computer's processor. Find the **Wait Until Next ms Multiple** function and wire in 100 ms of delay. This will suspend execution of the VI for 100 ms so that the processor can handle other operations and programs. Your VI should look similar to the completed block diagram shown in Figure A.8.

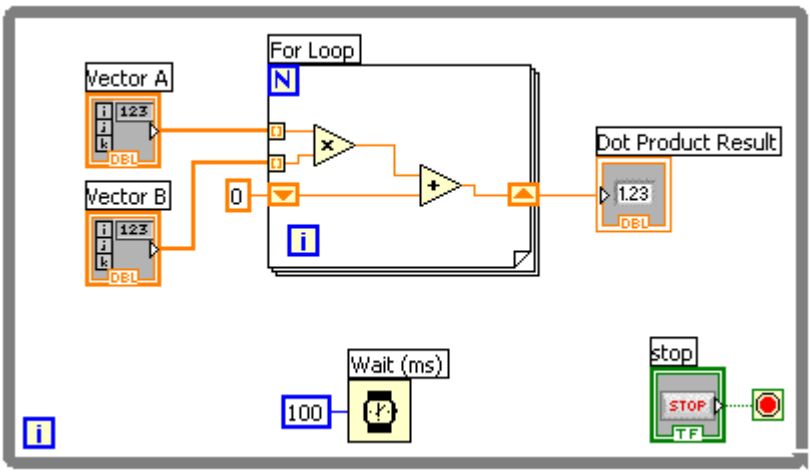
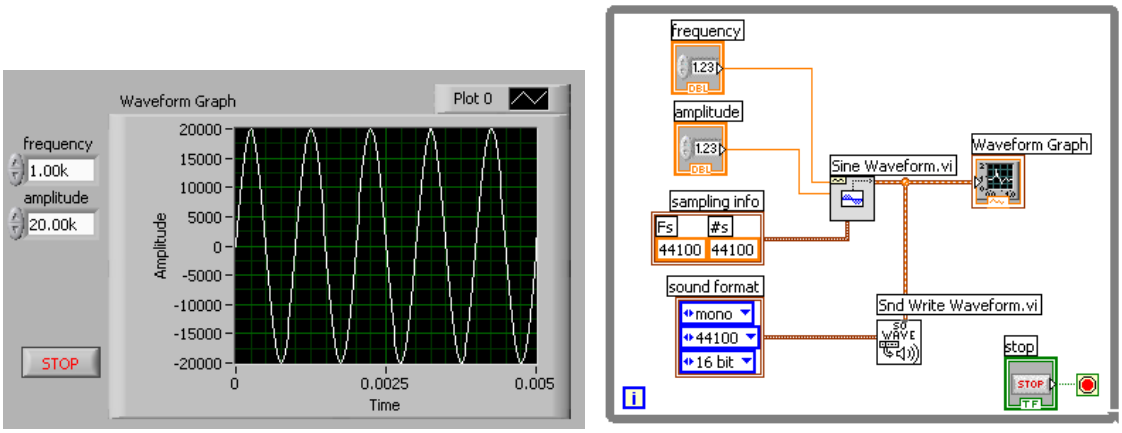


Figure A.8 Block diagram of the vector dot product VI.

If you have successfully completed the Dot Product VI, the **Run** arrow on the toolbar should appear solid and white, indicating that there are no errors in the code that would prevent the program from compiling. A broken **Run** arrow means that there is an error that you can investigate and fix by clicking this error to report what is causing the problem. Click the **Run** arrow to run the VI and experiment by placing different values into **Vector A** and **Vector B**. Compute the dot product of two small vectors by hand and compare the results with VI you created. Do they match?
Click the **stop** button to halt the VI execution. Click View → **Block Diagram** to switch to the block diagram. Enable the highlight execution feature by clicking the toolbar icon with the light bulb icon. Now press **run** again. What do you notice? How can this feature help you during development? Right-click a wire and select **Probe**. You can now see a small dialog box that displays the data value of that wire as the VI executes. Experiment with the other debugging features of the LabVIEW environment and stop the program when you are finished.

Hands-on Experiment A.2:

In this experiment, you will create the VI as shown in Figure A.9. This VI generates a one-second sine wave of a given frequency, displays its time-domain plot, and outputs the signal to the sound card.



(a) (b)

Figure A.9 Screen shots of the (a) signal generator and (b) its block diagram.

Open a new blank VI. Start by placing the `Sine Waveform.vi` on the block diagram. The `Sine Waveform.vi` can be obtained by bringing up the Functions palette and navigating to Programming → Waveform → Analog Waveform → Waveform Generation → Sine Waveform.vi. Right-click the frequency input of this function and select Create → Control. This automatically creates a numeric control of double representation with the name frequency. Switch to the front panel to see the actual control. Do the same for the amplitude input. Finally, right-click the Sampling Info input and select Create → Constant. This input happens to be a cluster data type that allows you to specify the sampling frequency and the number of samples for the generated sine wave. Clusters are a type of data unique to LabVIEW but are similar to structures in text based programming. They allow to bundle together different data elements that are related in some way. Clusters can consist of the same or different data types. Search the *LabVIEW Help* for more information on clusters. Change both values within the cluster to `44100`. The first is Fs, the sampling rate, and the second is #s, the number of samples to generate. Setting both values to the same number generates 1 second of 44100 sample/second data as shown in Figure A.9.

