VisualDSP++™ 3.5
Getting Started Guide for 16-Bit Processors

Revision 1.0, October 2003

Part Number
82-000035-05

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

ANALOG DEVICES

Copyright Information

Printed in the USA.

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

Contents

PREFACE

FEATURES AND TOOLS

BASIC TUTORIAL

ADVANCED TUTORIAL

PREFACE

Thank you for purchasing VisualDSP++™, the development software for Analog Devices processors.

Purpose of This Manual

The *VisualDSP++ 3.5 Getting Started Guide for 16-Bit Processors* provides basic and advanced tutorials that highlight many VisualDSP++ features. By completing the step-by-step procedures, you will become familiar with the VisualDSP++ environment and learn how to use these features in your own digital signal processing (DSP) development projects.

Intended Audience

This manual is intended for DSP programmers who are familiar with Analog Devices processors. The manual assumes that the audience has a working knowledge of Analog Devices processor architecture and instruction set.

DSP programmers who are unfamiliar with Analog Devices processors should refer to their processor's Hardware Reference and Instruction Set Reference, which describe the processor architecture and instruction set. Note that the *ADSP-BF533 Blackfin Processor Hardware Reference* includes information about the ADSP-BF531 and ADSP-BF532 processors.

Manual Contents

This guide contains the following chapters.

- Chapter 1, "Features and Tools"

  Provides an overview of VisualDSP++ features and code development tools

- Chapter 2, "Basic Tutorial"

  Provides step-by-step instructions for creating sessions, and for building and debugging projects by using examples of C/C++ and assembly sources

  The tutorial is organized to follow the steps that you take in developing a typical programming project. Before you begin actual programming, you should be familiar with the architecture of your particular processor and the other software development tools.

- Chapter 3, "Advanced Tutorial"

  Provides step-by-step instructions for using profile-guided optimization (PGO) and background telemetry channel (BTC).

What's New in This Manual

A new "Advanced Tutorial" chapter has been added to the guide. This chapter covers some of the more advanced VisualDSP++ features and techniques.

Technical or Customer Support

You can reach DSP Tools Support in the following ways.

- Visit the DSP Development Tools website at `http://www.analog.com/technology/dsp/developmentTools/index.html`
- Email questions to `dsptools.support@analog.com`
- Phone questions to **1-800-ANALOGD**
- Contact your ADI 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 name "*Blackfin*" refers to a family of Analog Devices 16-bit, embedded processors. VisualDSP++ currently supports the following Blackfin processors.

ADSP-BF531

ADSP-BF532 (formerly ADSP-21532)

ADSP-BF533

ADSP-BF535 (formerly ADSP-21535)

ADSP-BF561

AD6532

VisualDSP++ currently supports the following ADSP-21xx processors.

ADSP-2181

ADSP-2183

ADSP-2184/84L/84N

ADSP-2185/85L/85M/85N

ADSP-2186/86L/86M/86N

ADSP-2187L/87N

ADSP-2188L/88N

ADSP-2189M/89N

ADSP-2191

ADSP-2192-12

ADSP-2195

ADSP-2196

ADSP-21990

ADSP-21991

ADSP-21992

Product Information

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

Analog Devices is online at `www.analog.com`. Our website 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 website that allows customization of a webpage to display only the latest information on products you are interested in. You can also choose to receive weekly email notification containing updates to the webpages 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 email address.

DSP Product Information

For information on digital signal processors, visit our website at `www.analog.com/dsp`, 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.

- Email questions or requests for information to
  `dsp.support@analog.com`

- Fax questions or requests for information to
  1-781-461-3010 (North America)
  089/76 903-557 (Europe)

- Access the Digital Signal Processing Division's FTP website at
  `ftp ftp.analog.com` or **ftp 137.71.23.21**
  `ftp://ftp.analog.com`

Related Documents

For information on product related development software, see the following publications.

VisualDSP++ 3.5 User's Guide for 16-Bit Processors

VisualDSP++ 3.5 C/C++ Compiler and Library Manual for Blackfin Processors

VisualDSP++ 3.5 C Compiler and Library Manual for ADSP-218x Processors

VisualDSP++ 3.5 C/C++ Compiler and Library Manual for ADSP-219x Processors

VisualDSP++ 3.5 Assembler and Preprocessor Manual for Blackfin Processors

VisualDSP++ 3.5 Assembler and Preprocessor Manual for ADSP-21xx Processors

VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors

VisualDSP++ 3.5 Loader Manual for 16-Bit Processors

VisualDSP++ 3.5 Product Bulletin

VisualDSP++ Kernel (VDK) User's Guide

VisualDSP++ Component Software Engineering User's Guide

Quick Installation Reference Card

For hardware information, refer to your processor's *Hardware Reference*, *Programming Reference*, and data sheet.

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

Online Technical Documentation

Online documentation comprises VisualDSP++ Help system and tools manuals, Dinkum Abridged C++ library and 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 for the tools manuals are also provided.

A description of each documentation file type is as follows.

File	Description
`.CHM`	Help system files and VisualDSP++ tools manuals.
`.HTML`	Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the `.HTML` files require a browser, such as Internet Explorer 4.0 (or higher).
`.PDF`	VisualDSP++ tools manuals in Portable Documentation Format, one `.PDF` file for each manual. Viewing and printing the `.PDF` files require 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 rerunning the Tools installation.

Access the online documentation from the VisualDSP++ environment, Windows Explorer, or Analog Devices website.

From VisualDSP++

- Access VisualDSP++ online Help from the Help menu's **Contents**, **Search**, and **Index** commands.
- Open online Help from context-sensitive user interface items (toolbar buttons, menu commands, and windows).

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 files) are located in the Help folder, and .PDF files are located in the Docs folder of your VisualDSP++ installation. The Docs folder also contains the Dinkum Abridged C++ library and FlexLM network license manager software documentation.

Using Windows Explorer

- Double-click any file that is part of the VisualDSP++ documentation set.
- Double-click the vdsp-help.chm file, which is the master Help system, to access all the other .CHM files.

Using the Windows Start Button

- Access VisualDSP++ online Help by clicking the **Start** button and choosing **Programs**, **VisualDSP**, and **VisualDSP++ Documentation**.
- Access the .PDF files by clicking the **Start** button and choosing **Programs**, **VisualDSP**, **Documentation for Printing**, and the name of the book.

From the Web

To download the tools manuals, point your browser at:

www.analog.com/technology/dsp/developmentTools/gen_purpose.html

Select a DSP 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.

VisualDSP++ Documentation Set

VisualDSP++ manuals may be purchased through Analog Devices Customer Service at **1-781-329-4700**; ask for a Customer Service representative. The manuals can be purchased only as a kit. For additional information, call **1-603-883-2430**.

If you do not have an account with Analog Devices, you will be referred to Analog Devices distributors. To get information on our distributors, log onto `http://www.analog.com/salesdir/continent.asp`.

Hardware Manuals

Hardware reference and instruction set reference manuals can be ordered through the Literature Center or downloaded from the Analog Devices website. The phone number is **1-800-ANALOGD** (**1-800-262-5643**). The manuals can be ordered by a title or by product number located on the back cover of each manual.

Data Sheets

All data sheets can be downloaded from the Analog Devices website. As a general rule, any data sheet with a letter suffix (L, M, N) can be obtained from the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**) or downloaded from the website. Data sheets without the suffix can be downloaded from the website only—no hard copies are available. You can ask for the data sheet by a part name or by product number.

If you want to have a data sheet faxed to you, the phone number for that service is **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. Call the Literature Center first to find out if requested data sheets are available.

Contacting DSP Publications

Please send your comments and recommendations for improving our manuals and online Help. You can contact us by sending an email to:

```
dsp.techpubs@analog.com
```

Notation Conventions

The following table identifies and describes text conventions used in this manual.

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

Example	Description
Close command (**File** menu) or **OK**	Text in **bold** style indicates 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`.
`[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 ellipsis; read the example as an optional comma-separated list of `this`.
`.SECTION`	Registers, commands, directives, keywords, code examples, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
ⓘ	A note, providing information of special interest or identifying a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
🛇	A caution, providing information about critical design or programming issues that influence operation of a product. In the online version of this book, the word **Caution** appears instead of this symbol.

1 FEATURES AND TOOLS

This chapter contains the following topics.

- "VisualDSP++ Features" on page 1-1
- "New Features in Release 3.5" on page 1-5
- "Code Development Tools" on page 1-7

VisualDSP++ Features

VisualDSP++ provides the following features.

- **Extensive editing capabilities.** Create and modify source files by using multiple language syntax highlighting, drag-and-drop, bookmarks, and other standard editing operations. View files generated by the *code development* tools.
- **Flexible project management.** Specify a project definition that identifies the files, dependencies, and tools that you will use to build projects. Create this project definition once or modify it to meet changing development needs.

- **Easy access to code development tools.** Analog Devices provides these code development tools: C/C++ compiler, assembler, linker, splitter, and loader. Specify options for these tools by using dialog boxes instead of complicated command line scripts. Options that control how the tools process inputs and generate outputs have a one-to-one correspondence to command line switches. Define options for a single file or for an entire project. Define these options once or modify them as necessary.

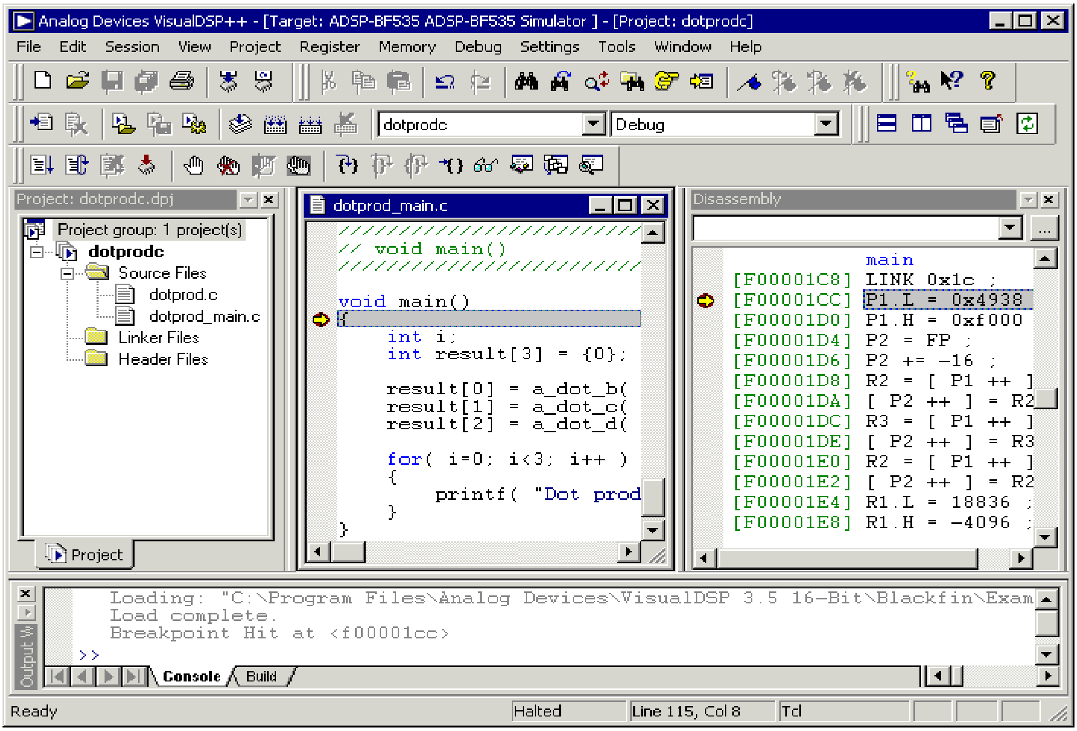
- **Flexible project build options.** Control builds at the file or project level. VisualDSP++ enables you to build files or projects selectively, update project dependencies, or incrementally build only the files that have changed since the previous build. View the status of your project build in progress. If the build reports an error, double-click on the file name in the error message to open that source file. Then correct the error, rebuild the file or project, and start a debug session.

- **VisualDSP++ Kernel (VDK) Support**. Add VDK support to a project to structure and scale application development. The **Kernel** tab page of the **Project** window enables you to manipulate events, event bits, priorities, semaphores, and thread types.

- **Flexible workspace management.** Create up to ten workspaces and quickly switch between them. Assigning a different project to each workspace enables you to build and debug multiple projects in a single session.

- **Easy movement between debug and build activities.** You start the debug session and move freely between editing, build, and debug activities.

Figure 1-1 shows the Integrated Development and Debugging Environment.


Analog Devices VisualDSP++ - [Target: ADSP-BF535 ADSP-BF535 Simulator] - [Project: dotprodc]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
dotprodc
Debug
Project: dotprodc.dpj
Project group: 1 project(s)
dotprodc
Source Files
dotprod.c
dotprod_main.c
Linker Files
Header Files
Project
dotprod_main.c
Disassembly
Output Window
Console
Build
Ready
Halted
Line 115, Col 8
Tcl

```
//////////////////////////
// void main()
//////////////////////////

void main()
{
    int i;
    int result[3] = {0};

    result[0] = a_dot_b(
    result[1] = a_dot_c(
    result[2] = a_dot_d(

    for( i=0; i<3; i++ )
    {
        printf( "Dot prod
    }
}
```

```
                       main
[F00001C8] LINK 0x1c ;
[F00001CC] P1.L = 0x4938
[F00001D0] P1.H = 0xf000
[F00001D4] P2 = FP ;
[F00001D6] P2 += -16 ;
[F00001D8] R2 = [ P1 ++ ]
[F00001DA] [ P2 ++ ] = R2
[F00001DC] R3 = [ P1 ++ ]
[F00001DE] [ P2 ++ ] = R3
[F00001E0] R2 = [ P1 ++ ]
[F00001E2] [ P2 ++ ] = R2
[F00001E4] R1.L = 18836 ;
[F00001E8] R1.H = -4096 ;
```

```
   Loading: "C:\Program Files\Analog Devices\VisualDSP 3.5 16-Bit\Blackfin\Exam
   Load complete.
   Breakpoint Hit at <f00001cc>
>>
```

Figure 1-1. The VisualDSP++ IDDE

VisualDSP++ reduces your debugging time by providing these key features.

- **Easy-to-use debugging activities.** Debug with one common, easy-to-use interface for all processor simulators and emulators, or hardware evaluation and development boards. Switch easily between these targets.
- **Multiple language support.** Debug programs written in C, C++, or assembly, and view your program in machine code. For programs written in C/C++, you can view the source in C/C++ or mixed C/C++ and assembly, and display the values of local variables or evaluate expressions (global and local) based on the current context.
- **Effective debug control.** Set breakpoints on symbols and addresses and then step through the program's execution to find problems in coding logic. Set watchpoints (conditional breakpoints) on registers, stacks, and memory locations to identify when they are accessed.
- **Tools for improving performance.** Use the trace, profile, and linear and statistical profiles to identify bottlenecks in your DSP application and to identify program optimization needs. Use plotting to view data arrays graphically. Generate interrupts, outputs, and inputs to simulate real-world application conditions.

New Features in Release 3.5

Release 3.5 includes the following new features and enhancements.

- **New processor support**. The ADSP-BF561 processor is supported by this software version. Refer to the *ADSP-BF561 Blackfin Hardware Reference* and chip data sheet for details.

- **Multiple project support**. VisualDSP++ provides the ability to switch among multiple open projects in the same IDDE session. The **Project** window displays active projects.

- **Data streaming and logging**. VisualDSP++ now offers the ability to stream and log data from a target DSP without halting the DSP. The IDDE takes advantage of this capability in plot windows. If the target supports background telemetry channel (BTC), the plot window is updated while the target is running.

- **License management in the IDDE**. License management (installation and validation) has been integrated into the VisualDSP++ IDDE. Installing a FlexLM license server is still handled by the separate installation application.

- **Profile-guided optimization (PGO) in the IDDE**. VisualDSP++ includes facilities to run common PGO scenarios simply and also provides a mechanism for advanced applications that require more control over the profiling process via scripting. The techniques relies on setting up and executing data sets to produce an optimized application.

