Getting Started with ADSP-BF548 EZ-KIT Lite®

Revision 1.0, November 2007

Part Number
82-000206-02

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

ANALOG DEVICES

Copyright Information

Printed in the USA.

Limited Warranty

The EZ-KIT Lite evaluation system is warranted against defects in materials and workmanship for a period of one year from the date of purchase from Analog Devices or from an authorized dealer.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

The Analog Devices icon bar and logo, VisualDSP++, the VisualDSP++ logo, Blackfin, the CROSSCORE logo, EZ-KIT Lite, and EZ-Extender are registered trademarks of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.

Regulatory Compliance

The ADSP-BF548 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-BF548 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-BF548 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: Z600ANA1.029

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

PREFACE

Thank you for purchasing the ADSP-BF548 EZ-KIT Lite®, Analog Devices, Inc. evaluation system for ADSP-BF548 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-BF548 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-BF548 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-BF548 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-BF548 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-BF548 EZ-KIT Lite provides example programs to demonstrate the capabilities of the evaluation board.

The ADSP-BF548 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, the linker restricts a user's program to 60 KB of memory for code space with no restrictions for data space

The board features:

- Analog Devices ADSP-BF548 Blackfin processor
  - Core performance up to 600 MHz
  - External bus performance up to 133 MHz
  - 400-pin mini-BGA package
  - 25 MHz crystal
- Double data rate (DDR) synchronous dynamic random access memory (SDRAM)
  - Micron MT46V32M16 – 64 MB (8M x 16-bits x 4 banks)
- Burst flash memory
  - Intel PC28F128K3C115 – 32 MB (16M x 16-bits)
- NAND flash memory
  - ST Micro NAND02 – 2 Gb

- SPI flash memory
  - ST Micro M25P16 – 16 Mb
- Advanced technology attachment packet interface (ATAPI)
  - Toshiba 2.5” MK4032GAX – 40 GB HDD
- Analog audio interface
  - Analog Devices AD1980 SoundMAX codec
  - 6 DAC channels for 5.1 surround
  - 1 input stereo `MIC` jack
  - 1 input stereo `LINE IN` jack
  - 1 output stereo `LINE OUT`/`HEAD PHONE OUT` jack
  - 1 output stereo `SURROUND` jack
  - 1 output center and `LFE` jack
- TFT LCD display with touchscreen
  - Sharp LQ043T1DG01 – 480 x 272, 4.3” touchscreen LCD
  - Analog Devices AD7877 – touchscreen controller
- Ethernet interface
  - SMSC LAN9218 device
  - 10-BaseT and 100-BaseTX Ethernet controller
  - Integrated PHY and MAC
  - HP Auto-MDIX
- Keypad
  - ACT components– 4 x 4 keypad assembly

- Thumbwheel
  - CTS Corp rotary encoder
- 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 ADSP-BF548 processor signals
- Other features
  - JTAG ICE 14-pin header
  - USB OTG connector
  - `HOST` interface connector
  - Blackfin power measurement jumpers
  - `PPI1` IDC connector
  - `SPORT2` and `SPORT3` IDC connectors
  - TWI, SPI, timers, `UART3` IDC connectors

For information about the hardware components of the EZ-KIT Lite, refer to the *ADSP-BF548 EZ-KIT Lite Evaluation System Manual*.

Purpose of This Manual

The *Getting Started with ADSP-BF548 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-BF548 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:

- Chapter 1, "Programming ADSP-BF548 EZ-KIT Lite with VisualDSP++" on page 1-1
  Provides instructions for connecting the EZ-KIT Lite to VisualDSP++ running on your PC. The established connection allows you to download the example program (along with your own) to

the board and control the program execution. You will build and run the first example project using the VisualDSP++ debug and data display tools.

- Chapter 2, "Using ADSP-BF548 EZ-KIT Lite to Investigate Performance Factors" on page 2-1
  Explores the memory hierarchy of the EZ-KIT Lite and measures the ramifications of memory placement decisions. Demonstrates the Blackfin processor core's built-in voltage regulator and its ability to throttle the processor's power consumption and clock rate at run-time. The system services library (SSL), your API to the Blackfin processor, also is introduced in this chapter.

- Chapter 3, "Using ADSP-BF548 EZ-KIT Lite Peripherals" on page 3-1
  Explores the peripheral interfaces and devices available on the EZ-KIT Lite. Introduces the device driver software supplied with VisualDSP++. You will use one of the supplied device drivers to play an audio clip through the AD1980 audio codec on the EZ-KIT Lite.

- Chapter 4,"Using ADSP-BF548 EZ-KIT Lite As A Mass Storage Device" on page 4-1
  Examines the USB and hard disk functionality. You will connect the EZ-KIT Lite to your PC and have Windows use it as a removable mass storage device.

- Chapter 5, "Using ADSP-BF548 EZ-KIT Lite Hard Disk and LCD Screen" on page 5-1
  Introduces the LCD panel driver of the SSL. You will copy a bitmap file to the EZ-KIT Lite hard disk and display the file on the LCD panel.

- Chapter 6, "Using ADSP-BF548 EZ-KIT Lite Keypad and LED Indicator" on page 6-1
  Explores the keypad interface of the EZ-KIT Lite and introduces the VisualDSP++ real-time kernel (VDK). You will use the LEDs to communicate the application status.
- Chapter 7, "Creating A Bootable Application" on page 7-1
  Finally, you will construct an application, drawing from the previous chapters in the manual. A working bitmap photo viewer application will be created, including hard drive access, USB connectivity, push button navigation, and audio sample triggering. The application will be burned to the on-board flash devices. This will allow the application to run while not connected to VisualDSP++.

What's New in This Manual

This is the first revision of the *Getting Started with ADSP-BF548 EZ-KIT Lite*.

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-BF548 EZ-KIT Lite evaluation system supports Analog Devices ADSP-BF548 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-BF542/BF544/BF548/BF549 Embedded Processor Data Sheet	General functional description, pinout, and timing
ADSP-BF548 Blackfin Processor Hardware Reference	Description of internal processor architecture and all register functions

Table 2. Related VisualDSP++ Publications

Title	Description
ADSP-BF548 EZ-KIT Lite Evaluation System Manual	Description of the ADSP-BF548 EZ-KIT Lite hardware and software components
VisualDSP++ User's Guide	Description of VisualDSP++ features and usage
VisualDSP++ Assembler and Preprocessor Manual	Description of the assembler function and commands

Table 2. Related VisualDSP++ Publications (Cont'd)

Title	Description
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

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-BF548 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-BF548 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 WinZip, 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.

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

1 PROGRAMMING ADSP-BF548 EZ-KIT LITE WITH VISUALDSP++

In this chapter, you will connect your personal computer (PC) to the ADSP-BF548 EZ-KIT Lite evaluation system and write a simple C language program to perform two sorting algorithms.

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

- VisualDSP++ sessions and target types
- Plot windows
- Project configurations

The chapter includes the following sections.

Installing VisualDSP++ and the EZ-KIT Lite

If you have not already done so, install the ADSP-BF548 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.

There are two USB interfaces on the ADSP-BF548 EZ-KIT Lite. Be sure to use the debugger's interface (labelled `USB Debug Agent`) when connecting your computer to the board with provided USB cable. The other USB interface (labelled `USB-OTG`) is for applications use.

Once the installation is complete, the amber LED (labeled `MON`, 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++ Installation Quick Reference Card*, step 4.

Starting VisualDSP++ and Connecting to the EZ-KIT Lite

If you have not already done so, start VisualDSP++. Use the Windows **Start** menu to launch the VisualDSP++ environment (also called the Integrated Development and Debugging Environment or IDDE).

When VisualDSP++ first launches, it is disconnected from your EZ-KIT Lite or any other kind of debugging *session* (Figure 1-1).

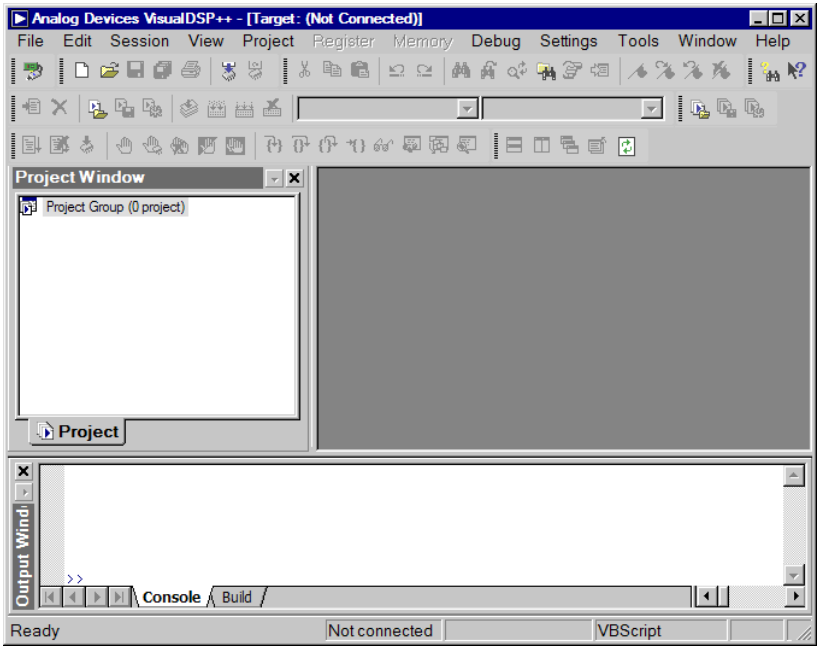

Figure 1-1. VisualDSP++ Main Window

A VisualDSP++ debugging session consists of a session type and target processor. There are three basic session types:

- **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 sacrifices 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.
- **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.

To inform VisualDSP++ that your target is an ADSP-BF548 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).

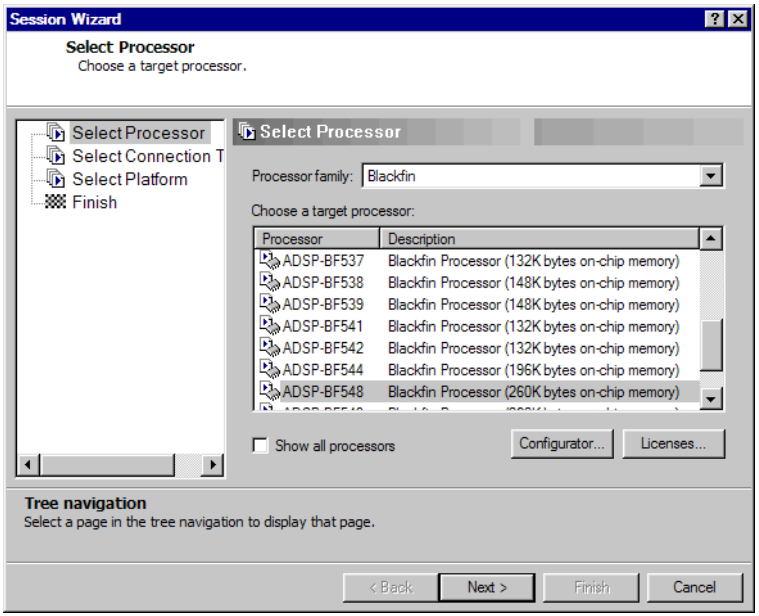

Figure 1-2. Session Wizard

2. On the **Select Processor** page, ensure that **Processor family** is set to **Blackfin**. In **Choose a target processor**, select **ADSP-BF548**. Click **Next**.

3. On the **Select Connection Type** page, select **EZ-KIT Lite**. Click **Next**.
4. On the **Select Platform** page, ensure that the selected platform is **ADSP-BF548 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-BF548 EZ-KIT Lite. Once connected, the main window's title is changed to include the session name set in step 4.

Creating a particular session is a one-time job. VisualDSP++ records a session's details between runs and, by default, tries to reconnect to the session that was in force when VisualDSP++ last ran. You can create more than one session, possibly changing the connection type to **Simulator** or **Emulator**. The **Session** menu allows you to swap between sessions as you wish.

Examples in this manual assume that you are using an ADSP-BF548 EZ-KIT Lite session.

Example 1: Building and Running an Application

Now it is time to start our first C program. "Example 1: Sorts.c File" on page 1-9 is the C program we start in this chapter and develop in the following one. 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 example 1 source code is included on the VisualDSP++ distribution CD. The source code and project files can be found in the `<install_path>\Blackfin\Examples\ADSP-BF548 EZ-Kit Lite\Getting Started Examples\Example_1` subdirectory. The default `<install_path>` is `C:\Program Files\Analog Devices\VisualDSP 5.0`.

Open example 1 by selecting **File–>Open–>Project**, browsing to the example's directory, and selecting the `Example_1.dpj` project file[1]. Once the project is opened, view its source code by double-clicking the **Sorts.c** label in the **Project** window (expand the **Source Files** tree control to see the label if necessary). Build the project 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 in the `main()` function, showing that the program is stopped there, awaiting your command to set it running or to single-step through the source.

To visualize the activities 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**.

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

Repeat this procedure to create a plot window for the `out_m` variable[1], modifying 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.

