Getting Started with
ADSP-BF537 EZ-KIT Lite®

**ANALOG
DEVICES**

Regulatory Compliance

The ADSP-BF537 EZ-KIT Lite is designed to be used solely in a laboratory environment. The board is not intended for use as a consumer end product or as a portion of a consumer end product. The board is an open system design which does not include a shielded enclosure and therefore may cause interference to other electrical devices in close proximity. This board should not be used in or near any medical equipment or RF devices.

The ADSP-BF537 EZ-KIT Lite has been certified to comply with the essential requirements of the European EMC directive 89/336/EEC amended by 93/68/EEC and therefore carries the "**CE**" mark.

The ADSP-BF537 EZ-KIT Lite has been appended to Analog Devices, Inc. Technical Construction File (TCF) referenced '**DSPTOOLS1**' dated December 21, 1997 and was awarded CE Certification by an appointed European Competent Body as listed below.

Technical Certificate No: Z600ANA1.021

Issued by: Technology International (Europe) Limited
60 Shrivenham Hundred Business Park
Shrivenham, Swindon, SN6 8TY, UK

The EZ-KIT Lite evaluation system contains ESD (electrostatic discharge) sensitive devices. Electrostatic charges readily accumulate on the human body and equipment and can discharge without detection. Permanent damage may occur on devices subjected to high-energy discharges. Proper ESD precautions are recommended to avoid performance degradation or loss of functionality. Store unused EZ-KIT Lite boards in the protective shipping package.

CONTENTS

CONTENTS

RUNNING A TCP/IP APPLICATION ON AN ADSP-BF537 EZ-KIT LITE

CONTROLLING A REAL-TIME APPLICATION VIA TCP/IP

CREATING A TCP/IP APPLICATION

INDEX

CONTENTS

PREFACE

Thank you for purchasing the ADSP-BF537 EZ-KIT Lite®, Analog Devices, Inc. evaluation system for ADSP-BF537 Blackfin® processors.

Blackfin processors are embedded processors that support a Media Instruction Set Computing (MISC) architecture. This architecture is the natural merging of RISC, media functions, and digital signal processing (DSP) characteristics towards delivering signal processing performance in a microprocessor-like environment.

The evaluation board is designed to be used in conjunction with the VisualDSP++® development environment to test the capabilities of ADSP-BF537 Blackfin processors. The VisualDSP++ development environment gives you the ability to perform advanced application code development and debug, such as:

- Create, compile, assemble, and link application programs written in C++, C, and ADSP-BF537 assembly

- Load, run, step, halt, and set breakpoints in application programs

- Read and write data and program memory

- Read and write core and peripheral registers

- Plot memory

Access to the ADSP-BF537 processor from a personal computer (PC) is achieved through a USB port or an optional JTAG emulator. The USB interface provides unrestricted access to the ADSP-BF537 processor and the evaluation board peripherals. Analog Devices JTAG emulators offer

faster communication between the host PC and target hardware. Analog Devices carries a wide range of in-circuit emulation products. To learn more about Analog Devices emulators and processor development tools, go to http://www.analog.com/dsp/tools/.

The ADSP-BF537 EZ-KIT Lite provides example programs to demonstrate the capabilities of the evaluation board.

(i) The ADSP-BF537 EZ-KIT Lite installation is part of the VisualDSP++ installation. The EZ-KIT Lite is a licensed product that offers an evaluation (temporary) license. Once the evaluation license expires, VisualDSP++ restricts simulator and emulator connections and limits the size of a user program to 20 KB of internal memory.

The board features:

- Analog Devices ADSP-BF537 Blackfin processor

    - Core performance up to 600 MHz
    - External bus performance to 133 MHz
    - 182-pin mini-BGA package
    - 25 MHz crystal

- Synchronous dynamic random access memory (SDRAM)

    - MT48LC32M8 – 64 MB (8M x 8-bits x 4 banks) x 2 chips

- Flash memory

    - 4 MB (2M x 16-bits)

- Analog audio interface

    - AD1871 96 kHz analog-to-digital codec (ADC)
    - AD1854 96 kHz digital-to-audio codec (DAC)

- ✓ 1 input stereo jack
- ✓ 1 output stereo jack

- Ethernet interface

  - ✓ 10-BaseT (10 Mbits/sec) and 100-BaseT (100 Mbits/sec) Ethernet Media Access Controller (MAC)
  - ✓ SMSC LAN83C185 device

- Controller Area Network (CAN) interface

  - ✓ Philips TJA1041 high-speed CAN transceiver

- National Instruments Educational Laboratory Virtual Instrumentation Suite (ELVIS) interface

  - ✓ LabVIEW™-based virtual instruments
  - ✓ Multifunction data acquisition device
  - ✓ Bench-top workstation and prototype board

- Universal asynchronous receiver/transmitter (UART)

  - ✓ ADM3202 RS-232 line driver/receiver
  - ✓ DB9 female connector

- LEDs

  - ✓ 10 LEDs: 1 power (green), 1 board reset (red), 1 USB (red), 6 general-purpose (amber), and 1 USB monitor (amber)

- Push buttons

  - ✓ 5 push buttons: 1 reset, 4 programmable flags with debounce logic

- Expansion interface

  - ✓ All processor signals

- Other features

  ✓ JTAG ICE 14-pin header

The EZ-KIT Lite board has flash memory with a total of 4 MB. The flash memory can be used to store user-specific boot code, allowing the board to run as a stand-alone unit. The board also has 64 MB of SDRAM, which can be used by the user at runtime.

SPORT0 interfaces with the audio circuit, facilitating development of audio signal processing applications. SPORT0 also connects to an off-board connector for communication with other serial devices.

The UART of the processor connects to an RS-232 line driver and a DB9 female connector, providing an interface to a PC or other serial device.

Additionally, the EZ-KIT Lite board provides access to all of the processor's peripheral ports. Access is provided in the form of a three-connector expansion interface.

For information about the hardware components of the EZ-KIT Lite, refer to the *ADSP-BF537 EZ-KIT Lite Evaluation System Manual*.

Purpose of This Manual

The *Getting Started with ADSP-BF537 EZ-KIT Lite* familiarizes users with the hardware capabilities of the evaluation system and demonstrates how to access these capabilities in the VisualDSP++ environment.

EZ-KIT Lite users should use this manual in conjunction with the *ADSP-BF537 EZ-KIT Lite Evaluation System Manual*, which describes the evaluation system's components in greater detail.

Intended Audience

The primary audience of this manual is a programmer with experience in desktop and/or embedded programming, but with little or no experience with the Blackfin architecture and/or VisualDSP++. A working knowledge of the C and C++ programming languages will be extremely helpful in understanding the examples and source code blocks referenced in this manual.

Manual Contents

The manual consists of:

- Exercise 1, "Programming ADSP-BF537 EZ-KIT Lite with Visu-alDSP++" on page 1-1
  Provides instructions for connecting the EZ-KIT Lite to a PC and writing a C program to perform two sorting algorithms.

- Exercise 2, "Running a TCP/IP application on an ADSP-BF537 EZ-KIT Lite" on page 2-1
  Provides instructions for creating a TCP/IP application using the open source LwIP stack and VisualDSP++ Kernel.

- Exercise 3, "Controlling a real-time application via TCP/IP" on page 3-1
  Provides instructions for creating a complex application with audio pass-through VDK threads that operate concurrently and indepen-dently with the TCP/IP stack's operations.

- Appendix A, "Creating a TCP/IP Application" on page A-1
  Familiarizes the user with the concepts utilized in the creation of a TCP/IP application.

What's New in This Manual

The *Getting Started with ADSP-BF537 EZ-KIT Lite* has been updated for the current release of VisualDSP++.

Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at
  `http://www.analog.com/processors/technicalSupport`

- E-mail tools questions to
  `processor.tools.support@analog.com`

- E-mail processor questions to
  `processor.support@analog.com (World wide support)`
  `processor.europe@analog.com (Europe support)`
  `processor.china@analog.com (China support)`

- Phone questions to **1-800-ANALOGD**

- Contact your Analog Devices, Inc. local sales office or authorized distributor

- Send questions by mail to:
  ```
  Analog Devices, Inc.
  One Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106
  USA
  ```

Supported Processors

The ADSP-BF537 EZ-KIT Lite evaluation system supports Analog Devices ADSP-BF537 Blackfin processors.

Product Information

You can obtain product information from the Analog Devices Web site, from the product CD-ROM, or from printed publications (manuals).

Analog Devices is online at `www.analog.com`. Our Web site provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

MyAnalog.com

MyAnalog.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information on products you are interested in. You can also choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests, including documentation errata against all manuals. You can also choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Registration:

Visit `www.myanalog.com` to sign up. Click **Register** to use MyAnalog.com. Registration takes about five minutes and serves as means for you to select the information you want to receive.

If you are already a registered user, just log on. Your user name is your e-mail address.

Processor Product Information

For information on embedded processors and DSPs, visit our Web site at www.analog.com/processors, which provides access to technical publications, data sheets, application notes, product overviews, and product announcements.

You may also obtain additional information about Analog Devices and its products in any of the following ways.

- E-mail questions or requests for information to
  processor.support@analog.com (World wide support)
  processor.europe@analog.com (Europe support)
  processor.china@analog.com (China support)

- Fax questions or requests for information to
  1-781-461-3010 (North America)
  +49-89-76903-157 (Europe)

Related Documents

For information on product related development software and hardware, see these publications:

Table 1. Related Processor Publications

Title	Description
ADSP-BF536/ADSP-BF537 Embedded Processor Data Sheet	General functional description, pinout, and timing.
ADSP-BF537 Blackfin Processor Hardware Reference	Description of internal processor architecture and all register functions.

Table 2. Related VisualDSP++ Publications

Title	Description
ADSP-BF537 EZ-KIT Lite Evaluation System Manual	Description of the ADSP-BF537 EZ-KIT Lite hardware and software components.
VisualDSP++ Assembler and Preprocessor Manual	Description of the assembler function and commands.
VisualDSP++ C/C++ Complier and Library Manual for Blackfin Processors	Description of the complier function and commands for Blackfin processors.
VisualDSP++ Linker and Utilities Manual	Description of the linker function and commands.
VisualDSP++ Loader and Utilities Manual	Description of the loader/splitter function and commands.
VisualDSP++ Device Drivers and System Services Manual for Blackfin Processors	Description of the device drivers' and system services' functions and commands

(i) If you plan to use the EZ-KIT Lite board in conjunction with a JTAG emulator, also refer to the documentation that accompanies the emulator.

All documentation is available online. Most documentation is available in printed form.

Visit the Technical Library Web site to access all processor and tools manuals and data sheets:
http://www.analog.com/processors/technicalSupport/technicalLibrary/.

Online Technical Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, the Dinkum Abridged C++ library, and Flexible License Manager (FlexLM) network license manager software documentation. You can easily search across the entire VisualDSP++ documentation set for any topic of interest. For easy printing, supplementary .pdf files of most manuals are also provided.

Each documentation file type is described as follows.

File	Description
.chm	Help system files and manuals in Help format
.htm or .html	Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the .html files requires a browser, such as Internet Explorer 6.0 (or higher).
.pdf	VisualDSP++ and processor manuals in Portable Documentation Format (PDF). Viewing and printing the .pdf files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

If documentation is not installed on your system as part of the software installation, you can add it from the VisualDSP++ CD-ROM at any time by running the Tools installation. Access the online documentation from the VisualDSP++ environment, Windows® Explorer, or the Analog Devices Web site.

Accessing Documentation From VisualDSP++

To view VisualDSP++ Help, click on the **Help** menu item or go to the Windows task bar and navigate to the VisualDSP++ documentation via the **Start** menu.

To view ADSP-BF537 EZ-KIT Lite Help, which is part of the VisualDSP++ Help system, use the **Contents** or **Search** tab of the Help window.

Accessing Documentation From Windows

In addition to any shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.

Help system files (`.chm`) are located in the `Help` folder, and `.pdf` files are located in the `Docs` folder of your VisualDSP++ installation CD-ROM. The `Docs` folder also contains the Dinkum Abridged C++ library and the FlexLM network license manager software documentation.

Your software installation kit includes online Help as part of the Windows interface. These help files provide information about VisualDSP++ and the ADSP-BF537 EZ-KIT Lite evaluation system.

Accessing Documentation From Web

Download manuals at the following Web site:
http://www.analog.com/processors/technicalSupport/technicalLibrary/.

Select a processor family and book title. Download archive (`.zip`) files, one for each manual. Use any archive management software, such as Win-Zip, to decompress downloaded files.

Printed Manuals

For general questions regarding literature ordering, call the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**) and follow the prompts.

Hardware Tools Manuals

To purchase EZ-KIT Lite and in-circuit emulator (ICE) manuals, call **1-603-883-2430**. The manuals may be ordered by title or by product number located on the back cover of each manual.

Processor Manuals

Hardware reference and instruction set reference manuals may be ordered through the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**), or downloaded from the Analog Devices Web site. Manuals may be ordered by title or by product number located on the back cover of each manual.

Data Sheets

All data sheets (preliminary and production) may be downloaded from the Analog Devices Web site. Only production (final) data sheets (Rev. 0, A, B, C, and so on) can be obtained from the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**); they also can be downloaded from the Web site.

To have a data sheet faxed to you, call the Analog Devices Faxback System at **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. If the data sheet you want is not listed, check for it on the Web site.

Notation Conventions

Text conventions used in this manual are identified and described as follows.

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
{this | that}	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as this or that. One or the other is required.
[this | that]	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional this or that.
[this,…]	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of this.
.SECTION	Commands, directives, keywords, and feature names are in text with letter gothic font.
filename	Non-keyword placeholders appear in text with italic style format.
ⓘ	**Note:** For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
⚡	**Caution:** Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
🚫	**Warning:** Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for the devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

Notation Conventions

(i) Additional conventions, which apply only to specific chapters, may appear throughout this document.

1 PROGRAMMING ADSP-BF537 EZ-KIT LITE WITH VISUALDSP++

In exercise 1, you will connect your personal computer (PC) to the ADSP-BF537 EZ-KIT Lite evaluation system and write a simple C language program to perform two sorting algorithms. You will be presented with techniques to visualize graphically the effects of the sorts. Next, you will measure the Blackfin processor's performance and learn how the program's placement (within the processor memory hierarchy) impacts performance. Finally, you will study the processor performance in terms of speed and voltage trade-offs.