- **Integrated Source Code Control** (**SCC**). VisualDSP++ uses the Microsoft Common Source Code Control (MCSCC) interface to provide a connection from the IDDE to SCC applications (such as Visual SourceSafe, PVCS Version Manager, and ClearCase) installed on your machine. You can now conveniently access commonly-used SCC features from VisualDSP++ without leaving the IDDE. Advanced and application-specific SCC features not available from the IDDE must be run directly from the SCC applications.

- **Automation aware scripting engine**. VisualDSP++ includes a scripting engine that uses the Microsoft ActiveX script host framework. The engine enables you to use multiple scripting languages (such as VBScript, JavaScript, and so on) to access the VisualDSP++ Automation API.

  You can interact with the IDDE by using a single command or a script file similar to the Tcl scripting functionality, which was available in previous versions of VisualDSP++.

- **Profiling code with Expert Linker**. You can use Expert Linker to profile object sections in a program. When the program halts, Expert Linker graphically displays how much time was spent in each object section. You can use this display to locate code “hotspots” and then move that code to faster, internal memory.

- **Address bar in Disassembly and Memory windows**. When enabled, an address bar is displayed in **Disassembly** windows and memory windows. You can use the address bar to navigate by address, symbol, or expression. The address bar maintains a most recently used history of visited locations.

- **Menus with Icons**. Icons now appear beside menu commands that have corresponding toolbar buttons.

Code Development Tools

Code development tools for 16-bit processors include:

- C/C++ compiler
- Runtime library with over 100 math, DSP, and C runtime library routines
- Assembler
- Linker
- Splitter
- Loader
- Simulator
- Emulator (must be purchased separately from VisualDSP++)

These tools enable you to develop applications that take full advantage of your processor's architecture.

The VisualDSP++ linker supports multiprocessing, shared memory, and memory overlays.

The code development tools provide the following key features.

- **Easy-to-program C, C++, and assembly languages.** Program in C/C++, assembly, or mix C/C++ and assembly in one source. The assembly language is based on an algebraic syntax that is easy to learn, program, and debug.
- **Flexible system definition.** Define multiple types of executables for a single type of processor in one Linker Description File (`.LDF`). Specify input files, including objects, libraries, shared memory files, overlay files, and executables.
- **Support for overlays, multiprocessors, and shared memory executables.** The linker places code and resolves symbols in multiprocessor memory space for use by multiprocessor systems. The loader enables you to configure multiprocessors with less code and faster boot time. Create host, link port, and PROM boot images.

Software and hardware tool kits include context-sensitive Help and manuals in PDF format.

For details about assembly syntax, refer to the *VisualDSP++ 3.5 Assembler and Preprocessor Manual* for your target processor.

2 BASIC TUTORIAL

This chapter contains the following topics.

Overview

The Basic Tutorial demonstrates key features and capabilities of the VisualDSP++ Integrated Development and Debugging Environment (IDDE). The exercises use sample programs written in C, C++, and assembly for Blackfin processors.

You can use different Blackfin processors with only minor changes to the Linker Description Files (`.LDFs`) included with each project. VisualDSP++ includes basic Linker Description Files for each processor type in the `ldf folder`. For Blackfin processors, the folder's default installation path is:

```
Program Files\Analog Devices\VisualDSP 3.5 16-Bit\Blackfin\ldf
```

The source files for these exercises are installed during the VisualDSP++ software installation.

The tutorial contains five exercises:

- In **Exercise One**, you will start up VisualDSP++, build a project containing C source code, and profile the performance of a C function.
- In **Exercise Two**, you will create a new project, create a Linker Description File to link with the assembly routine, rebuild the project, and profile the performance of the assembly language routine.
- In **Exercise Three**, you will plot the various waveforms produced by a Finite Impulse Response (FIR) algorithm.
- In **Exercise Four**, you will use linear profiling to examine the efficiency of the FIR algorithm used in Exercise Three. Using the collected linear profile data, you will pinpoint the most time-consuming areas of the algorithm, which are likely to require hand tuning in the assembly language.
- In **Exercise Five**, you will install a VCSE component on your system and add the component to the project. Then you will build and run the program with the component.

The ADSP-BF53x Family Simulator and ADSP-BF535 processor are used for all exercises.

Tip: Become familiar with the VisualDSP++ toolbar buttons, shown in Figure 2-1 on page 2-3. They are shortcuts for menu commands such as **Open** a file and **Run** a program. Toolbar buttons and menu commands that are not available for tasks that you want to perform are disabled and displayed in gray.

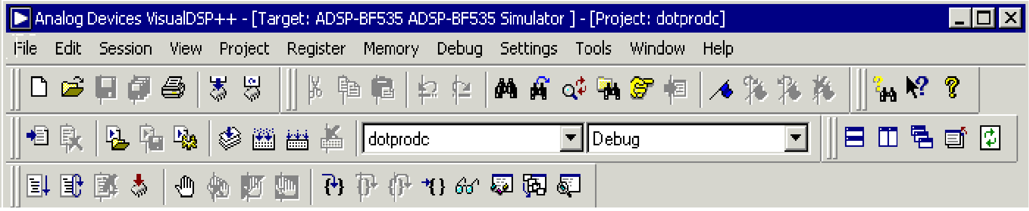

Analog Devices VisualDSP++ - [Target: ADSP-BF535 ADSP-BF535 Simulator] - [Project: dotprodc]
File
Edit
Session
View
Project
Register
Memory
Debug
Settings
Tools
Window
Help
dotprodc
Debug

Figure 2-1. VisualDSP++ Toolbar Buttons

VisualDSP++ is a licensed software product. To run the software, you must have a valid license installed on your system. If you try to run VisualDSP++ and a license is not installed, a message window opens to let you add a license. For details about license management, see the *VisualDSP++ 3.5 User's Guide for 16-Bit Processors* or the VisualDSP++ online Help.

Exercise One: Building and Running a C Program

In this exercise, you will:

- Start up the VisualDSP++ environment
- Open and build an existing project
- Examine windows and dialog boxes
- Run the program

The sources for this exercise are in the `dot_product_c` folder. The default installation path is:

```
Program Files\Analog Devices\VisualDSP 3.5 16-Bit\Blackfin\
Examples\Tutorial\dot_product_c
```

Step 1: Start VisualDSP++ and Open a Project

To start VisualDSP++ and open a project:

1. Click the Windows **Start** button and select **Programs**, **Analog Devices**, **VisualDSP++ 3.5 for 16-bit Processor**, and **VisualDSP++ Environment.**

   If you are running VisualDSP++ for the first time, the **New Session** dialog box (Figure 2-6 on page 2-11) opens to enable you to set up a session.

   a. Select the values shown in Table 2-1.

Table 2-1. Session Specification

Box	Value
Debug Target	ADSP-BF53x Family Simulator
Platform	ADSP-BF535 Simulator
Session Name	ADSP-BF535 ADSP-BF535 Simulator
Processor	ADSP-BF535

   b. Click **OK**. The VisualDSP++ main window appears.

   If you have already run VisualDSP++ and the **Reload last project at startup** option is selected on the **Project** page under **Settings** and **Preferences**, VisualDSP++ opens the last project that you worked on. To close this project, choose **Close** from the **Project** menu, and then click **No** when prompted to save the project. Since you have made no changes to the project, you do not have to save it.

2. From the **Project** menu, choose **Open**.

   VisualDSP++ displays the **Open Project** dialog box.

3. In the **Look in** box, open the `Program Files\Analog Devices` folder and double-click the following subfolders in succession.

   ```
   VisualDSP 3.5 16-Bit\Blackfin\Examples\Tutorial\
   dot_product_c
   ```

This path is based on the default installation.

4. Double-click the `dotprodc` project (`.dpj`) file.

   VisualDSP++ loads the project in the **Project** window, as shown in Figure 2-2. The environment displays messages in the **Output** window as it processes the project settings and file dependencies.

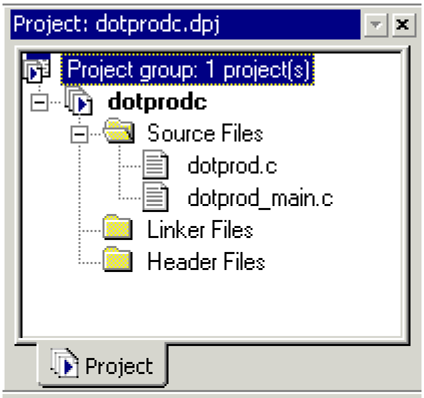
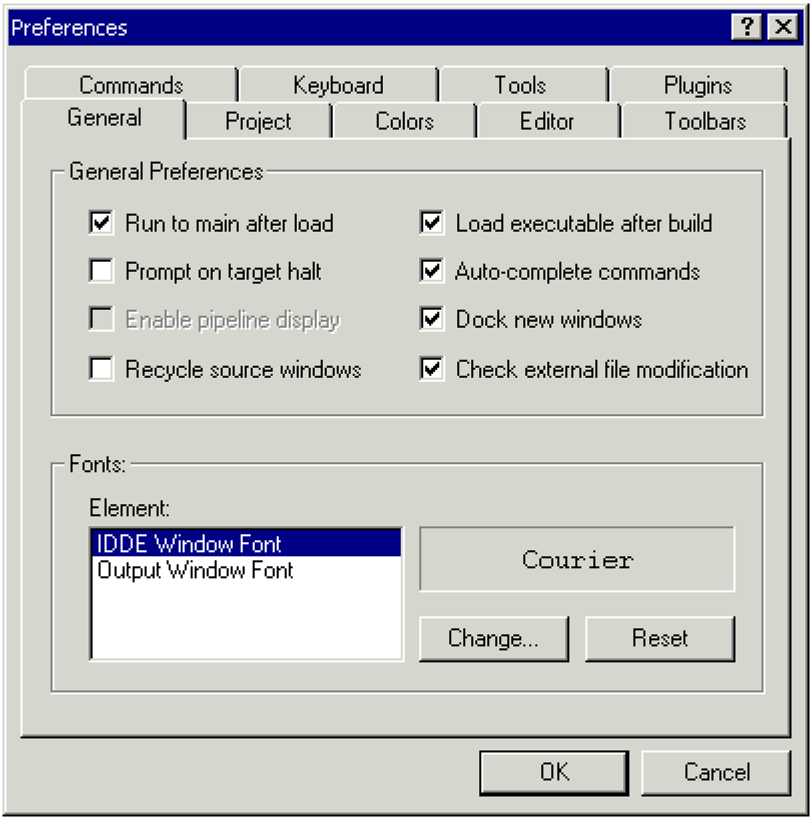


Figure 2-2. Project Loaded in the Project Window

The `dotprodc` project comprises two C language source files, `dotprod.c` and `dotprod_main.c`, which define the arrays and calculate their dot products.

5. From the **Settings** menu, choose **Preferences** to open the **Preferences** dialog box, shown in Figure 2-3.



General Preferences

☑ Run to main after load

☐ Prompt on target halt

☐ Enable pipeline display

☐ Recycle source windows

☑ Load executable after build

☑ Auto-complete commands

☑ Dock new windows

☑ Check external file modification

Fonts:

Element:

IDDE Window Font

Output Window Font

Courier

Figure 2-3. Preferences Dialog Box

6. On the **General** page, under **General Preferences**, make sure that the following options are selected.

    - **Run to main after load**
    - **Load executable after build**

7. Click **OK** to close the **Preferences** dialog box.

   The VisualDSP++ main window appears. You are now ready to build the project.

Step 2: Build the dotprodc Project

To build the `dotprodc` project:

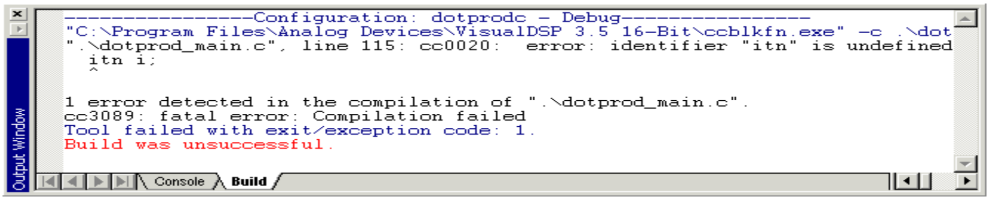
1. From the **Project** menu, choose **Build Project**.

   VisualDSP++ first checks and updates the project dependencies and then builds the project by using the project source files.

   As the build progresses, the **Output** window displays status messages (error and informational) from the tools. For example, when a tool detects invalid syntax or a missing reference, the tool reports the error in the **Output** window.

   If you double-click the file name in the error message, VisualDSP++ opens the source file in an editor window. You can then edit the source to correct the error, rebuild, and launch the debug session. If the project build is up-to-date (the files, dependencies, and options have not changed since the last project build), no build is performed unless you run the **Rebuild All** command. Instead, you see the message "`Project is up to date`." If the build has no errors, a message reports "`Build completed successfully`."

In this example (Figure 2-4) notice that the compiler detects an undefined identifier and issues the following error in the **Output** window.



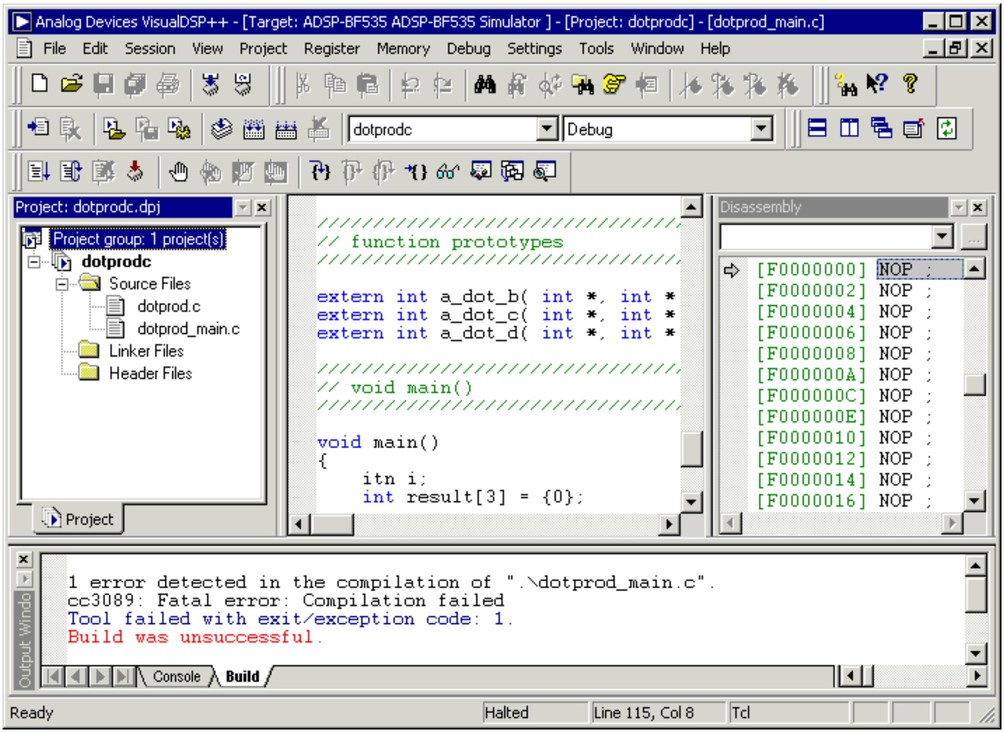
```
----------------Configuration: dotprodc - Debug----------------
"C:\Program Files\Analog Devices\VisualDSP 3.5 16-Bit\ccblkfn.exe" -c .\dot
".\dotprod_main.c", line 115: cc0020:  error: identifier "itn" is undefined
    itn i;
    ^

1 error detected in the compilation of ".\dotprod_main.c".
cc3089: fatal error: Compilation failed
Tool failed with exit/exception code: 1.
Build was unsuccessful.
```

Figure 2-4. Example of Error Message

2. Double-click the error message text in the **Output** window.

   VisualDSP++ opens the C source file `dotprod_main.c` in an editor window and places the cursor on the line that contains the error (see Figure 2-5 on page 2-9).

   The editor window in Figure 2-5 shows that the integer variable declaration `int` has been misspelled as `itn`.