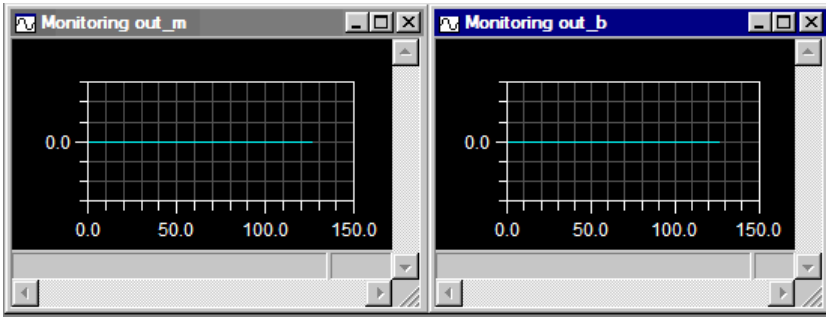

Figure 1-3. Plot Windows

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 statement 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 example 1 project uses the 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

[1] 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.

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 window, and others are available under the **View–>Debug Windows** menu selection.

Example 1: Sorts.c File

```
/*
 * Getting Started With the ADSP-BF548 EZ-KIT Lite
 * Example 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 )
   {
      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 );
      }
   }
```

2 USING ADSP-BF548 EZ-KIT LITE TO INVESTIGATE PERFORMANCE FACTORS

In this chapter, we will benchmark a program, examine memory types, and look at the effects of enabling a portion of fast internal memory as a cache for external memory. Finally we will study the processor performance in terms of clock speed and voltage trade-offs.

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

- Benchmarking code with a Blackfin processor's cycle counter and real-time clock
- Statistical profiling
- Blackfin processor's memory hierarchy, cache, and direct memory placement
- Blackfin processor's voltage regulator and the processor support library

The chapter includes the following sections.

ADSP-BF548 Processor Memory Hierarchy

The ADSP-BF548 processor supports a 'hierarchy' of three synchronous memories, where the term *synchronous* means that the memory operates in step with the edges of the clock signal on whichever processor bus is used to access the memory. Understanding the differences between the different memories is an important aspect of high-performance application development.

- **Internal L1 memory.** Consists of 196K bytes of SRAM within the processor, split into several different areas. L1 memory is the highest-performing memory available to the Blackfin core and can be accessed at core clock speeds. An application never stalls waiting for a memory read/write in L1 or for an instruction fetched from L1. Instructions (code) and data are held in separate areas of L1, and part of each area can be set aside and used as a cache for the lower level memories.

- **Internal L2 memory.** Consists of a single 128K byte area of SRAM within the processor. L2 is somewhat lower-performing than L1, requiring two core clock cycles for access. L2 also has longer latencies than L1. Instructions and data can co-exist in L2 memory.

- **External memory.** Sometimes referred to as L3, this is DDR SDRAM that exists external to the processor and can be found mounted on the EZ-KIT Lite board. 64M bytes of DDR SDRAM is supplied on the EZ-KIT Lite, but different sizes can be used on custom hardware to suit your application's specific needs, up to a maximum of 512M bytes. External memory operates synchronously with the processor's system clock rather than the core clock, causing access time to SDRAM to be relatively slower than to L1 or L2 memory. Similar to L2, external memory can hold instructions and data.

The ADSP-BF548 processor also supports *asynchronous* memories, the internal operations of which are not tied to either of the processor's main clocks. Typically, these are flash memory devices, of which the ADSP-BF548 processor supports a number of different types. The ADSP-BF548 EZ-KIT Lite is populated with these flash devices:

- **NOR.** The easiest to use but most costly form of flash memory. NOR can be read without any special handling in software. The final chapter of this tutorial, "Creating A Bootable Application" on page 7-1, shows how to build an application to burn to NOR flash. The application will be launched each time the EZ-KIT Lite is powered up or reset, without the need for the EZ-KIT Lite to connect to VisualDSP++.
- **NAND.** The largest and least expensive form of flash available, but prone to errors through bad blocks and wearing. Because of this, NAND flash is most often accessed as a file system, rather than as straight memory, through library calls such as `mkdir()` and `fopen()`. The EZ-KIT Lite NAND flash is not explored directly in this tutorial, but its usage (from a software point of view) is fundamentally identical to that of the hard disk explored in a later chapter.
- **SERIAL.** Some flash memories are designed to be accessed serially, receiving and supplying data one bit per clock cycle rather than in eight-bit or 16-bit units. Typically such memories are used for very low traffic tasks, such as storing configuration parameters that are read-only or modified infrequently. The EZ-KIT Lite has one such device connected to one of the SPI ports of the processor. This tutorial does not cover the serial flash device operations.

Note that access to the L1, L2, DDR SDRAM, and NOR flash memories can be made via the Blackfin processor's standard read and write instructions (as generated by the C/C++ compiler), whereas access to other memories generally requires special sequences of commands to be written to the processor's control registers.

Example 2: Benchmarking the Relative Performance of Memories

Now that you are familiar with basic VisualDSP++ operations, it is time to analyze and tweak a program's performance. Close the `Example_1.dpj` file using the **File–>Close–>Project** menu selection. Open the next project file (`Example_2.dpj`) located in the `<install_path>\Blackfin\Examples\ADSP-BF548 EZ-Kit Lite\Getting Started Examples\Example_1` subdirectory of your VisualDSP++ directory.

This project builds on the program discussed in the previous exercise. The same sequence of function calls (`randomize_arrays()`, `bubble_sort()`, and `quick_sort()`) now is enclosed in a loop that repeats the sequence sufficient number of times in order to profile the loop's execution. In addition, some code has been added to the program to obtain and print a rough measure of the loop's execution time in seconds, along with the number of clock cycles the loop required.

Looking at the new version of the C source file (`Sorts.c`) you see there are some additional `#include` directives at the top of the file. These are C declarations for using `printf()`, for starting and reading the processor's real-time clock (RTC), and for reading the cycle counter. The RTC is a straightforward device; the code in functions `start_real_time_clock()` and `get_real_time_clock_in_seconds()` shows how the clock's control registers and value are accessed from a C program by simple pointer de-referencing.

The control registers of all Blackfin peripherals are mapped into the standard address space and can be accessed in a similar manner; although, the set-up and control required for most of the other peripherals is more complicated than that for the RTC. In Chapter 3, "Using ADSP-BF548 EZ-KIT Lite Peripherals" on page 3-1, we will see that VisualDSP++ supplies libraries of service functions and device drivers to simplify and standardize the use of many processor peripherals (including the RTC)

and associated EZ-KIT Lite devices. Another library function (`clock()`) provides the current value of the processor's core clock cycle counter whenever the function is called.

Since we are examining performance in this example, we will build the project in the **Release** configuration, which invokes the compiler's optimizer. Ensure that the configuration drop-down list in the project toolbar is set to **Release** rather than **Debug**, then rebuild the project.

Before running the example program, open a **Statistical Profiler** window using the **Tools–>Statistical Profiling–>New Profile** menu selection. Position and resize the window as necessary. While a program is running, the Statistical Profiler repeatedly reads the processor's program counter value, associates the values with locations in the program, and displays a percentage of 'hits' for different parts of the program in the profiler window (Figure 2-1). The Statistical Profiler is an excellent tool for finding hotspots in a program's execution and assessing the performance effects of program changes.

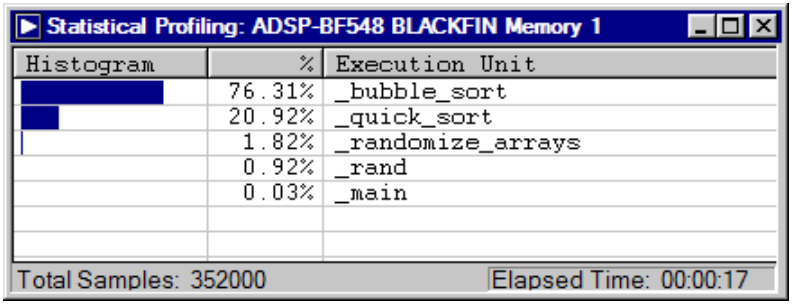

Figure 2-1. Statistical Profiler Window

Now run the program. While the program runs, the **Statistical Profiler** window is updated periodically to show the percentage of runtime being spent in each function. After 20 seconds or so, the program prints a message to the **Console** tab of the VisualDSP++ **Output** window (in green text) giving the approximate number of seconds the loop took to run and

the approximate number of millions of processor cycles executed (see Table 2-1). The first row in Table 2-1 shows typical values, along with the **Statistical Profiler**'s final estimates of the percentage of execution time spent in the `bubble_sort()` and `quick_sort()` functions. Your figures may differ slightly from those in Table 2-1 but still will show that, as expected, `bubble_sort()` is using most of the processor time.

Table 2-1. Typical Results from Example 2

Code and Data Disposition	Elapsed Seconds	Core Cycles	Bubble Sort %	Quick Sort %
Everything in external memory	19	9533	74	23
`bubble_sort()` in L2 memory	13	7079	66	30
`bubble_sort()` in L1 code memory	13	6866	64	32
Everything in external memory, plus code cache on	10	5367	87	11
All code in external memory, plus code cache on, plus data arrays in L1 data memory	1	612	68	25

By default, VisualDSP++ maps as much code and data as possible to internal L1 and L2 memories when building a program. However, this project's options contain a setting that maps all of the code and data from `Sorts.c` into external memory, and so the just obtained profiling results represent the worst-case performance for our application. In the following sections, we will modify the placement of some parts of the program and observe the effects.

One way to alter the default placement of an individual code function or data variable is to place a `section` directive in front of its definition in the C source file. This directive causes the VisualDSP++ compiler to place the item in the output section named in the directive. The available section names are defined in the linker description (`.ldf`) file that the project

employs. Since the profiler window shows that function `bubble_sort()` is using the largest percentage of processor time, we will move that function further up the memory hierarchy and examine the results.

There are two commented-out section directives in the middle of the `Sorts.c` file, immediately above the `bubble_sort()` definition. Uncomment the directive that places `bubble_sort()` in the section named `L2_sram` by removing the `//` characters, then rebuild and rerun the project. Because the bubble sort's instructions now are fetched from L2 memory rather than external memory, the processor spends less time waiting, resulting in a faster elapsed time and smaller cycle count. The profiler window also shows that `bubble_sort()` is not dominating the execution of the program as much as before.

Now replace the comment at the start of the `L2_sram` directive and uncomment the directive on the line above. This will result in the bubble sort's instructions being placed in the portion of L1 memory reserved for code. Rebuild and rerun the project. Your new results probably are not much different from the previous set. One reason for this is that the access time difference between L2 and L1 is rather less than the difference between external memory and L2. However, other factors are at work as we shall see in the next two steps.

All Blackfin processors have the option of using part of the L1 code and data memory areas as caches. Enabling the caches allows the second and subsequent accesses to external memory locations to happen at L1 speeds. We are going to examine this now. Ensure that both section directives at the start of `bubble_sort()` are commented out—all of the program's code will be placed in external memory. Next we are going 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**.

3. In the **Instruction cache memory** drop-down list, select **Enable instruction cache**.
4. Click **OK** to set the new project option. VisualDSP++ will regenerate the necessary files to enable the instruction cache when the program runs.
5. Rebuild the project by selecting **Project–>Rebuild Project** or by right-clicking the project name **Example_2** in the **Project** window and selecting **Rebuild Project**.

Rerunning the example shows a further reduction in execution time and core cycles because now both the bubble sort and quick sort are benefiting from L1 code memory via the instruction cache.

Overlooking data placement can be a mistake in application development. Accordingly, as a final step in this exercise, we will observe the effect of placing the two data arrays, `out_b` and `out_m`, in L1 memory. Find the array definitions in the program's source file (towards the end, just above the `main()` definition) and uncomment the section directives. This will place the arrays in the `L1_data_a` section of L1 data memory. Rebuild and rerun the project, then marvel at the overall program performance improvement: from 19 seconds down to 1 second or so. Excellent result!

Example 3: Using the Blackfin Processor Voltage Regulator

Of course, speed is not everything in a high-performance application. Sometimes the power required to accomplish a task is more important than the time taken, particularly for mobile devices. Blackfin processors feature an internal voltage regulator that controls the voltage level at which the core operates. The supported levels and the corresponding maximum core and system clock rates for each level are documented in the datasheet for each processor.

The voltage level can be changed on the fly as a program runs; for example, to suit the current workload or battery state. Changing the voltage regulator setting is not difficult but becomes a more complicated task with respect to coordinating the change with any necessary adjustments of the core and system clock rates. Fortunately, the system services library (SSL) that Analog Devices supplies as part of VisualDSP++ contains a power management module that includes many really useful functions. One SSL function sets a particular voltage level **and** changes the clocks to the maximum speed supported at the new voltage. Our next example will use this function in addition to another SSL function that reports back the current clock rates. VisualDSP++ online help contains a comprehensive description of the SSL.

Close any open projects. Locate the next project file (`Example_3.dpj`) in the `Example_3` directory. Double-click the project file name or drag it to the **Project Window** of VisualDSP++. Expand the **Source Files** and **Header Files** nodes to investigate the file contents. In addition to the example application program (`Sorts.c`), there is a new C source file (`adi_ssl_Init.c`) and corresponding header (`adi_ssl_Init.h`). As with most libraries, the SSL requires initialization before any of the library facilities are used. Initialization takes the form of instructing the various SSL managers about the hardware platform (such as the SDRAM timings for the external bus manager) and supplying them with sufficient memory to service the interrupts and peripherals that we wish to employ.