Now that we have all of the necessary inputs, navigate to the front panel and place a Waveform Graph on the front panel. Navigate back to the block diagram and wire the signal out terminal of the Sine Waveform VI to the Waveform Graph. The VI should now run. Test it once. Zoom in on the graph to see a smaller piece of the data. Consider clicking and changing the right-most value on the x axis to 0.005.

In order to play the sine wave on the sound card, we must use the `Snd Write Waveform.vi`. Right-click the sound format input and select Create → Constant. The default values will need to be changed to `mono`, `44100`, and `16`-bit. Then, wire the output signal to the mono 16-bit input on the top right of the Snd Write Waveform VI. Finish the VI by enclosing the block diagram code with a While Loop so that it will run until the stop button is pressed. Right-click the termination terminal of the loop and select Create → Control so that a stop button appears on the front panel. Arrange the front panel objects so that they look like those shown in Figure A.9.

As in the previous experiment, fix any errors that might be causing the **Run** arrow to be broken. Plug speakers or headphones into audio output of the computer so that you can listen to the generated sine wave signal. Click the **Run** button to start the VI. Experiment by changing values of the input controls on the front panel. Does the VI behave as expected?

A.3 Introduction to the LabVIEW Embedded Module for ADI Blackfin Processors

This section examines the state-of-the-art software that allows algorithms to be developed in the LabVIEW graphical environment and downloaded onto the Blackfin EZ-KITs.

A.3.1 What is the LabVIEW Embedded Module for ADI Blackfin Processors?

The LabVIEW Embedded Module for Analog Devices Blackfin Processors allows programs written in the LabVIEW to be executed and debugged on Blackfin processors. The software was jointly developed by Analog Devices and National Instruments, to take advantage of the NI LabVIEW Embedded technology and the convergence of capabilities of the Blackfin EZ-KIT, for control and signal processing applications. With the LabVIEW Embedded Module for ADI Blackfin, you can design and test algorithms in Windows and then compile and run them on the Blackfin Processor. Advanced debugging features such as viewing of live front panel updates via JTAG, serial, or TCP/IP, as well as single-stepping through code with VisualDSP++ allow complex embedded systems to be rapidly designed, prototyped, and tested.

A.3.2 Targeting the Blackfin Processor

Begin by launching the LabVIEW 8.5 by navigating to **Start → All Programs → National Instruments LabVIEW 8.5**. Once open, you will see the menu shown in Figure A.10.