Analog Devices VisualDSP++ - [Target: ADSP-BF535 ADSP-BF535 Simulator] - [Project: dotprodc] - [dotprod_main.c]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
dotprodc
Debug
Project: dotprodc.dpj
Project group: 1 project(s)
dotprodc
Source Files
dotprod.c
dotprod_main.c
Linker Files
Header Files
Project
Disassembly
Output Window
Console
Build
Ready
Halted
Line 115, Col 8
Tcl

```
/////////////////////////////////
// function prototypes
/////////////////////////////////

extern int a_dot_b( int *, int *
extern int a_dot_c( int *, int *
extern int a_dot_d( int *, int *

/////////////////////////////////
// void main()
/////////////////////////////////

void main()
{
    itn i;
    int result[3] = {0};
```

```
[F0000000] NOP ;
[F0000002] NOP ;
[F0000004] NOP ;
[F0000006] NOP ;
[F0000008] NOP ;
[F000000A] NOP ;
[F000000C] NOP ;
[F000000E] NOP ;
[F0000010] NOP ;
[F0000012] NOP ;
[F0000014] NOP ;
[F0000016] NOP ;
```

```
1 error detected in the compilation of ".\dotprod_main.c".
cc3089: Fatal error: Compilation failed
Tool failed with exit/exception code: 1.
Build was unsuccessful.
```

Figure 2-5. Output Window and Editor Window

3. In the editor window, click on `itn` and change it to `int`. Notice that `int` is now color coded to signify that it is a valid C keyword.

4. Save the source file by choosing **Save** from the **File** menu.

5. Build the project again by choosing **Build Project** from the **Project** menu. The project is now built without any errors, as reported on the **Build** page in the **Output** window.

Now that you have built your project successfully, you can run the example program.

Step 3: Run the Program

In this procedure, you will:

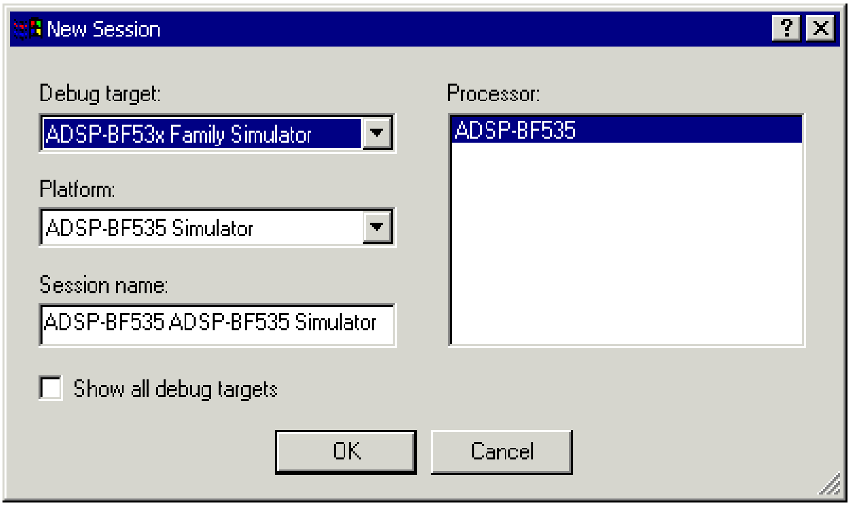
- Set up the debug session before running the program
- View debugger windows and dialog boxes

Since you enabled **Load executable after build** on the **General** page in the **Preferences** dialog box, the executable file `dotprodc.dxe` is automatically downloaded to the target. If the debug session's processor does not match the project's build target, VisualDSP++ reports this discrepancy and asks if you want to select another session before downloading the executable to the target. If VisualDSP++ does not open the **Session List** dialog box, skip steps 1–4.

To set up the debug session:

1. **In the Session List** dialog box, click **New Session** to open the **New Session** dialog box, shown in Figure 2-6.

   For subsequent debugging sessions, use the **New Session** command on the **Sessions** menu to open the **New Session dialog box.**



Debug target: ADSP-BF53x Family Simulator

Platform: ADSP-BF535 Simulator

Session name: ADSP-BF535 ADSP-BF535 Simulator

Processor: ADSP-BF535

□ Show all debug targets

Figure 2-6. New Session Dialog Box

2. Specify the target and processor information listed in the Table 2-2.

Table 2-2. Session Specification

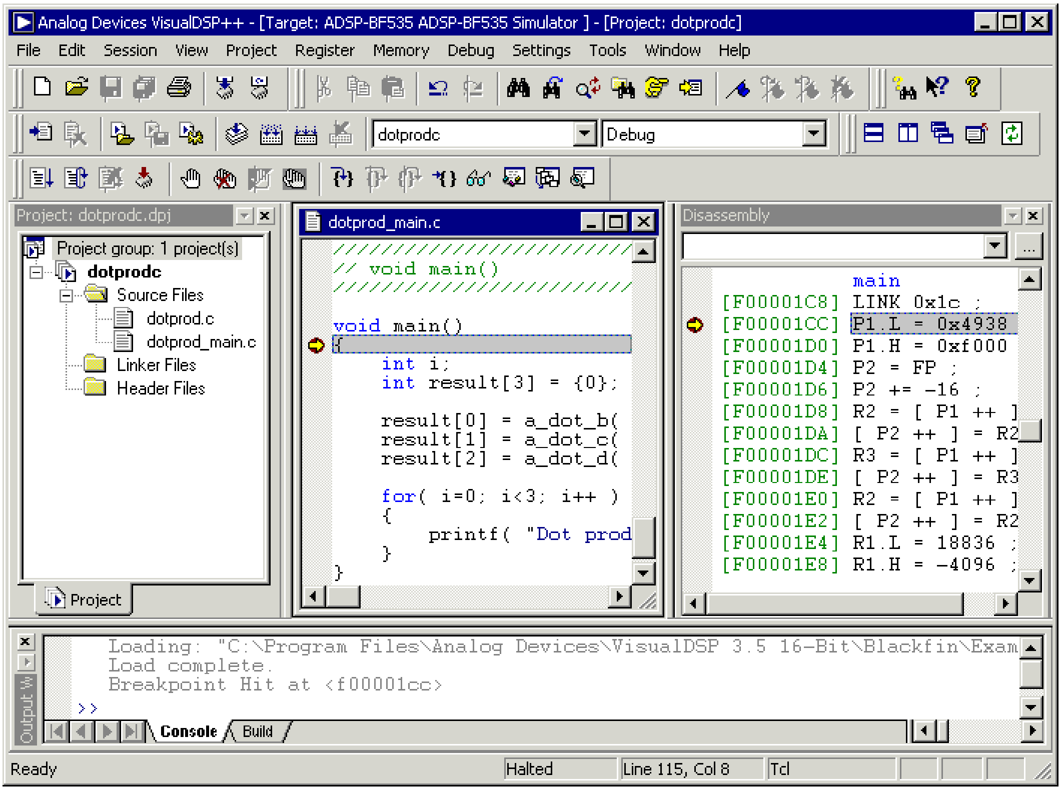
Box	Value
Debug Target	ADSP-BF53x Family Simulator
Platform	ADSP-BF535 Simulator
Session Name	ADSP-BF535 ADSP-BF535 Simulator
Processor	ADSP-BF535

3. Click **OK** to close the **New Session** dialog box and return to the **Session List** dialog box.

4. With the new session name highlighted, click **Activate**.

If you do not click **Activate**, the session mismatch message appears again.

VisualDSP++ closes the **Session List** dialog box, automatically loads your project's executable file (`dotprodc.dxe`), and advances to the main function of your code (see Figure 2-7).


Analog Devices VisualDSP++ - [Target: ADSP-BF535 ADSP-BF535 Simulator] - [Project: dotprodc]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
dotprodc
Debug
Project: dotprodc.dpj
Project group: 1 project(s)
dotprodc
Source Files
dotprod.c
dotprod_main.c
Linker Files
Header Files
Project
dotprod_main.c
Disassembly
Output W
Console
Build
Ready
Halted
Line 115, Col 8
Tcl

```
///////////////////////////
// void main()
///////////////////////////

void main()
{
      int i;
      int result[3] = {0};

      result[0] = a_dot_b(
      result[1] = a_dot_c(
      result[2] = a_dot_d(

      for( i=0; i<3; i++ )
      {
          printf( "Dot prod
      }
}
```

```
                main
[F00001C8] LINK 0x1c ;
[F00001CC] P1.L = 0x4938
[F00001D0] P1.H = 0xf000
[F00001D4] P2 = FP ;
[F00001D6] P2 += -16 ;
[F00001D8] R2 = [ P1 ++ ]
[F00001DA] [ P2 ++ ] = R2
[F00001DC] R3 = [ P1 ++ ]
[F00001DE] [ P2 ++ ] = R3
[F00001E0] R2 = [ P1 ++ ]
[F00001E2] [ P2 ++ ] = R2
[F00001E4] R1.L = 18836 ;
[F00001E8] R1.H = -4096 ;
```

```
    Loading: "C:\Program Files\Analog Devices\VisualDSP 3.5 16-Bit\Blackfin\Exam
    Load complete.
    Breakpoint Hit at <f00001cc>
>>
```

Figure 2-7. Loading dotprodc.dxe

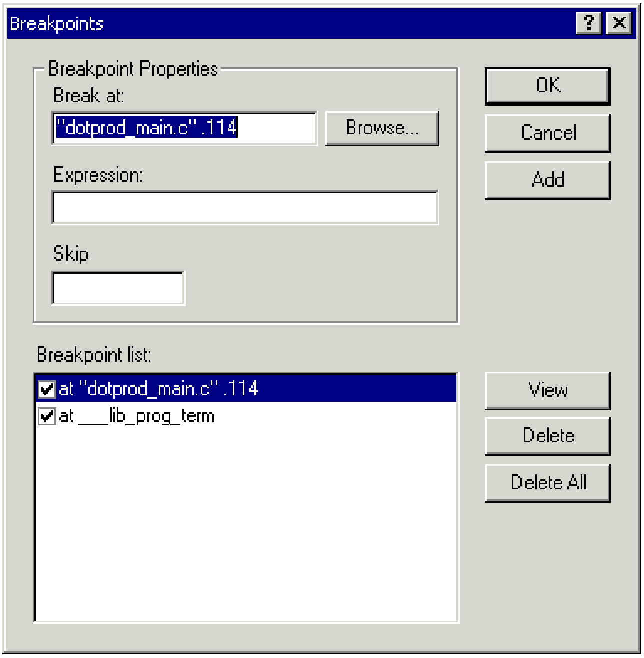
5. Look at the information in the open windows.

   The **Output** window's Console page contains messages about the status of the debug session. In this case, VisualDSP++ reports that the `dotprodc.dxe` load is complete.

   The **Disassembly** window displays the assembly code for the executable. Use the scroll bars to move around the **Disassembly** window.

   Note that a solid red circle and a yellow arrow appear at the start of the program labeled "`main`". The solid red circle ● indicates that a breakpoint is set on that instruction, and the yellow arrow ⇨ indicates that the processor is currently halted at that instruction. When VisualDSP++ loads your C program, it automatically sets two breakpoints, one at the beginning and one at the end of code execution. Your breakpoint locations may differ slightly from those shown in the examples in this book.

6. From the **Settings** menu, choose **Breakpoints** to view the breakpoints set in your program. VisualDSP++ displays the **Breakpoints** dialog box, shown in Figure 2-8 on page 2-14.



Breakpoints

Breakpoint Properties

Break at: "dotprod_main.c".114

Expression:

Skip

Breakpoint list:

☑ at "dotprod_main.c".114

☑ at ___lib_prog_term

Figure 2-8. Breakpoints Dialog Box

The breakpoints are set at these C program labels:

- `at “dotprod_main.c” 114`
- `at __lib_prog_term`

The **Breakpoints** dialog box enables you to view, add, and delete breakpoints and to browse for symbols. In the **Disassembly** and editor windows, double-clicking on a line of code toggles (adds or deletes) breakpoints. In the editor window, however, you must place the mouse pointer in the gutter before double-clicking.

Use these tool buttons to set or clear breakpoints:

- Toggles a breakpoint for the current line
- Clears all breakpoints

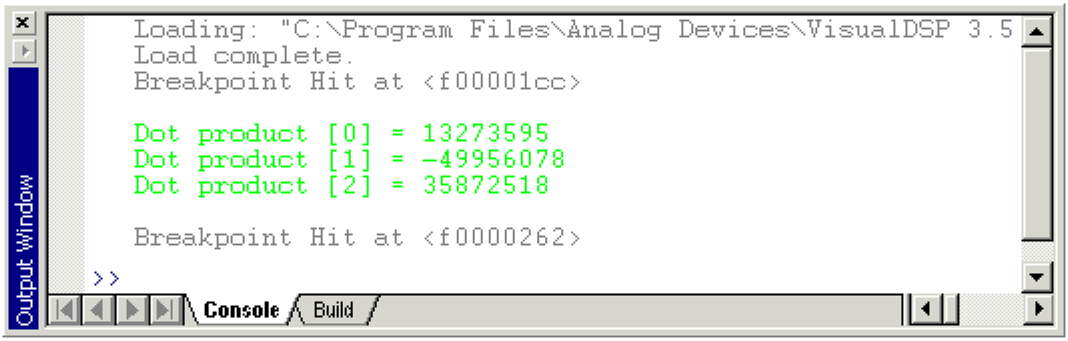
7. Click **OK** or **Cancel** to exit the **Breakpoints** dialog box.

Step 4: Run dotprodc

To run `dotprodc`, click the **Run** button or choose **Run** from the **Debug** menu.

VisualDSP++ computes the dot products and displays the following results on the **Console** page (Figure 2-9) in the **Output** window.

```
Dot product [0] = 13273595
Dot product [1] = -49956078
Dot product [2] = 35872518
```



```
Loading: "C:\Program Files\Analog Devices\VisualDSP 3.5
Load complete.
Breakpoint Hit at <f00001cc>

Dot product [0] = 13273595
Dot product [1] = -49956078
Dot product [2] = 35872518

Breakpoint Hit at <f0000262>
>>
```

Figure 2-9. Results of the dotprodc Program

You are now ready to begin Exercise Two.

Exercise Two: Modifying a C Program to Call an Assembly Routine

In Exercise One, you built and ran a C program. In this exercise, you will:

- Modify the C program to call an assembly language routine
- Create a Linker Description File to link with the assembly routine
- Rebuild the project

The project files are largely identical to those of Exercise One. Minor modifications illustrate the changes needed to call an assembly language routine from C source code.

Step 1: Create a New Project

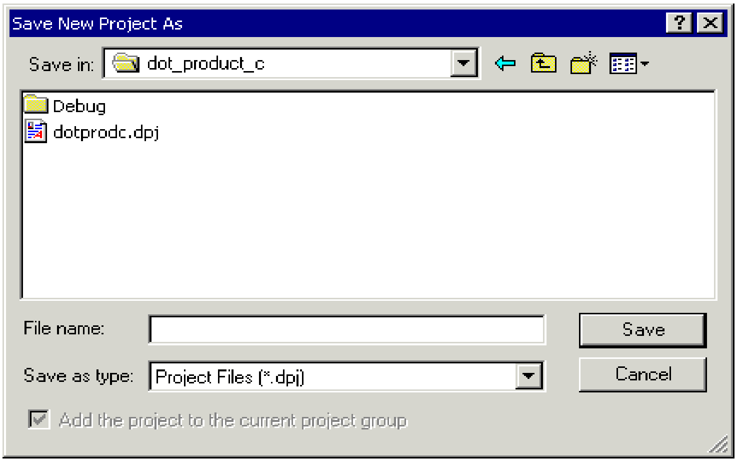
To create a new project:

1. From the **Project** menu, choose **Close** to close the `dotprodc` project.

   Click **Yes** when prompted to close all open source windows.

   If you have modified your project during this session, you are prompted to save the project. Click **No**.