In the exercise, you will learn about the following concepts.

- VisualDSP++ sessions and target types

- Plot windows

- Project configurations

- Benchmarking code with a Blackfin processor's cycle counter and real-time clock

- Statistical profiling

- Blackfin processor's memory hierarchy, cache, and direct L1 memory placement

- Blackfin processor's voltage regular and the processor support library

Installing VisualDSP++ and ADSP-BF537 EZ-KIT Lite

If you have not done so, install the ADSP-BF537 EZ-KIT Lite. Ensure the board is disconnected from your PC, then install VisualDSP++ by following instructions in the *VisualDSP++ Installation Quick Reference Card* steps 1 to 3. Connect the EZ-KIT Lite to the PC with the provided USB cable.

Once the installation is complete, the amber LED (labeled `USB_MONITOR`, found near the USB jack) is illuminated, indicating a successful physical connection between the PC and EZ-KIT Lite.

Install your license following instructions in the *VisualDSP++ Licensing Guide*.

Starting VisualDSP++ and Connecting to ADSP-BF537 EZ-KIT Lite

From the **Start** menu, navigate to the VisualDSP++ environment via the **Programs** menu. After a second or two, the main VisualDSP++ window appears on the screen. When VisualDSP++ first launches, it is disconnected from your EZ-KIT Lite or any *debugging session* (Figure 1-1).
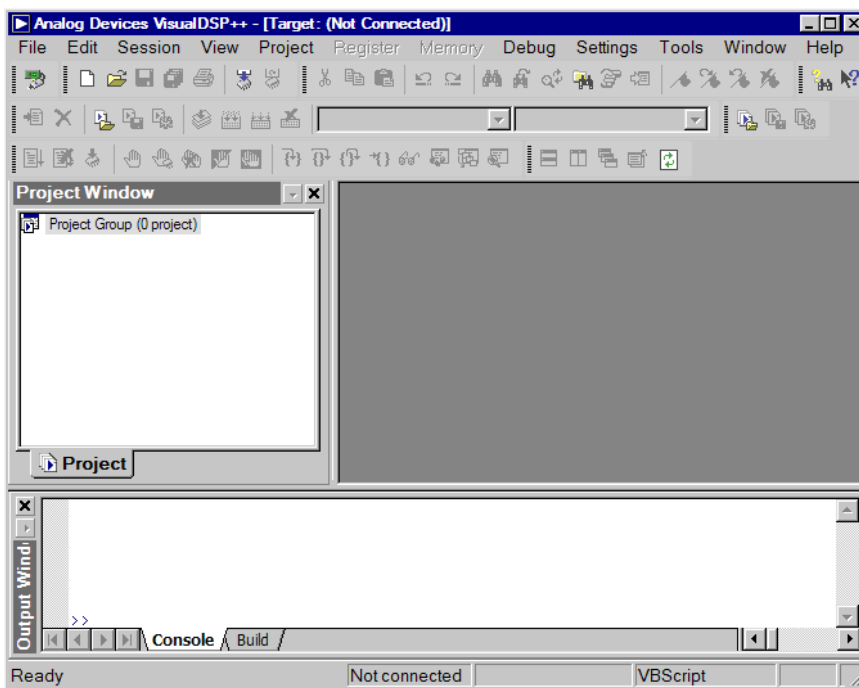

Figure 1-1. VisualDSP++ Main Window

VisualDSP++ is able to connect to a number of different debug sessions, where each session has its own application and benefits. The session types available with VisualDSP++ are[1]:

- **EZ-KIT Lite**. This is the dedicated USB connection between the PC and EZ-KIT Lite. The connection is simple to manage and is an integral part of the EZ-KIT Lite. However, the connection is available with the EZ-KIT Lite only. Once your custom hardware board is available for development, you will use an emulator session (description follows) to connect to the custom hardware.

- **Simulator**. This is a software model of the processor. Simulators offer unique advantages, the first being that no external hardware is required, a great benefit when using VisualDSP++ on the road. Furthermore, simulators offer a deep insight to the internal workings of the processor (pipelines, caches, and more), which is not possible with hardware-based sessions. The downside is that a simulator is several orders of magnitude slower than actual hardware. The software model simulates only the processor, making it difficult to accurately simulate a complex system that involves more than the processor.

  VisualDSP++ includes two types of Blackfin simulators: a cycle-accurate, interpreted simulator and a functional, compiled simulator. A cycle-accurate simulator is a completely accurate model of the Blackfin processor and allows you to fully visualize the inner-workings of the processor. The compiled simulator sacri-

---

[1] Third-party software may add additional session types.

fices the detailed view but allows you to simulate much more quickly, at millions of simulated cycles per second depending on the speed of your PC.

For a more comprehensive discussion and exercises concerning the simulators' unique features, refer to the *VisualDSP++ Getting Started Guide*, available in the VisualDSP++ Help system.

- **Emulator**. This is a JTAG emulator, the ideal device for connecting to hardware, giving the best performance and maximum flexibility. An emulator is a separate module that provides a high-bandwidth USB or PCI connection between the PC and the target device. An emulator is required to connect to a non-EZ-KIT Lite target.

- **Legacy target**. This is a target created in VisualDSP++ 4.0 or a prior version.

Throughout this tutorial, we will use the EZ-KIT Lite connection.

To tell VisualDSP++ that your target is an ADSP-BF537 processor on an EZ-KIT Lite, you must create and activate a new session. Follow these steps:

1. From VisualDSP++' **Session** menu, select **New Session**, which launches the **Session Wizard** (Figure 1-2).

2. On the **Select Processor** page, ensure that **Processor family** is set to **Blackfin**. In **Choose a target processor**, select **ADSP-BF537**. Click **Next**.

3. On the **Select Connection Type** page, select **EZ-KIT Lite**. Click **Next**.
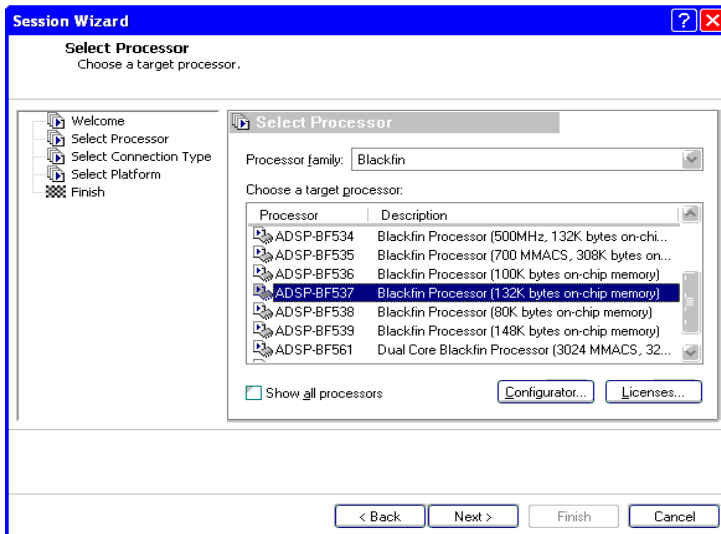
Figure 1-2. Session Wizard

4. On the **Select Platform** page, ensure that the selected platform is **ADSP-BF537 EZ-KIT Lite via Debug Agent**. Specify your own **Session name** for your session or accept the default name. Click **Next**.

5. On the **Finish** page, check the presented information and click **Finish**. VisualDSP++ creates the new session and connects to the ADSP-BF537 EZ-KIT Lite. Once connected, the main window's title is changed to include the session name set is step 4.

Exercise 1, Part 1: Building and Running the First Program

Now it is time to start our first C program. "Listing 1-1. Exercise 1, Part 1" on page 1-19 is the C program we start and expand throughout this exercise. The program randomizes and sorts two arrays using classic sorting algorithms: the bubble sort and the quick sort. If you are familiar with the algorithms, you know that the quick sort, true to its name, is the faster of the two algorithms (on average, $O(n \log n)$ versus $O(n^2)$).

To spare you from typing in the program, the entire Exercise 1 source code is included on the VisualDSP++ distribution CD. The part 1 program can be found in the `<install_path>\Blackfin\Examples\ADSP-BF537 EZ-Kit Lite\Getting Started Examples\Part_1_1` subdirectory. The default `<install_path>` is `C:\Program Files\Analog Devices\VisualDSP 5.0`.

Open the project file for the first part of Exercise 1 by selecting **File**–>**Open**–>**Project**, browsing to the exercise directory, and selecting the `Sorts_1_1.dpj` project file[1]. Once the project is opened, you can view its source code by double-clicking the **Sorts.c** label in the **Project** window (expand the **Source Files** tree control if necessary).

Build and load the executable program to the EZ-KIT Lite using the **Project** –> **Build Project** command (or use the **F7** hotkey). Once the program is loaded, observe the blue bar on the first executable statement the `main()` function, showing that the program is stopped there awaiting your command to set it running or to single-step through the source.

---

[1] If your PC is used by multiple VisualDSP++ users and/or you do not have write privileges, copy the entire Getting Started Examples folder to a location you can use without influencing other users.

Exercise 1, Part 1: Building and Running the First Program

To visualize the activity discussed in the exercise, create two plot windows, one for the out_b array and one for the out_m array. To create a plot window for the out_b array:

1. Select the **View–>Debug Windows–>Plot–>New** menu item. The **Plot Configuration** dialog box appears.

2. Change **Title** to Monitoring out_b.

3. Type out_b in the **Address** field.

4. Type 128 (the length of the out_b array) in the **Count** field.

5. Change **Data** to int (the type of our data).

6. Click **Add**, then click **OK**.

Repeat this procedure to create a plot window for the out_m variable[1], adjusting steps 2 and 3 accordingly. Once the plot windows are created, adjust them to comfortable sizes. Your plot windows look similar to those in Figure 1-3.

Note that both line plots are flat at zero because the data arrays are zero-initialized by VisualDSP++. We will see VisualDSP++ update the windows as we step into the program. Issue the **Debug–>Step Over** command (or use the **F10** hotkey) three times to highlight the call to the bubble_sort() function as the next instruction to execute. The two plot windows show the random values to which the arrays are initialized. **Step Over** again to observe that the out_b array is now sorted. **Step Over** one more time to observe that out_m is also sorted.

Note that the part 1 project uses the debug *configuration* (Figure 1-4).

---

[1] Note that you can add both plots to a single window. However, this is undesirable when two plots have the same results, causing the plot lines to overwrite each other.

Figure 1-3. Plot Window

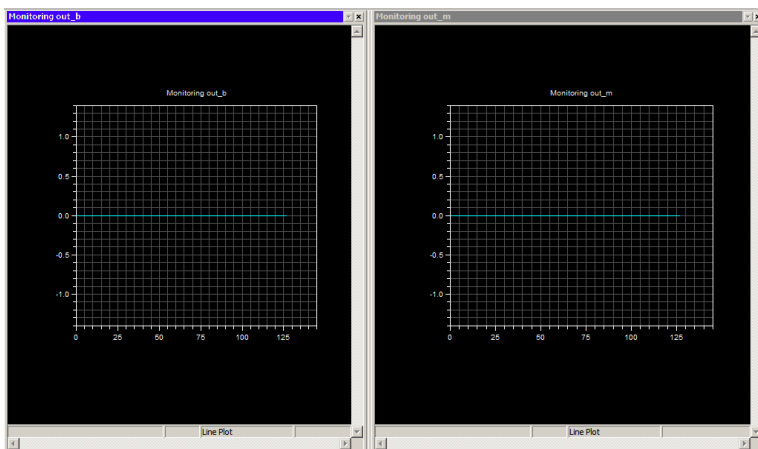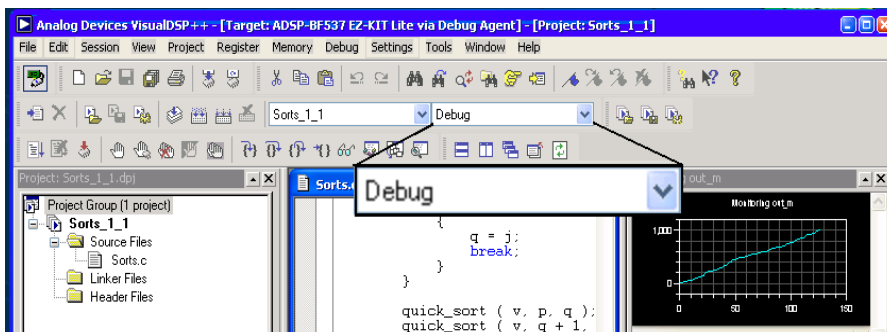Figure 1-4. Project Debug Configuration

The debug configuration is one of the two configurations VisualDSP++ provides for projects. You can create more configurations. A configuration is a set of project build options, similar in concept to a makefile target. It is often desirable to maintain different types of configurations for your system. For example, while debugging, you may want to include trace or

other debugging information, which is not desired in the released product. A VisualDSP++ configuration allows you to create alternate build settings without interfering with the build settings of your final product.

VisualDSP++ automatically adds two configurations for every project it creates. These configurations are:

- **Debug**. Used for functional debugging of your system. Compiler optimizations are off, giving you and the debugger the most linear and easily-debugged code.

- **Release**. Used for your production system. Compiler optimizations are on and maximally aggressive, sacrificing readability and some debugger support.

At this point, feel free to experiment further with the debugger, familiarizing yourself with the windows and basic mechanics of running, halting, stepping, and reloading. C/C++ language debugging windows, such as local variable and expression monitors, function call stack windows, and others are available under the **View–>Debug Windows** menu selection.

Exercise 1, Part 2: Analyzing Performance and Memory Hierarchy Impact

Once you have familiarized yourself with basic VisualDSP++ operations, it is time to use VisualDSP++ to analyze and tweak the program's performance. Close the part 1 project with the **File–>Close–>Project** command. Open the new project, `<istall_path>\Blackfin\Examples\ADSP-BF537 EZ-Kit Lite\Getting Started Examples\Part_1_2\Sorts_1_2.dpj`.