Most of the VisualDSP++ example programs using the SSL services follow a similar pattern, so standard code for initialization and termination of the library is provided in the common source file (`adi_ssl_Init.c`). Of course, the different examples use different library facilities, so each project includes its own copy of the header file `adi_ssl_Init.h`, in which the values of various preprocessor macros vary to suit particular application requirements. For example 3 we use only the SSL's power management facility, which does not need any extra information, so all the configuration macros are set to zero.

Our application (`Sorts.c`) looks similar to the application in the previous example: a work loop in `main()` repeatedly executes the bubble sort and quick sort functions. Additional calls at the start and end of `main()` initialize and terminate the SSL library as mentioned above; additionally, the work loop is wrapped inside another loop. This new outer loop cycles through a number of core voltage settings and uses the SSL function `adi_pwr_SetMaxFreqForVolt()` to adjust the voltage regulator **and** set the clocks to the maximum supported by the new level. Then the loop calls the `adi_pwr_GetFreq()` function to get the new clock rates, runs the work loop, and prints voltage, clock, elapsed time, and cycles values to the **Console** window.

Table 2-2 shows typical results from example 3. Since the entire program is in L1 memory, the **Elapsed Seconds** column shows a decrease as the core voltage (and hence the core clock rate) increases, while the number of cycles executed remains more or less constant. However, the **System MHz** column serves as a reminder that for a real application with code and data in external memory (and perhaps with other peripherals dependant on the system clock rate), core clock speed is not the only determining factor when balancing power consumption against speed.

Table 2-2. Typical Results From Example 3

Core Volts	Core MHz	System MHz	Elapsed Seconds	Core Cycles
0.85	250.0	83.3	12	2936M
0.95	325.0	81.3	9	2936M
1.05	400.0	100.0	7	2935M
1.25	500.0	125.0	6	2935M

3 USING ADSP-BF548 EZ-KIT LITE PERIPHERALS

Chapters 1 and 2 of this tutorial introduce the ADSP-BF548 processor's features that affect code execution: the memory hierarchy, instruction and data caches, and internal voltage regulator. This chapter introduces the I/O peripheral controllers and devices integrated with the processor and accessible on the EZ-KIT Lite.

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

- Peripheral integration on the ADSP-BF548 processor
- Peripheral accessibility on the ADSP-BF548 EZ-KIT Lite
- VisualDSP++ software services and drivers for controlling and using the peripherals

The chapter includes the following sections.

ADSP-BF548 Processor I/O Peripherals

The ADSP-BF548 processor's I/O controllers and peripherals fall into four categories:

1. Serial data interfaces

    - **Two-wire interface** (**TWI**). A TWI interface is compatible with the widely-used I^2C bus standard. The processor supports up to two TWI interfaces.
    - **Serial peripheral interface** (**SPI**) bus standard. The processor supports up to three SPI interfaces.
    - **Synchronous serial port** (**SPORT**). This is a powerful serial port containing two sets of input/output pins that can operate in I^2S mode and has eight-deep receive and transmit buffers. A SPORT offers considerable control over word length and framing options. The processor can support up to four SPORT interfaces.
    - **Full-duplex universal asynchronous receiver/transmitter** (**UART**) interface. With external voltage level shifters, the interface can provide RS-232 style communications. The processor can support up to four UART interfaces.
    - **Universal Serial Bus** (**USB**) 2.0 controller. The controller operates in device or host (on-the-go) mode at low, full, or high speed. The processor has one USB controller.

2. Parallel data interfaces

    - **Enhanced parallel peripheral interface** (**EPPI**). An EPPI supports direct connection of LCD panels, parallel analog-to-digital (A/D) and digital-to-analog (D/A) converters,

and other peripherals. The interface is configurable and supports data widths up to 24 bits. The ADSP-BF548 processor can support up to three EPPI interfaces.

- **Industry-standard ATA/ATAPI-6** interface. An ATA/ATAPI-6 interface is for control of a CD, DVD, hard disk drive, or any other ATAPI-compatible device.

3. Specialized interfaces

    - **Host DMA port**. A host DMA port allows an external processor to control DMA transfers to and from an ADSP-BF548 system.
    - **Secure digital** (**SD**) I/O controller
    - **Controller Area Network** (**CAN**) controllers. The CAN controllers implement the CAN 2.0B (active) protocol. The processor supports up to two CAN controllers.

4. Miscellaneous interfaces

    - A 32-bit up/down counter interface for standard manual rotary switches and mechanical drives
    - A keypad interface supporting a matrix of up to 8 x 8 keys
    - General-purpose I/O (GPIO) pins. Individual pins from any unused processor peripheral can be used as input or output data signals. The pins can be configured as edge- or level-sensitive and used in a polled- or interrupt-driven manner. The GPIO pins also are called the programmable flags (PFs).

Due to limitations on the number of pins available on the processor package, some peripherals are multiplexed onto the same pins and cannot be operated simultaneously.

For complete information about the ADSP-BF548 processor and peripherals, consult the processor's data sheet and hardware reference manuals. The data sheet and manuals are available at http://www.analog.com/processors/technicalSupport/technicalLibrary/; the hardware reference manuals are part of the VisualDSP++ online help.

ADSP-BF548 Peripheral Interfaces on the EZ-KIT Lite

In addition to the ADSP-BF548 processor and DDR and flash memories mentioned in Chapter 2, "ADSP-BF548 Processor Memory Hierarchy" on page 2-2, the EZ-KIT Lite contains the interface circuitry and sockets required for using most of the processor's peripheral interfaces:

- An RS-232 line driver and DB9 socket for one of the UARTs
- A USB on-the-go (OTG) socket for using the EZ-KIT Lite as a USB device or host
- An SD memory card connector (plus a memory card) for SD/SDIO
- Two sockets for the CAN controller
- IDC connectors for various processor interfaces

Furthermore, the board includes user devices for the following interfaces.

- An LCD screen on one of the EPPIs
- A 40 GB hard drive on the ATA/ATAPI-6

- A rotary switch for the up/down counter
- A 4 x 4 keypad matrix for the keypad interface
- An audio codec for one of the SPORTs
- Push buttons and LED indicators for the GPIO pins

In addition, the EZ-KIT Lite board includes a 10/100 Ethernet controller with associated magnetics and RJ-45 socket. The controller connects to the processor's asynchronous memory bus. See the *ADSP-BF548 EZ-KIT Lite Evaluation Manual* for a full list of the board features.

In the remainder of this and following chapters, we will learn how to use some of these peripherals and devices using VisualDSP++.

Using ADSP-BF548 Peripherals on the EZ-KIT Lite

Writing code for the simpler peripheral interfaces, such as the GPIO pins, is a straightforward process. Writing code for the data transfer interfaces, such as ATA/ATAPI-6 or EPPI, is a more complex and time-consuming process because it requires additional management of DMA channels and interrupt handling.

Analog Devices supplies software to aid the use of most of the peripherals on a Blackfin EZ-KIT Lite. The software is structured as two packages: the system services library (SSL) and device drivers (DDs); when referred to as a whole, the software is known as SSL/DD.

The SSL presents a set of application programming interfaces (APIs) for core peripherals and features common across the Blackfin processors. The APIs facilitate configuration and operation of:

- The core and peripheral interrupt systems and associated interrupt service routines
- DMA channels to and from the data-oriented peripherals and between the various memories
- The general-purpose timers
- The real-time clock
- The internal voltage regulator and core and system clocks
- The external memory bus controller
- The GPIO pins (programmable flags)
- A file system service for those devices or memories that are best accessed via a file system (such as FAT16/32)

In addition, the SSL manages a delayed callback service, a decoupling process between the interrupt and application levels.

The device drivers are individual units of code that facilitate configuration and operation of peripherals specific to one Blackfin processor or one EZ-KIT Lite board. They adhere to a common API for opening and closing the associated peripherals, passing configuration data and reading/writing application-supplied data buffers. The drivers use the SSL facilities to perform such common tasks as registering interrupt handlers, performing DMA, using GPIO pins to reset peripherals, and more. Each driver also contains device-specific code for performing its device-specific tasks.

Some device drivers use other drivers; for example, on the ADSP-BF548 EZ-KIT Lite, the AD1980 audio codec connects to the processor's SPORT0 interface for data transfers. The AD1980 device driver uses the supplied SPORT driver for managing the data transfers rather than driving the SPORT itself, or requiring the application to do so.

Refer to the VisualDSP++ online help for more information about the SSL/DD.

Example 4: Overview

This chapter's example project (`Example_4.dpj`) uses the AD1980 audio codec device driver to send a buffer of audio samples through the D/A converter to the headphone socket.

The purpose of this example is to show how to use a data-transfer peripheral's device driver. When built and run, the application initializes the SSL, opens the AD1980 audio codec driver, configures the AD1980 hardware, and sends a buffer of digital audio samples to the codec. The codec converts the digital signals to analog form and plays back the signals through the headphone/line out socket on the EZ-KIT Lite.

Navigate to the `<install_path>\Blackfin\Examples\ADSP-BF548 EZ-Kit Lite\Getting Started Examples\Example_4` directory and open the project file `Example_4.dpj`. The **Project** window lists all of the source files included in the project. Open the `Audio.c` file by double clicking its name.

Example 4: Audio.c File

Four functions in `Audio.c` interact with the AD1980 audio codec device driver: `InitAudio()` (see "InitAudio()—Opening the Driver"), `TerminateAudio()`, `PlayBuffer()`, and `AD1980Callback()`. The following sections describe some parts of these functions.

InitAudio()—Opening the Driver

The audio functions assume that the SSL is initialized. The initialization process is described in Chapters 4 and 5 on page 4-1 and on page 5-1, respectively. `InitAudio()` opens an instance of the Analog Devices-supplied device driver for the AD1980 audio codec and uses the driver to configure the hardware. `InitAudio()` also uses the driver to prepare the SSL data transfer functions to handle the buffers of PCM audio samples that the application will send to the codec. Listing 3-1 shows the call to the `adi_dev_Open()` function that opens the AD1980 driver.

Listing 3-1. Opening the Device Driver

```
/* open the AD1980 driver */
if((Result = adi_dev_Open(  adi_dev_ManagerHandle,
                            &ADIAD1980EntryPoint,
                            0,
                            NULL,
                            &AD1980DriverHandle,
                            ADI_DEV_DIRECTION_OUTBOUND,
                            adi_dma_ManagerHandle,
                            DCBQueueHandle,
                            AD1980Callback  ))
            != ADI_DEV_RESULT_SUCCESS)
{
  printf("Failed to open audio output device, "
         "Error Code: 0x%08X\n", Result);
  break;
}
```

A successful call to `adi_dev_Open()` results in a value identifying the initialized driver instance. We will use this value in future device manager calls to identify the driver instance.

The `adi_dev_Open()` function takes nine arguments. The values supplied by `InitAudio()` are:

- `adi_dev_ManagerHandle`: The `adi_dev_Open()` function is part of the device manager section of the SSL. All device driver-related calls are made to the device manager, not directly to the drivers. The device manager has been initialized elsewhere in the example; the first argument in the call is a variable containing a reference to the initialized device manager.
- `&ADIAD1980EntryPoint`: Every device driver contains a static table of information, such as function pointers defining its entry-points. Device manager uses this information to identify and manage each driver instance. The second argument passes the address of the AD1980 device driver's information table to the device manager.
- `0`: There are multiple instances of various processor peripherals; for example, multiple SPORTs and UARTs. The device manager and the drivers for these peripherals need to know which SPORT or UART is being opened. The third argument in the `adi_dev_Open()` call is an integer that distinguishes between the multiple peripherals. There is only one AD1980 audio codec on the ADSP-BF548 EZ-KIT Lite, so we pass the value of 0.
- `NULL`: An application that has opened two or more devices of the same type must be able to distinguish between the devices when the device manager signals a processing or error event. The fourth argument in the `adi_dev_Open()` call attaches a value of type `void *` to the instance of the device driver being opened. The device manager remembers the value and presents it back when a

processing or error event occurs. For the single instance of the AD1980 audio codec on the ADSP-BF548 EZ-KIT Lite, we do not need a distinguishing value, so we supply `NULL`.

- `&AD1980DriverHandle`: Argument five informs the device manager where to store the device driver instance value (handle). As noted earlier, we will use this handle to identify the driver instance in later device manager calls.

- `ADI_DEV_DIRECTION_OUTBOUND`: For some peripherals, the device manager (and sometimes the device driver) requires to know whether the application is going read data from the peripheral, write data to the peripheral, or both. If data is to flow in one direction only, interrupt and memory resources can be saved. The device manager header file (`<drivers/adi_dev.h>`) contains definitions of three identifiers to enumerate the possibilities. We select this identifier to pass as the sixth argument to `adi_dev_Open()` because we only are sending data to the AD1980 codec.

- `adi_dma_ManagerHandle`: Each of the data transfer peripherals on the ADSP-BF548 processor has at least one dedicated direct memory access (DMA) channel. DMA relieves the device driver from writing bytes or words of data to or reading them from the peripheral programmatically. All DMA channels operate in an identical manner, so the SSL provides a DMA manager for the drivers. The DMA manager must be initialized before use—the seventh argument to `adi_dev_Open()` passes in the DMA manager handle obtained elsewhere in the application.

- `DCBQueueHandle`: The eighth argument is the handle of another SSL service—the delayed callback manager. Device drivers contain code that runs when the controlled peripheral raises an interrupt. The driver does what is required to service the peripheral but often needs to inform the driver's user—the application—that a data transfer has completed or an error has occurred. The application's response to the information may be some lengthy operation, such

as preparing the next data buffer. Such an operation can affect the application's response to interrupts from other peripherals. The delayed callback manager provides a mechanism for drivers to place a function pointer and associated arguments in a queue. The callback manager removes entries from the queue and calls the functions at a later time when no other interrupts are being serviced.