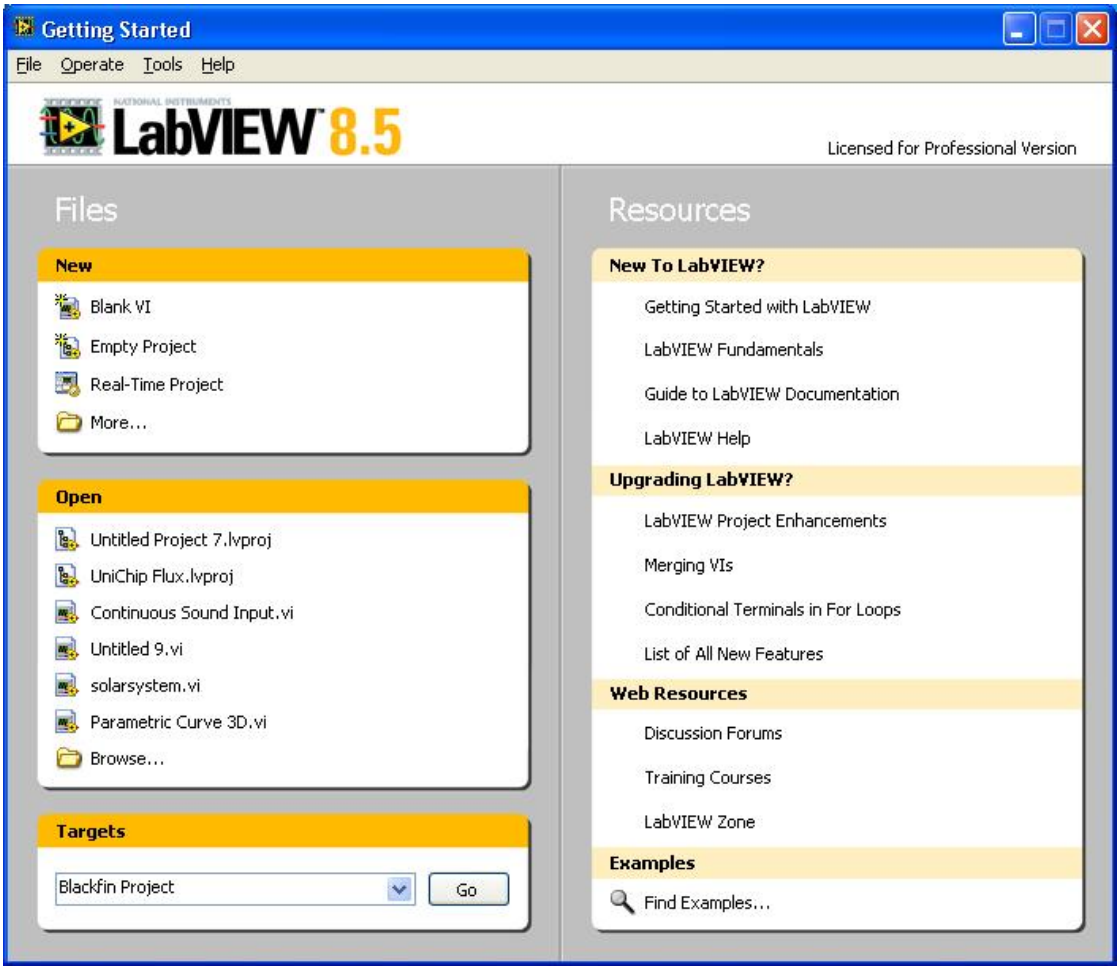


Figure A.10 LabVIEW Getting Started screen.

Select **Empty Project** in the **Getting Started** window to open an empty LabVIEW project as shown in Figure A.11.

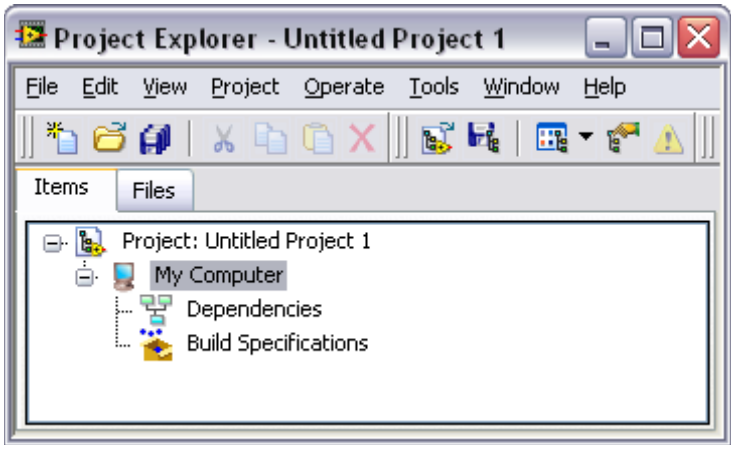


Figure A.11 LabVIEW Project Explorer Window.

Right-click **Project:Untitled Project 1** in the **Project Explorer** window and select **New → Targets and Devices** from the shortcut menu to open the **Add Targets and Devices**

dialog box. Expand the **Embedded** folder and select **Analog Devices ADSP-BF537**. Select the OK button and add the target to your project. To open an empty VI, right-click the **ADSP-BF537 Target** and select **New → VI**.

The Project Explorer provides a way to manage the target settings and all of the files included in the application. First, check the **Target** settings to be sure the **Blackfin Target** is setup appropriately. The two most important target settings are **Target → Build Options** and **Target → Configure Target**. Settings changed in the **Configure Target** dialog box are unique to the development machine, while **Build Options** are unique to the individual LabVIEW Embedded project file. Any time you switch between targets, you will need to verify target configuration.

A.3.3 Build Options

The Build specifications specify how the LabVIEW C Code Generator generates the C code and how to build the Blackfin VI into a Blackfin application. Right-click **Build Specifications** under the ADSP-BF537 target and select **New → VDK Application (Debug)** from the shortcut menu. LabVIEW might prompt you to configure the target. Click the **Yes** button and refer to (A.3.4) Target Configuration section for more information on configuring the target. The **Build Specification Properties** dialog box, shown in Figure A.12, is separated into three Application Information configurations: **General**, **Advanced**, and **TCP/IP**. The **General** tab allows the **Debugging** mode to be configured and provides some options for code generation optimizations. The **Advanced** tab provides higher level debugging and optimization options to be set, as well as compiler and linker flag options for more experienced users. The **TCP/IP** tab is provides options when TCP/IP instrumental debugging is selected.