The part 2 project builds on the program discussed in the previous exercise. "Listing 1-2. Exercise 1, Part 2" on page 1-21 includes some new lines of code (delineated in *italics*). This example's code has been placed (for now) into external SDRAM to demonstrate the effects of memory placement. Since the exercise concerns run-time code performance, we will build the project in the **Release** configuration, which invokes the compiler optimizer.
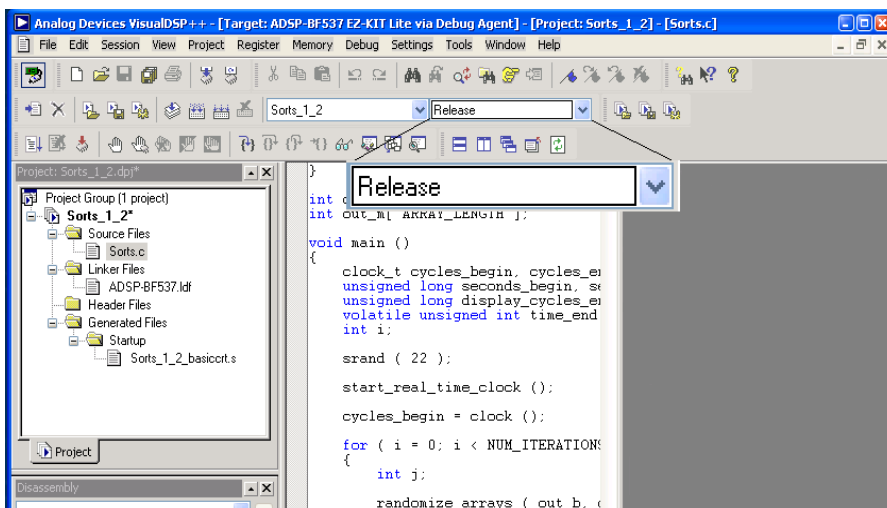

Figure 1-5. Project Release Configuration

Exercise 1, Part 2: Analyzing Performance and Memory Hierarchy Impact

Select the configuration for the project by choosing the **Release** configuration from the drop-down box (Figure 1-5). Then use **Project**–>**Rebuild All** to build the optimized version of the program.

Now you can benchmark and profile the program by:

- Taking advantage of the built-in cycle counters of the Blackfin processor by including benchmark-gathering code in the program.

- Using the real-time clock (RTC) of the ADSP-BF537 processor. This clock measures "human-scale" time (seconds, minutes, and so on).

  Both cycle count and real-time are being measured because, as we learn later, the relationship between the values is not necessarily a constant multiplier.

- Using the statistical profiler. The statistical profiler is a unique tool that polls the Blackfin processor hundreds of times per a second. The data is used to paint a statistical view of the program to determine where the program spends the majority of its time.

  The statistical profiler has a distinct advantage over traditional profiling techniques because it operates non-intrusively (tradition profiling techniques require an instrumentation of your project), requiring zero overhead and not influencing your program's operation. However, because the profiling is statistical in nature, it cannot be relied on as a code-coverage tool, and it cannot show caller information. For this kind of analysis, traditional, instrumented profiling is also available.

For the part 2 program, we use the statistical profiler. Enable the statistical profiler by selecting **Tools**–>**Statistical Profiling**–>**New Profile**. Move and resize the new window until the viewing space is comfortable to continue the exercise.

Run the program using **Debug**–>**Run** (or use the **F5** hotkey). The programs may take 30 seconds or more to run to completion (by design). When the program runs, its status information displays in the lower-right corner of the VisualDSP++ main window (Figure 1-6).
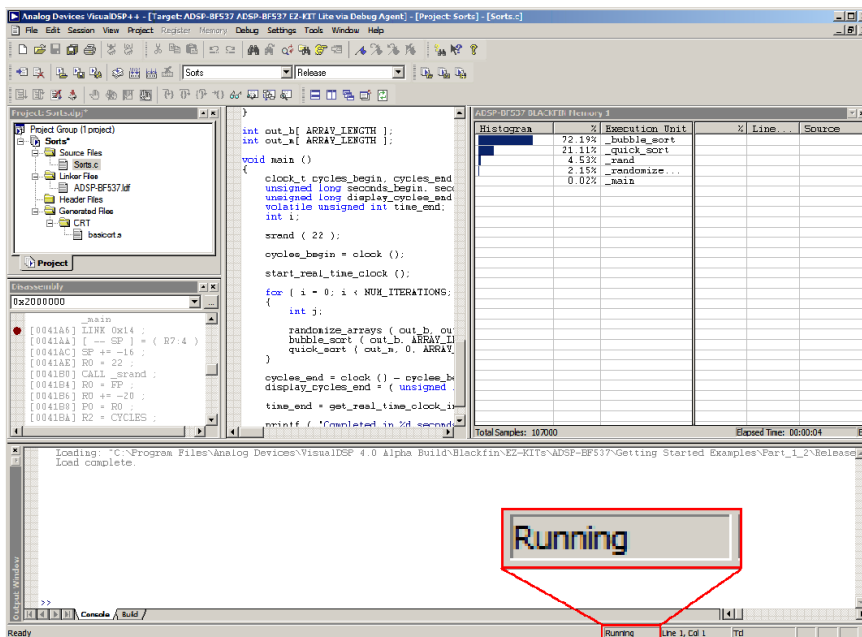
Figure 1-6. Program Status Information

When the program completes, the **Console** page of the **Output** window displays an informative message (in green text, similar to "xx seconds in approx. xxxx cycles"). The message is the output of the printf() call placed at the end of the main() function. Take note of the timing presented. Now look at the **Statistical Profiling** window (Figure 1-7).

Results may vary slightly on your computer.

Not surprisingly, the vast majority of the project's time is spent in the bubble sort algorithm.

Exercise 1, Part 2: Analyzing Performance and Memory Hierarchy Impact
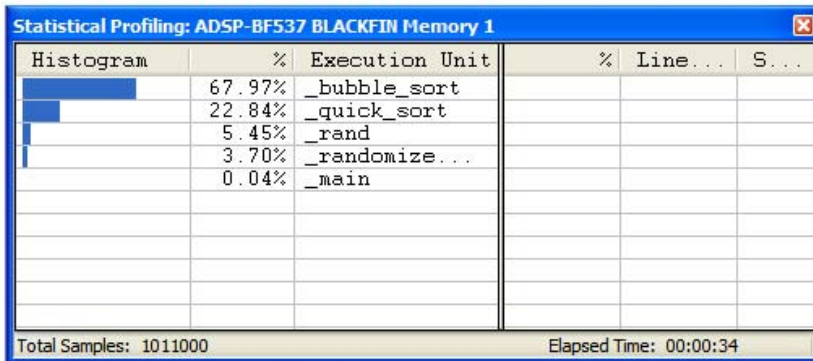
Figure 1-7. Statistical Profiling: Initial Results

The Blackfin processor has a memory hierarchy with a small amount of internal SRAM (L1) flexible in its configuration. You can configure the SRAM as a cache for slower external SDRAM or as "straight" memory for completely optimal, zero latency, access speed. The fact is, the Sorts_1_2.dpj program currently is running under somewhat artificial "worse case" conditions. The program's code has been placed into external SDRAM, while internal memory of the Blackfin processor remains completely unused.

The next steps demonstrate how the proper utilization of the memory hierarchy of the Blackfin processor can improve dramatically the program performance. The "easiest" way to improve the performance is to enable the instruction cache[1] of the Blackfin processor.

---

[1] The Blackfin processor also has a data cache, but its discussion is beyond this exercise. Refer to the *ADSP-BF537 Blackfin Processor Hardware Reference* for more information.

To enable the instruction cache:

1. Open the project wizard (**Project**–>**Project Options** or **ALT-F7**)

2. In the **Project** tree control, locate the **Startup Code Settings** node. Click **Cache and Memory Protection** under **Startup Code Settings** (Figure 1-8).
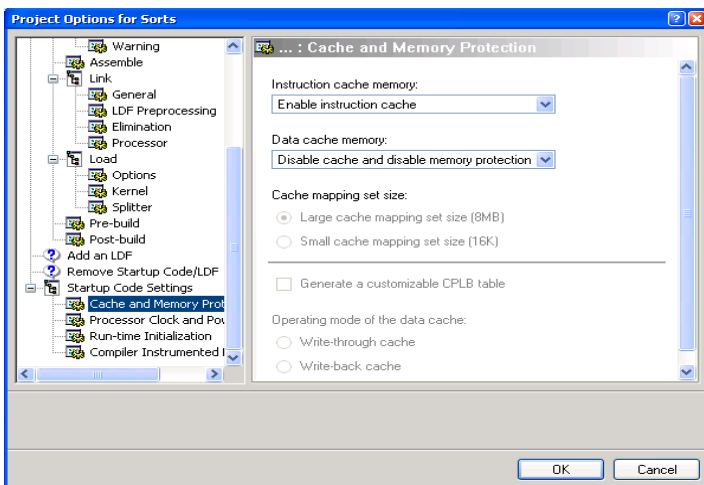
Figure 1-8. Cache and Memory Protection Page

3. In the **Instruction cache memory** drop-down box, select **Enable instruction cache.**

4. Click **OK** to set the new project option. VisualDSP++ will regenerate the necessary files to enable the instruction case when the program runs.

5. Rebuild the project by selecting **Project**–>**Rebuild All** or by right-clicking the project name in the **Project** window and selecting **Rebuild All**.

Exercise 1, Part 2: Analyzing Performance and Memory Hierarchy Impact

Rerunning the program shows a dramatic improvement in the program's performance, reflected by the output of the `printf()` statement. The **Statistical Profiling** window's results are largely unchanged. Note that since this is a statistical view, minor differences between the program runs are expected.

Another way to tune the performance of a Blackfin processor is to place key algorithms into internal memory. In this program, `bubble_sort()` is the "key algorithm" in that it consumes the majority of the Blackfin's processing power. You can demonstrate the effect of placing the key algorithm in internal memory by first "undoing" the cache set-up and then placing the `bubble_sort()` function into L1 internal memory:

1. In the **Project**–>**Project Options**, navigate to the **Startup Code Settings**–>**Cache and Memory Protection** page and change **Instruction cache memory** to **Disable cache and disable memory protection**.

2. In the `sorts.c` file, go to the declaration of `bubble_sort()` and add the section qualifier to place this function (and this function only) into internal L1 memory:
   ```
   section("L1_code") void bubble_sort(int *v,
                                       unsigned int length)
   ```

3. Build and run your program to completion. Now observe that the overall performance is better than the first, non-cached run, but not as good as the cached run because a significant portion of the program still runs from external memory without the benefit of a cache.

The statistical profile shows an interesting effect of the change: the `bubble_sort()` function no longer uses the largest percentage of the processor time—`quick_sort()` is now the most time-consuming portion of the application (Figure 1-9).
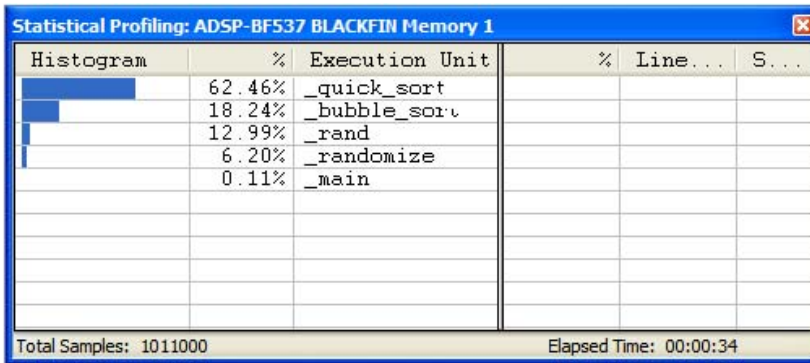
Figure 1-9. Statistical Profiling: Results after Moving bubble_sort()

If `quick_sort()` is moved to internal memory as well, overall execution time again improves and the relative speed of `bubble_sort()` versus `quick_sort()` returns to the expected ratio.

The exercise program is small enough to fit entirely into the internal memory of the processor. In this scenario, VisualDSP++ places the program into internal memory by default. However, this is not likely to be the case for a complex application, where the techniques utilized in this exercise come into play. The optimal memory configuration varies from an application to application, possibly involving a blend of caching and direct L1 placement.

Exercise 1, Part 3: Working with Blackfin Voltage Regulator

All Blackfin processors, including the ADSP-BF537, include an on-chip voltage regulator that allows you to control the processor's power consumption and clock. The trade-off is non-linear, that is, an increase in clock frequency requires a larger voltage which is non-linearly greater than the clock frequency gain. Conversely, dropping the clock frequency cre-

ates an even greater drop in the required voltage (see the *ADSP-BF536/ADSP-BF537 Embedded Processor Data Sheet* for specifics). VisualDSP++ allows you to easily change the voltage and clock speed in a running application. The remainder of this exercise uses the System Services Library to run the application at a variety of voltage levels.

Close the part 2 project and statistical profile window and open the project `<install_path>\Blackfin\Examples\ADSP-BF537 EZ-Kit Lite\Getting Started Examples\Part_1_3\Sorts_1_3.dpj`. Again, the new project builds on the project of the previous exercise, but now with the entire application loading into the L1 memory of the ADSP-BF537 processor. "Listing 1-3. Exercise 1, Part 3" on page 1-25 delineates the new lines of code in italics. Significantly, the bulk of the `main()` function is placed in a loop. The goal is to run the previous exercise at four different voltages (see the `voltage_levels` struct declared above `main()`).

You again use the **Release** configuration for this exercise. Build and run the program. It takes a minute or more to run the program to completion, using `printf()` to output a table of results to the VisualDSP++ **Console** window. The results are shown in Figure 1-10.