2. From the **Project** menu, choose **New** to open the **Save New Project As** dialog box, shown in Figure 2-10.



Save New Project As

Save in: dot_product_c

Debug

dotprodc.dpj

File name:

Save as type: Project Files (*.dpj)

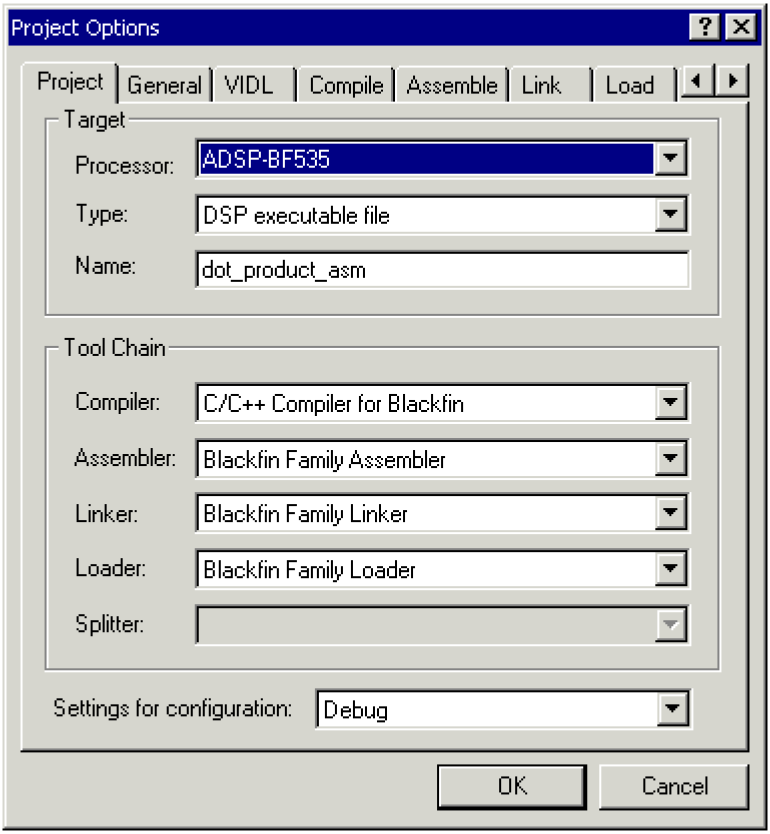
☑ Add the project to the current project group

Figure 2-10. Save New Project As Dialog Box

3. Click the up-one-level button until you locate the `dot_product_asm` folder, and then double-click this folder.

4. In the **File name** box, type `dot_product_asm`, and click **Save**.

   The **Project Options** dialog box (Figure 2-11) appears.



Processor: ADSP-BF535

Type: DSP executable file

Name: dot_product_asm

Compiler: C/C++ Compiler for Blackfin

Assembler: Blackfin Family Assembler

Linker: Blackfin Family Linker

Loader: Blackfin Family Loader

Splitter:

Settings for configuration: Debug

Figure 2-11. Project Options Dialog Box: Project Page

   This dialog box enables you to specify project build information.

5. Take a moment to view the tabbed pages in the **Project Options** window: **Project**, **General**, **Compile**, **Assemble**, **Link**, **Load**, **Compiled Simulation**, and **Post Build**. On each page, you specify the tool options used to build the project.

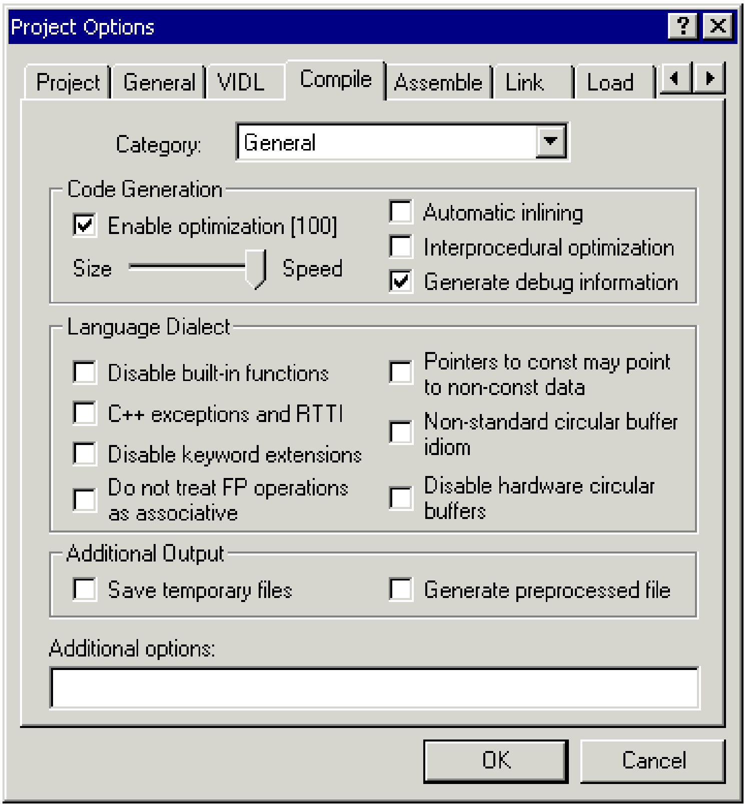
6. On the **Project** page (Figure 2-11 on page 2-18), specify the values shown in Table 2-3.

Table 2-3. Completing the Project Page

Box	Value
Processor	ADSP-BF535
Type	DSP executable file
Name	dot_product_asm
Settings for configuration	Debug

These settings specify information for building an executable file for the ADSP-BF535 processor. The executable contains debug information, so you can examine program execution.

7. Click the **Compile** tab to display the **Compile** page, shown in Figure 2-12 on page 2-20.



Category: General

Code Generation

☑ Enable optimization [100]

Size — Speed

☐ Automatic inlining

☐ Interprocedural optimization

☑ Generate debug information

Language Dialect

☐ Disable built-in functions

☐ C++ exceptions and RTTI

☐ Disable keyword extensions

☐ Do not treat FP operations as associative

☐ Pointers to const may point to non-const data

☐ Non-standard circular buffer idiom

☐ Disable hardware circular buffers

Additional Output

☐ Save temporary files

☐ Generate preprocessed file

Additional options:

Figure 2-12. Project Options Dialog Box: Compile Page

8. Complete the **General** group box as follows.

    a. Select the **Enable optimization** check box to enable optimization.

    b. Select the **Generate debug information** check box, if it is not already selected, to enable debug information for the C source.

These settings direct the C compiler to optimize code for the ADSP-BF535 processor. Because the optimization takes advantage of DSP architecture and assembly language features, some of the C debug information is not saved. Debugging is, therefore, performed through debug information at the assembly language level.

9. Click **OK** to apply changes to the project options and to close the **Project Options** dialog box. When prompted to add support for the VisualDSP++ kernel, click **No**.

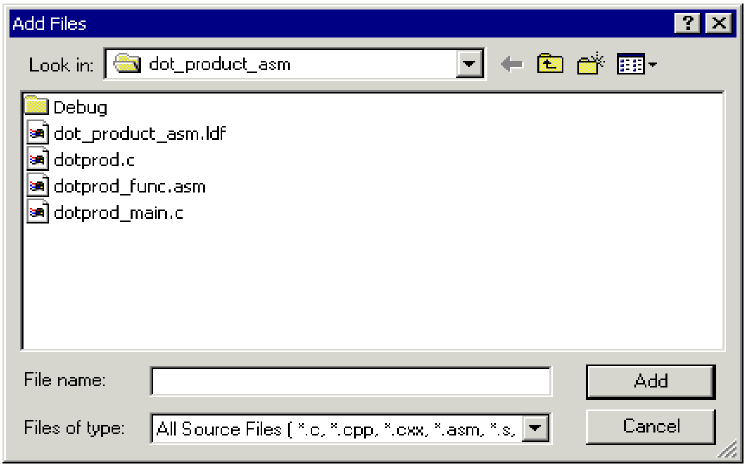
You are now ready to add the source files to the project.

Step 2: Add Source Files to dot_product_asm

To add the source files to the new project:

1. Click the **Add File** button , or from the **Project** menu, choose **Add to Project**, and then choose **File(s)**.

   The **Add Files** dialog box (Figure 2-13) appears.



Look in: dot_product_asm

Debug
dot_product_asm.ldf
dotprod.c
dotprod_func.asm
dotprod_main.c

Files of type: All Source Files (*.c, *.cpp, *.cxx, *.asm, *.s,

Figure 2-13. Add Files Dialog Box: Adding Source Files to the Project

2. In the **Look in** box, locate the project folder, `dot_product_asm`.

3. In the **Files of type** box, select **All Source Files** from the drop-down list.

4. Hold down the **Ctrl** key and click `dotprod.c` and `dotprod_main.c`. Then click **Add**.

   To display the files that you added in step 4, open the `Source Files` folder in the **Project** window.

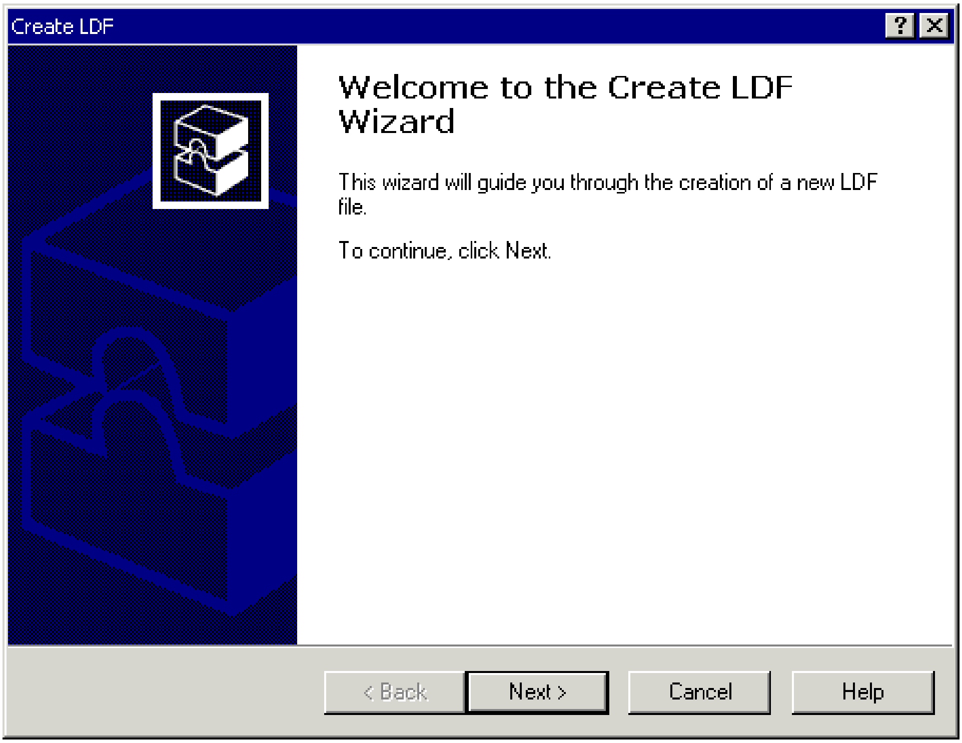
You are now ready to create a Linker Description File for the project.

Step 3: Create a Linker Description File for the Project

In this procedure, you will use the Expert Linker to create a Linker Description File for the project.

To create a Linker Description File:

1. From the **Tools** menu, choose **Expert Linker** and then choose **Create LDF** to open the **Create LDF Wizard**, shown in Figure 2-14 on page 2-23.



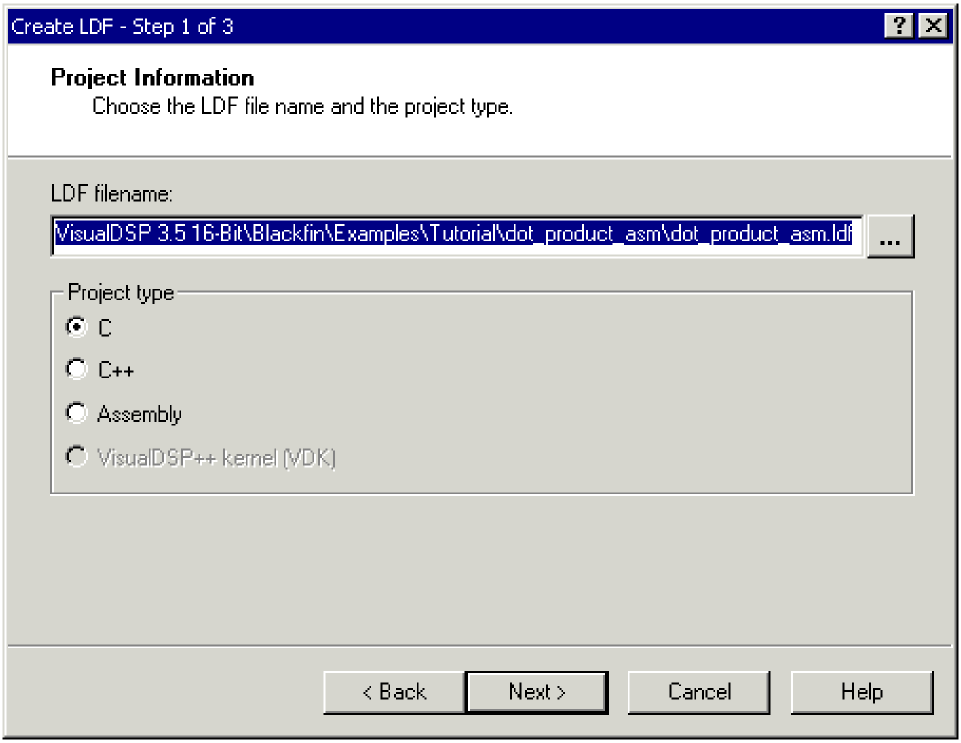
Welcome to the Create LDF Wizard

This wizard will guide you through the creation of a new LDF file.

To continue, click Next.

Figure 2-14. Create LDF Wizard

2. Click **Next** to display the **Create LDF – Step 1 of 3** page, shown in Figure 2-15.



Project Information

Choose the LDF file name and the project type.

LDF filename:

VisualDSP 3.5 16-Bit\Blackfin\Examples\Tutorial\dot_product_asm\dot_product_asm.ldf

Project type

● C

○ C++

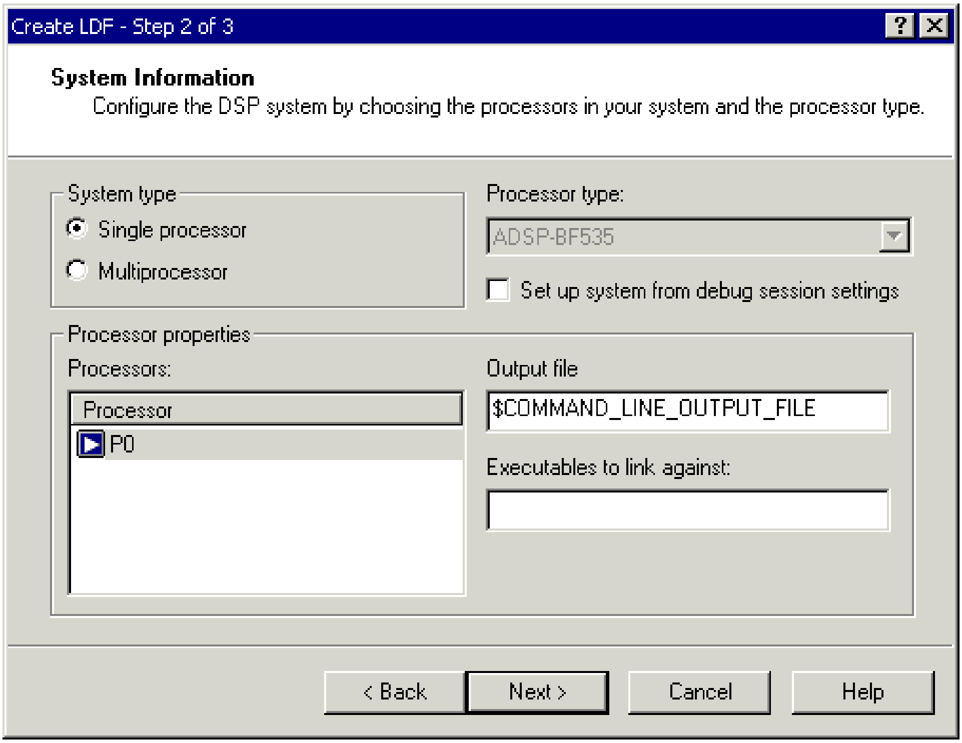
○ Assembly

○ VisualDSP++ kernel (VDK)

Figure 2-15. Create LDF – Step 1 of 3 Page

This page enables you to assign the **LDF file name** (based on the project name) and to select the **Project type**.