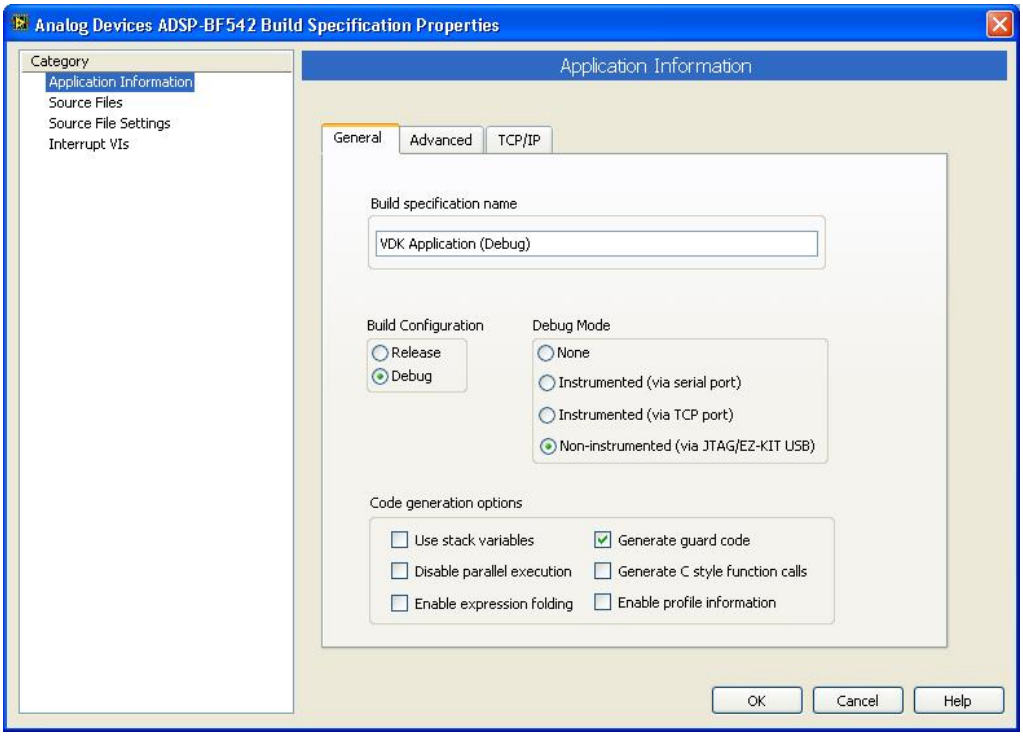


Figure A.12 LabVIEW Embedded target build options.

Select **Source Files** from the **Category** list and select your top level VI in the source files list. Click the blue right arrow button, shown at left, to move the VI from the source files list to the **Top-level VI** text box as shown in Figure A.13.

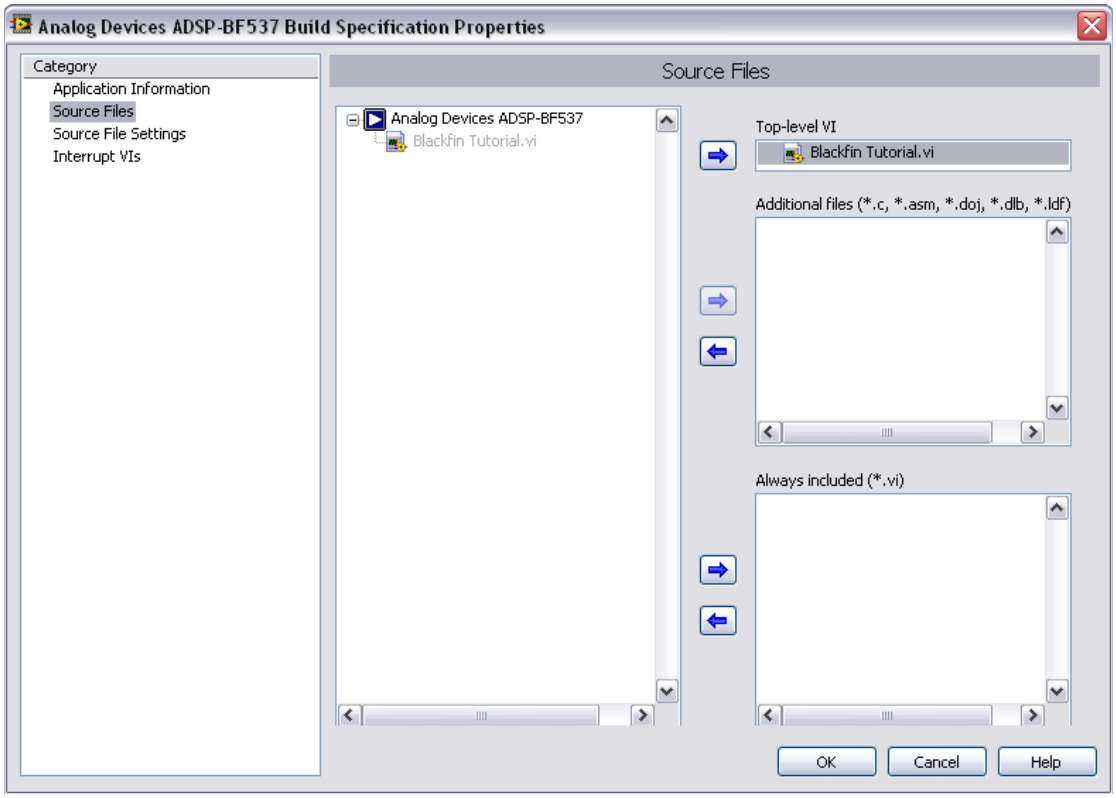


Figure A.13 Selecting the Top-Level VI for the Build Specification

A.3.4 Target Configuration

The **Target Configuration** dialog box is separated into three configuration tabs, **Target Settings**, **Debug Options and Hardware** as shown in Figure A.14. The **Target Settings** must be verified each time the target is changed from one processor type to another. The silicon revision chosen in the project and that of your Blackfin processor must match. The processor model and revision can be found on the silkscreen of the Blackfin chip at the center of the EZ-KIT circuit board.