ⓘ Results may vary slightly on your computer.

```
Voltage           Seconds        Cycles (x 1m)
   0.85               36                   8777
   0.95               21                   8777
   1.05               18                   8777
   1.25               14                   8777
```

Figure 1-10. Sorts Results in Console Window

The key observation are:

- As voltage increases, execution time decreases

- Cycle count remains constant regardless of the voltage

Listing 1-1. Exercise 1, Part 1

```
/*
 * Getting Started With the ADSP-BF537 EZ-KIT Lite
 * Part 1, Exercise 1
 */
#include <stdlib.h>
#define NUM_ITERATIONS    1
#define ARRAY_LENGTH      128
/* Initialize two arrays to the same set of random values */
void randomize_arrays ( int *v1, int *v2, unsigned int length )
{
   unsigned int i;
   for ( i = 0; i < length; ++i )
   {
      v1[ i ] = v2[ i ] = rand () % 1024;
   }
}


/* A standard bubble sort algorithm, O(n^2) */
void bubble_sort ( int *v, unsigned int length )
{
   unsigned int i, j;
   for ( i = 0; i < length - 1; ++i )
   {
      for ( j = i + 1; j < length; ++j )
      {
         if ( v[ i ] > v[ j ] )
         {
            int temp = v[ i ];
            v[ i ] = v[ j ];
            v[ j ] = temp;
         }
      }
   }
}
/* A standard quick sort algorithm, O(n*log(n)) */
void quick_sort ( int *v, unsigned int p, unsigned int r )
{
   if ( p < r )
```

Listing 1-1. Exercise 1, Part 1

```
   {
      unsigned int x, i, j, q;
      x = v[ p ];
      i = p - 1;
      j = r + 1;
      for ( ;; )
      {
         do { --j; } while ( v[ j ] > x );
         do { ++i; } while ( v[ i ] < x );

         if ( i < j )
         {
            int temp = v[ i ];
            v[ i ] = v[ j ];
            v[ j ] = temp;
         }
         else
         {
            q = j;
            break;
         }
      }
      quick_sort ( v, p, q );
      quick_sort ( v, q + 1, r );
   }
}
int out_b[ ARRAY_LENGTH ];
int out_m[ ARRAY_LENGTH ];
void main (){
   int i;
   srand ( 22 );
   for ( i = 0; i < NUM_ITERATIONS; ++i )
   {
      randomize_arrays ( out_b, out_m, ARRAY_LENGTH );
      bubble_sort ( out_b, ARRAY_LENGTH );
      quick_sort ( out_m, 0, ARRAY_LENGTH - 1 );
   }
}
```

Listing 1-2. Exercise 1, Part 2

```
/*
 * Getting Started With the ADSP-BF537 EZ-KIT Lite
 * Part 1, Exercise 2
 */

#include <stdlib.h>
#include <stdio.h>
#include <ccblkfn.h>
#include <cdefbf533.h>
#include <sysreg.h>
#include <time.h>

#define NUM_ITERATIONS    5000
#define ARRAY_LENGTH      128

void start_real_time_clock(void);
unsigned int get_real_time_clock_in_seconds(void);

/* Helper function to enable the real-time clock and reset it to
time "zero" */
#define WAIT_FOR_RTC_WRITE_COMPLETE()
   {while ( *pRTC_ISTAT & 0x8000 ); }

void start_real_time_clock ()
{
   if ( !*pRTC_PREN )
   {
      *pRTC_PREN = 1;

      WAIT_FOR_RTC_WRITE_COMPLETE();
   }

   *pRTC_STAT = 0;
   WAIT_FOR_RTC_WRITE_COMPLETE();
}

/* Help function to get the number of seconds since "zero" time.
 * Only works up to one hour of time. */
```

Listing 1-2. Exercise 1, Part 2

```c
unsigned int get_real_time_clock_in_seconds ()
{
   unsigned int clock = *pRTC_STAT;

   /* second */
   unsigned int seconds = ( clock & 0x3f );

   /* minutes */
   seconds += 60 * ( clock & 0xfc0 ) >> 6;

   return seconds;
}

/* Initialize two arrays to the same set of random values */
void randomize_arrays ( int *v1, int *v2, unsigned int length )
{
   unsigned int i;

   for ( i = 0; i < length; ++i )
   {
       v1[ i ] = v2[ i ] = rand () % 1024;
   }
}

/* A standard bubble sort algorithm, O(n^2) */
/* section("L1_code") */

void bubble_sort ( int *v, unsigned int length )
{
   unsigned int i, j;

   for ( i = 0; i < length - 1; ++i )
   {
      for ( j = i + 1; j < length; ++j )
      {
          if ( v[ i ] > v[ j ] )
          {
              int temp = v[ i ];
              v[ i ] = v[ j ];
              v[ j ] = temp;
```

Getting Started with ADSP-BF537 EZ-KIT Lite

```
         }
       }
    }
}

/* A standard quick sort algorithm, O(n*log(n)) */
void quick_sort ( int *v, unsigned int p, unsigned int r )
{
   if ( p < r )
   {
      unsigned int x, i, j, q;
      x = v[ p ];
      i = p - 1;
      j = r + 1;

      for ( ;; )
      {
         do { --j; } while ( v[ j ] > x );
         do { ++i; } while ( v[ i ] < x );

         if ( i < j )
         {
            int temp = v[ i ];
            v[ i ] = v[ j ];
            v[ j ] = temp;
         }
         else
         {
            q = j;
            break;
         }
      }

      quick_sort ( v, p, q );
      quick_sort ( v, q + 1, r );
   }
}

int out_b[ ARRAY_LENGTH ];
int out_m[ ARRAY_LENGTH ];
```

Listing 1-2. Exercise 1, Part 2

```
void main ()
{
    clock_t cycles_begin, cycles_end;
    unsigned long seconds_begin, seconds_end;
    unsigned long display_cycles_end;
    volatile unsigned int time_end;
    int i;

    srand ( 22 );
    start_real_time_clock ();
    cycles_begin = clock ();

    for ( i = 0; i < NUM_ITERATIONS; ++i )
    {
        int j;

        randomize_arrays ( out_b, out_m, ARRAY_LENGTH );
        bubble_sort ( out_b, ARRAY_LENGTH );
        quick_sort ( out_m, 0, ARRAY_LENGTH - 1 );
    }

    cycles_end = clock () - cycles_begin;
    display_cycles_end = ( unsigned long)( cycles_end / 1000000 );
    time_end = get_real_time_clock_in_seconds ();
    printf ( "Completed in %d seconds and approx. %u million
            cycles.\n", time_end, display_cycles_end );
}
```

Listing 1-3. Exercise 1, Part 3

```
/*
 * Getting Started With the ADSP-BF537 EZ-KIT Lite
 * Part 1, Exercise 3
 */

#include <stdlib.h>
#include <stdio.h>
#include <ccblkfn.h>
#include <cdefbf537.h>
#include <sysreg.h>
#include <time.h>
#include <services/adi_pwr.h>

#define NUM_ITERATIONS   50000
#define ARRAY_LENGTH     128

/* Helper function to enable the real-time clock and reset it to
time "zero" */

#define WAIT_FOR_RTC_WRITE_COMPLETE()
{ while ( ! ( *pRTC_ISTAT & 0x8000 ) ); }

void start_real_time_clock ()
{
   if ( !*pRTC_PREN )
   {
      *pRTC_PREN = 1;
      WAIT_FOR_RTC_WRITE_COMPLETE();
   }

   *pRTC_STAT = 0;
   WAIT_FOR_RTC_WRITE_COMPLETE();
}

/* Helper function to get the number of seconds since zero time.
 * Only works up to one hour of time. */
unsigned int get_real_time_clock_in_seconds ()
{
```

Listing 1-3. Exercise 1, Part 3

```
   unsigned int clock = *pRTC_STAT;

   /* seconds */
   unsigned int seconds = ( clock & 0x3f );

   /* minutes */
   seconds += 60 * ( ( clock & 0xfc0 ) >> 6 );

   return seconds;
}

/* Initialize two arrays to the same set of random values */
void randomize_arrays ( int *v1, int *v2, unsigned int length )
{
   unsigned int i;

   for ( i = 0; i < length; ++i )
   {
      v1[ i ] = v2[ i ] = rand () % 1024;
   }
}

/* A standard bubble sort algorithm, O(n^2) */
void bubble_sort ( int *v, unsigned int length )
{
   unsigned int i, j;

   for ( i = 0; i < length - 1; ++i )
   {
      for ( j = i + 1; j < length; ++j )
      {
         if ( v[ i ] > v[ j ] )
         {    int temp = v[ i ];
              v[ i ] = v[ j ];
              v[ j ] = temp;
         }
      }
   }
}
```

```
/* A standard quick sort algorithm, O(n*log(n)) */
void quick_sort ( int *v, unsigned int p, unsigned int r )
{
   if ( p < r )
   {
      unsigned int x, i, j, q;

      x = v[ p ];
      i = p - 1;
      j = r + 1;
      for ( ;; )
      {
         do { --j; } while ( v[ j ] > x );
         do { ++i; } while ( v[ i ] < x );

         if ( i < j )
         {
            int temp = v[ i ];
            v[ i ] = v[ j ];
            v[ j ] = temp;
         }
         else
         {
            q = j;
            break;
         }
      }

      quick_sort ( v, p, q );
      quick_sort ( v, q + 1, r );
   }
}

int out_b[ ARRAY_LENGTH ];
int out_m[ ARRAY_LENGTH ];

ADI_PWR_COMMAND_PAIR ezkit_init[] =
{
   /*600Mhz ADSP-BF537 EZ-KIT Lite */
   {ADI_PWR_CMD_SET_EZKIT,  (void*)ADI_PWR_EZKIT_BF537_600MHZ},
```

Listing 1-3. Exercise 1, Part 3

```
    /* command to terminate the table */
    {ADI_PWR_CMD_END,0 }
};

typedef struct
{
    ADI_PWR_VLEV v;
    const char *n;
} voltage_levels_type;

voltage_levels_type voltage_levels[] =
{
   { ADI_PWR_VLEV_085, "0.85" },
   { ADI_PWR_VLEV_095, "0.95" },
   { ADI_PWR_VLEV_105, "1.05" },
   { ADI_PWR_VLEV_125, "1.25" },
};

void main ()
{
   unsigned long long cycles_begin, cycles_end;
   unsigned long seconds_begin, seconds_end;
   unsigned long display_cycles_end;
   volatile unsigned int time_end;
   int i, v;

   adi_pwr_Init(ezkit_init);

   srand ( 22 );

   printf("%20s%20s%20s\n","Voltage","Seconds","Cycles (x 1m)");

   for ( v = 0; v < sizeof ( voltage_levels ) / sizeof (
voltage_levels_type ); ++v )
   {
      adi_pwr_SetMaxFreqForVolt ( voltage_levels[ v ].v );

      start_real_time_clock ();
      cycles_begin = clock ();
```

```
for ( i = 0; i < NUM_ITERATIONS; ++i )
{
   randomize_arrays ( out_b, out_m, ARRAY_LENGTH );
   bubble_sort ( out_b, ARRAY_LENGTH );
   quick_sort ( out_m, 0, ARRAY_LENGTH - 1 );
}

cycles_end = clock () - cycles_begin;
display_cycles_end = (unsigned long)(cycles_end / 1000000);
time_end = get_real_time_clock_in_seconds ();

printf ( "%20s%20u%20u\n", voltage_levels[ v ].n,
         time_end, display_cycles_end );
   }
}
```

Listing 1-3. Exercise 1, Part 3

2 RUNNING A TCP/IP APPLICATION ON AN ADSP-BF537 EZ-KIT LITE

In Exercise 2, we will use VisualDSP++ to create a basic TCP/IP application using the open source LwIP stack and the VisualDSP++ Kernel (VDK). Then, we will connect the EZ-KIT Lite to the Ethernet network, receive an IP address from a Dynamic Host Configuration Protocol (DHCP) server, and ping the EZ-KIT from another computer. Finally, we will implement a simple Caesar Cipher program and run the program via telnet.

In the exercise, you will learn about the following concepts.

- TCP/IP and the LwIP stack

- VDK and its relationship to LwIP

- TCP/IP project type

- Determining Ethernet Media Access Control (MAC) and IP addresses

- Sockets programming and input/output management

Part 1: A Primer on TCP/IP, OS Threads and Semaphores, and Sockets API

The ADSP-BF537 EZ-KIT Lite includes an on-chip 10/100 Mbits/sec Ethernet MAC. The interface is exposed on the board, providing an easy connection between the EZ-KIT Lite and an existing TCP/IP network. VisualDSP++ includes an open source TCP/IP software stack, LwIP, ported to the Blackfin architecture. This stack relies on the presence of an underlying operating system, and the VisualDSP++ Kernel serves as the operating system. Therefore, the EZ-KIT Lite and VisualDSP++ provide out-of-the-box software and a hardware connection for TCP/IP networking.[1]

The LwIP stack's interface to both VDK and the underlying EZ-KIT Lite hardware is well-abstracted into libraries with defined APIs. This makes the EZ-KIT Lite and VisualDSP++ a good test and evaluation vehicle for your application. The application can be ported later to alternative hardware and/or operating systems without modifying the internals of the LwIP stack. Furthermore, the LwIP stack is programmed using the industry-standard Berkley Socket (or just "sockets") APIs, so existing code bases can be adapted to LwIP quickly.

If you are unfamiliar with the basic concepts surrounding TCP/IP, operating system threads, or sockets, a primer can be found in "Creating a TCP/IP Application" on page A-1.

---

[1] Some third parties provide operating systems and/or TCP/IP stack solutions that can be more feature-rich than VisualDSP++.

Part 2: Creating a TCP/IP Application and Connecting to a DHCP Server

VisualDSP++ includes a TCP/IP project type to handle the creation of a framework for your TCP/IP-aware application. The generated project is a working application that:

1. Initializes all hardware and software needed to service the stack

2. Reads the unique MAC address from your EZ-KIT Lite's firmware. The MAC address uniquely identifies your hardware from every other Ethernet-aware device in the world.

3. Connects to your network's DHCP server and receives an IP address. The IP address is used to connect to the target board.

4. Continues running the application

To start, physically connect the ADSP-BF537 EZ-KIT Lite hardware to the 10/100 Mbits/sec network (the same network your PC connects to) and then create the TCP/IP support software application using the supplied non-cross-over cable.

 Close any open projects. To create a skeleton TCP/IP project:

1. Select **File**–>**New**–>**Project** to start the **Project Wizard**. On the **Project: Select Type** page, select **CP/IP Stack application using LwIP and VDK** as the project type and choose the name and directory for the project (Figure 2-1). Click **Next**.