- `AD1980Callback`: The ninth, and final, argument to `adi_dev_Open()` is a pointer to the callback function (AD1980Callback()) that we want the AD1980 driver to invoke. See the callback function description on page 3-16 for more information.

adi_dev_Control()—Configuring the Driver

The SSL's device manager API has one function (`adi_dev_Control()`) for configuring and controlling a device driver. An application can call `adi_dev_Control()` multiple times in order to set up the operating parameters of the peripheral. Thereafter, the application uses the same function to instruct the device manager to enable and disable data flow to and from the peripheral, if required.

The `adi_dev_Control()` function takes three arguments:

- The first argument is the device handle returned by a successful call to `adi_dev_Open()` (see Listing 3-1 on page 3-8). The value identifies the device driver instance to the device manager.
- The second argument is an integer command value that specifies the requested configuration or control action. There are two sets of command values for `adi_dev_Control()`.

- The first set is commands to the device manager, requesting the device manager to take certain actions on behalf of the device driver instance. These commands are defined in the device manager header file (<drivers/adi_dev.h>) and start with "ADI_DEV_CMD_".
- The second set is commands specific to the device driver and start with "ADI_xxx_CMD_", where "xxx" identifies the driver. In our example, the driver identifiers start with "ADI_AD1980_CMD_". There is a separate header file for each device driver where these commands are defined. For the AD1980 codec driver, the header is <drivers/codec/adi_ad1980.h>.

• The third argument is a pointer-sized value that acts as the operand for the command. If the operand is not needed, the argument is ignored.

Listing 3-2 shows a call to adi_dev_Control(), which is using a device manager command (ADI_DEV_CMD_SET_DATAFLOW_METHOD) and operand (ADI_DEV_MODE_CHAINED). Together they specify how the device manager should manage the flow of data buffers that the application will send later.

Listing 3-2. Configuring the Device Manager

```
if((Result = adi_dev_Control(
                  AD1980DriverHandle,
                  ADI_DEV_CMD_SET_DATAFLOW_METHOD,
                  (void*)ADI_DEV_MODE_CHAINED))
      != ADI_DEV_RESULT_SUCCESS)
{
  printf("Failed to set dataflow method for audio output device,"
         "Error Code: 0x%08X\n",Result);
  break;
}
```

Other device manager commands include ADI_DEV_CMD_REGISTER_TABLE_WRITE whose operand is the address of an application-defined table of register *name/value* pairs, and ADI_DEV_CMD_REGISTER_FIELD_TABLE_WRITE which takes a table of register *name/field name/value* triples. The application can supply tables of whole-register or part-register values to modify an entire set of device registers with one call to adi_dev_Control(). The InitAudio() function contains a call (not shown) setting up the AD1980 audio codec's registers appropriately for the two channel stereo audio samples supplied at a later time.

The device driver-specific control commands are individual to each driver. Many of them, however, are concerned with passing extra configuration information. Listing 3-3 shows an example. First, a structure is populated with the information that the driver requires. The structure layout is defined in the AD1980 audio codec device driver header file (<drivers/codec/adi_ad1980.h>) included in Audio.c. The information held in the structure includes the memory address and size of some work areas that the driver instance needs for operation. Also often required is information regarding the target hardware configuration. In Listing 3-3, we inform the driver that the AD1980 audio codec connects to SPORT0 for data transfers, and that programmable flag PB3 connects to the codec's reset pin. (Identifiers SPORT_DEVICE_NUMBER and AD1980_RESET_FLAG are defined and initialized elsewhere in the application.)

Listing 3-3. Initializing Specific Diver

```
InitAD1980.pDataFrame       = (void *)AudioFrameBuffer;
InitAD1980.DataFrameSize    = AUDIO_FRAME_BUFFER_SIZE;
InitAD1980.pAC97            = &AC97_Instance;
InitAD1980.ResetFlagId      = AD1980_RESET_FLAG;
InitAD1980.SportDevNumber   = SPORT_DEVICE_NUMBER;
if((Result = adi_dev_Control( AD1980DriverHandle,
                              ADI_AD1980_CMD_INIT_DRIVER,
                              (void*)&InitAD1980 ))
      != ADI_DEV_RESULT_SUCCESS)
```

```
{
  printf("Failed to initialize audio output device, "
         "Error Code: 0x%08X\n", Result);
  break;
}
```

TerminateAudio()

When an application no longer needs a peripheral, the application terminates the peripheral's driver and reclaims the driver's resources with two function calls. A call to `adi_dev_Control()` stops data flow to the peripheral. A call to another device manager API (`adi_dev_Close()`) makes any memory work areas given to the device driver instance available for reuse.

PlayBuffer()

Two final basic device manager API functions are `adi_dev_Read()` and `adi_dev_Write()`. As their names suggest, the application supplies a memory buffer to be filled with data from the peripheral (`adi_dev_Read()`), or a buffer whose contents are to be sent to the peripheral (`adi_dev_Write()`). Listing 3-4 shows the entire `PlayBuffer()` function from `Audio.c`. The function accepts the address and length of a buffer filled with audio samples and passes the information to `adi_dev_Write()`.

Listing 3-4. Submitting a Data Buffer

```
u32 PlayBuffer(u8* buffer, int buffer_length)
{
  u32 Result = ADI_DEV_RESULT_SUCCESS;
  int i = buff_ndx;

  /* set up our buffer descriptor */
  desc[i].Data                = buffer;
  desc[i].ElementCount        = buffer_length;
  desc[i].ElementWidth        = 1;
  desc[i].CallbackParameter   = (void*)&buff_free[i];
```

```
  desc[i].ProcessedFlag       = FALSE;
  desc[i].pNext               = NULL;

  if ((Result = adi_dev_Write(AD1980DriverHandle,
                              ADI_DEV_1D,
                              (ADI_DEV_BUFFER *)&desc[i]))
                != ADI_DEV_RESULT_SUCCESS)
  {
    printf("Failed to submit buffer descriptor, "
           "Error Code: 0x%08X\n", Result);
        return 1;
  }
    return (Result != ADI_DEV_RESULT_SUCCESS);
}
```

The data transfer functions require more information than just the application buffer's address and length—the functions need to know whether the application wants notified when the transfer is complete. To accommodate this per-transfer information, the device manager API defines a structure (`ADI_DEV_BUFFER`) for a buffer descriptor containing appropriate fields. The `PlayBuffer()` function in Listing 3-4 fills in the required fields and passes the address of the buffer descriptor to `adi_dev_Write()`.

Note that the call to `adi_dev_Write()` is asynchronous with respect to the actual data transfer. A successful function return does not imply that the data has been written to the peripheral; rather the buffer is queued for transfer at some future point. Similarly, a call to `adi_dev_Read()` does not return any data: the function only queues the buffer for future filling. How does an application know when data has been received or written? By setting the `CallbackParameter` field of the buffer descriptor to a non-`NULL` value, the application signals that it wants its callback function, which it registered when it opened the device driver, to be called when the buffer has been processed.

AD1980Callback()

Callback functions have the following arguments.

- The first argument identifies which driver instance has invoked the callback. This value was supplied as the fourth argument to `adi_dev_Open()` (see on page 3-8) when the driver instance was created.
- The second argument identifies the reason for the callback. It is an integer value: similar to the values of the control commands, the value is either a common event value defined by SSL's device manager or a value specific to the device driver raising the event. The event identifier indicates a successful event, such as completion of a data transfer, or an unsuccessful event, such as a hardware error. Either way, at this point it is up to the application to take some appropriate action.
- The final argument identifies the buffer descriptor (if any) that triggered the event. The application can process, reuse, or discard the buffer descriptor and the application data buffer it describes, as required.

Listing 3-5 shows the callback function from example 4. The event descriptor `ADI_DEV_EVENT_BUFFER_PROCESSED` is one of the device manager's standard events, while `ADI_AC97_EVENT_REGISTER_ACCESS_COMPLETE` is specific to the audio codec driver.

Listing 3-5. Callback Function

```
static void AD1980Callback(
  void *AppHandle,
  u32  Event,
  void *pArg)
{
  ADI_DEV_1D_BUFFER*  pdesc;
  volatile bool*      pflag;

  switch (Event)
```

```
  {
    case (ADI_AC97_EVENT_REGISTER_ACCESS_COMPLETE):
        reg_update_complete = true;
        break;
    case (ADI_DEV_EVENT_BUFFER_PROCESSED):
        /* signal that this transfer has completed     */
        /* ..get address of this buffer descriptor     */
        pdesc = (ADI_DEV_1D_BUFFER*)pArg;
        /* ..extract address of flag and set it        */
        pflag  = (volatile bool*)pdesc->CallbackParameter;
        *pflag = true;
        break;
    default:
        printf("Unexpected audio device callback. "
               "Event code: 0x%08X\n",Event);
        break;
  }
  /* return */
}
```

Example 4: Running

It has taken a lot more effort to describe how to use a typical device driver than it does to actually use it! To observe the device driver in action, open and build the `Example_4.dpj` project located in `<install_path>\Blackfin\Examples\ADSP-BF548 EZ-Kit Lite\Getting Started Examples\Example_4`. On the EZ-KIT Lite, plug a pair of headphones or powered speakers into the audio socket labelled `LINE OUT` (it is the single-tier socket beside the two double-tier sockets) and press **F5** to run the project. You will hear the phrase "An orange-colored sky" spoken five times with a short pause between repetitions.

4 USING ADSP-BF548 EZ-KIT LITE AS A MASS STORAGE DEVICE

In this chapter, we will make the EZ-KIT Lite hard disk accessible from a PC as a removable mass storage device (MSD) with minimum application code. Then we will copy some files from the PC to the disk in preparation for later examples.

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

- ADSP-BF548 processor's USB interface
- USB standard software
- USB data transfers to or from the EZ-KIT Lite

The chapter includes the following sections.

ADSP-BF548 Processor USB Interface

The ADSP-BF548 processor incorporates a USB on-the-go (OTG) controller capable of operating at high speed (480 Mbps), full speed (12 Mbps), and low speed (1.5 Mbps, host mode only). The controller operates in peripheral mode or OTG mode. In peripheral mode, the controller appears as a single or multi-function device on a USB bus. In OTG host mode, the controller directs data transfers between the ADSP-BF548 processor and some attached USB device, such as a digital camera. On the EZ-KIT Lite, the USB controller connects to a standard OTG-style socket, which allows either host or device mode.

For more information about the USB interface, refer to the ADSP-BF548 processor's hardware reference and EZ-KIT Lite manuals.

Analog Devices USB Software

In order to support a wide variety of plug-and-play devices, the USB standard defines a fast bus architecture supporting guaranteed data transfer bandwidth (or latency) and asynchronous control messaging. The USB standard describes several layers of software that control:

- The low-level protocol, involving physical data transfer and device addressing
- The enumeration of connected devices, involving device class specification
- The proper reaction of devices to standard control messages
- The proper operation of USB hosts

The Analog Devices USB software (supplied with VisualDSP++ 5.0) uses the system services library (SSL) and device driver models to create an architecture to which you can add your own class- and device-specific

code. The supplied software includes mass storage class and bulk transfer class device drivers. In this exercise, we will use the mass storage device driver.

The main principle behind the operation of a USB mass storage device is that communication from the host to the device takes the form of basic Small Computer Systems Interface (SCSI) commands wrapped inside standard USB transactions. The SCSI is a long-established standard interface for the interconnection of system peripherals. The USB mass storage class uses some SCSI basic commands for reading sectors from and writing sectors to the device's storage medium, and for finding out the medium's storage capacity. The USB host software is responsible for dealing with the higher-level aspects of mass storage, such as file system organization. The USB device software is responsible for implementing the low-level sector read and write commands, appropriate for the medium being used. The VisualDSP++ 5.0 mass storage device class software includes code that implements the necessary SCSI commands for the EZ-KIT Lite hard disk.

Example 5: USB Project

This chapter's example project (`Example_5.dpj`) is a straightforward piece of code that makes the EZ-KIT Lite hard disk available to a PC as a removable storage device. All the code has to do is to open and configure the supplied USB mass storage device class driver; the underlying support code does the hard work. The source files included in the project are:

- **Example_5.c.** The source file contains all of the project-specific code in function `main()`.
- **adi_ssl_Init.h.** The project-specific header file defines the numbers of various SSL resources that the project requires.
- **adi_ssl_Init.c.** The common source file defines functions for initializing and terminating the SSL components. The SSL-related examples in this tutorial use a project-specific copy of

adi_ssl_Init.h to specify their requirements. The common functions in adi_ssl_Init.c perform the SSL initialization, according to the requirements, and the final closing.

The code in main() is also straightforward. After initializing the SSL, the code does the following.

1. Uses the device manager to open and obtain a handle for the USB mass storage device driver. The only unusual aspect is that the adi_dev_Open() function requires the address of a callback function while the USB mass storage device driver does not invoke the callback. Our example supplies the address of a dummy callback function (Callback()).

2. Uses the standard configuration mechanism, adi_dev_Control(), to pass the ADI_USB_MSD_CMD_INIT_RAW_PID command to the USB mass storage device driver. The command instructs the driver to initialize a lower-level driver responsible for physical access to the EZ-KIT Lite hard disk, then set a flag indicating the operation's success.