The **Debug Options** pane provides configuration settings for update rate, communications port, and advanced debug options. Remember, these settings are unique to the development machine, so they are not saved in the project file and not changed when a new project file is loaded. The **Hardware** pane specifies the default or customized hardware settings.

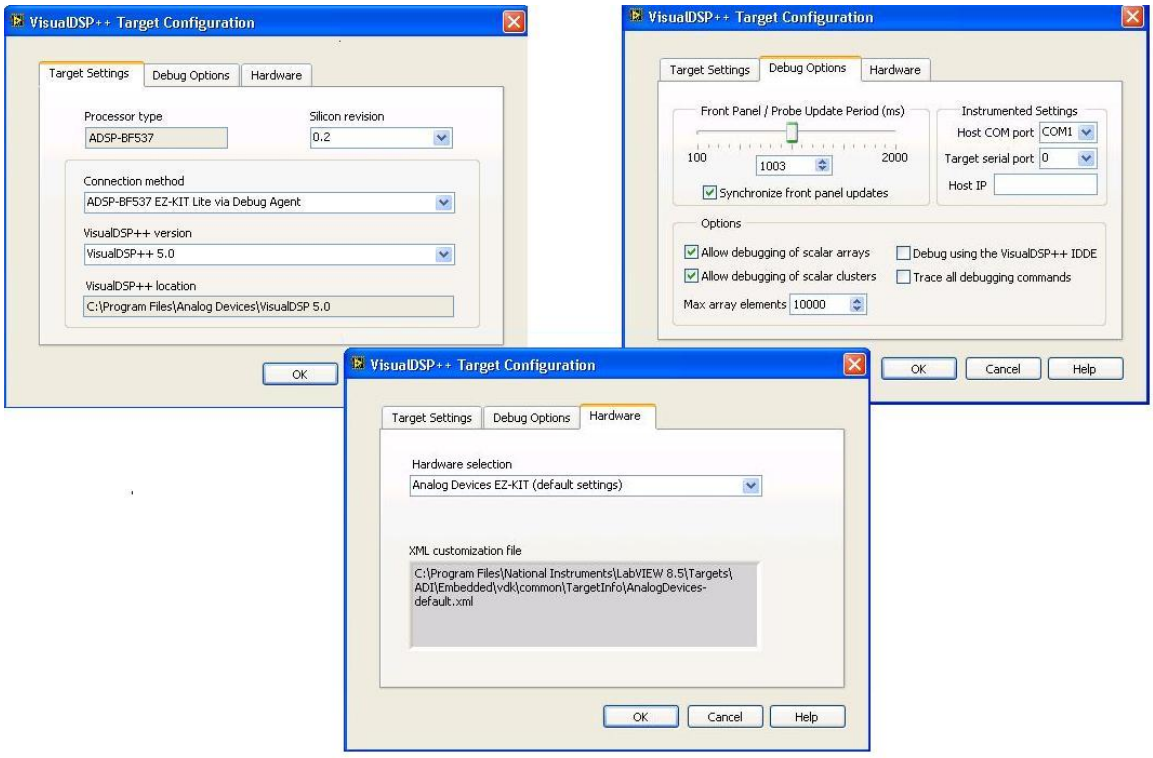


Figure A.14 LabVIEW Embedded target configuration options.

LabVIEW embedded applications can be executed by clicking the **Run** button in the Project Explorer window. But if you need to **Debug** the application, right-click the **VDK Application (Debug)** build specification in the Project Explorer window and select

Debug. LabVIEW Embedded applications are first translated from graphical LabVIEW to C code, compiled, and then downloaded to the embedded controller for execution.

For additional reference material and more specific help using LabVIEW Embedded ADI Blackfin Processors, refer to the *Getting Started with the LabVIEW Embedded Module for Analog Devices Blackfin Processors* manual, reference [69]. This manual is also installed in the National Instruments folder on your computer and can be found at `\NationalInstruments\LabVIEW8.5\manuals\Blackfin_Getting_Started.pdf`.
The manual can also be downloaded from `http://ni.com/manuals.`

A.3.6 Debugging in the LabVIEW Embedded Module for ADI Blackfin Processors

Different debugging options are available when using the LabVIEW Embedded Module for ADI Blackfin Processors to develop a Blackfin application. Debugging modes available in the module include the standard non-instrumented debugging that employs the JTAG/EZ-KIT USB port, as well as a unique instrumented debugging mode that uses the serial or TCP port. The debugging method can be selected using the **Build Specification Properties** as shown previously in Figure A.12. Debugging provides a window into the processes and programs running on the DSP during execution. The LabVIEW Embedded Module for Blackfin allows you to interact with and view the values of front panel controls and indicators, as well as probed data wires on the block diagram. This type of debugging control allows the programmer to visualize and understand every step of an algorithm graphically, which can be more intuitive than conventional register-based debugging. Data is transferred from the target to the Windows environment based on the polling rate and vector size specified in the **Debug Options** tab of the Target **Configuration** window, shown in Figure A.14.