2. On the **Project: Select Processor** page, set **ADSP-BF537 Blackfin Processor** as the processor type. Click **Next**.
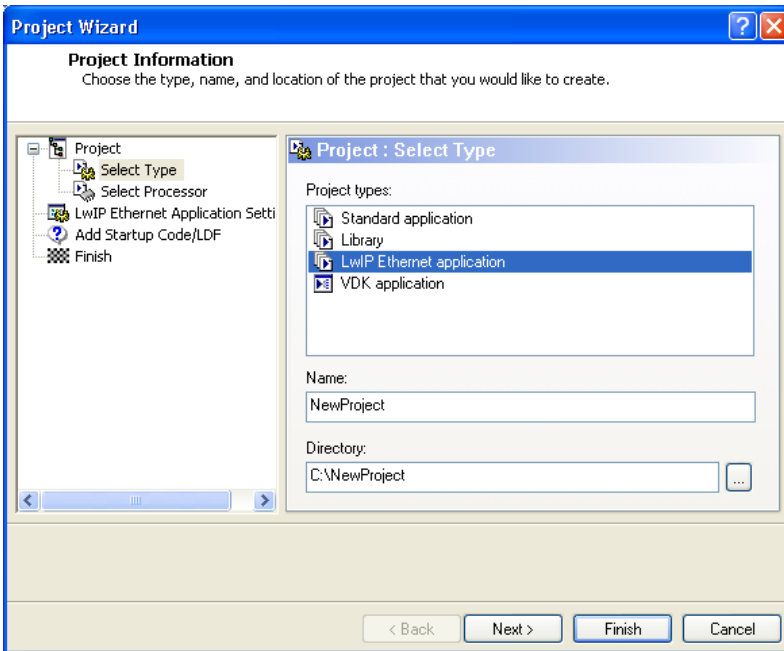
Figure 2-1. VDK Project with LwIP TCP/IP Stack

3. On the **LwIP Ethernet Application Settings** page, click **Next**.

4. On the **Add Startup Code/LDF** page, ensure that **Add an LDF and startup code** is selected. Click **Finish** to close the wizard.

The LwIP stack, as distributed with VisualDSP++, relies on the presence of a DHCP server on your network. If your network does not support DHCP, the LwIP library must be rebuilt using a static IP address assigned by the network administrator. For details regarding rebuilding the LwIP library with a static IP address, refer to the `LWIP_UserGuide.doc` file in the `<install_path>\Black-fin\lib\src\lwip\docs` directory.

VisualDSP++ creates the skeleton application. Build the project with
Project–>Build. Once the project is built and loaded to the EZ-KIT Lite,
run the example. Within a few seconds, output is emitted in the Visu-
alDSP++ **Output** window, similar to:

```
IP ADDRESS xxx.xxx.xxx.xxx
```
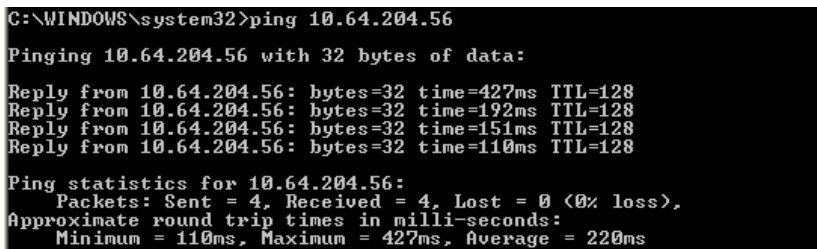
Leave the EZ-KIT Lite running. Now, "ping" the EZ-KIT Lite from your
PC. Open a **Command Prompt** window (DOS application) and type the
following command, using the IP address reported in the VisualDSP++
Output window:

```
ping xxx.xxx.xxx.xxx
```

Again, the EZ-KIT Lite's IP address is assigned by the DHCP server.

(i) Because network configurations are dynamic, the IP address can
change from run to run. Remember to check the address each time
the program runs.

The output of the ping command is emitted in the **Command Prompt**
window with output similar to Figure 2-2.


```
C:\WINDOWS\system32>ping 10.64.204.56

Pinging 10.64.204.56 with 32 bytes of data:

Reply from 10.64.204.56: bytes=32 time=427ms TTL=128
Reply from 10.64.204.56: bytes=32 time=192ms TTL=128
Reply from 10.64.204.56: bytes=32 time=151ms TTL=128
Reply from 10.64.204.56: bytes=32 time=110ms TTL=128

Ping statistics for 10.64.204.56:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 110ms, Maximum = 427ms, Average = 220ms
```

Figure 2-2. Ping Output

Examine the source code of this skeleton application in the **Project** window. The `lwip_sysboot_threadtype.c` file is worthy of examination because the file needs to be modified in order to add functionality to the stack.

Specifically, open the source file and scroll to (or search for) the function `lwip_sysboot_threadtype_RunFunction()`. Observe the comment block,

```
/**
 *  Add Application Code here
 **/
```

where, in the next exercise, you will insert code to add functionality to the application.

The remainder of the source code is this file is beyond the scope of this book's exercises. The code relies heavily on the system services library that is touched upon in the previous exercises.

Part 3: Communicating with Sockets API

Now it is time to add functionality to the created application. First, you create a simple Caesar Cipher program, then run the program, and access it from the computer via telnet. The Caesar Cipher is a simple data encryption algorithm, which increments each input letter by a value (the increment of one is used in this implementation). The letter A becomes B, B becomes C, and so on. The algorithm wraps at the end of the alphabet, with Z becoming A.

To enable multiple clients (computers) to connect concurrently to the same host (the EZ-KIT Lite), TCP/IP applications typically use multiple operating system threads of execution as follows.

1. The main application thread begins polling the port number(s) relevant to the application.

2. When a client connects, a new "worker" thread is spawned to interact with the client.

3. Meanwhile, the main application thread continues polling for new connections.

The following procedure creates a Caesar Cipher application from the skeleton template. The steps implement a thread to interact with the client, producing a Caesar Cipher on input received from the client. Alternatively, the project `<install_path>\Blackfin\Examples\ADSP-BF537 EZ-Kit Lite\Getting Started Examples\Part_2_1\Caesar_Cipher.dpj` can be loaded to the PC if you are unconcerned about the details herein and want to avoid typing in source code. (If you load the project, skip the following procedure).

To create a new thread type, `Caesar_Cipher_ThreadType`:

1. From the **Kernel** page of VisualDSP++ **Project** window, navigate to **Kernel**–>**Threads**–>**Thread Types**. Right-click and select **New** thread type. Type `Caesar_Cipher_ThreadType` in the **Name** field, leave **Source File** and **Header File** as is, automatic source code generation as **Yes**, and leave the **Language** as **C++** (Figure 2-3).

2. Click **OK**. The new thread displays in the **VisualDSP++ Kernel** tab.

3. Switch back to the **Project** page and open the newly-created source file `Caesar_Cipher_ThreadType.cpp`. Replace the method `Caesar_Cipher_ThreadType::Run()` with the contents of "Listing 2-1. Caesar_Cipher_ThreadType::Run() Implementation" on
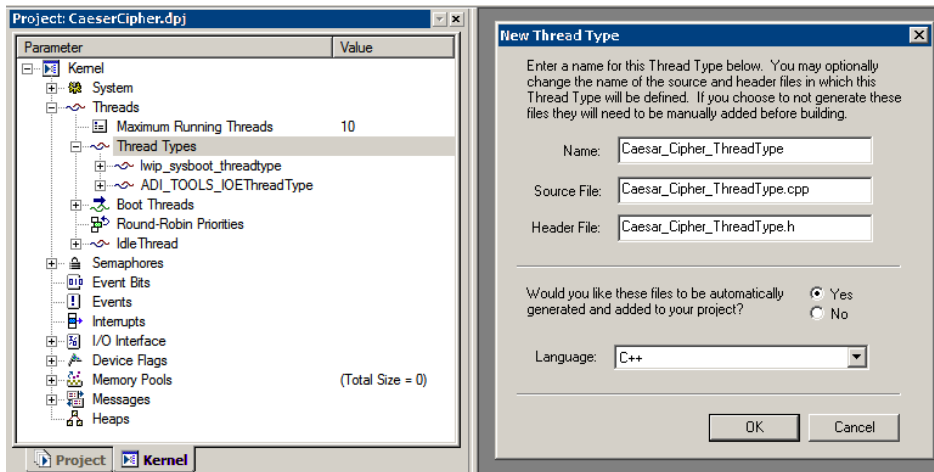
Figure 2-3. New Thread Type

page 2-10. Next, replace the constructor
`Caesar_Cipher_ThreadType::Caesar_Cipher_ThreadType()` with
the contents of "Listing 2-2. Caesar_Cipher_ThreadType() Imple-
mentation" on page 2-11. The new lines of code are delineated in
italics.

4. Open `Caesar_Cipher_ThreadType.h` and add a few new member
variables (see "Listing 2-3. VDK::Thread Implementation" on
page 2-11).

5. Go back to `Caesar_Cipher_ThreadType.cpp` and add
`#include <lwip/sockets.h>` near the top of the file, with the other
`#include` directives, to make the sockets API "known" to this
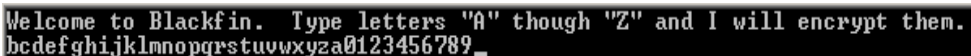source file.

6. Open the source file `lwip_sysboot_threadtype.c`. Add the following line to the top of the file: `VDK_ThreadID g_AttenuatorID`.

7. Replace the function `lwip_sysboot_threadtype_RunFunction()` with the contents of "Listing 2-4. lwip_sysboot_threadtype_RunFunction() Implementation" on page 2-12. The new lines of code are delineated in italics.

Build and run the new project. As described earlier, the assigned IP address is echoed to VisualDSP++ **Output** window.

Open the **Command Prompt** window again. This time, use the telnet application to connect to the EZ-KIT Lite:

```
telnet xxx.xxx.xxx.xxx
```

Once connected, you receive the welcome message implemented in `Caesar_Cipher_ThreadType::Run()`. Type characters in the **Output** window—the letters A through Z (upper and lowercase) are incremented by one, while non-letters are echoed back undisturbed (see Figure 2-4).


```
Welcome to Blackfin.  Type letters "A" though "Z" and I will encrypt them.
bcdefghijklmnopqrstuvwxyza0123456789_
```

Figure 2-4. Telnet Command Prompt

When finished, close the telnet session with **Ctrl+]**, then type `quit` at the telnet prompt.

You can test the EZ-KIT Lite's capability to handle multiple concurrent connections by opening multiple command prompt windows and running a telnet session in each of the windows. If you have access to multiple computers on the same network, this is another means to effectively demonstrate the ability to maintain multiple concurrent connections.

Listing 2-1. Caesar_Cipher_ThreadType::Run() Implementation

Listing 2-1. Caesar_Cipher_ThreadType::Run() Implementation

```
void
Caesar_Cipher_ThreadType::Run()
{
   static char *pszWelcome = "Welcome to Blackfin.  Type letters
\"A\" though \"Z\" and I will encrypt them.\xa\xd";
   if ( 0 >= send ( m_iSocket, pszWelcome, strlen ( pszWelcome ),
0 ) )
      return;
   while (1)
   {
      int iCount;
      if ( ( iCount = recv ( m_iSocket, m_vInBuf, sizeof (
m_vInBuf ) / sizeof ( char ), 0 ) ) >= 1 )
      {
         int iCharNum;
         char c;

         for ( iCharNum = 0; iCharNum < iCount; ++iCharNum )
         {
            c = m_vInBuf [ iCharNum ];

            if ( ( c >= 'A' && c <= 'Y' ) || ( c >= 'a' && c <=
'y' ) )
                ++c;
            else if ( c == 'Z' || c == 'z' )
                c -= ( 'Z' - 'A' );
            m_vOutBuf [ iCharNum ] = '\x8';  /* telnet back-
space control character to overwrite the character sent */
            m_vOutBuf [ iCharNum + iCount ] = c;
         }

         if ( send (m_iSocket, m_vOutBuf, iCount * 2, 0) <= 0 )
            break;
      }
```

```
        else
    {
        break;
    }
    }

    close ( m_iSocket );
}
```

Listing 2-2. Caesar_Cipher_ThreadType() Implementation

```
Caesar_Cipher_ThreadType::Caesar_Cipher_ThreadType(VDK::Thread::
ThreadCreationBlock &tcb)
  : VDK::Thread(tcb)
{
    m_iSocket = (int) tcb.user_data_ptr;
}
```

Listing 2-3. VDK::Thread Implementation

```
class Caesar_Cipher_ThreadType : public VDK::Thread
{
public:
  Caesar_Cipher_ThreadType(VDK::Thread::ThreadCreationBlock&);
  virtual ~Caesar_Cipher_ThreadType();
  virtual void Run();
  virtual int ErrorHandler();
  static VDK::Thread* Create(VDK::Thread::ThreadCreationBlock&);

/* The following declarations are specific to this example */

protected:
    int m_iSocket;
    char m_vInBuf[16];
    char m_vOutBuf[16*2];
```

**Listing 2-4. lwip_sysboot_threadtype_RunFunction()
Implementation**

```
    int m_iBufLen;

};
```

Listing 2-4. lwip_sysboot_threadtype_RunFunction() Implementation

```
void
lwip_sysboot_threadtype_RunFunction(void **inPtr)
{
   char  ip[32];
   /* Initializes the TCP/IP Stack and returns */
   if(system_init() == -1)
   {
      printf("Failed to initialize system\n");
      return;
   }

   /* start stack */
   start_stack();

   /*
    * For debug purposes, printf() IP address to the VisualDSP++
    * console window. Likely not needed in final application.
    */

   memset(ip,0,sizeof(ip));
   if(gethostaddr(0,ip))
   {
      printf("IP ADDRESS: %s\n",ip);
   }

   /**
    *  Add Application Code here
    **/
```

```
{
    struct sockaddr_in saddr;
    int listenfd;

    if ( 0 > ( listenfd = socket ( AF_INET, SOCK_STREAM, 0 ) ) )
    {
        printf ( "Call to socket() failed.\n" );
        abort();
    }

    memset ( &saddr, 0, sizeof ( saddr ) );
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = htonl ( INADDR_ANY );
    saddr.sin_port = htons ( 23 ); /*  listening on port 23
                              (well-known default "telnet") */

    if ( -1 == bind ( listenfd, (struct sockaddr*) &saddr,
sizeof(saddr) ) )
    {
         printf ( "Call to bind() failed.\n" );
         abort();
    }

    if ( -1 == listen ( listenfd, 0 ) )
    {
        printf ( "Call to listen() failed.\n" );
        abort();
    }

    for ( ;; )
    {
        struct sockaddr cliaddr;
        int clilen;
        int iSocket;

        iSocket = accept (listenfd, &cliaddr, &clilen);
        if ( -1 == iSocket )
        {
            printf ( "Call to accept() failed.\n" );
            abort();
```

Listing 2-4. lwip_sysboot_threadtype_RunFunction() Implementation

```
        }

        VDK_ThreadCreationBlock TCB = {
            kCaesar_Cipher_ThreadType,
            (VDK_ThreadID)0,
            0,
            (VDK_Priority)0,
            (void *) iSocket,
            0
        };

        if ( UINT_MAX == VDK_CreateThreadEx ( &TCB ) )
        {
            printf( "Call to VDK_CreateThreadEx()failed.\n" );
            abort();
        }
    }
}

/* Put the thread's exit from "main" HERE */
/* A thread is automatically Destroyed when it exits its run
function */
}
```

3 CONTROLLING A REAL-TIME APPLICATION VIA TCP/IP

In Exercise 3 you will build upon the Ethernet application constructed in the previous exercise. The application is augmented with audio pass-through VDK threads that operate concurrently with and independently from the stack. Then you will modify the stack's functionality to change, on the fly, the audio pass-through volume.