3. Uses adi_dev_Control() again, this time passing a table of commands rather than a single instance. Some commands in the table are generic and relate to the common device driver model, the rest are specific to the USB mass storage device driver. Table 4-1 lists the commands passed to the table and gives a brief description of each command.
   Note that the USB vendor ID, peripheral/class ID and peripheral serial number values used in this example program are for demonstration purposes and must not be used in your production software.

4. Prints a message to the VisualDSP++ **Console** window indicating that the EZ-KIT Lite is now ready to connect to a USB host.

5. Calls `adi_dev_Control()` again, this time in a loop, using the `ADI_USB_MSD_CMD_IS_DEVICE_CONFIGURED` command until the driver signals that a USB host has enumerated and configured the EZ-KIT Lite as a mass storage device.
6. Calls `adi_dev_Control()` for the final time, using the `ADI_USB_MSD_CMD_SET_BUFFER` command to pass the address of a data structure to the driver. Among other data, the structure includes the address of a memory buffer that the USB software will use (and reuse) while accepting and responding to commands from the USB host. This is a non-standard use of the device driver API—usually the data structures are passed dynamically between the application and the driver in response to incoming or outgoing data transfer events. An application that invokes the MSD driver, however, takes no part in the actual data transfer; the alternative means of passing the data structure is used instead.
7. Prints another message indicating that a USB host has successfully recognized and configured the EZ-KIT Lite as a mass storage device.
8. Enters an idle loop, leaving the interrupt-driven mass storage device driver and support software to accept commands from the USB host and to access the EZ-KIT Lite hard disk.

Table 4-1. Example 5 Commands

Configuration Command	Description
`ADI_USB_MSD_CMD_SCSI_INIT`	Directs the MSD driver to initialize a subsystem that responds to SCSI disk access requests from the USB host
`ADI_USB_MSD_CMD_SET_VID`	Supplies a USB vendor ID for the driver to use
`ADI_USB_MSD_CMD_SET_PID`	Supplies a USB peripheral/class ID for the driver to use

Table 4-1. Example 5 Commands (Cont'd)

Configuration Command	Description
ADI_USB_MSD_CMD_SET_SERIAL_NUMBER	Supplies a USB peripheral serial number for the driver to use
ADI_DEV_CMD_SET_DATAFLOW_METHOD	Describes the format of the data buffer that will be supplied for device manager
ADI_DEV_CMD_SET_DATAFLOW	Enables data transfers within device manager
ADI_USB_USB_CMD_ENABLE_USB	Directs MSD driver to enable operation
ADI_DEV_CMD_END	End-of-table marker

Example 5: Running

To run `Example_5.dpj` you need a USB cable with an on-the-go mini-B connector at one end and a series A connector at the other end. The ADSP-BF548 EZ-KIT Lite is supplied with a 5-in-1 USB cable and suitable connectors. Plug the mini-B end of the cable into the socket marked `USB OTG` on the EZ-KIT Lite and leave the other end unconnected. Build and run the project. The message "`USB Mass Storage Device ready to be connected to PC`" appears in the **Console** window of VisualDSP++. Now connect the free end of the USB cable to a USB 2.0 socket on your PC. The message "`USB Mass Storage Device is connected to PC`" appears in the **Console** window, indicating that the USB host Windows software has recognized the EZ-KIT Lite as a mass storage device.

What happens next depends on how you have configured Windows to react when a new USB device is plugged in. Windows may display a dialog (see Figure 4-1), asking which of several actions to follow, or may display a new **Windows Explorer** window with the contents of the new device (that is, the EZ-KIT Lite hard disk). It is possible that Windows only displays the disk as a new entry in the **Hard Disk Drives** section of **My**

Computer (default action). Figure 4-2 shows **My Computer** after Windows has recognized the EZ-KIT Lite disk and assigned it a drive letter ("E" in this case).

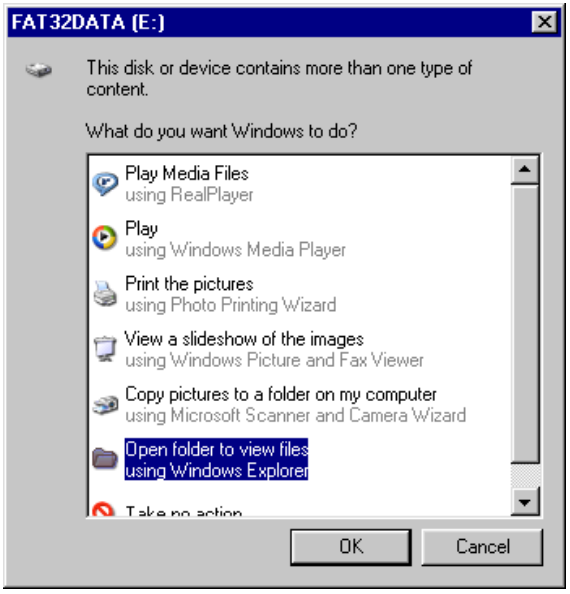

Figure 4-1. Windows Explorer Dialog Box

If Windows displays a message indicating that the new device is not formatted, stop the example and locate the ADSP-BF548 EZ-KIT Lite disk formatting utility in the `<install_path>\Blackfin\Examples\ADSP-BF548 EZ-Kit Lite\Services\File System\HardDisk\HardDiskFormat` directory. Refer to the readme file before building and running the utility. Then come back and rerun `Example_5.dpj`.

Now you can treat the EZ-KIT Lite hard drive as any other hard drive connected to your PC. Inspect the disk's contents (if any) using Windows Explorer, copy files using drag-and-drop, save files, and so on. The real purpose of this example, however, is to copy some specific data files to the

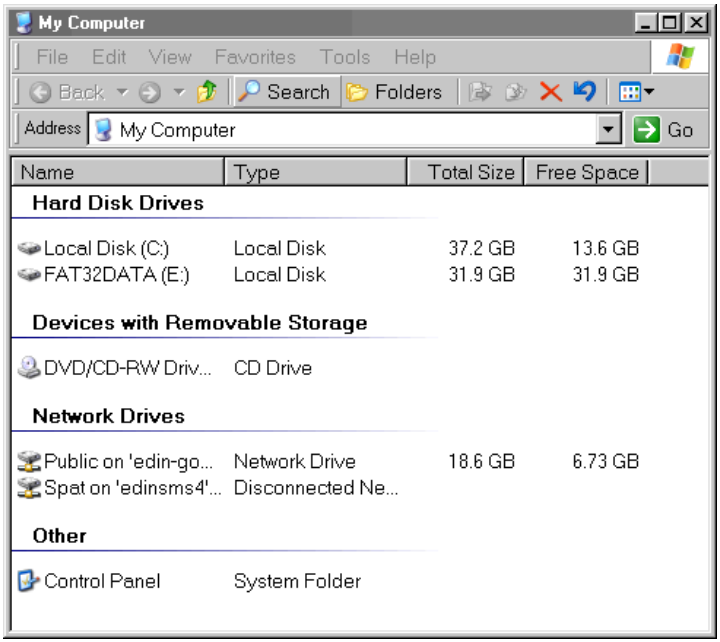

Figure 4-2. Windows My Computer Display

EZ-KIT Lite drive for later use. On the PC, navigate to example 5's directory and drag the `BMP Files` folder to the top level of the EZ-KIT Lite drive, thus creating the `E:\BMP Files` directory if "`E`" is the letter that Windows assigned to the EZ-KIT Lite drive.

It shouldn't take long to copy the folder and its contents; when finished, left-click the **Safely Remove Hardware** icon in the system tray area of your Windows task bar and select **Safely remove USB Mass Storage Device - Drive (E:)**. (Choose the option with the appropriate drive letter if there is more than one option.) When Windows indicates that the hardware can be safely removed, unplug the cable and use VisualDSP++ to halt example 5.

5 USING ADSP-BF548 EZ-KIT LITE HARD DISK AND LCD SCREEN

In this chapter we will use a major new component of the system services library (SSL)—the file system service (FSS)—in addition to one of the supplied device drivers to create a program that reads a bitmap image from the EZ-KIT Lite hard disk and displays the image on the LCD screen.

In order to run this program successfully, first run "Example 5: USB Project" on page 4-3 and follow the instructions to copy the folder containing the bitmap file from your PC to the EZ-KIT Lite hard disk.

The concept introduced in this exercise is that of pluggable or stackable components:

- The file system service can be plugged into the C I/O library in place of the standard, emulator-based facilities
- Alternative device drivers can be plugged into the file system service to target different physical devices and to support different file system organizations
- The LCD driver is stacked on top of a driver for its data source—one of the ADSP-BF548 processor's enhanced parallel peripheral interfaces (EPPIs)

The chapter includes the following sections.

SSL File System Service

The file system service is a major addition to the SSL in VisualDSP++ 5.0. Embedded applications targeting Blackfin processors now have a direct means of accessing data held on mass storage devices using standard C library functions or custom APIs.

Built on the existing system service and device driver models, the FSS can be configured to use different physical devices and different file system organizations by changing only a few settings in its initialization parameters. The initial release of the FSS is accompanied by physical device drivers for ATA/ATAPI attached devices, such as the ADSP-BF548 EZ-KIT Lite hard disk, SD devices, and USB mass storage host (for flash/pen drives plugged into the EZ-KIT Lite). Also included is a software driver supporting FAT 12/16/32 file systems. The open architecture of the FSS means that the FSS can be used with different combinations of drivers to support a flash file system on NAND flash memory or an ISO-9660 file system on an attached CD drive with minimal changes to application-level code.

At the application level, the easiest way to use FSS is to exploit its 'pluggability' and hook an initialized instance of FSS into the C library's I/O system via a single function call. Thereafter, all of the file-oriented functions of the standard header `<stdio.h>` (such as `fopen()`, `fread()`, and

`fwrite()`) and many common directory-oriented functions (such as `mkdir()`, `rmdir()`, `opendir()`, `readdir()`, and `closedir()`) will apply to the file system/device driver combination supported by the FSS instance.

The FSS also supports an API of its own, which allows more detailed control of the underlying file system and device drivers. The API description is included in the FSS documentation accompanying VisualDSP++. Note that this tutorial does not cover the API's use.

Sharp LQ043T1DG01 LCD Device Driver

Most of the ADSP-BF548 processor's on-chip peripherals and most of the devices mounted on the EZ-KIT Lite are supplied with device drivers, which are included in VisualDSP++ 5.0. One of the supplied drivers controls the display functions of the Sharp LQ0431T1DG01 LCD display. (Another driver controls the touch-screen functionality of the display, but this tutorial does not cover that driver's functionality.)

The LCD driver is a typical driver for managing a bulk-data device: it is layered on top of another driver for its data provider and uses standard device manager facilities for controlling the DMA-based data flow. In this instance, the LCD's data provider is one of the ADSP-BF548 processor's enhanced parallel peripheral interfaces. The LCD driver is responsible for opening and running an instance of the EPPI driver but provides the application with an opportunity to configure the EPPI first, as we will see in the following example on page 5-4.

Using the LCD driver typically involves a specific set-up for the Sharp LCD (and for the EPPI to which the LCD connects), then generic control of the DMA-based I/O. The latter requires deciding which of the standard data flow methods (chained buffers, chained buffers with loopback, circular buffers) will be used, then providing data to the device driver in a timely manner.

VisualDSP++ 5.0 includes full documentation about the LCD device driver.

Example 6: Displaying a Bitmap File on the EZ-KIT Lite

This chapter's example program (`Example_6.dpj`) uses the facilities of the FSS and LCD driver to read a bitmap image from the EZ-KIT Lite hard disk and to display the image on the LCD. There is remarkably little application-level code needed to achieve the result, but before examining the code, we will examine the project set-up. Note that FSS requires only a few non-default project settings.

Example 6: Project Options

Open the `Example_6.dpj` file in VisualDSP++ and examine the **Project** window (Figure 5-1).

There is a new **Generated Files** folder with source files in the **Startup** and **User Heap** subfolders. What are these source files and why do we need them? The files were generated automatically when the project was created in response to customization options for an extra user-defined heap and for modifiable memory cache control tables. We need an extra heap for FSS's use; we also need to modify the cache characteristics of the memory area that the heap occupies. The project's linker description file (`Example_6.ldf`) was also created automatically; the `.ldf` file plays an important part in defining the system and user heaps.

To see the customizations made when the project was created, open the project's options viewer using the **Project–>Project Options** menu selection or by right-clicking the project name in the **Project** window and selecting **Project Options**. In the project tree control, scroll down to the **LDF Settings** node and select the **User Heap** item (Figure 5-2).

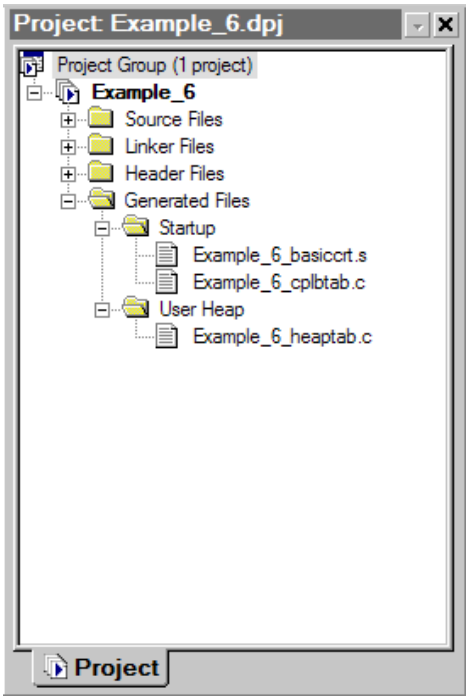

Figure 5-1. Example 6 Project Window

The table at the top of the window shows that in addition to the system heap, which the C library functions `malloc()` and `free()` use by default, there is a user heap (`FSSGeneralHeap`) allocated 4 MB of space in DDR SDRAM memory (L3). FSS operates a cache of disk sectors to speed up its operation, but instead of making FSS allocate and free memory for the cache from the system heap (which may have a disruptive effect on other parts of the application), we give FSS a heap of its own. The FSS's heap does not require to be as large as 4 MB, but 4 MB turns out to be a convenient size for other reasons.