A.3.7 Non-instrumented Debugging

Debug via JTAG/EZ-KIT USB is the same type of debugging available in the VisualDSP++ environment, but with the added benefit of LabVIEW's interactive debugging features. Keep in mind that JTAG mode interrupts the processor to perform debug data transfers, which can disrupt real-time processes. The faster the update rate and the more data transferred in each vector, the more JTAG debugging could interfere with the execution of your process. For this reason, Instrumented Debugging was developed for faster communication between the host and embedded processor. Refer to the release notes for additional non-instrumented debugging troubleshooting information.

A.3.8 Instrumented Debugging

Instrumented debugging allows the user to fully interact with a running embedded application without disrupting the processor during execution. This feature, unique to the LabVIEW 8.5 Embedded Edition, is achieved by adding additional code, as much as a 40% increase in overall code size, to the application before compilation. This added code captures and transfers values over an interface other than USB, such as serial or TCP ports. Occasionally, this form of debugging may slow down a time critical process, but it is extremely useful for debugging applications at higher data rates with the greatest ease and flexibility. When using Instrumented Debugging, the USB cable must remain attached for execution control and error checking during debugging.

Hands-on Experiment A.3:

In this experiment, you will create the VI shown in Figure A.15, which calculates the dot product of two vectors and displays the result on the front panel. This VI will be compiled, downloaded, and executed on the Blackfin processor with front panel interaction available through the debugging capabilities of the LabVIEW Embedded Module for ADI Blackfin Processors. The dot product result will also be passed back to the host computer through the standard output port using the Inline C Node.

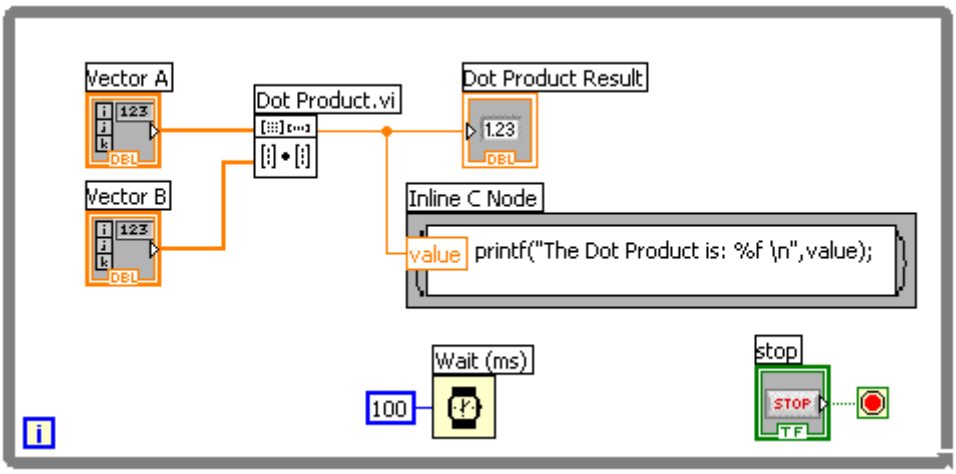


Figure A.15 Dot product VI block diagram written in LabVIEW Embedded.

Open a new project by selecting Empty Project from the Getting Started window and save it as `dotproduct.lvproj`. Right-click **Project:Untitled Project 1** in the **Project Explorer** window and select **New → Targets and Devices** from the shortcut menu to open the **Add Targets and Devices** dialog box. Expand the **Embedded** folder and select **Analog Devices ADSP-BF537**. Select the OK button and add the target to your project. You can right-click the **ADSP-BF537** target and select **New → VI**.

The next step is to recreate parts of the Hands-on Experiment A.1 following those instructions, or you may start with a saved version of that VI. The goal is to modify the VI to that found in Figure A.15. For the purposes of this exercise, we will replace the For Loop implementation of the dot product with the dot product LabVIEW analysis function. The controls and indicators will be identical to those used in the previous exercise and the outer While Loop will be reused.

Once you have recreated the new DotProd.vi complete with controls, indicators, and while loop, place the `Dot Product.vi,` located on the **Mathematics → Linear Algebra** palette, in the center of the VI. Click to select and then click to drop the `Dot Product.vi` on the block diagram. Configure the function inputs and outputs by wiring **Vector A** to the **X array** input, **Vector B** to the **Y array** input, and **Dot Product Result** to the **X*Y** output of the Dot Product VI. Click and drag each of these icons to move and arrange them as seen in Figure A.15.

Next, we will add console output capability to our application using an Inline C Node, which will allow us to enter text-based C code directly into the LabVIEW application. Navigate to the **Structures** palette, select the **Inline C Node**, and drag the mouse across the block diagram to create a rectangular region roughly the size shown in Figure A.15. Click inside the Inline C Node and enter `printf("The Dot Product is: %f \n",value);`, which will output to the debugging window the value of the dot product during every iteration of the While Loop. To pass the value of the dot product to this C code, right-click the left border of the **Inline C Node** and select **Add Input**. This creates a new input terminal for the C node, which is named as `value` to match the name referenced in the C code.

Verify that you have enclosed the dot product controls and indicators with a While Loop structure to allow the code to execute continuously. Finally, add time delay to the loop if you have not already done so. Add a **Wait (ms)** function, located in the **Time, Dialog & Error** palette and create a constant input of `100` milliseconds to provide a short pause between each loop iteration.