3. Accept the values selected for your project and click **Next** to display the **Create LDF – Step 2 of 3** page, shown in Figure 2-16.



Create LDF - Step 2 of 3

System Information

Configure the DSP system by choosing the processors in your system and the processor type.

System type

● Single processor

○ Multiprocessor

Processor type: ADSP-BF535

□ Set up system from debug session settings

Processor properties

Processors:

Processor
P0

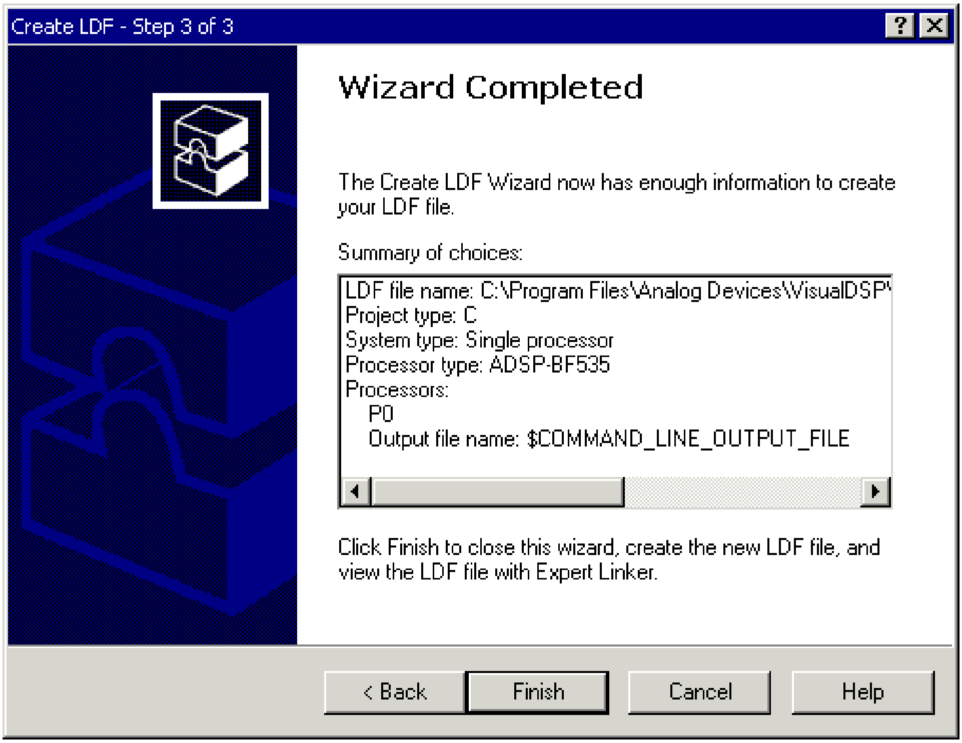
Output file: $COMMAND_LINE_OUTPUT_FILE

Executables to link against:

Figure 2-16. Create LDF – Step 2 of 3 Page

This page enables you to set the **System type** (defaulted to **Single processor**), the **Processor type** (defaulted to **ADSP-BF535** to match the project), and the name of the linker **Output file** (defaulted to the name selected by the project).

4. Accept the default values and click **Next** to display the **Create LDF – Step 3 of 3** page, shown in Figure 2-17.



The Create LDF Wizard now has enough information to create your LDF file.

Summary of choices:

```
LDF file name: C:\Program Files\Analog Devices\VisualDSP\
Project type: C
System type: Single processor
Processor type: ADSP-BF535
Processors:
   P0
   Output file name: $COMMAND_LINE_OUTPUT_FILE
```

Click Finish to close this wizard, create the new LDF file, and view the LDF file with Expert Linker.

Figure 2-17. Create LDF – Step 3 of 3 Page

5. Review the **Summary of choices** and click **Finish** to create the `.LDF` file.

   You now have a new `.LDF` file in your project. The new file is in the **Linker Files** folder in the **Project** window.

   The **Expert Linker** window opens with a representation of the `.LDF` file that you created. This file is complete for this project. Close the **Expert Linker** window.

6. Click the **Rebuild All** button to build the project. The C source file opens in an editor window, and execution halts.

The C version of the project is now complete. You are now ready to modify the sources to call the assembly function.

Step 4: Modify the Project Source Files

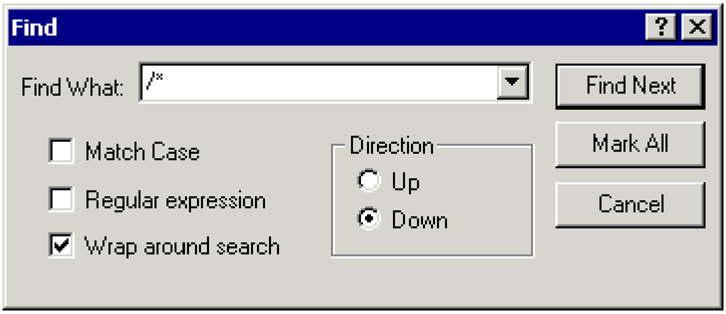
In this procedure, you will:

- Modify `dotprod_main.c` to call `a_dot_c_asm` instead of `a_dot_c`
- Save the modified file

To modify `dotprod_main.c` to call the assembly function:

1. Resize or maximize the editor window for better viewing.

2. From the **Edit** menu, choose **Find** to open the **Find** dialog box, shown in Figure 2-18.



Find What: /*

☐ Match Case

☐ Regular expression

☑ Wrap around search

Direction

○ Up

● Down

Figure 2-18. Find Dialog Box: Locating Occurrences of /*

3. In the **Find What** box, type `/*`, and then click **Mark All**.

   The editor bookmarks all lines containing `/*` and positions the cursor at the first instance of `/*` in the `extern int a_dot_c_asm` declaration.

4. Select the comment characters /* and use the **Ctrl+X** key combination to cut the comment characters from the beginning of the a_dot_c_asm declaration. Then move the cursor up one line and use the **Ctrl+V** key combination to paste the comment characters at the beginning of the a_dot_c declaration. Because syntax coloring is turned on, the code will change color as you cut and paste the comment characters.

   Repeat this step for the end-of-comment characters */ at the end of the a_dot_c_asm declaration. The a_dot_c declaration is now fully commented out, and the a_dot_c_asm declaration is no longer commented.

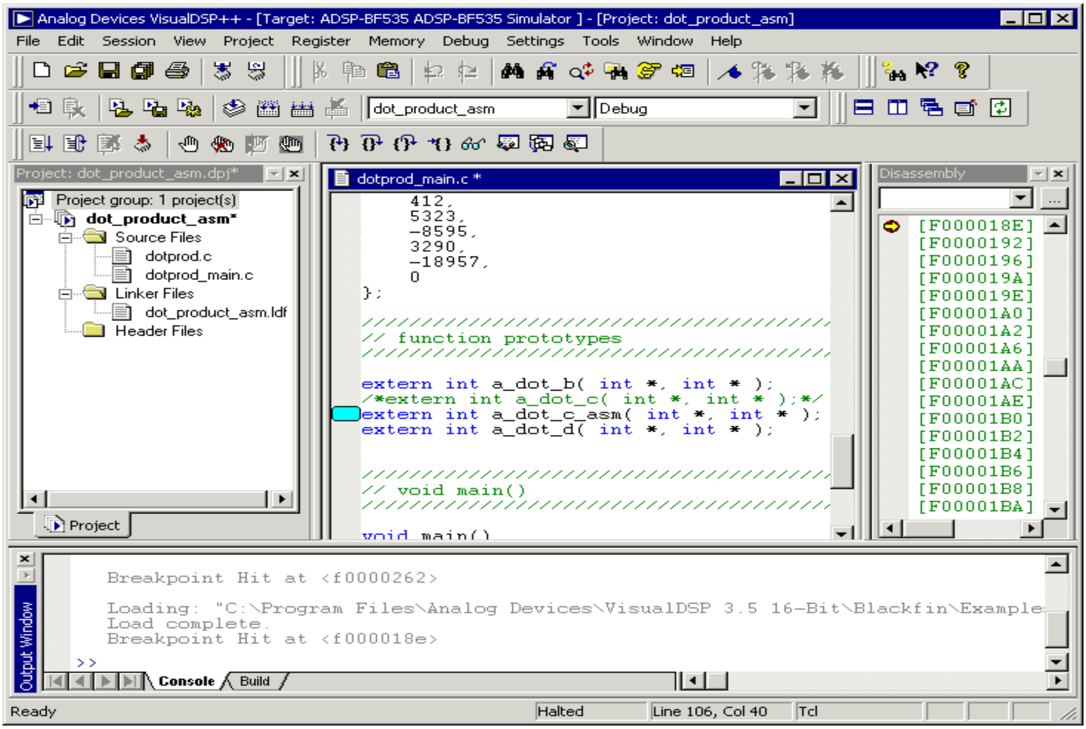
5. Press **F3** to move to the next bookmark.

   The editor positions the cursor on the /* in the function call to a_dot_c_asm, which is currently commented out. Note that the previous line is the function call to the a_dot_c routine.

6. Press **Ctrl+X** to cut the comment characters from the beginning of the function call to a_dot_c_asm. Then move the cursor up one line and press **Ctrl+V** to paste the comment characters at the beginning of the call to a_dot_c.

   Repeat this step for the end-of-comment characters */. The main() function should now be calling the a_dot_c_asm routine instead of the a_dot_c function, previously called in Exercise One.

   Figure 2-19 on page 2-30 shows the changes made in step 6.


Analog Devices VisualDSP++ - [Target: ADSP-BF535 ADSP-BF535 Simulator] - [Project: dot_product_asm]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
dot_product_asm
Debug
Project: dot_product_asm.dpj*
Project group: 1 project(s)
dot_product_asm*
Source Files
dotprod.c
dotprod_main.c
Linker Files
dot_product_asm.ldf
Header Files
Project
dotprod_main.c *
Disassembly
Output Window
Console
Build
Ready
Halted
Line 106, Col 40
Tcl

```
         412,
         5323,
         -8595,
         3290,
         -18957,
         0
};

//////////////////////////////////////////
// function prototypes
//////////////////////////////////////////

extern int a_dot_b( int *, int * );
/*extern int a_dot_c( int *, int * );*/
extern int a_dot_c_asm( int *, int * );
extern int a_dot_d( int *, int * );


//////////////////////////////////////////
// void main()
//////////////////////////////////////////

void main()
```

```
[F000018E]
[F0000192]
[F0000196]
[F000019A]
[F000019E]
[F00001A0]
[F00001A2]
[F00001A6]
[F00001AA]
[F00001AC]
[F00001AE]
[F00001B0]
[F00001B2]
[F00001B4]
[F00001B6]
[F00001B8]
[F00001BA]
```

```
Breakpoint Hit at <f0000262>

Loading: "C:\Program Files\Analog Devices\VisualDSP 3.5 16-Bit\Blackfin\Example
Load complete.
Breakpoint Hit at <f000018e>
>>
```

Figure 2-19. Editor Window: Modifying dotprod_main.c to Call a_dot_c_asm

7. From the **File** menu, choose **Save** to save the changes to the file.

8. Place the cursor in the editor window. Then, from the **File** menu, choose **Close** to close the `dotprod_main.c` file.

You are now ready to modify `dotprodasm.ldf`.

Step 5: Use the Expert Linker to modify dot_prod_asm.ldf

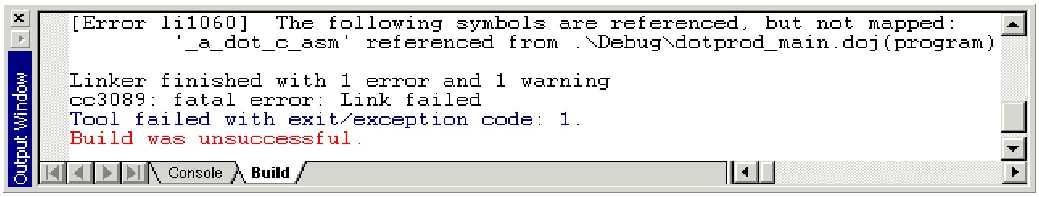
In this procedure you will:

- View the Expert Linker representation of the `.LDF` file that you created
- Modify the `.LDF` file to map in the section for the `a_dot_c_asm` assembly routine

To examine and then modify `dot_prod_asm.ldf` to link with the assembly function:

1. Click the **Add File** button .
2. Select `dotprod_func.asm` and click **Add**.
3. Try to build the project by performing one of these actions:
    - Click the **Build Project** button .
    - From the **Project** menu, choose **Build Project**.

    Notice the linker error in the **Output** window, shown in Figure 2-20.

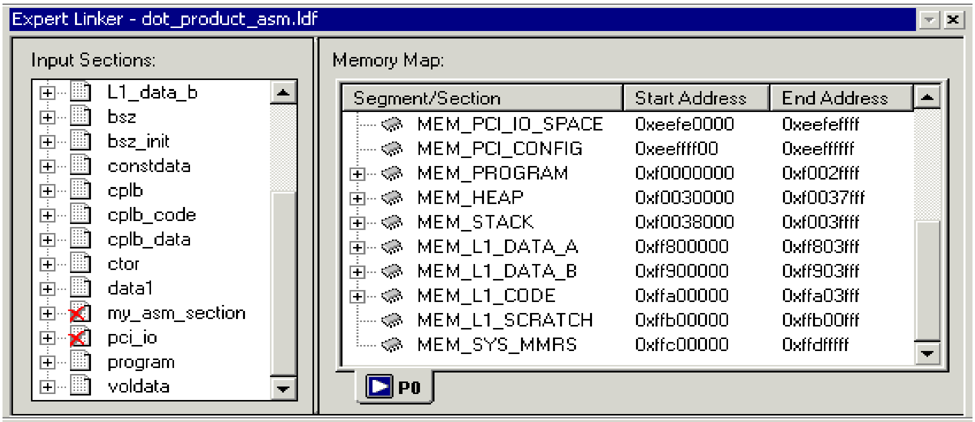


```
[Error li1060]  The following symbols are referenced, but not mapped:
        '_a_dot_c_asm' referenced from .\Debug\dotprod_main.doj(program)

Linker finished with 1 error and 1 warning
cc3089: fatal error: Link failed
Tool failed with exit/exception code: 1.
Build was unsuccessful.
```

Figure 2-20. Output Window: Linker Error

4. In the **Project** window, double-click in the `dot_prod_asm.ldf` file. The **Expert Linker** window (Figure 2-21) opens with a graphical representation of your file.

   Resize the window to expand your view and change the view mode. To display the tree view shown in Figure 2-21, right-click in the right pane, choose **View Mode**, and then choose **Memory Map Tree**.