In this exercise, you will learn:

- How to manage an audio encoder and decoder with device drivers

- How to use VDK history views to examine and trace system behavior

- How thread priorities impact system behavior

- How threads communicate with VDK messages

Part 1: Creating an Audio Pass-Through Application with VDK

The ADSP-BF537 EZ-KIT Lite includes both an analog-to-digital converter (ADC) and a digital-to-analog converter (DAC). These converters allow the EZ-KIT Lite to digitize an incoming audio signal, perform operations on the signal, and convert the signal back to analog, allowing you to hear the results.

The most elementary type of audio application is a *pass-through*. The audio signal is digitized, brought into the Blackfin processor, and then simply converted back to analog with no alteration of the audio stream performed. While this basic program does not seem to have any practical application, its source code can serve as a framework, a useful starting point, for creating a "serious" application.

In this exercise, we start with a program that features the Caesar Cipher algorithm from page 2-6 and augment the program with an audio pass-through that is running concurrently and independently from the Caesar Cipher.

First, the EZ-KIT Lite hardware is connected to the network, audio source, and audio destination:

1. Connect the EZ-KIT Lite to the network as described in Exercise 2 on page 2-3.

2. Connect an audio source to the mini-din (mini-headphone) connector on the EZ-KIT Lite. The connector is labeled LINE IN. Any two-channel stereo audio source is suitable for the exercise, but the most convenient source is your PC (for example, playing back an

MP3 file). If your audio source uses a mini-din connector for out-
put, you can connect the source to the EZ-KIT Lite with the
provided connector.

3. Connect an audio destination, such as the provided headphones, to
   the mini-din connector on the EZ-KIT Lite. The connector is
   labeled `LINE OUT`.

Your completed hardware interconnections look similar to the diagram in
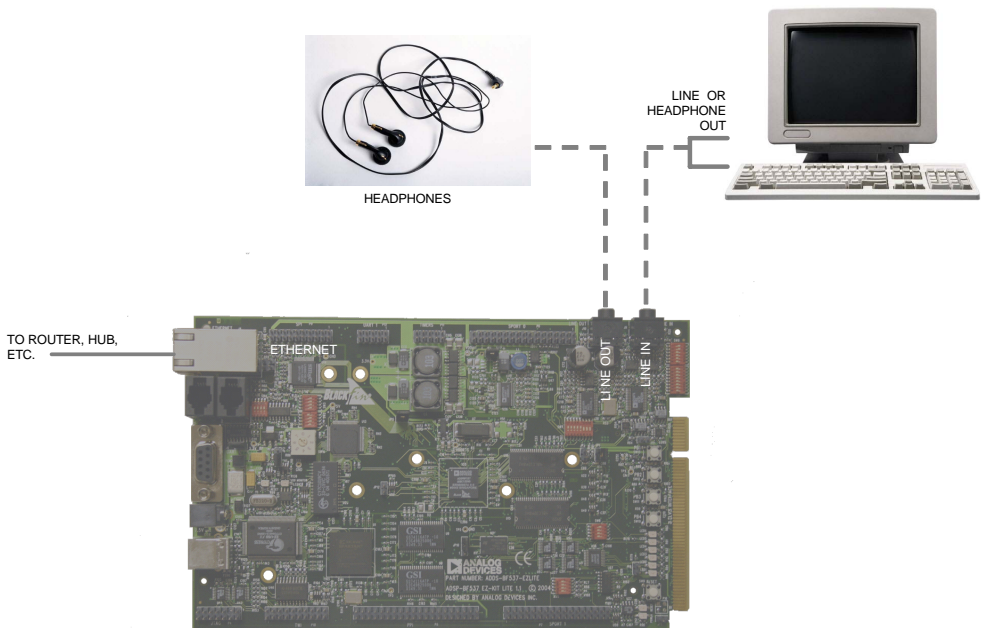Figure 3-1.

Figure 3-1. EZ-KIT Lite Connection Diagram

Next, load the project `<install_path>\Blackfin\Examples\ADSP-BF537
EZ-Kit Lite\Getting Started Exam-
ples\Part_3_1\TalkThrough_3_1.dpj` to the EZ-KIT Lite. Build and run
the application. While the application is running, enable the audio play-

back on your audio source. You will hear the output on the output device connected to the EZ-KIT Lite. Concurrently, use the telnet session and Caesar Cipher algorithm to communicate to the EZ-KIT Lite, as described in Exercise 2 on .

🚫 If you hear no sound, confirm that the Blackfin processor is running. Double-check all audio connections. Ensure the volume on the audio source is not muted and turned up to a suitable level.

Taking the EZ-KIT Lite "out of the circuit" by connecting the audio source directly to the audio destination may also be helpful when troubleshooting the application. If the problem persists, consult the *ADSP-BF537 EZ-KIT Lite Evaluation System Manual* and confirm that all of the DIP switches are in the correct positions.

While the pass-through application is running, examine its relation to the existing Caesar Cipher application from Exercise 2. Switch to the **Kernel** page of the **Project** window to see three new thread types added to the Caesar Cipher (Figure 3-2).

These new thread types are:

- **Input** to manage the audio input to the ADSP-BF537 processor

- **Output** to manage the audio output from the ADSP-BF537 processor

- **Attenuator** to manage volume control in software and move data from the input thread to the output thread. Right now, the volume control functionality is dormant because we have no means (yet) of getting volume control messages to the attenuator thread.

Note that three threads, one for each new type, are created after the stack is up by creating the threads in the `lwip_sysboot_threadtype_RunFunction()` function.
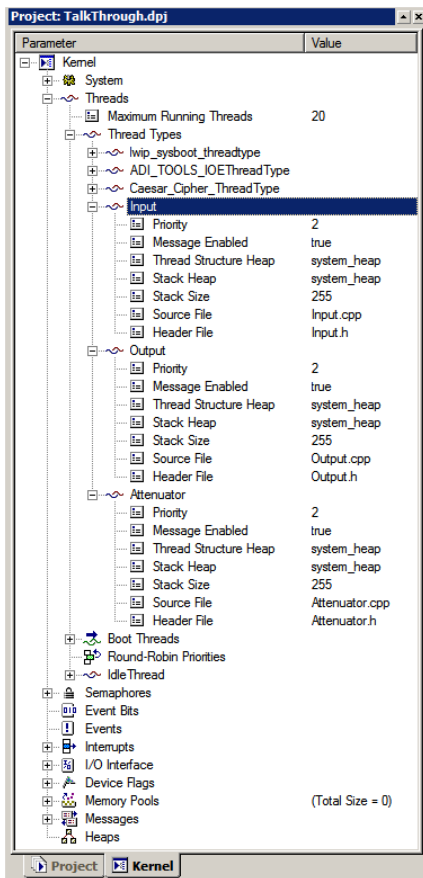
Figure 3-2. New Caesar Cipher Thread Types

Observe that the relative thread priorities are different between the audio thread types and Ethernet thread types, with the audio thread types being given a lower number (higher priority). Proper assignment of thread priority is a hallmark of a good application design and often is one of the most important system-level design decisions. In this case, it is more important

to service the audio stream, which receives around 24,000 samples (or data points) per second, than to service the Ethernet stack, which is operating on a "human" scale and perhaps a few keystrokes (data points) per second.

Furthermore, considerably more processing is done per data point within the Ethernet software. If the Ethernet stack is given a higher priority, the keystroke processing may take too much time, leaving the audio stream un-serviced for an appreciable amount of time. Ultimately, this can result in audible clicks in the final audio output whenever a key is pressed, degrading the user experience.

The software interface to the DAC (AD1854) and ADC (AD1871) converters is managed by device drivers written specifically for these devices (Figure 3-3).
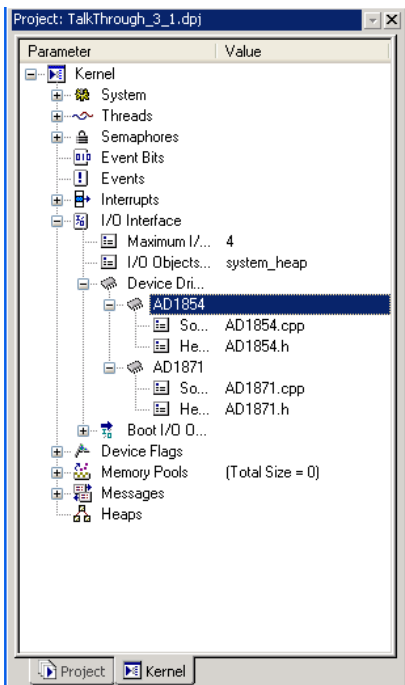

Figure 3-3. AD1854 and AD1871 Device Drivers

The source code for the drivers is part of your project and is available for examination in `AD1854.cpp` and `AD1871.cpp`. Luckily, the complexity of device intercommunication is abstracted by these device drivers, allowing interaction with the device without needing to understand its underlying complexity.

A *message* is a form of inter-thread communication that is available within VDK. Messages have a type (header) and an arbitrarily-sized *payload* (body), allowing data of any type and length to be shared between threads. The audio-processing threads use messages to communicate among themselves.

Part 2: Controlling Pass-Through Volume via Telnet

The example in its current state includes an audio pass-through and a Caesar Cipher operating concurrently but independently. The next step is to control the audio pass-through via the Ethernet connection. At this point, we remove the Caesar Cipher algorithm and replace it with control code that allow you to change the output volume of the application via Ethernet. Communications between the stack thread and the output thread are managed by a new message.

If you are unconcerned about implementation details, close the current project, load the project `<install_path>\Blackfin\Examples\ADSP-BF537 EZ-Kit Lite\Getting Started Examples\Part_3_2\TalkThrough_3_2.dpj`, and skip the procedure to upgrade the application with inter-thread messages.

(i) You must close the project from the previous exercise (part 1 of exercise 3) before opening the part 2 project.

To generate `VOL_CHANGE` messages:

1. Open the `Caesar_Cipher_ThreadType.cpp` file.

2. Replace the implementation of
   `Casesar_Cipher_ThreadType::Run()` with the contents of .

   Note how the + (plus) and – (minus) characters are used to change the `volume` variable in the range of `0.0` and `48.0` (inclusive), with zero being no volume (mute) and `48.0` being maximum volume. After changing the variable, its value is passed to the audio threads by posting a `VOL_CHANGE` message. Since the entire payload of the message is the volume change only (a four-byte `float`), the message has a size of four bytes, with the payload being the volume value itself. `VOL_CHANGE` is declared in `AudioMsg.h`, so the header file must be `#include`'d in `Caesar_Cipher_ThreadType.cpp`.

Build and run the project. When the audio output starts, telnet to the EZ-KIT Lite and use the + and – on your keyboard to change the volume of the output. Now you are running an interactive real-time application (audio pass-through) that communicates through a TCP/IP connection.

Now we use monitoring and analysis tools to understand the application's behavior. We already are familiar with the statistical profiler used in the earlier exercise. Using the statistical profiler, we will show that the application spends the majority of its time in the `VDK::IdleThread` functions. The `Idle Thread` is where a VDK application goes when all the processing is complete. It represents the amount of spare capacity available for other work to be done in the system.

Halt the EZ-KIT Lite with **Debug**–>**Halt**. We can now examine the behavior of the system's thread with the **View**–>**VDK Windows**–>**History** menu item. The **VDK State History** window displays the recent history of the application. Resize the window to a comfortable size. Turn on the legend by right-clicking in the window and selecting **Legend**. The window

looks similar to the window in Figure 3-4. The X axis represents time, and the Y axis represents system threads, one row for a thread. You recognize some of the threads created earlier in this exercise by name, as well as a `kCaesar_Cipher_ThreadType` if you halt the EZ-KIT Lite while connecting via telnet. Other threads (`kADI_TOOLS_IOEThreadType`) are unfamiliar to you. These threads are maintained by the LwIP Ethernet library.
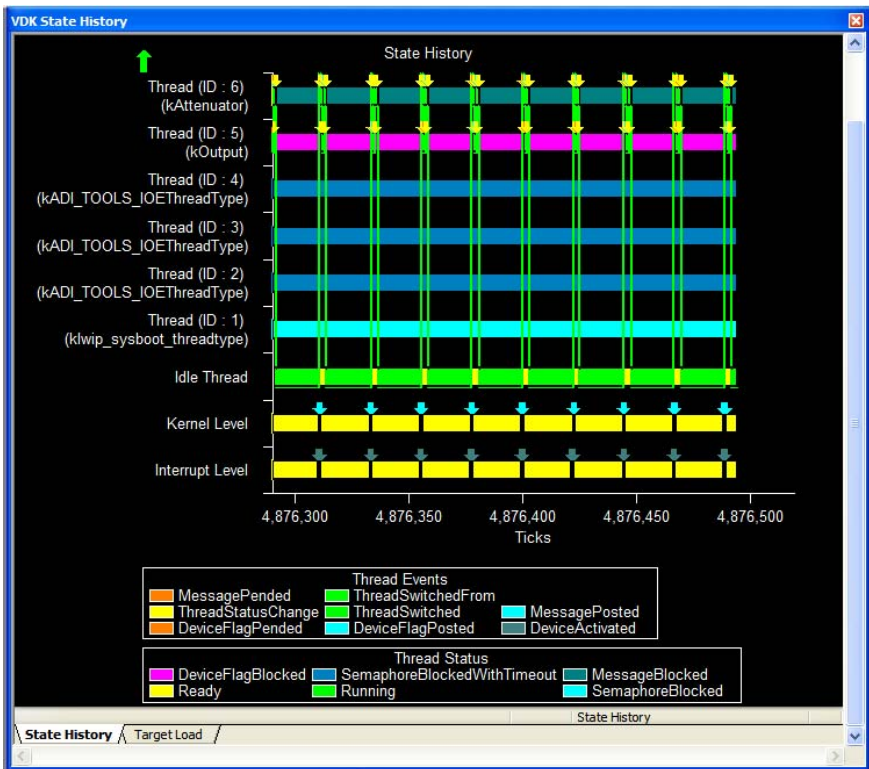

Figure 3-4. VDK State History Window of Talk-Through Activity

The graph in Figure 3-4 visualizes the statistical profiling data. Green vertical lines indicate a change between two threads; the thread running at any given time is also green. Thus, we can see that the `Idle Thread` gets the most activity, with intermittent activity among the three audio

threads. To zoom in on any particular time slice, move the mouse to the edge of the range you want to zoom in on, then click and drag the mouse to the other edge of the range.