Now scroll down the project option tree control and locate the **Cache and Memory Protection** item under **Startup Code Settings**. You can see that we have enabled the instruction cache and both the data caches to improve

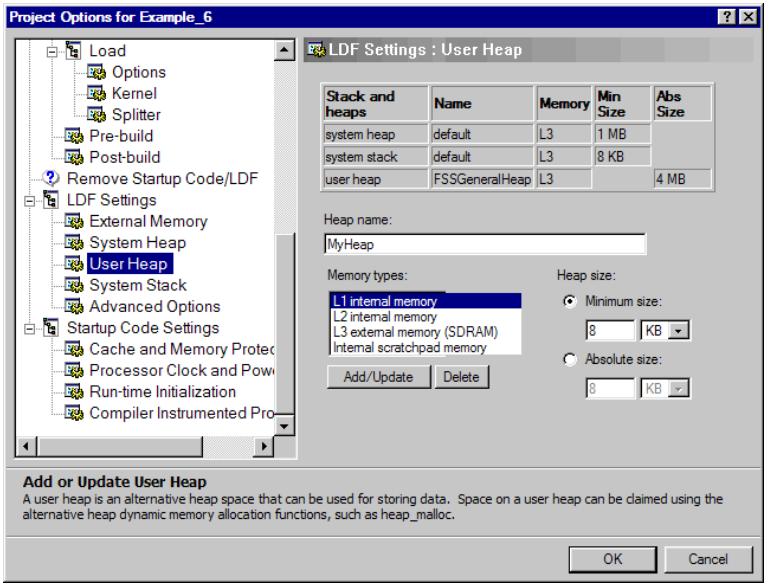

Figure 5-2. User Heap Settings

the application's general performance. The processor's caches or memory protection features require a control table, the cacheability and protection lookaside buffer (CPLB), to be set up. Typically this is done automatically by VisualDSP++ using appropriate default settings. However, current FSS and some device drivers cannot operate correctly when the data sections they read to or write from are cacheable. To allow this, the option **Generate a customizable CPLB table** has been selected (in the lower half of the **Cache and Memory Protection** window). The option provides the opportunity to modify the appropriate CPLB table entries.

When you finish browsing the project options, click **Cancel** rather than **OK** to dismiss the dialog box in case you unintentionally modified any project settings.

Now double-click the **Example_6_heaptab.c** filename to open a source window containing the file's contents and scroll to the bottom of the file (Listing 5-1). There are external declarations for two symbol pairs: one pair for the default system heap, the other pair for our user-defined heap. The symbols are defined by the linker when the example is built, acting on commands placed in the linker description file by VisualDSP++. The rest of the file consists of the initialization of a heap descriptors table, as detailed under "*Defining Heaps at Link Time*" in the *VisualDSP++ C/C++ Compiler for Blackfin Processors Manual* in the online help.

Note that the generated `Example_6_heaptab.c` requires no editing.

Listing 5-1. Example_6_heaptab.c File

```
/* Address and length of system user heaps */
extern “asm” int ldf_heap_space;
extern “asm” int ldf_heap_length;
extern “asm” int FSSGeneralHeap_space;
extern “asm” int FSSGeneralHeap_length;

struct heap_table_t
{
  void           *base;
  unsigned long  length;
  long int       userid;
};

/* Program’s heap descriptor table */
#pragma file_attr(“libData=HeapTable”)
#pragma section(“constdata”)
struct heap_table_t heap_table[3] =
{
  { &ldf_heap_space, (int) &ldf_heap_length, 0 },
  { &FSSGeneralHeap_space, (int) &FSSGeneralHeap_length, 1 },

  { 0, 0, 0 }
};
```

The other generated C file (`Example_6_cplbtab.c`) does require modification—this has already been done in the `Example_6.dpj` project file distributed with VisualDSP++ 5.0: the properties of the data cache CPLB entries covering the FSS heap and frame buffers for the LCD have been changed to disable caching (Listing 5-2). The startup code that runs before `main()` transfers the contents of the CPLB tables into the appropriate internal registers of the ADSP-BF548 processor before enabling the caches.

Listing 5-2. Example_6_cplbtab.c File

```
...
// CPLBs covering 48MB
{0x01000000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
{0x01400000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
{0x01800000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},

// LCD frame (DMA use)
{0x01c00000, (PAGE_SIZE_4MB | CPLB_DNOCACHE)},
{0x02000000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
{0x02400000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
{0x02800000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},

// LCD frame (DMA use)
{0x02c00000, (PAGE_SIZE_4MB | CPLB_DNOCACHE)},

// FSS Heap (DMA use)
{0x03000000, (PAGE_SIZE_4MB | CPLB_DNOCACHE)},
{0x03400000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
{0x03800000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
{0x03c00000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
...
...
```

Example 6: Application Structure

The structure of the example 6 application is straightforward; source file `Example_6.c` contains `main()`, which:

- Calls functions to initialize the main components of the SSL, such as the interrupt and device managers, and set up the services and drivers for the specific devices (file system service, LCD display, and an I/O flag for a push button)
- Calls a function to read a specific `.bmp` image file from the EZ-KIT Lite hard disk and send the image data to the LCD to display
- Loops, checking once per second for a pushbutton event to terminate the loop
- Calls functions to close down the device drivers and services

The two other major source files used in this project are located in the `Common` folder, at the same level as the `Example_6` folder. These files are FileSystem.c and LCD.c

FileSystem.c

The file's major function (`InitFileSystem()`) initializes and configures the file system service. Note that this needs only the inclusion of the appropriate header files, plus four function calls to:

1. Obtain identifier of the heap created for the FSS (see "Example 6: Project Options" on page 5-4)
2. Create a delayed callback queue for the FSS

3. Call the FSS's initialization function, passing in a table of seven configuration parameters
4. Call a C-library function to install the FSS instance as the default `STDIO` driver

That is all! Our application's calls to the `STDIO` file handling functions and the directory manipulation functions now apply to the EZ-KIT Lite hard disk. The FSS automatically takes care of initializing and controlling the physical device driver (for the disk) and the file system driver (for the FAT file system).

LCD.c

The `InitLCD()` function contains code to initialize and configure the driver for the LCD display. In addition, the driver layering (see "Sharp LQ043T1DG01 LCD Device Driver" on page 5-3) allows the function to configure an EPPI driver, acting through the LCD driver. The configuration parameters are commented in the source file. For complete details, see the Sharp LQ043T1DG01 LCD and the EPPI drivers documentation in the `<install_path>\Blackfin\docs` folder of VisualDSP++.

The most interesting part of the configuration process is how the DMA chains for the frame buffers are initialized, and how the DMA data flow is set up. In `LCD.c`, the data flow configuration is described in a comment headed `DMA buffer preparation`. The *DMA Manager* section of the *Device Drivers and System Services Manual for Blackfin Processors* in the online help contains introductory and reference material about managing DMA in your applications.

The `DisplayBMP()` function is responsible for opening a named file containing a suitable bitmap image, reading the file's contents into a temporary frame buffer, and causing the LCD driver to start displaying the new image. The function's code is straightforward, with the only com-

plications involving centering, cropping the image and filling the frame buffer 'backwards' to accommodate a line ordering difference between the `.bmp` file format and the LCD.

In the `InitLCD()` function, we request the DMA manager to call function `LcdCallback()` after the final transfer in each of the two DMA chains. This callback function checks whether `DisplayBMP()` has prepared a new frame to be displayed and, if so, updates the inactive chain's data buffer descriptor with the address of the new frame. During this time, the other chain's buffer is being displayed and, when that chain terminates, `LcdCallback()` will update this chain's descriptor too.

adi_ssl_Init.h

The final point to note is about the section of the `adi_ssl_Init.h` header file that defines the number of SSL resources required by our application (see Listing 5-3): because many drivers themselves use drivers, it is vital to read each driver's documentation to determine the total resources required for your application.

Listing 5-3. adi_ssl_Init.h File

```
...
#define ADI_SSL_INT_NUM_SECONDARY_HANDLERS   (6)
  // number of secondary interrupt handlers
  //   LCD : EPPI DMA data transfer
  //   LCD : EPPI DMA error
  //   LCD : Timer
  //   Disk: ATAPI read
  //   Disk: ATAPI write
  //   Disk: ATAPI error
#define ADI_SSL_DCB_NUM_SERVERS               (2)
  //   number of DCB servers
  //   one for LCD and one for disk
#define ADI_SSL_DMA_NUM_CHANNELS              (3)
  //   number of DMA channels
```

```
  //   one for EPPI and two for Disk
#define ADI_SSL_FLAG_NUM_CALLBACKS            (0)
  //   number of flag callbacks
#define ADI_SSL_DEV_NUM_DEVICES               (4)
  //   number of device drivers
  //   LCD, EPPI, file system, ATAPI
...
...
```

Example 6: Running

Build and run the project. The LCD screen first turns black and then displays the image from the bitmap file (Figure 5-3). To clear the screen and terminate the program, press and hold for a second the PB4 button on the EZ-KIT Lite.

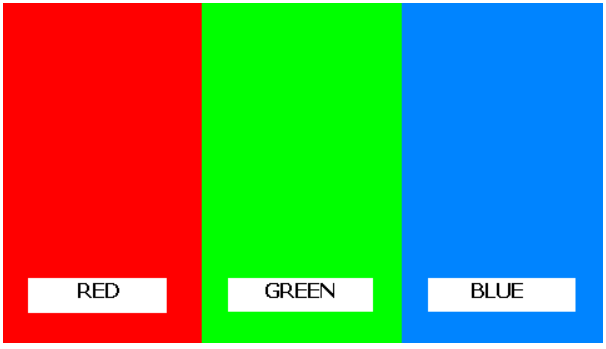

Figure 5-3. LCD Screen Image

If the program prints an error message, stating it cannot open the bitmap file, ensure that you have followed the instructions in Chapter 4, "Using ADSP-BF548 EZ-KIT Lite As A Mass Storage Device" on page 4-1 and copied the file from your PC to the EZ-KIT Lite hard disk.

6 USING ADSP-BF548 EZ-KIT LITE KEYPAD AND LED INDICATOR

In this chapter, we will use the ADSP-BF548 EZ-KIT Lite 4 x 4 matrix keypad and six LED indicators. We will use a simple event-driven driver to detect keypad presses and the system services library's (SSL) flag service to display each press's row and column numbers on the LEDs. In addition, we will use the VisualDSP++ real-time kernel (VDK) to structure our example program.

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

- Event-driven as opposed to data-driven SSL drivers
- SSL device drivers within the VDK device model
- VDK inter-thread message-passing

The chapter includes the following sections.

Event-Driven Device Drivers

There are two classes of peripheral device within the ADSP-BF548 processor: those whose primary purpose is to control a flow of data in or out of the processor, and those that generate or react to individual events. In "Example 6: Displaying a Bitmap File on the EZ-KIT Lite" on page 5-4, we have learned how the SSL's device manager and DMA manager work seamlessly with the EPPI device driver to control the flow of data to an external peripheral (the LCD). With the help of the SSL device and DMA managers, the application was responsible only for providing the required data.

For event-driven peripherals that are common across the Blackfin family of processors, the SSL provides generic API sets, such as the programmable flag service, timer service, and port control service. Each service API set includes useful functions for managing and controlling the relevant peripheral(s). This chapter's example project (`Example_7.dpj`) uses the programmable flag service to control the state of LED indicators on the ADSP-BF548 EZ-KIT Lite.

Event-driven peripherals specific to one or several Blackfin processors are supported by device drivers. For ADSP-BF548 processors these peripherals include the keypad and rotary counter interfaces. The drivers for the event-driven peripherals are used in the same manner as the drivers for data-driven peripherals. In some respects, it is easier to use event-driven drivers because an application uses the same device manager APIs to open, configure, enable, disable, and close the device, and there is no need to set up chains of DMA requests and to configure a dataflow model. This is because all interactions with the device are event-based and handled in the application's callback function.

The callback function is where the difference between the data-driven and event-driven device drivers is most evident. For data-driven events, the emphasis is on processing the successful termination of data transfer operations and initiating new transfers. In this case, errors or exception

conditions are, by and large, not expected to happen. For event-driven drivers, the emphasis is on reacting appropriately to each of the status or exception events that the device can generate. The driver passes any information supplied with the event notification to the rest of the application for processing.

Note that with both device forms, data-driven and event-driven, the Analog Devices-supplied driver obtains status information from the device and clears any interrupt conditions. Applications are free to operate at a higher level to handle data transfer and control events.