After the VI has been wired correctly, clicking the **Run** button will result in a dialog box asking if you want to create a build specification. Choose Yes, to bring up the **Build Specification Properties** dialog Box. You can choose what level of debugging support to use, as well as various compilation options for the project. For **Debug mode**, choose **Non-instrumented (via JTAG/EZ-KIT USB)** from the pull-down menu and make sure that the **Build configuration** is set to **Debug**. This option allows for debugging our application without the need for any additional cables or hardware. You will also need to uncheck **Redirect stdout to serial port** on the **Advanced** tab. Before running the application open the **Processor Status** dialog box from the **Target** menu to view the console output transferred over the standard output port from the application.

After the build options have been set for the project, right-click the newly created **Build Specification** and select **Debug** from the menu to compile, build, download, run, and debug the application. Switch to the front panel and observe how the calculated value for the dot product changes as you enter different values for the input vectors. Note also the console output in the **Processor Status** dialog box. Do the values calculated by this VI match those you calculated in Hands-on Experiment A.1?

Hands-on Experiment A.4:

In this final experiment, we will modify an audio pass-through example to generate and output a custom sine wave to the audio out port of the Blackfin EZ-KIT using the LabVIEW code shown in Figure A.16. To begin, open the `Audio Talkthrough – BF537.lvproj` project file from `c:\adsp\chap2\exp2_6`. Select **File → Save Project As** and save the project as `Sine Generator.lvproj at a different location`. From the **Project Explorer** window, open `Audio Talkthrough – BF537.vi` and select **File → Save As** to save the VI as `Sine Generator.vi` in the same directory as your embedded project file. We are now ready to add the sine wave generator functionality to the code.

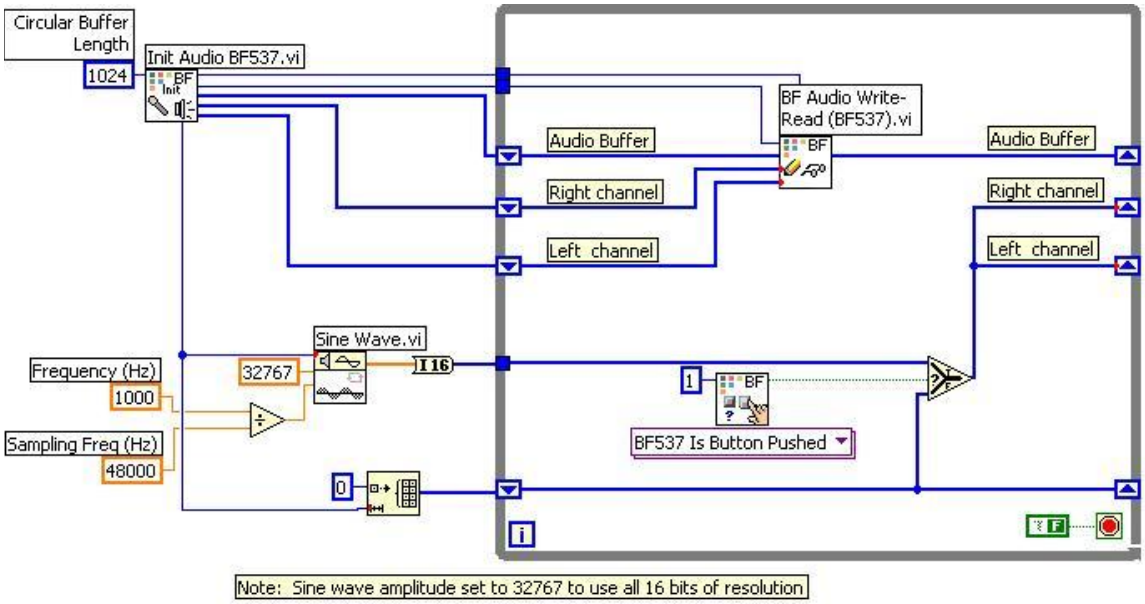


Figure A.16: Completed `Sine Generator.vi` block diagram.

Start by first adding the `Sine Wave.vi`, located on the **Signal Processing → Signal Generation** palette, to the block diagram. Place this VI outside of the **While** loop, below the **Init Audio** function. At this point you should expand the size of the While Loop to make more room for additional code. This is easily done by placing the mouse over the While Loop and dragging down the lower handle until the loop is approximately the same size as shown in Figure A.16. Move the mouse over the sine wave output of `Sine Wave.vi`, and notice that the terminal has an orange color, denoting a floating-point data type. Because the Blackfin processor is a fixed-point processor, we need to convert the floating-point array of samples into a fixed-point format, as explained in Chapter 6. To accomplish this, place a **To Word Integer** VI, located on the **Numeric → Conversion** palette, next to the **sine wave** output terminal and wire them together. Next,

a wire between the output of the conversion function to the border of the While Loop. What data type is this wire?