Figure 3-5. Audio Thread Interaction Detail

Now, zoom in on one of the bursts of audio thread activity to visualize the interaction between the audio threads (a relevant portion shown in Figure 3-5). In the figure, you can see a regular pattern in the dataflow from the `Input` thread, through to the `Attenuator` thread, to the `Output` thread, and back to the `Attenuator`. The colored arrows represent various VDK events (refer to the legend for details).

It is also interesting to look at the interaction between the `Caesar Cipher` thread and the audio threads when volume change messages are processed. It is highly unlikely that you can halt the processor in the midst of this activity (which occurs once or twice a second). Instead, we will set a breakpoint at a key location in the program. Open the `Caesar_Cipher_ThreadType.cpp` file and place a breakpoint at the instruction

```
if ( 0 >= send (m_iSocket, m_vOutBuf, strlen (m_vOutBuf), 0) )
```

at the end of the `Caesar_Cipher_ThreadType::Run()` member function, around line 72. You can set a breakpoint by positioning the cursor on the desired line and pressing the **F9** key. (Note the red circle placed in the gutter bar of the editor window).

To reset your program, choose the **Debug–>Reset** menu item, then **File–>Reload Program**. If you still are connecting via telnet, terminate the session by pressing **Ctrl+]** and typing `quit` in the DOS window. Run the program.

After halting this application (any application), you may find it impossible to run the program again and pick up where you left off. Typically, this is because various interrupts in the system went un-serviced when you halted, placing the application into an unknown state. When in doubt, reset and reload.

As done previously, wait for the assigned IP address to appear in the **Output** window and then telnet to that address. You get the welcome message. Then press the + key. VisualDSP++ quickly halts at the just-set breakpoint (the audio pass-through, of course, also stops). Going back to the **VDK State History** window, notice an interesting flurry of activity on the right

side of the display, worthy of some discussion. Zoom in on the activity detail, which looks similar to that in Figure 3-6. Information in the window, reading left-to-right, can be interpreted as:
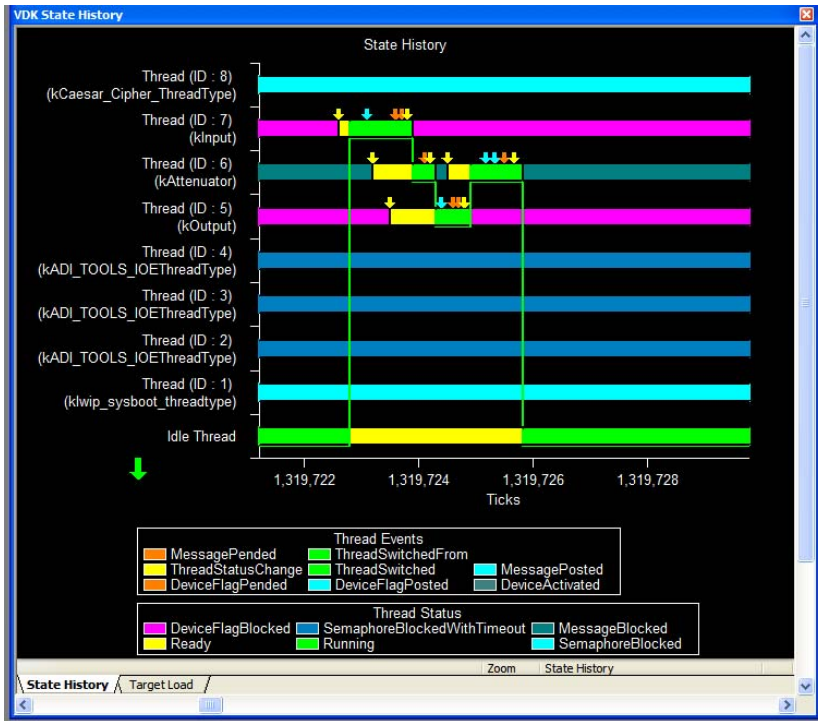

Figure 3-6. Volume Change Thread Interaction

1. The `LwIP stack` (thread ID 4, then 2) receives the keystroke from the Ethernet peripheral.

2. The stack's processing is interrupted temporarily by the processing of an audio block (threads ID 5–7). This interruption occurs because the audio processing thread has a higher priority than the Ethernet threads. After servicing the audio, execution resumes in the Ethernet thread.

3. The LwIP stack services the keystroke by passing it to our `Caesar Cipher` thread (thread ID 8). Control moves between thread IDs 8 and 2 to get the keystroke to the worker thread.

4. The `Caesar Cipher` thread posts a message (`VOL_CHANGE`) to the `Attenuator` (thread ID 6), where it is serviced.

5. In the meantime, another block of audio has become ready to processor. Rather than returning to the (lower priority) Caesar Cipher thread, control moves to the `Input` thread, and the block of audio is processed much like it was in step 2.

6. When the audio block is finished, control returns to the lower priority `Caesar Cipher` thread. Our breakpoint was set just after this point.

Part 3: Tuning Application

Although we have plenty of available spare capacity in the application as is, it is worth revisiting optimization techniques introduced in the earlier chapters. The three "easy" ways to performance-tune this application are:

- Turn on the compiler's optimizations

- Enable the ADSP-BF537 processor's instruction cache

- Increase the speed at which the processor runs

Click the **Target Load** page of the **VDK State History** window. Observe the percentage load over time. In its current non-optimized state, average load is around 25%, sometimes peaking to 100% for brief periods of time when processing Ethernet packets.

First, turn on the compiler's optimizations. Switch from the Debug configuration to the Release configuration, then build, load, and run the application. After a few seconds of audio, halt and revisit the **Target Load**

page. Note that utilization is modestly less, around 20%. The improvement was not considerable since the processor spent only a minority of its time in our code, so there was little gain to be made.

Finally, we will increase the speed of the ADSP-BF537 processor. By default, the EZ-KIT Lite operates at around 270 MHz. We'll use the custom start-up wizard again to increase the voltage supplied to the part (and, thus, the clock rate of the processor):

1. Select **Project–>Project Options** and navigate to the **Startup Code Settings–>Processor Clock and Power** page.

2. Enable **Configure clock and power settings**. Leave **EZ-KIT** as `ADSP-BF537 EZ-KIT`. Set **Clock and power settings** to **Optimize for speed**. Click **OK**.

3. Because we need to make VDK aware of the change, from the **Kernel** page of the **Project** window, navigate to **Kernel–>System->Clock Frequency**. Change the clock frequency value to `600` (MHz).

Build, run, listen, and halt as before. You can see a modest improvement in performance.

If you still have the **Statistical Profiling** window open, you may have noticed that `VDK::HistoryBuffer::AddEntry()` function consumes upward of 5% of the processor's capacity (which is a significant fraction of the processing our application is using). The function is the history-gathering routine within VDK. We can eliminate the impact of the function by going to the **Kernel** page, navigating to **Kernel–>System**, and setting **Instrumentation** to **None**. Note, however, that you no longer can use the VDK state history displays introduced earlier in this exercise.

Part 4: Running an Application from Flash Memory

The ADSP-BF537 EZ-KIT Lite is able to load your application to the flash memory using the Flash Programmer utility. The utility runs the program directly from the processor's memory without using the VisualDSP++ tool suite. In part 4 of exercise 3, the TalkThrough example will be placed into and run from the flash memory.

Before placing the application into the flash memory, create a loader (.ldr) file. Loader files are boot-loadable files created from processor executable (.dxe) files. The **Project Options** dialog box is used to specify the loader file settings.

To create a loader file for the TalkThrough example:

1. Open the project file <install_path>\Blackfin\Examples\ADSP-BF537 EZ-Kit Lite\Getting Started Examples\Part_3_2\TalkThrough_3_2.dpj.

2. From the **Project** menu, select **Project Options**.

    - In the **Project** tree control, select the **General** node. On the **Project** page, select **Loader file** as the project target type.

    - In the **Project** tree control, select the **Load** node. On the **Load Options** page set:

    Boot Mode to **Flash/PROM**,
    Boot Format to **Intel hex**,
    Output Width to **16-bit**,
    Initialization file to
    <install_path>\Blackfin\ldr\INIT_CODE_BF537_EZ-KIT_LITE.dxe.

3. Click **OK** to save the changes as your project options.

4. Rebuild the project file.

Now the created loader file is ready to be placed into the flash memory.

To place the file using the Flash Programmer:

1. Select **Flash Programmer** from the **Tools** menu. The Flash Programmer dialog box (Figure 3-7) appears on the screen.
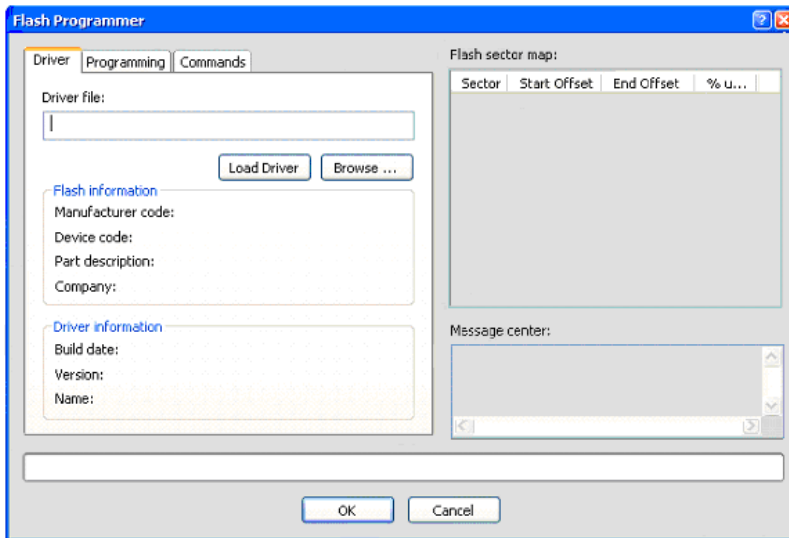

Figure 3-7. Flash Programmer Dialog Box – Driver Tab

2. On the **Driver** page, click **Browse** to locate the driver file
   `<install_path>\Blackfin\Examples\ADSP-BF537 EZ-Kit`
   `Lite\Flash Programmer\BF537EzFlashDriver.dxe`

3. Click **Load Driver** to load the driver file.
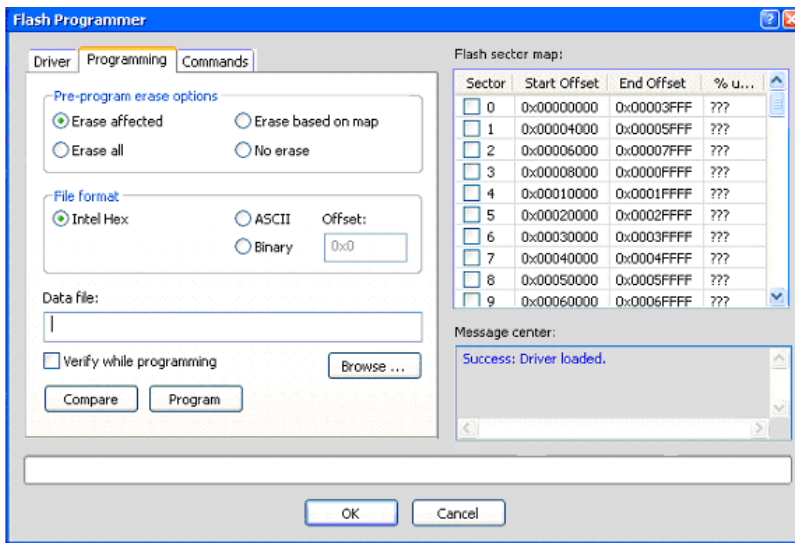
4. Click the **Programming** tab (Figure 3-8).

Figure 3-8. Flash Programmer Dialog Box – Programming Tab

5. Use **Browse** to locate the loader file …`\Blackfin\Exam-ples\ADSP-BF537 EZ-Kit Lite\Getting Started Examples\Part_3_2\Debug\TalkThrough_3_2.ldr`.

6. Click **Program** to load the program to the flash memory.

7. Click **OK** to exit the Flash Programmer.

When the application is loaded to the flash memory, it can be run from the flash without VisualDSP++. To run your application:

1. Close VisualDSP++.

2. Reset the EZ-Kit Lite by pushing the board's reset button.

   The two illuminated lights of the Ethernet jack indicate that the TalkThrough application is running.

   ⓘ You need to know the IP address of the board to run the application. If you use a DHCP router, the IP address is most likely the one that was used in previous examples in this tutorial.

You can change the volume of the audio using a TCPIP connection via telnet when the application is running. When the audio starts, telnet to the EZ-Kit Lite and use the + and – keys on the keyboard to change the volume.