Input Sections:

- L1_data_b
- bsz
- bsz_init
- constdata
- cplb
- cplb_code
- cplb_data
- ctor
- data1
- ☒ my_asm_section
- ☒ pci_io
- program
- voldata

Memory Map:

Segment/Section	Start Address	End Address
MEM_PCI_IO_SPACE	0xeefe0000	0xeefeffff
MEM_PCI_CONFIG	0xeeffff00	0xeeffffff
MEM_PROGRAM	0xf0000000	0xf002ffff
MEM_HEAP	0xf0030000	0xf0037fff
MEM_STACK	0xf0038000	0xf003ffff
MEM_L1_DATA_A	0xff800000	0xff803fff
MEM_L1_DATA_B	0xff900000	0xff903fff
MEM_L1_CODE	0xffa00000	0xffa03fff
MEM_L1_SCRATCH	0xffb00000	0xffb00fff
MEM_SYS_MMRS	0xffc00000	0xffdfffff

Figure 2-21. Expert Linker Window

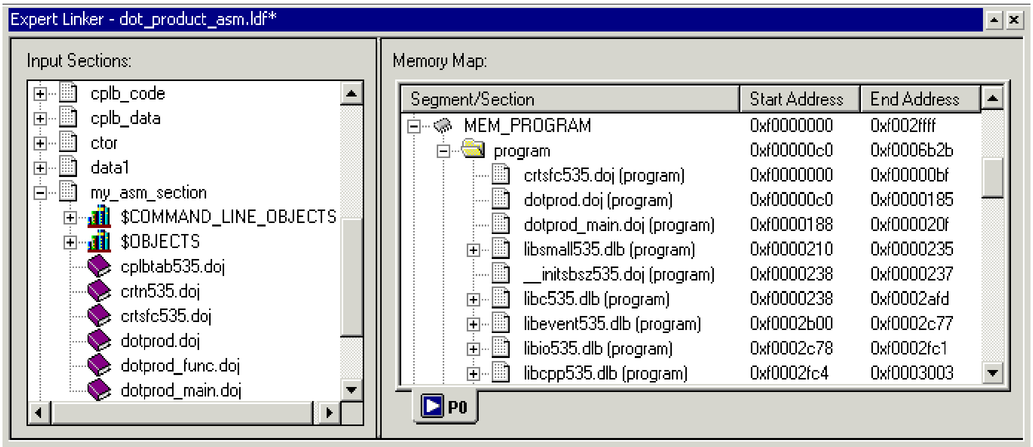
The left pane contains a list of the **Input Sections** that are in your project or are mapped in the `.LDF` file. A red `X` is over the icon in front of the section named "`my_asm_section`" because the Expert Linker has determined that the section is not mapped by the `.LDF` file.

The right pane contains a representation of the memory segments that the Expert Linker defined when it created the `.LDF` file.

5. Map the section `my_asm_section` into the memory segment named `MEM_PROGRAM` as follows.

   Open the my_asm_section input section by clicking on the plus sign in front of it. The input section expands to show that the linker macros `$COMMAND_LINE_OBJECTS` and `$OBJECTS` and the object file `dotprod_func.doj` have a section that has not been mapped. Expand `MEM_PROGRAM` in the memory map pane and drag the icon in front of `$OBJECTS` onto the `program` output section under `MEM_PROGRAM`.

   As shown in Figure 2-22, the red `X` should no longer appear because the section `my_asm_section` has been mapped.



Expert Linker - dot_product_asm.ldf*

Input Sections:

- cplb_code
- cplb_data
- ctor
- data1
- my_asm_section
  - $COMMAND_LINE_OBJECTS
  - $OBJECTS
  - cplbtab535.doj
  - crtn535.doj
  - crtsfc535.doj
  - dotprod.doj
  - dotprod_func.doj
  - dotprod_main.doj

Memory Map:

Segment/Section	Start Address	End Address
MEM_PROGRAM	0xf0000000	0xf002ffff
program	0xf00000c0	0xf0006b2b
crtsfc535.doj (program)	0xf0000000	0xf00000bf
dotprod.doj (program)	0xf00000c0	0xf0000185
dotprod_main.doj (program)	0xf0000188	0xf000020f
libsmall535.dlb (program)	0xf0000210	0xf0000235
__initsbsz535.doj (program)	0xf0000238	0xf0000237
libc535.dlb (program)	0xf0000238	0xf0002afd
libevent535.dlb (program)	0xf0002b00	0xf0002c77
libio535.dlb (program)	0xf0002c78	0xf0002fc1
libcpp535.dlb (program)	0xf0002fc4	0xf0003003

Figure 2-22. Dragging $OBJECTS onto Program Output Section

6. From the **Tools** menu, choose **Expert Linker** and **Save** to save the modified file. Then close the **Expert Linker** window.

   If you forget to save the file and then rebuild the project, VisualDSP++ will see that you modified the file and will automatically save it.

You are now ready to rebuild and run the modified project.

Step 6: Rebuild and Run dot_product_asm

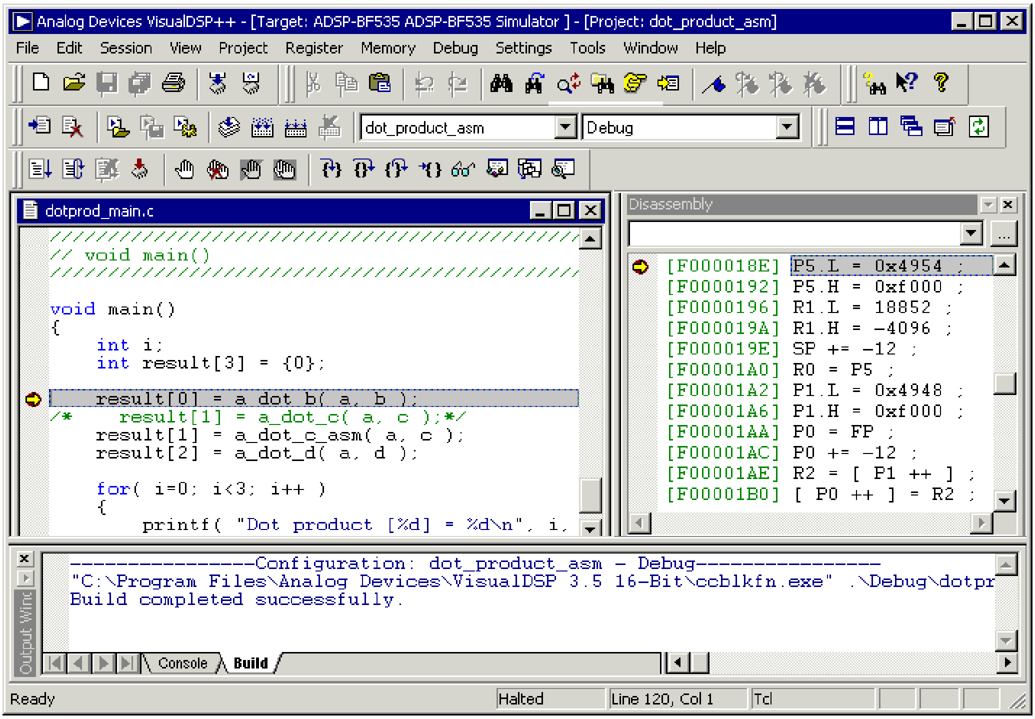
To run `dot_product`:

1. Build the project by performing one of these actions:
   - Click the **Build Project** button .
   - From the **Project** menu, choose **Build Project**.

   At the end of the build, the **Output** window displays the following message on the **Build** page.

   “`Build completed successfully.`”

   VisualDSP++ loads the program, runs to main, and displays the **Output**, **Disassembly**, and editor windows (shown in Figure 2-23 on page 2-35).


Analog Devices VisualDSP++ - [Target: ADSP-BF535 ADSP-BF535 Simulator] - [Project: dot_product_asm]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
dot_product_asm
Debug
dotprod_main.c
Disassembly
Output Wind
Console
Build
Ready
Halted
Line 120, Col 1
Tcl

```
//////////////////////////////////////////////////////////////
// void main()
//////////////////////////////////////////////////////////////

void main()
{
    int i;
    int result[3] = {0};

    result[0] = a_dot_b( a, b );
/*    result[1] = a_dot_c( a, c );*/
    result[1] = a_dot_c_asm( a, c );
    result[2] = a_dot_d( a, d );

    for( i=0; i<3; i++ )
    {
        printf( "Dot product [%d] = %d\n", i,
```

```
[F000018E] P5.L = 0x4954 ;
[F0000192] P5.H = 0xf000 ;
[F0000196] R1.L = 18852 ;
[F000019A] R1.H = -4096 ;
[F000019E] SP += -12 ;
[F00001A0] R0 = P5 ;
[F00001A2] P1.L = 0x4948 ;
[F00001A6] P1.H = 0xf000 ;
[F00001AA] P0 = FP ;
[F00001AC] P0 += -12 ;
[F00001AE] R2 = [ P1 ++ ] ;
[F00001B0] [ P0 ++ ] = R2 ;
```

```
----------------Configuration: dot_product_asm - Debug----------------
"C:\Program Files\Analog Devices\VisualDSP 3.5 16-Bit\ccblkfn.exe" .\Debug\dotpr
Build completed successfully.
```

Figure 2-23. Windows Left Open from the Previous Debugger Session

2. Click the **Run** button to run `dot_product_asm`.

   The program calculates the three dot products and displays the results on the **Console** page in the **Output** window. When the program stops running, the message "`Halted`" appears in the status bar at the bottom of the window. The results, shown below, are identical to the results obtained in Exercise One.

```
Dot product [0] = 13273595
Dot product [1] = -49956078
Dot product [2] = 35872518
```

You are now ready to begin Exercise Three.

Exercise Three: Plotting Data

In this exercise, you will:

- Load and debug a pre-built program that applies a simple Finite Impulse Response (FIR) filter to a buffer of data
- Use VisualDSP++'s plotting engine to view the different data arrays graphically, both before and after running the program

Step 1: Load the FIR Program

To load the FIR program:

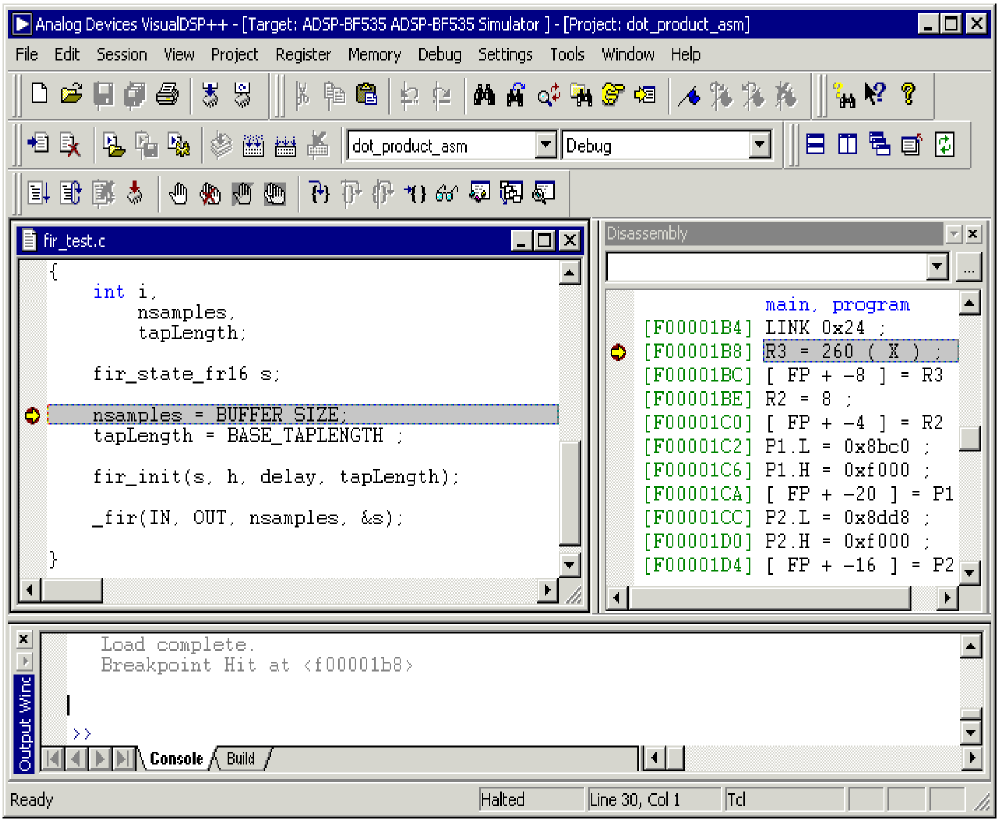
1. Keep the **Disassembly** window and **Console** page (in the **Output** window) open, but close all other windows.
2. From the **File** menu, choose **Load Program** or click . The **Open a Processor Program** dialog box appears.
3. Select the FIR program to load as follows.
   a. Open your `Analog Devices` folder and double-click:

      `VisualDSP 3.5 16-Bit\Blackfin\Examples\Tutorial\fir`

   b. Double-click the `Debug` subfolder.

   c. Double-click `FIR.DXE` to load the program.

      If VisualDSP++ does not open an editor window (shown in Figure 2-24 on page 2-37), right-click in the **Disassembly** window and select **View Source**.


Analog Devices VisualDSP++ - [Target: ADSP-BF535 ADSP-BF535 Simulator] - [Project: dot_product_asm]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
dot_product_asm
Debug
fir_test.c
Disassembly
Output Window
Console
Build
Ready
Halted
Line 30, Col 1
Tcl

```
{
    int i,
        nsamples,
        tapLength;

    fir_state_fr16 s;

    nsamples = BUFFER_SIZE;
    tapLength = BASE_TAPLENGTH ;

    fir_init(s, h, delay, tapLength);

    _fir(IN, OUT, nsamples, &s);

}
```

```
                main, program
[F00001B4] LINK 0x24 ;
[F00001B8] R3 = 260 ( X ) ;
[F00001BC] [ FP + -8 ] = R3
[F00001BE] R2 = 8 ;
[F00001C0] [ FP + -4 ] = R2
[F00001C2] P1.L = 0x8bc0 ;
[F00001C6] P1.H = 0xf000 ;
[F00001CA] [ FP + -20 ] = P1
[F00001CC] P2.L = 0x8dd8 ;
[F00001D0] P2.H = 0xf000 ;
[F00001D4] [ FP + -16 ] = P2
```

```
    Load complete.
    Breakpoint Hit at <f00001b8>

>>
```

Figure 2-24. Loading the FIR Program

4. Look at the source code of the FIR program.

   You can see two global data arrays:

   - `IN`

   - `OUT`

   You can also see one function, `fir`, that operates on these arrays.

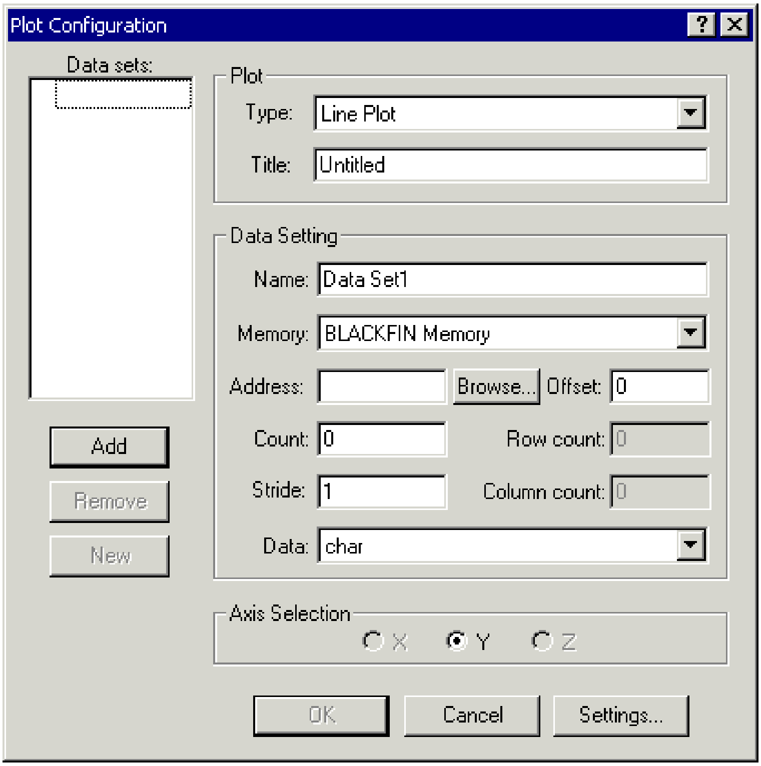
You are now ready to open a plot window.

Step 2: Open a Plot Window

To open a plot window:

1. From the **View** menu, choose **Debug Windows** and **Plot**. Then choose **New** to open the **Plot Configuration** dialog box, shown in Figure 2-25.

   Here you will add the data sets that you want to view in a plot window.