You must now define the signal parameters for the sine wave you want to generate. To ensure that the sine wave generated is the appropriate number of samples, connect a wire between the **samples** input of the `Sine Wave VI` and the **Half buffer size** output of the `Init Audio VI.` Refer to Figure A.16 if you need help locating the terminals. Next, define the amplitude of the sine wave by right-clicking the **amplitude** terminal and selecting **Create → Constant** from the shortcut menu. The amplitude has a default value of 1, but we change it to 32767 to use all 16 bits of resolution. Finally, we need to define the frequency of sine wave. From the **Numeric** palette, place two numeric constants and a Divide function and wire the constants to the inputs of the divide function. Wire the output of the Divide function to the **Frequency** input terminal of the `Sine Wave VI`. The units that this VI uses for frequency are cycles/sample, which we will calculate by dividing the desired sine wave frequency in Hz by the sampling frequency of our audio system. Use 1000 for the desired frequency and 48000 for the sampling frequency. At this point, the block diagram should look like Figure A.17.

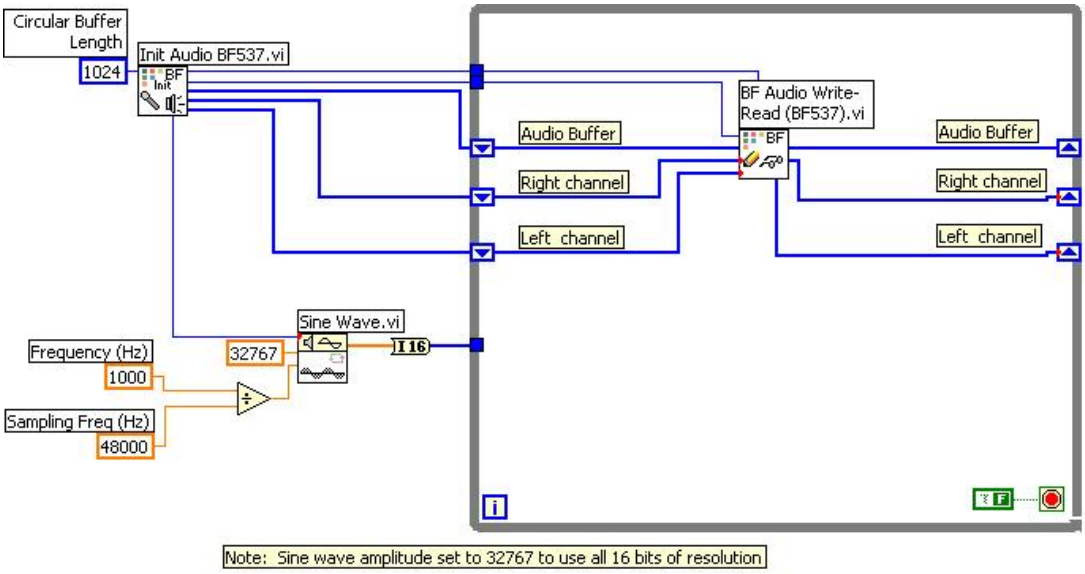


Figure A.17: Sine wave generator – intermediate stage.

Now that the sine wave generation is implemented, we need to add in the ability to play the signal using the audio capabilities built into the Blackfin EZ-KIT when the user holds down a push button. Place an **Initialize Array** function from the **Array** palette below the Sine Wave function and wire the **dimension size** terminal to the **half buffer size** output of the `Init Audio VI.` You can connect to the same wire that connects the Init Audio and Sine Wave VIs. Create a constant for the **element** input and change it to `0` of `I16` data type by right-clicking the constant and selecting **Representation → I16**. This creates an empty array of the same size as the generated sine wave that we can use to output a silent signal when the button is not pressed. Connect the output of the **Initialize Array** function to the border of the **While** loop.

Right-click the tunnel created on the While Loop border and select **Replace with Shift Register**, then wire the left shift register to the right shift register. Next, we need to determine if a button has been pushed. Place a BF is Button Pushed VI, located on the **Blackfin → EZ-KIT → Button** palette, inside the While Loop and select the instance appropriate for the hardware from the pull-down menu. This VI will return a Boolean output telling us if a particular button is currently being pressed. To choose which button to monitor, create a constant on the **button number** input. Valid ranges are 1-4 for the BF537 EZ-KIT, corresponding to the PB and SW numbers of the buttons, respectively. To choose which signal to output, add a **Select** function located on the **Comparison** palette and wire the **button pushed** output to the **s** input of the Select function. Now, wire the **f** input of the Select function to the wire connecting the two shift registers we just created, and wire the **t** input of the Select function to the converted sine array. Delete the connections between the right and left audio channel outputs of the `BF Audio Write-Read VI` and the right shift registers. Finally, connect the output of the Select function to the right shift registers for both the right channel and the left channel. The block diagram is now complete, and should match Figure A.15.

Compile, download, and run the application on the Blackfin processor and test the VI by pressing the button you chose to monitor and listen for the generated sine wave to be output on the audio out line of the Blackfin EZ-KIT. Experiment with generating different sine wave frequencies. Can you hear the difference? Why do most combination of parameters distort the audio tone? Consider the combination of buffer size, sampling rate, and the frequency of the sine wave. How would you modify the VI to remove this phenomenon?