Listing 3-1.
Caesar_Cipher_ThreadType::Run() New
Implementation

```
void
Caesar_Cipher_ThreadType::Run()
{
  static char *pszWelcome = "Welcome to Blackfin. Press \"+\" and
\"-\" to change volume level.\xa\xd";
  if ( 0 >= send ( m_iSocket, pszWelcome, strlen ( pszWelcome ),
0 ))
    return;

    float volume = 48.0;
    VDK::MessageID msg = VDK::CreateMessage ( VOL_CHANGE, volume,
NULL );
    while (1)
    {
      int iCount;

      if ( ( iCount = recv ( m_iSocket, m_vInBuf, sizeof (
m_vInBuf ) / sizeof ( char ), 0 ) ) >= 1 )
      {
        int iCharNum;
        char c;
        for ( iCharNum = 0; iCharNum < iCount; ++iCharNum )
        {
          c = m_vInBuf [ iCharNum ];

          if ( c == '+' || c == '-' )
          {
            if ( c == '+' )
              volume += 1.0;
            else
              volume -= 1.0;

              /* Volume must in a range from 0.0 to 48.0 */
```

Listing 3-1. Caesar_Cipher_ThreadType::Run() New Implementation

```
            if ( volume < 0.0 )
                volume = 0.0;
            else if ( volume > 48.0 )
                volume = 48.0;
        /*
         * Send a message to the Attenuator thread.
         * Note that zero is maximum volume and 48.0 is mute,
         * the opposite of "expected" behavior, so we
         * inverse the argument here.
         */

            extern VDK::ThreadID g_AttenuatorID;
            float message_args[] = { 48.0 - volume };

            VDK::SetMessagePayload ( msg, VOL_CHANGE,
sizeof(message_args)/sizeof(float), message_args );
            VDK::PostMessage ( g_AttenuatorID, msg,
CONTROL_CHANNEL );
            msg = VDK::PendMessage ( RETURN_BUFF_CHANNEL, 0 );

            sprintf ( m_vOutBuf, "\xa\xdVolume is now %g of
%g.\xa\xd", volume, 48.0 );

            if ( send ( m_iSocket, m_vOutBuf, strlen ( m_vOutBuf
), 0 ) <= 0 )
                break;
            }
        }
    }
    else
    {
        break;
    }
}

    close ( m_iSocket );
}
```

What's Next?

Analog Devices believes that you have found this *Getting Started with ADSP-BF537 EZ-KIT Lite* a helpful introduction to your ADSP-BF537 EZ-KIT Lite and the VisualDSP++ development suite. The next steps are:

- For a more detailed look at other capabilities of your EZ-KIT Lite and example code for its other peripherals (for example, the UART and CAN bus interfaces), look in the …\Blackfin\Examples\ADSP-BF537 EZ-KIT Lite subdirectory of the VisualDSP++ installation directory.

- The exercises in this book focus on the EZ-KIT Lite. Another set of exercises are available to set the focus on VisualDSP++, including some of the unique features of the Blackfin simulators. The *VisualDSP++ Getting Started Guide* is available in the online Help under the **Getting Started** folder. A PDF version of the book also is available in the Docs directory on the VisualDSP++ installation CD, or on the Analog Devices Web site.

- Users of the LabVIEW application from National Instruments can install a separate *Analog Devices Blackfin Test Integration Toolkit for LabVIEW* included with the EZ-KIT Lite package. It acts as a bridge between LabVIEW and VisualDSP++, allowing the LabVIEW software to simulate and otherwise drive VisualDSP++. The *Analog Devices Blackfin Test Integration Toolkit for LabVIEW* includes an example application.

- Visit Analog Devices Web site, www.analog.com, regularly for additional information and education materials, including example applications, technical publications, and live Web seminars.

Happy coding.

What's Next?

A   CREATING A TCP/IP APPLICATION

This appendix is an overview of the theory behind the TCP/IP framework provided by VisualDSP++. Examples 2 and 3 (on page 2-1 and page 3-1, respectively) of this book build upon the information presented in the following sections.

You will learn about the following concepts:

- TCP/IP framework

- VisualDSP++ Kernel (VDK) operating system

- Sockets API

TCP/IP Framework

The Ethernet Media Access Control (MAC) device of the ADSP-BF537 processor provides a 10/100-Mbits/s Ethernet interface between the Media-Independent Interface (MII) and the processor's peripheral system. You can create an embedded program, taking advantage of the Ethernet MAC and using the TCP/IP stack support provided by VisualDSP++.

TCP/IP is a communication protocol that allows applications to transfer data regardless of their underlying communication media. To implement a TCP/IP application, several different elements are required. For an EZ-KIT Lite system, the TCP/IP application framework consists of:

Application. This is the actual end-user application program. For the VisualDSP++ model, this includes using the VisualDSP++ Kernel as an operating system and BSD Socket API as a means to program communications with Transport and Network layers of TCP/IP.

Data Transport and Network libraries. In the VisualDSP++ model, two libraries allow communication and data transfer between the host and client applications. TCP/IP libraries are used for data transporting, and the `LwipBF537` libraries (or stack libraries) are used for network communications.

Network Connection. The network connection includes hardware and software components. The ADSP-BF537 EZ-KIT Lite is the hardware component that provides the physical connection between the processor and the Local Area Network (LAN). The software component is a set of libraries and device drivers to provide the communications between the application and hardware.

TCP/IP Communications

TCP/IP communication occurs between a client application and a host application. The host listens for a connection from the client application. For each application, the following steps are taken to set up the stack and initiate inter-process communications.

1. Create a stack. Each application must initiate and start its own stack. The stack is the conduit through which all communications pass.

2. Create an IP address. In order for the host and client to communicate, each application must to have a unique identifier (an IP address). A port address is assigned also to the application and is relative to the IP address.

3. Establish a host–client link. The host application acknowledges that it has been contacted by the client.

4. Transfer data. Data can be sent and received by the host and client.

5. Close the connection. When the host and client communication completes, the host and client must close the connection with each other.

VisualDSP++ Kernel (VDK) Overview

To program a TCP/IP application, a Real Time Operating system (RTOS or kernel), is needed. VisualDSP++ provides an RTOS called VDK. VDK is a preemptive multitasking kernel that incorporates scheduling and resource allocation techniques tailored for the memory and time constraints of programming with Analog Devices processors, such as the ADSP-BF537 processor. The kernel facilitates development of fast-performance structured applications using frameworks of template files.

VDK allows more efficient use of hardware by enabling improved scheduling of work, as well as a development framework containing implementations of common synchronization and scheduling paradigms. For more information, see the *VisualDSP++ Kernel (VDK) User's Guide*.

BSD Socket API

The TCP/IP stack uses the standard Berkeley Sockets Interface (or *sockets*) to provide an interface to the application. This generic interface may be used with any inter-process communication protocol. The sockets API is included with VisualDSP++ for use in your TCP/IP stack application.

As mentioned earlier, in the network realm, there are host and client applications. The difference between these applications is that the host "listens" for connections and the client "contacts" the host. Both applications send and receive information using socket API calls once a connection is established.

The basic flow of control for a typical host–client connection is as follows:

1. Applications set up sockets

2. The host waits for a client to contact it

3. The client connects to the host application

4. The host acknowledges the client contact

5. Data flow between the server and client begins

6. The server and client connections are closed

1. Create Sockets

To create a socket, first instantiate the socket and then give the socket a unique IP address. The "socket" command is used to instantiate the socket. Next, the "bind" command is used to specify the type and the addressing format of the socket. The IP address contains the actual address assigned by the network, as well as a port address—the location on the actual machine where the application is running. The server and client applications must create their own sockets.

2. Host Waits for Client

A host application is placed in listening mode, which lets other applications know that the host is available for inter-process communications. The API command for this is the "listen" command.

3. Client Connects with Host

A client application can connect to a host application using the "connect" command. The "connect" command specifies the IP address of the host application.

4. Host Acknowledges Client

When the host application gets a connection attempt from the client application, it accepts the incoming call and obtains the socket address from the client. At this point, the handshaking between the client and host is established and data transfer begins.

5. Inter-Process Communications

The host and client applications are free to send and receive data using the socket API calls. For more information on socket APIs, refer to Exercise 2 on page 2-2.

6. Close Connection

When the applications complete the data exchange, the sockets must be closed. The client application closes the socket it has created. The host, on the other hand, close its own socket and the socket created by the connect call.

Figure A-1 illustrates the flow of control for the host and client applications.

In both applications, a TCP/IP stack must be created before any socket communications can take place. The TCP/IP libraries provide calls to create and start the stack.

```
    HOST APPLICATION              CLIENT APPLICATION

┌──────────────────────┐      ┌──────────────────────┐
│  INITIALIZE SOCKET AND│      │   INITIALIZE SOCKET   │
│ LISTEN FOR CONNECTIONS│      │                       │
└──────────────────────┘      └──────────────────────┘
           │                              │
           │                   ┌──────────────────────┐
           │◄──────────────────│   CONNECT TO HOST     │
           │                   └──────────────────────┘
┌──────────────────────┐
│ ESTABLISH CONNECTION  │
│      BETWEEN          │
│   CLIENT AND HOST     │
└──────────────────────┘
           │
┌──────────────────────┐
│   HOST AND CLIENT     │
│   EXCHANGE DATA       │
│      PACKETS          │
└──────────────────────┘
           │
┌──────────────────────┐
│   HOST AND CLIENT     │
│   CLOSE SOCKETS       │
└──────────────────────┘
```
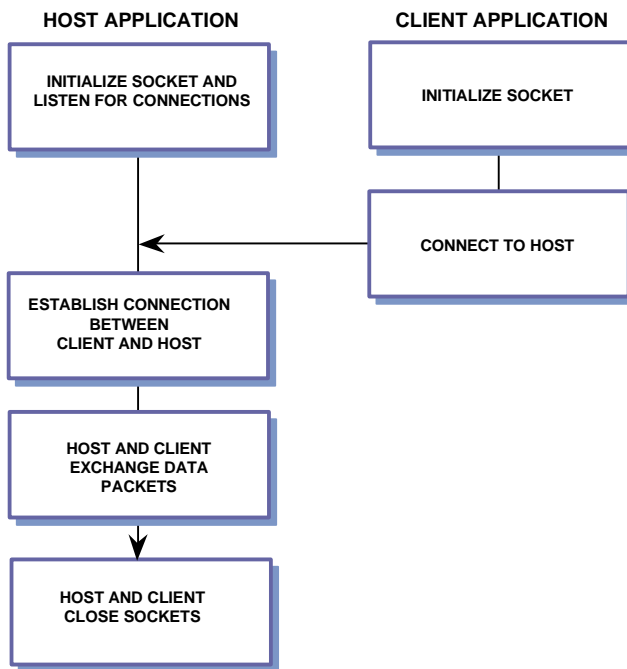
Figure A-1. Host and Client Communication Control Flow

I INDEX

A

AD1854 digital-to-analog converters (DACs), 3-2, 3-6

AD1871 analog-to-digital converters (ADCs), 3-2, 3-6

ADSP-BF537 processors
  built-in cycle counters, 1-12
  clock frequency, 1-17
  internal memory (SRAM), 1-14, 1-16
  peripheral systems, A-1
  voltage regulators, 1-17

analog audio, *See* audio

audio
  applications, 3-2, 3-4
  converters, 3-2, 3-6
  interface, xii
  playback, 3-3
  sources, 3-2, 3-4
  thread types, 3-5, 3-8

AUDIO_OUT connector, 3-3

B

Berkley sockets, 2-2, A-3

bind command, A-4

Blackfin processor simulators
  cycle-accurate interpreted, 1-4
  functional compiled, 1-4

bubble sort algorithm, 1-7, 1-16

build options, 1-9

Build (project) command, 1-7, 2-5

built-in cycle counters, 1-12

C

cache, configuring, 1-16

Cache and Memory Protection dialog box, 1-15, 1-16

Caesar Cipher algorithm, 2-1, 2-6, 3-2, 3-4

CAN
  interface, xi
  transceiver devices, xi

client applications, A-2

client/host connections, closing, A-3, A-5

clock frequency, 1-17, 3-14

Close (project file) command, 1-11

Command Prompt window, 2-5, 2-9

compiled simulators, 1-4

compiler optimizations, 1-10

configurations
  debug, 1-8, 1-10
  release, 1-10

connect command, A-4

connector, LINE IN (mini-headphone), 3-2

Console page, 1-13, 2-5

customer support, xiv

custom hardware, 1-4

cycle-accurate simulators, 1-4

cycle counts, 1-18

D

data
  points, 3-6
  transfers, A-5
  transport libraries, A-2

INDEX

DB9 (UART) connector, xii
debug, project configuration, 1-8, 1-10
Debug (Run) command, 1-13
debug session types
    emulator, 1-5
    EZ-KIT Lite, 1-4
    simulator, 1-4
Debug windows (View) menu, 1-10
DHCP servers, 2-1, 2-3, 2-4
DOS applications, 2-5

E

Educational Laboratory Virtual
    Instrumentation Suite interface, *See* ELVIS
ELVIS interface, xi
emulators, 1-5
Enable instruction cache command, 1-15
encryption algorithms, 2-6
Ethernet
    interface, xi
    MACs, 2-2, A-1
    networks, 2-1
    stacks, 3-6
    thread types, 3-5
external memory (SDRAM), 1-11, 1-14, 1-16
EZ-KIT Lite
    audio sources/destinations, 3-2
    connecting to networks, 2-3
    Ethernet MACs, 2-2

F

features, of this EZ-KIT Lite, x
functional compiled simulators, 1-4

H

handshakes between client/host, A-5
headers, of VDK messages, 3-7
host applications, A-2

host/client interactions, 2-7, A-2, A-4

I

instruction cache memory, 1-15
interfaces, *See* audio, CAN, ELVIS, Ethernet,
    expansion, SDRAM
internal memory, 1-16
inter-process communications, A-5
IP address, 2-1, 2-4, 2-5, 2-9, A-2, A-4

J

JTAG emulators, 1-5

K

Kernel (VDK), 2-1, 2-2

L

L1 internal memory, 1-16
LabVIEW virtual instruments, -xi
LED USB_MONITOR, 1-2
LINE IN connector, 3-2
listen command, A-4
local area networks (LANs), A-2
LwIP
    stacks, 2-1, 2-2
    wizard, 2-4

M

MAC address, xi, 2-3
main() function, 1-18
Media Access Controller, *See* MAC
Media-Independent Interface (MII), A-1
memory placements, 1-11
message
    payload (body), 3-7, 3-8
    type (header), 3-7
mini-headphones, 3-2

U

universal asynchronous receiver transmitter, *See* UART
USB jack, 1-2
USB_MONITOR LED, 1-2

V

VDK State History window, 3-13

VisualDSP++
debug sessions, 1-4
distribution CD, 1-7
Kernel (VDK), 2-1, A-3
main window, 1-3
program code placement, 1-17
project configurations, 1-9
simulators, 1-4
TCP/IP software stacks, 2-2
voltage levels, 1-18