Plot Configuration

Data sets:

Plot

Type: Line Plot

Title: Untitled

Data Setting

Name: Data Set1

Memory: BLACKFIN Memory

Address:

Offset: 0

Count: 0

Row count: 0

Stride: 1

Column count: 0

Data: char

Axis Selection

○ X ● Y ○ Z

Figure 2-25. Plot Configuration Dialog Box

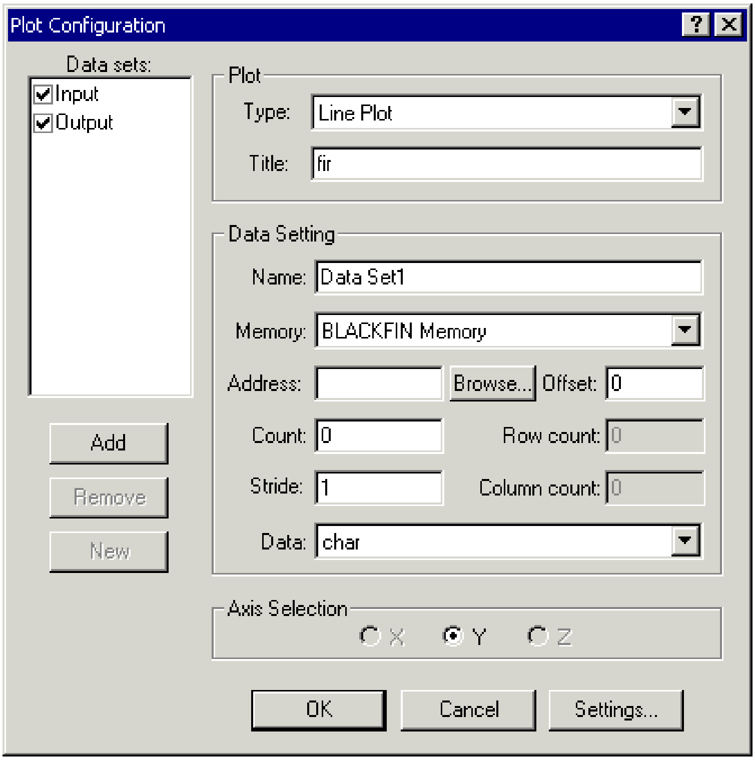
2. In the **Plot** group box, specify the following values.
   - In the **Type** box, select **Line Plot** from the drop-down list.
   - In the **Title** box, type `fir`.
3. Enter two data sets to plot by using the values in Table 2-4.

Table 2-4. Two Data Sets: Input and Output

Box	Input Data Set	Output Data Set	Description
Name	Input	Output	Data set
Memory	BLACKFIN Memory	BLACKFIN Memory	Data memory
Address	IN	OUT	The address of this data set is that of the Input or Output array. Click **Browse** to select the value from the list of loaded symbols.
Count	128	128	The array is 260 elements long, but you are plotting the first 128 elements.
Stride	1	1	The data is contiguous in memory.
Data	short	short	Input and Output are arrays of int values.

After entering each data set, click **Add** to add the data set to the **Data sets** list on the left of dialog box.

The **Plot Configuration** dialog box should now look like the one in Figure 2-26 on page 2-40.



Plot Configuration

Data sets:

☑ Input

☑ Output

Plot

Type: Line Plot

Title: fir

Data Setting

Name: Data Set1

Memory: BLACKFIN Memory

Address:

Offset: 0

Count: 0

Row count: 0

Stride: 1

Column count: 0

Data: char

Axis Selection

○ X ● Y ○ Z

Figure 2-26. Plot Configuration Dialog Box with Input/Output Data Sets

4. Click **OK** to apply the changes and to open a plot window with these data sets.

   The plot window now displays the two arrays. Since, by default, the simulator initializes memory to zero, the Output data set appears as one horizontal line, shown in Figure 2-27 on page 2-41.

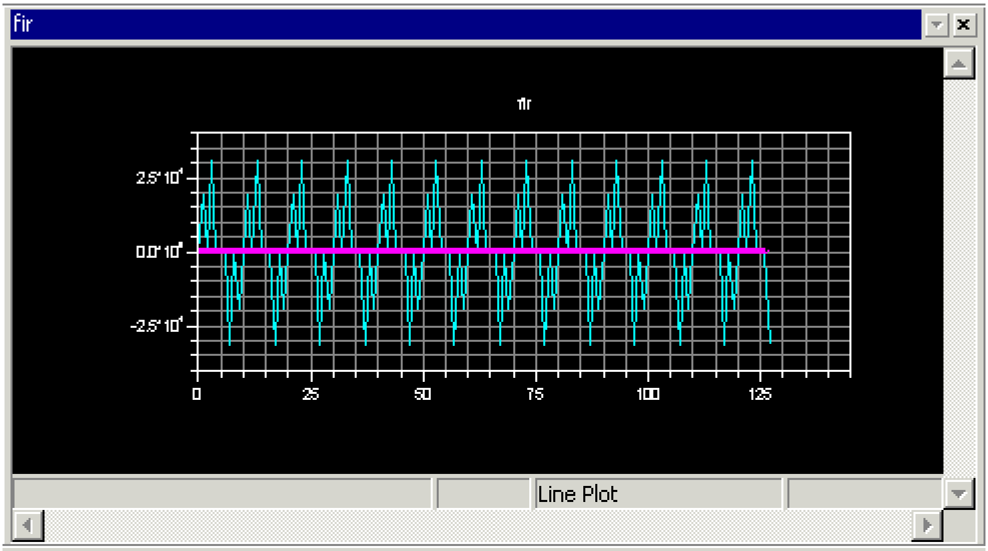


Figure 2-27. Plot Window: Before Running the FIR Program

Resizing the plot window changes the scale on the x and y axis.

5. Right-click in the plot window and choose **Modify Settings**. On the **General** page, in the **Options** group box, select **Legend** and click **OK** to display the legend box.

Step 3: Run the FIR Program and View the Data

To run the FIR program and view the data:

1. Press **F5** or click the **Run** button to run to the end of the program.

   When the program halts, you see the results of the FIR filter in the Output array. The two data sets are visible in the plot window, as shown in Figure 2-28 on page 2-42.

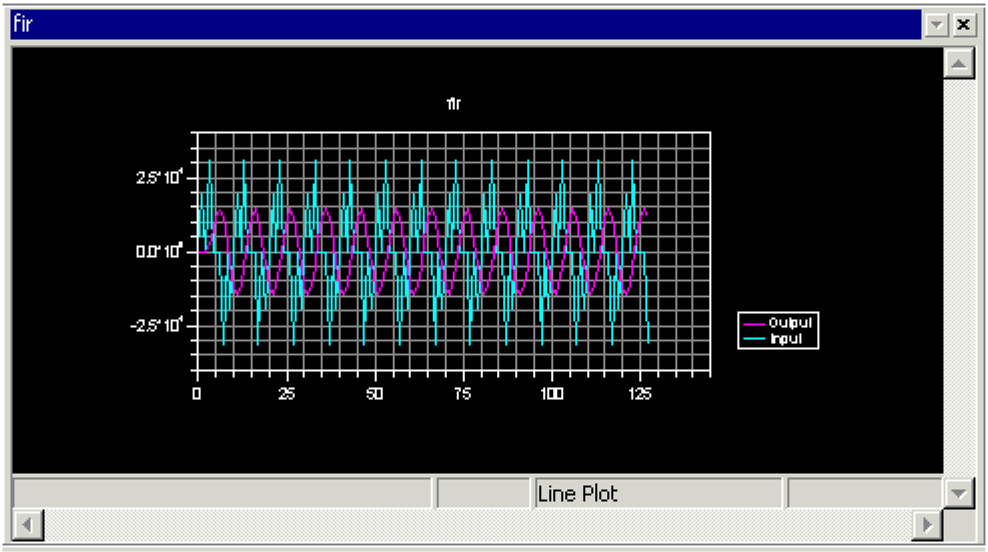


Figure 2-28. Plot Window After Running the FIR Program to Completion

Next you will zoom in on a particular region of interest in the plot window to focus in on the data.

2. Click the left mouse button inside the plot window and drag the mouse to create a rectangle to zoom into. Then release the mouse button to magnify the selected region.

   Figure 2-29 shows the selected region, and Figure 2-30 on page 2-44 shows the magnified result.

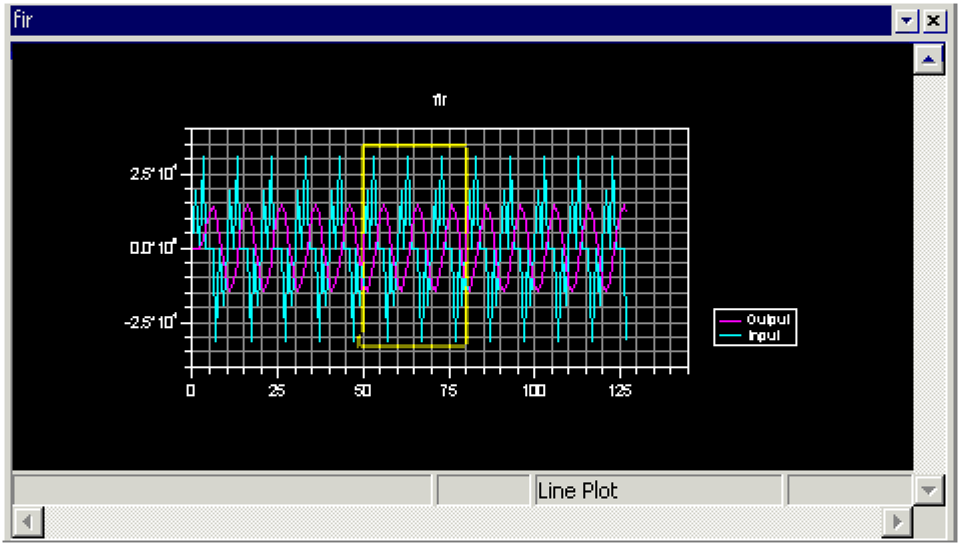


Figure 2-29. Plot Window: Selecting a Region to Magnify

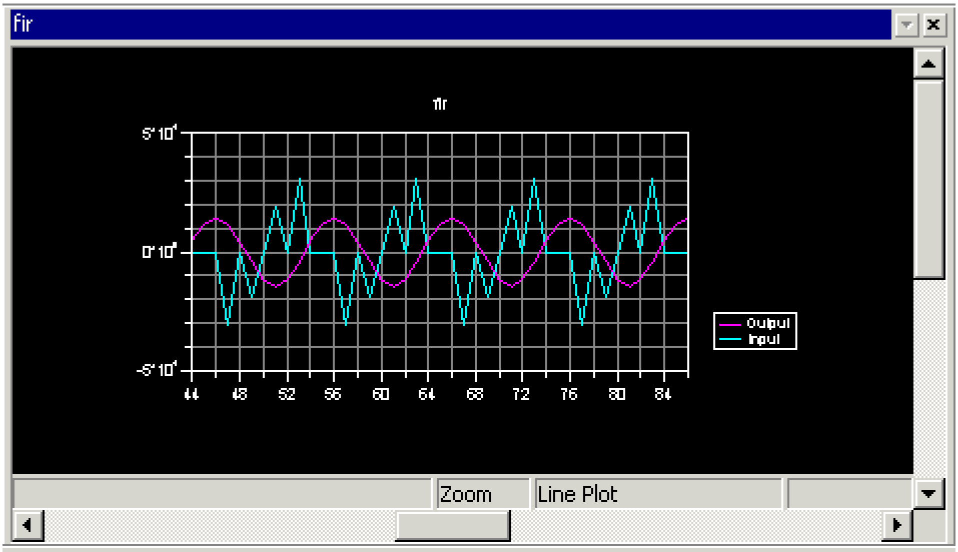
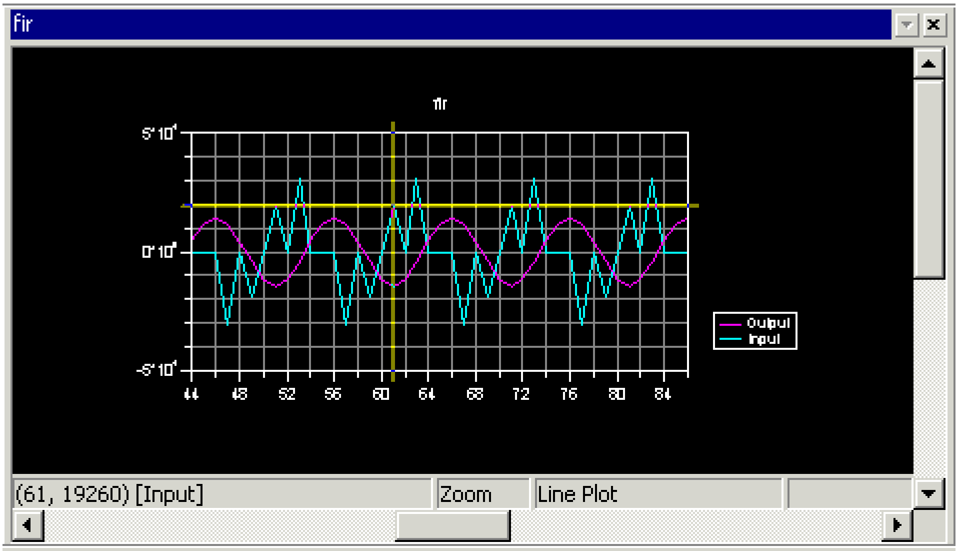


Figure 2-30. Plot Window: Magnified Result

To return to the previous view (before magnification), right-click in the plot window and choose **Reset Zoom** from the pop-up menu. You can view individual data points in the plot window by enabling the data cursor, as explained in the next step.

3. Right-click inside the plot window and choose **Data Cursor** from the popup menu. Move to each data point in the current data set by pressing and holding the Left (←) or Right (→) arrow key on the keyboard. To switch data sets, press the Up (↑) or Down (↓) arrow key. The value of the current data point appears in the lower-left corner of the plot window, as shown in Figure 2-31 on page 2-45.



(61, 19260) [Input]

Figure 2-31. Plot Window: Using the Data Cursor Feature

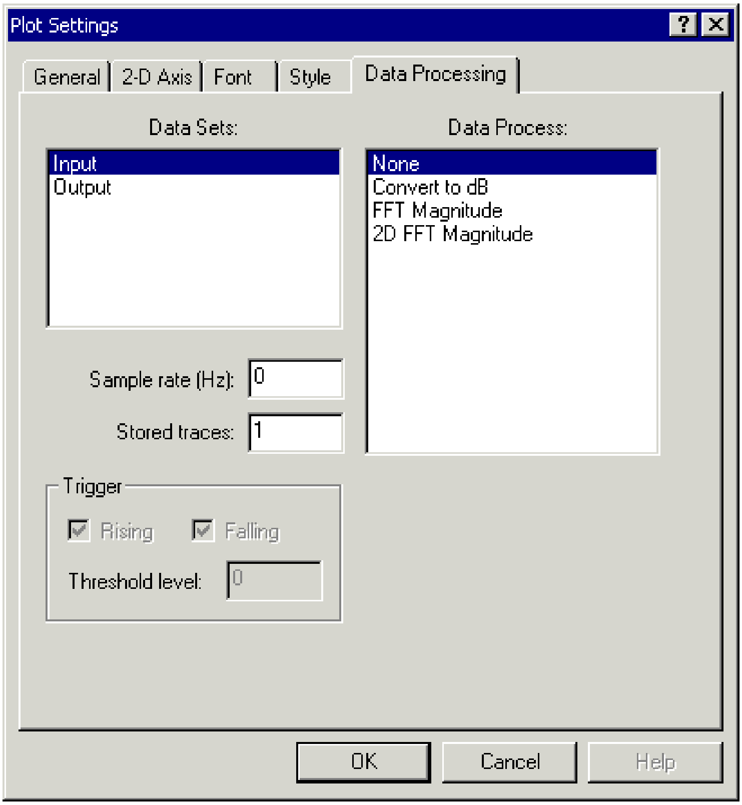
4. Right-click in the plot window and choose **Data Cursor** from the pop-up menu.

   Next you will look at data sets in the frequency domain.

5. Right-click in the plot window and choose **Modify Settings** to open the **Plot Settings** dialog box.

6. Complete these steps:

   a. Click the **Data Processing** tab to display the **Data Processing** page, shown in Figure 2-32.