VDK Device Model

VDK is a real time kernel supplied with VisualDSP++ that supports:

- The definition of thread types and the creation of thread instances
- A priority-based thread scheduler
- Inter-thread synchronization mechanisms such as semaphores, mutexes, messages and events
- A smooth interrupt domain to thread domain transition
- A device model around which to structure applications

The SSL/device driver libraries provide low-level code for configuring and driving specific devices but impose no particular organization on the applications that use the code. In contrast, the VDK device model provides a means to structure VDK applications around synchronous access to data streams. VDK supplies appropriate synchronization methods and a driver framework for each device defined in the application, leaving the user to supply the low-level device access code.

It is possible to use the two complementary features (VDK's higher-level device model and the SSL/device drivers' low-level code) together in one application, as shown in example 7.

VDK Message Passing

As mentioned before, VDK supports a variety of inter-thread synchronization methods. For example, two different threads in your application may require access to the same shared resource; for example, a data buffer. In this case, a semaphore or mutex can be used to protect the resource against concurrent accesses. Or, your application takes the form of a control thread that creates data for worker threads to process and requires a synchronized access to a queue of work requests (messages). VDK provides such a mechanism: the required message queues are defined at project creation and created during application start-up. VDK automatically coordinates the flow of messages in to and out from the queue as the application executes.

Example 7 uses VDK messages to pass information about keypad presses from the boot thread to a worker thread.

Introductory and reference material about VDK can be found in the *VisualDSP++ Kernel (VDK) User's Guide* that is part of the VisualDSP++ online help under **Manuals–>Software Tools Manuals**.

Example 7: Creating a VDK Application

Example 7 was created as a VDK project by selecting the project type **VDK application** on the initial page of the **New Project** wizard. Whenever VisualDSP++ creates a new VDK project or opens an existing one, a new tab labelled **Kernel** is created in the **Project** window. The **Kernel** tab

is where VDK-specific attributes of the project are entered and edited. Other general attributes of the project are entered or edited in the **Project Wizard**.

Figure 6-1 shows the **Kernel** tab for example 7. The VDK project attributes are displayed in a tree structure and grouped into subsections according to function. We will examine the **Threads**, **I/O Interface**, **Device Flags**, and **Messages** subsections:

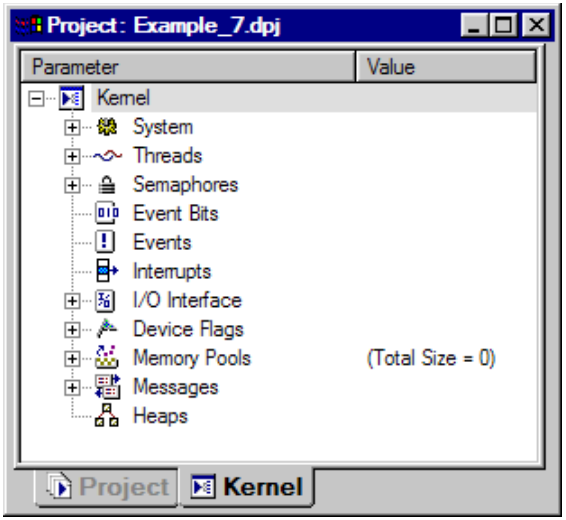

Figure 6-1. VDK Kernel Tab for Example 7

- The **Threads** subsection defines two types of thread (`BootThreadType` and `LEDThreadType`) according to their initial priorities, stack-space requirements, and ability to receive VDK messages.

The VDK runtime system is instructed to create and start execution of one instance of `BootThreadType` as a boot thread when initializing the application.

- The **I/O Interface** subsection defines one type of VDK device driver (`KeypadDriver`) and instructs VDK to create one instance of the device driver during application initialization.
- The **Device Flags** subsection defines the flag (`KeypadFlag`), which is the means of synchronization between the interrupt-domain and thread-domain code in `KeypadDriver`.
- The **Messages** subsection specifies the maximum number of VDK messages that can exist simultaneously in the application and directs VDK to allocate space for messages from the standard memory heap.

When the project was created, the **Project Wizard** generated header files and skeleton code files for the two thread types and device driver type. The wizard can produce C++, C, or assembly versions of the files; C version was selected for this project. The wizard also generated the `VDK.h` and `VDK.cpp` files. The header file contains pre-processor definitions that enable or disable VDK features, enumerations for identifier values for the application's thread and device types, and gives access to the whole VDK public API. The source file contains the VDK objects that correspond to the application's boot time thread, device, and device flag.

Example 7: Source Files

A VDK application typically does not define a `main()` function: VDK contains a default `main()` that performs the necessary initialization and starts the scheduler, resulting in execution of one boot thread. However, the default version only initializes the VDK system, while we want our example program to initialize the SSL in the same manner as the previous examples. Therefore, we provide our own `main()` in the `main.c` file (see Listing 6-1).

Listing 6-1. main.c File

```
#include "VDK.h"
#include "adi_ssl_Init.h"

int main(void)
{
    adi_ssl_Init();   /* initialize the SSL                    */
    VDK_Initialize(); /* initialize VDK and boot-time objects */
    VDK_Run();        /* start VDK's thread scheduler          */

    return 0;
}
```

Listing 6-1 shows the two essential VDK API calls preceded by a call to the common code that initializes the SSL, as used in several other examples. The `adi_ssl_Init.h` include file defines the SSL resources required for the project.

The source files generated by the **Project Wizard** for the project's thread types and device driver consist essentially of empty definitions of the functions that the VDK thread control and device handling models require. See the *VisualDSP++ 5.0 Kernel (VDK) User's Guide* for more information.

The VDK-style device driver has only one skeleton function requiring completion—`KeypadDriver_DispatchFunction()`. VDK calls `KeypadDriver_DispatchFunction()` at specific points in the application, passing an argument to indicate the reason (text *in italics* describes the code added to the function's skeleton):

- At boot time, when VDK is creating the device object.
  No additional code for the device driver is required at this point.
- When the application opens or closes the device.
  Our device driver calls the SSL driver to open and enable (or disable and close) the physical device driver. Opening the physical driver also registers our callback function (ISRCallback()) with the SSL.
- When the physical device needs attention.
  When a key press is detected, the physical device driver calls the callback function. The callback function appends the key press details to a buffer and activates this driver according to the VDK model, causing an entry to the VDK driver at a later time. All the VDK driver needs to do is signal when a key press is available by posting the device driver flag (KeypadFlag).
- When the application requests data transfers to or from the device.
  In our case, this means that a thread has issued a read on our VDK driver in order to get details of the next key press. The driver checks the buffer of key presses; if one is available, the key press is removed from the buffer and passed back to the calling thread. Otherwise, the driver pends on KeypadFlag, which blocks the calling thread until a subsequent activation entry posts the flag again.

Each source file that the **Project Wizard** generates in response to a new thread type definition in the **Kernel** tab contains skeleton definitions of four functions:

- An initialization function called when an instance of the thread type is created by VDK as a boot thread (or by some other thread in the application)
- A termination function called when an instance of the thread type is destroyed automatically by VDK (or programmatically by the application)
- An error function called by VDK API functions to report an error condition
- A run function called by VDK to start execution of the thread instance. The thread exists until the run function returns (unless terminated programmatically)

The `BootThreadType.c` file of example 7 adds no extra code to the initialization, termination, or error functions—the default code in the functions is sufficient for the application. Listing 6-2 shows the code added to the thread's run function.

Listing 6-2. BootThreadType's Run() Function

```
void
BootThreadType_RunFunction(void **inPtr)
{
  bool running = true;

  /* get device descriptor for keypad from VDK-style driver */
  VDK_DeviceDescriptor keypad_dd =
                                VDK_OpenDevice(kKeypad, NULL);

  /* create thread instance that will display row and         */
  /* column values on LEDs                                    */
  VDK_ThreadID   LEDthread = VDK_CreateThread(kLEDThreadType);
```

```
while (running){
  unsigned     int r;
  keycoords_t  keypress;

  /* get next keypress                                          */
  r = VDK_SyncRead(
            keypad_dd,      /* device descriptor for keypad */
            (char*)&keypress, /* where to put next keypress */
            1,                /* number required (ignored)  */
            0);               /* no timeout - wait forever  */

  if (r == 0) {
    /* pack row/col info into type word to avoid needing    */
    /* a payload                                            */
    int type = ((keypress.row << 8) | keypress.col);

    VDK_MessageID msg = VDK_CreateMessage(
                     type,  /* type is row/col packed in int */
                     0,     /* no payload, size is zero      */
                     NULL); /* no payload, address is NULL   */

    /* check that message was created OK                    */
    r = (msg == UINT_MAX);

    if (r == 0) {
       /* post message to LED display thread                */
       VDK_PostMessage(
           LEDthread,         /* destination thread instance */
           msg,               /* the message's ID            */
           VDK_kMsgChannel1); /* queue to use in dest thread */
    }
  }

  /* terminate loop if any errors detected                  */
  running = (r == 0);
}

VDK_CloseDevice(keypad_dd);
```

```
  VDK_DestroyThread(LEDthread, false);

  /* This thread is automatically Destroyed when it exits its */
  /* run function                                             */
}
```

The code in Listing 6-2 does the following.

- Opens the VDK-style keypad device which, as described above, opens the SSL-style physical device driver for the keypad.
- Creates an instance of `LEDThreadType`. The thread is created programmatically (rather than being created by VDK as a boot thread) in order to obtain the thread's identifier. The thread's ID is required to send messages to the thread. The newly created thread is responsible for displaying key press row and column numbers on the EZ-KIT Lite LED indicators.
- Enters a loop to read details of the next key press from the keypad driver (which blocks this thread if no key press is available), creates a VDK message to hold the key press details, and posts the message to the LED thread's message queue.
- Closes the device and terminates the LED thread before exiting back to VDK. Note that this code runs only when an error is detected in the preceding loop because no way of terminating the program 'cleanly' has been implemented.

Code has been added to the initialization function in `LEDThreadType.c`, which:

- Initializes the SSL's programmable flag manager
- Opens each of the six flags, which in turn control the EZ-KIT Lite LED indicators

- Sets the flag directions to ‘out’
- Clears the flags (turns each indicator off). Corresponding code appears in the termination function to close each flag and terminate the flag manager

The run function of the `LEDThreadType` thread consists of an endless loop that pends on the thread’s message queue waiting for the boot thread to post a message. Then the run function retrieves the message, extracts the row and column information, displays the row and column numbers on the LED indicators, and disposes of the message. The message format is simple since there is only one type of message involved, and all the data required by `LEDThreadType` is packed into the message’s type word. VDK also supports multiple message queues per thread, multiple user-defined message types, and messages with data payloads.

When run, the example program first extinguishes any of the six LED indicators that are lit. The six indicators are labelled `LED 1` through `LED 6` and located in a single row along the short edge of the EZ-KIT Lite board (furthermost from the LCD screen). Then the program waits for one of the keys on the 4 x 4 keypad to be pressed. The program displays that key’s row number (in binary) on LEDs 6–4 and column number on LEDs 3–1. Since row and column numbers are in the range 0–3, which only requires two bits to display, LED 6 and LED 3 should never be lit. The key labelled `ENTER` is at (row 0, column 0). Table 6-1 shows the LEDs that will light for each key.

Table 6-1. Keypad Row/Column Display

Key Label	LED					
	6	5	4	3	2	1
1						
2						

Table 6-1. Keypad Row/Column Display (Cont'd)

Key Label	LED					
	6	5	4	3	2	1
3						
(Arrow up)						
4						
5						
6						
(Arrow down)						
7						
8						
9						
2ND						
CLEAR						
0						
HELP						
ENTER						

7 CREATING A BOOTABLE APPLICATION

In this final chapter, we will step beyond executing programs under the control of the VisualDSP++ Integrated Development and Debugging Environment (IDDE) and learn how to create a program that runs whenever the EZ-KIT Lite is power cycled or reset. The chapter's example program builds on the previous examples to produce an application that displays a slide show of BMP images with optional audio captions, controlled from the EZ-KIT Lite keypad.

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

- Loader utility
- Flash Programmer utility

The chapter includes the following sections.

VisualDSP++ Utility Programs

The loader utility is a VisualDSP++ tool which takes one or more executable (`.dxe`) files and converts them into a loader (`.ldr`) file, in the format required by the processor's boot loader code. The boot loader can access a variety of peripherals and memory types, including flash memory. The Flash Programmer utility is another VisualDSP++ tool that controls the 'burning' of data to flash memories on the EZ-KIT Lite.

Executable and Loadable Program Files

For an application project type (standard application, LwIP Ethernet application, or VDK application), the VisualDSP++ **Project Wizard** offers a choice of project output type on the **Application Settings** page. The choices are **Executable (.dxe)** or **Loader file (.ldr)**. The project output type controls the end format of a project build. An executable (`.dxe`) file is written in Executable and Linkable Format (ELF), a standard format for executable files, and contains all code and data sections that comprise the program. In addition, an ELF file includes further headers and tables required by the ELF standard. If any program source files are compiled with the debugging option enabled, the executable file also contains tables of line number and variable location information, as required by the ELF-related DWARF standard debugging format.

By contrast, the end result of a loader project is a file (`.ldr`) containing only the program's code and initialized data sections in a simple data stream that the processor's boot code can handle. A description of the boot process, including the format of the boot stream, can be found in the *System Reset and Booting* chapter of the *ADSP-BF54x Blackfin Processor Hardware Reference Manual* in the VisualDSP++ online help.

Executable files are appropriate during the development, testing, and profiling phases of an application's life cycle. The extra information in the ELF/DWARF headers and tables is what enables VisualDSP++ to load the

code and data sections directly in L1, L2, and L3 (DDR) memories and provide such features as breakpoints, data variable display, and statistical profiling. Note that memory space for extra information is not required on the target hardware —the information is retained within VisualDSP++ on the PC.

Once the application is ready for release, the need arises for a loader file—a version of the code and initialized data—to be stored permanently on the target platform and be accessible to the ADSP-BF548 processor's boot code. Once this is done, each time the processor is reset (including power-on), the boot code will copy each application section to the appropriate part of L1, L2, or L3 memory and transfer control to the application's entry point. In this form, the application itself must ensure that all processor controllers and devices are initialized to an appropriate state (if their post-reset state is not sufficient) since there is no VisualDSP++ to perform that service.

Creating a Loadable Program File

A loadable program file (loader file for short) is created by VisualDSP++ in an extra step added to the end of the typical compile, assemble, and link steps of a project build. In the extra step, the loader utility reads the code and data segments from the executable file produced by the linker and writes the segments to a new file in a format that the processor's boot loader understands. The loader utility's operation is configured by options set on the **Loader** pages of the **Project Options** dialog box associated with a project. The **Loader** pages (also called loader property pages) are labelled **Options**, **Kernel**, and **Splitter** and accessed as nodes in the **Project Options** tree control. Figure 7-1 shows the **Loader: Options** page.

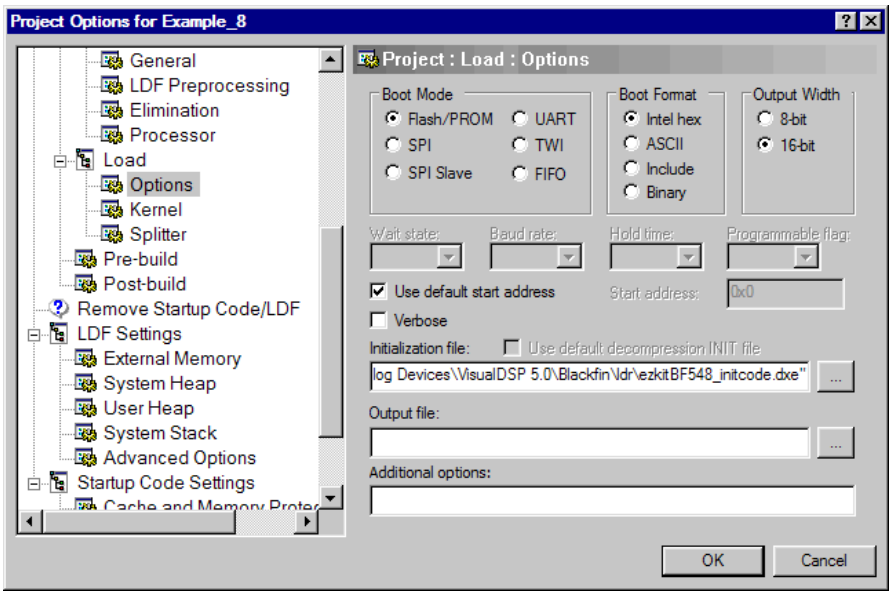

Figure 7-1. Loader: Options Dialog Box

On the **Loader: Options** page, the options grouped under the **Boot Mode** label select the memory or peripheral from which the boot loader will read the loader file. The boot loader can read from a UART, SPI-connected device, and host port among other options; a common choice is to place the loader file in flash memory.

The options under **Boot Format** choose between several representations of the loader file. Intel-hex is a widely used file representation and should be selected for loader files destined for flash memory.

The boot loader can read from eight- or 16-bit wide memories or devices—select the appropriate width option under the **Output Width** label. The burst flash memory on the ADSP-BF548 EZ-KIT Lite is 16 bits wide.

The boot stream processed by the processor's boot loader code can contain more than one program. The boot loader copies the first program into memory and calls the program's entry point. If the program returns, the boot loader copies the next program in memory and calls that program's entry point, and so on.

The **Initialization** file text box allows you to enter or browse for the name of an executable file for insertion in the loader file ahead of the project's executable file. This is one way to initialize the processor's hardware prior to an application being loaded and run. VisualDSP++ provides a default initialization executable file for the ADSP-BF548 EZ-KIT Lite hardware setup, including setting the DDR memory controller's registers. Consequently, this allows you to boot load and execute portions of your application in L3 DDR. The default initialization executable file can be found in `<install_path>\Blackfin\ldr\exkitBF548_initcode.dxe`.

For more information about the loader utility, including the loader property pages, refer to the *VisualDSP++ Loader and Utilities Manual* in the VisualDSP++ online help.

Writing a Loader File to Flash Memory

Following the procedures in this section will overwrite the current contents of the burst flash memory on your ADSP-BF548 EZ-KIT Lite. Review the contents of the `Readme.txt` file in `<install_path>\Blackfin\Examples\ADSP-BF548 EZ-KIT Lite\Power_On_Self_Test` to learn how to build and restore the factory contents of the burst flash memory if you wish to do so.

Once a loader file containing a boot stream (an optional initialization executable followed by your application) is written, the next step is to put the loader file in a location where the processor's boot loader can find the file. As mentioned previously, the boot code can process boot streams from a variety of memories and devices. The most convenient location for hold-

ing a boot stream on an ADSP-BF548 EZ-KIT Lite is the 32 MB, 16-bit wide burst flash memory. VisualDSP++ includes a Flash Programmer utility to program the loader file into flash memory ready for booting.

From the **Tools** menu, click **Flash Programmer** to open a three-page dialog box. Figure 7-2 shows the **Driver** page, the initial setup page of the dialog box. The **Flash Programmer** manages the overall flash programming process but relies on a driver, running on the ADSP-BF548 processor, to issue the appropriate commands to erase and write flash sectors or blocks. VisualDSP++ provides drivers for the EZ-KIT Lite flash memories. You specify the driver file on the **Driver** page of the **Flash Programmer** dialog box. For the burst flash memory, use the **Browse** command to navigate to `<install_path>\Blackfin\Examples\ADSP-BF548 EZ-KIT Lite\Flash Programmer\Burst\BF548EzFlashDriver_PC28f128.dxe`, then click **Load Driver**. After a few seconds, the message "`Success: Driver loaded.`" will appear in the **Message** center text box.

Now move to the next page by clicking the **Programming** tab of the dialog box (Figure 7-3). Flash memory must be erased before it can be re-programmed. The **Pre-program erase options** control the erasing process and specify flash memory sectors to erase. You can erase only sectors required to hold the new loader file, the entire flash memory, sectors listed on the right-hand side of the page, or none at all. **Erase affected** is the appropriate choice for simple flash programming tasks.

The middle panel of Figure 7-3 is where you specify the format of the file holding the data to be programmed into flash memory. This should match the format you direct the loader utility to create, with **Intel Hex** being the most commonly used format.

Now use the **Browse** command under the **Data file** text box to select the loader file created by your VisualDSP++ project. You can instruct the flash driver to check that the loader file data is being programmed correctly, on a sector-by-sector basis, by clicking the **Verify while programming** check box. Or, you can use the **Compare** command once programming is com-

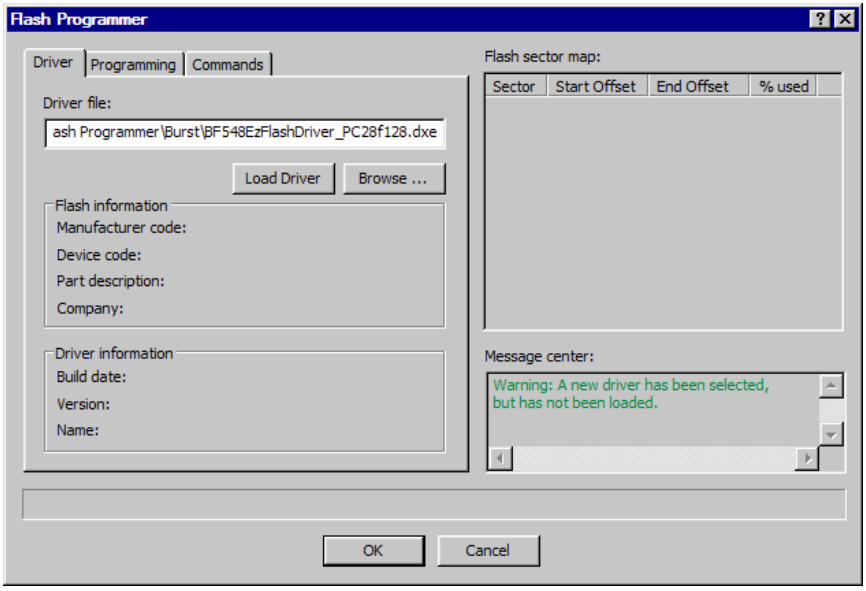

Figure 7-2. Flash Programmer: Driver Dialog Box

plete. Check your selections and click the **Program** command. After a short while, the message indicates that the erase and program operations have been successful. The **%used** column in the sector map list shows how much of each sector has been filled.

This completes the process of programming your loader file in flash memory. The third page of the **Flash Programmer** dialog box, **Commands**, enables such operations as erasing some or all of the flash memory when you have no new data to program.

For more information about the Flash Programmer, refer to the VisualDSP++ online help under **Graphical Environment –>Emulation Tools –>Flash Programmer**.

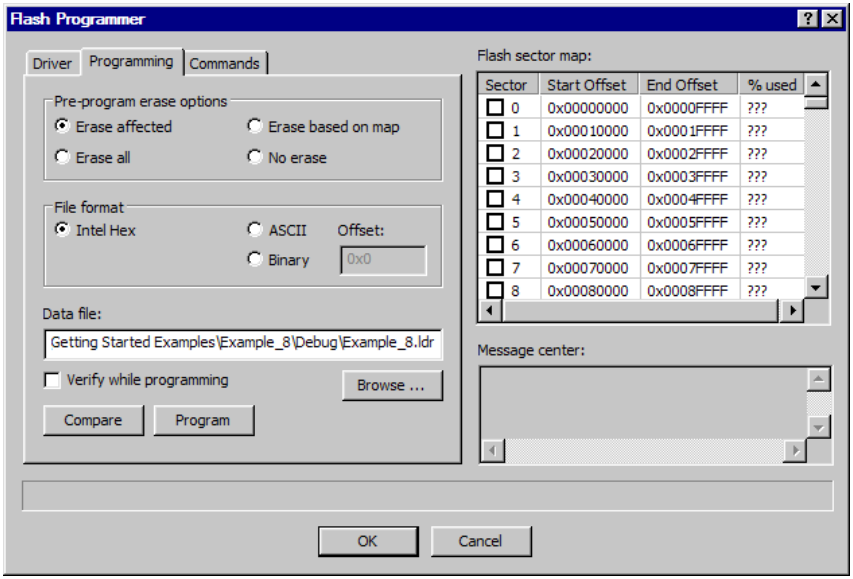

Figure 7-3. Flash Programmer: Programming Dialog Box

Booting From Burst Flash Memory

The BMODE0–3 input pins of the processor select the location accessed by the processor's boot loader at reset. On the ADSP-BF548 EZ-KIT Lite, rotary switch SW1 (labelled DSP BOOT) sets the values of the BMODE0–3 pins. SW1 position 1 (default) selects the burst flash memory. When SW1 is in position 1, every processor reset, including power-on, causes the boot loader to start processing the contents of flash memory as a boot stream.

Example 8: Loading

This chapter's example project (`Example_8.dpj`) reuses much of the code from previous examples to create a simple VDK and SSL/device driver based application that displays a series of `.bmp` images on the LCD screen. Each image has an optional audio caption, held in a `.wav` file and played through the audio output socket whenever the image is selected. Image selection is under user control from the keypad.

Note that you must have built and run the example 5 project according to the instructions in Chapter 4, "Using ADSP-BF548 EZ-KIT Lite As A Mass Storage Device" on page 4-1, in order for example 8 to have images to show and captions to play.

The `Example_8.dpj` project is supplied ready to create a loader file, as described in "Creating a Loadable Program File" on page 7-3. The `Example_8.dxe` file from which the loader file is built is retained by the build process, however. After building the project, but before using the Flash Programmer, use VisualDSP++ to load the executable file into the EZ-KIT Lite via the **File–>Load Program** menu command. Now you can run the program to check that the application works.

The application first displays an instruction screen, as shown in Figure 7-4.

The instruction screen shows the action associated with each of four buttons on the EZ-KIT Lite matrix keypad. Note that the main purpose of the **Redisplay Current Image** action is to replay any audio caption associated with the current `.bmp` file. Also note that exiting the program does not stop the processor: each VDK application thread terminates, but the VDK system's idle thread continues running.

Figure 7-4. Example 8 Instruction Screen

If the example works as expected, you can burn the loader file to burst flash memory as described in "Writing a Loader File to Flash Memory" on page 7-5. The application will be booted and run every time the EZ-KIT Lite is reset, assuming that the board is not connected to a VisualDSP++ session.

Epilogue

We hope that you have enjoyed exploring the ADSP-BF548 EZ-KIT Lite and learning about some of VisualDSP++ features. Refer to the VisualDSP++ online help for more tutorial material and reference information. Refer to the Analog Devices Web site (`www.analog.com`) for informative Application Notes and Engineer to Engineer articles.

Good luck with your future Blackfin projects!

I INDEX

F

G

H

I

J

K

L

M

N