Data Sets:
Input
Output

Data Process:
None
Convert to dB
FFT Magnitude
2D FFT Magnitude

Sample rate (Hz): 0

Stored traces: 1

Trigger

☑ Rising ☑ Falling

Threshold level: 0

Figure 2-32. Data Processing Page

   b. In the **Data Sets** box, make sure that **Input** (the default) is selected. In the **Data Process** box, choose **FFT Magnitude**.

   c. In the **Sample rate (Hz)** box, type **10000**.

   d. In the **Data Sets** box, select **Output**. In the **Data Process** box, choose **FFT Magnitude**

e. Click **OK** to exit the **Data Processing** page.

   VisualDSP++ performs a Fast Fourier Transform (FFT) on the selected data set before it is plotted. FFT enables you to view the signal in the frequency domain, as shown in Figure 2-33.

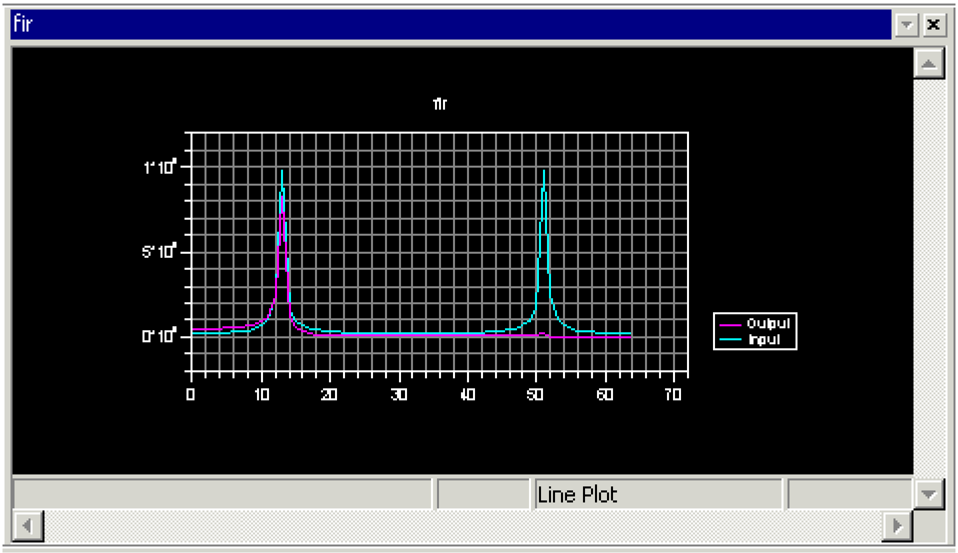
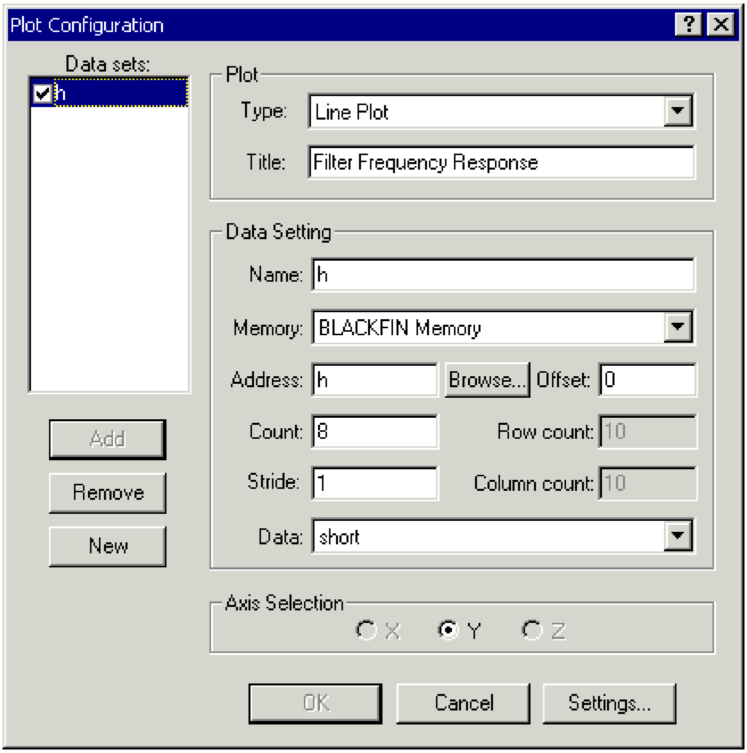


Figure 2-33. FFT Performed on a Selected Data Set

Now, complete the following steps to look at the FIR filter's response in the frequency domain.

1. From the **View** menu, choose **Debug Windows** and **Plot**. Then choose **New** to open the **Plot Configuration** dialog box.
2. Set up the Filter Frequency Response plot by completing the **Plot** and **Data Setting** group boxes as shown in Figure 2-34 on page 2-48.



Data sets:
☑ h

Plot
Type: Line Plot
Title: Filter Frequency Response

Data Setting
Name: h
Memory: BLACKFIN Memory
Address: h
Offset: 0
Count: 8
Row count: 10
Stride: 1
Column count: 10
Data: short

Axis Selection
○ X ● Y ○ Z

Figure 2-34. Filter Frequency Response Data Set

3. Click **Add** to add the data set to the **Data sets** box.
4. Click **OK** to apply the changes and to open the plot window with this data set.
5. Right-click in the plot window and choose **Modify Settings** to open the **Plot Settings** dialog box.

6. Click the **Data Processing** tab to display the **Data Processing** page, shown in Figure 2-32 on page 2-46. Complete this page as follows.

   a. In the **Data Sets** box, select **h**.

   b. In the **Data Process** box, choose **FFT Magnitude**.

   c. In the **Sample rate** (**Hz**) box, type **10000**.

   d. Click **OK** to exit the **Data Processing** page.

   VisualDSP++ performs a Fast Fourier Transform (FFT) on the selected data set, and enables you to view the filter response plot in the *frequency* domain, as shown in Figure 2-35.

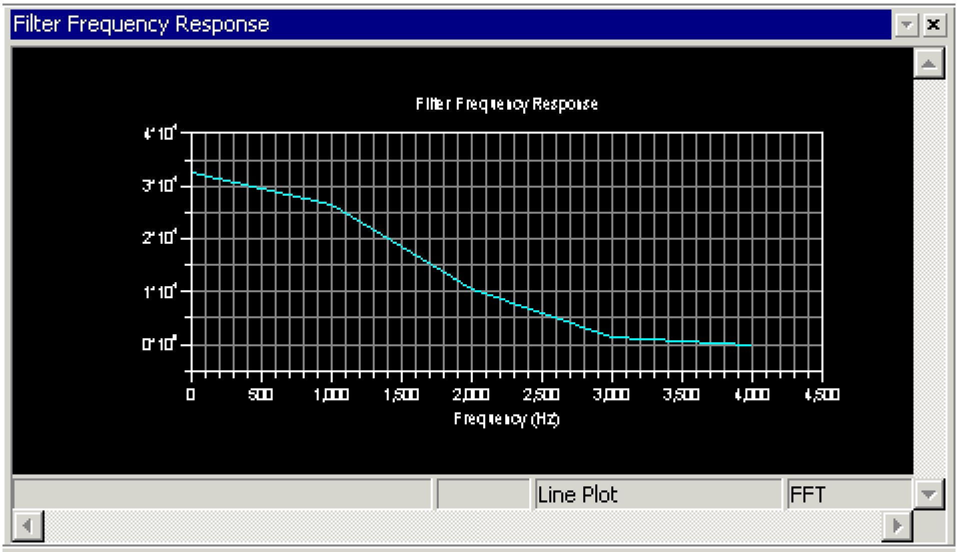


Figure 2-35. Filter Frequency Response Plot

This plot shows that the low pass FIR filter cuts off all frequency components above 4,000 Hz. When you apply a low pass filter to the input signal, the resulting signal has no output above 4,000 Hz.

You are now ready to begin Exercise Four.

Exercise Four: Linear Profiling

In this exercise, you will:

- Load and debug the FIR program from the previous exercise
- Use linear profiling to evaluate the program's efficiency and to determine where the application is spending the majority of its execution time in the code

VisualDSP++ supports two types of profiling: linear and statistical.

- You use linear profiling with a simulator. The count in the **Linear Profiling Results** window is incremented every time a line of code is executed.
- You use statistical profiling with a JTAG emulator connected to a processor target. The count in the **Statistical Profiling Results** window is based on random sampling.

Step 1: Load the FIR Program

To load the FIR program:

1. Close all open windows except for the **Disassembly** window and the **Output** window.
2. From the **File** menu, choose **Load Program**, or click . The **Open a Processor Program** dialog box appears.

3. Select the program to load as follows.

   a. Open the `Analog Devices` folder and double-click:

      `VisualDSP 3.5 16-Bit\Blackfin\Examples\Tutorial\fir`

   b. Double-click the `Debug` subfolder.

   c. Double-click `FIR.DXE` to load and run the FIR program.

      If VisualDSP++ does not open an editor window (shown in Figure 2-37 on page 2-52), right-click in the **Disassembly** window and select **View Source**.

You are now ready to set up linear profiling.

Step 2: Open the Profiling Window

To open the linear profiling window:

1. From the **Tools** menu, choose **Linear Profiling** and then choose **New Profile**.

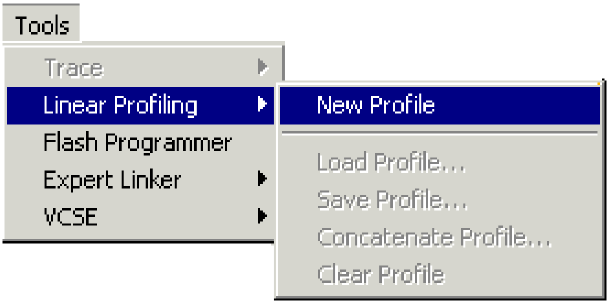
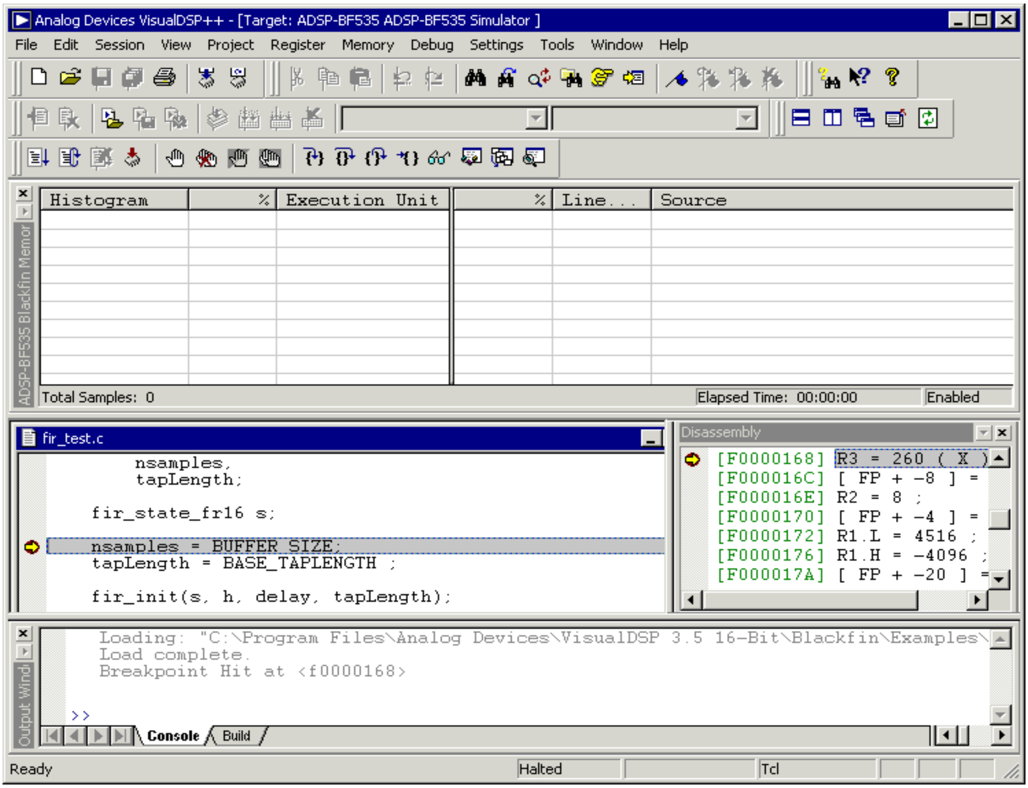


Figure 2-36. Setting Up Linear Profiling for the FIR Program

The **Linear Profiling Results** window opens.

2. For a better view of the data, use the window's title bar to drag and dock the window to the top of the VisualDSP++ main window, as shown in Figure 2-37.


Analog Devices VisualDSP++ - [Target: ADSP-BF535 ADSP-BF535 Simulator]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
ADSP-BF535 Blackfin Memor
Histogram
%
Execution Unit
%
Line...
Source
Total Samples: 0
Elapsed Time: 00:00:00
Enabled
fir_test.c
Disassembly
Output Windo
Console
Build
Ready
Halted
Tcl

```
        nsamples,
        tapLength;

    fir_state_fr16 s;

    nsamples = BUFFER_SIZE;
    tapLength = BASE_TAPLENGTH ;

    fir_init(s, h, delay, tapLength);
```

```
[F0000168] R3 = 260 ( X )
[F000016C] [ FP + -8 ] =
[F000016E] R2 = 8 ;
[F0000170] [ FP + -4 ] =
[F0000172] R1.L = 4516 ;
[F0000176] R1.H = -4096 ;
[F000017A] [ FP + -20 ] =
```

```
Loading: "C:\Program Files\Analog Devices\VisualDSP 3.5 16-Bit\Blackfin\Examples\
Load complete.
Breakpoint Hit at <f0000168>

>>
```

Figure 2-37. Linear Profiling Results Window (Empty)

The **Linear Profiling Results** window is initially empty. Linear profiling will be performed when you run the FIR program. After you run the program and collect data, this window displays the results of the profiling session.

You are now ready to collect and examine linear profile data.

Step 3: Collect and Examine the Linear Profile Data

To collect and examine the linear profile data:

1. Press **F5** or click to run to the end of the program.

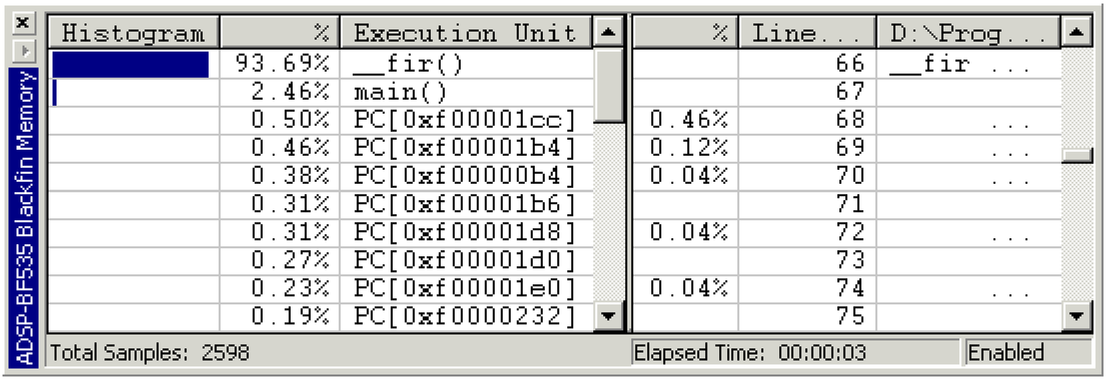
   When the program halts, the results of the linear profile appear in the **Linear Profiling Results** window.

2. Examine the results of your linear profiling session.

   The **Linear Profiling Results** window is divided into two, three-column panes.

   The left pane shows the results of the profile data. You can see the percentages of total execution time consumed, by function and by address.

   Double-clicking a line with a function enables you to display the source file that contains that function. For example, double-click the `fir` function to display the assembly source file `fir.asm` in the right pane, as shown in Figure 2-38.



Histogram	%	Execution Unit
█	93.69%	__fir()
▏	2.46%	main()
	0.50%	PC[0xf00001cc]
	0.46%	PC[0xf00001b4]
	0.38%	PC[0xf00000b4]
	0.31%	PC[0xf00001b6]
	0.31%	PC[0xf00001d8]
	0.27%	PC[0xf00001d0]
	0.23%	PC[0xf00001e0]
	0.19%	PC[0xf0000232]

%	Line...	D:\Prog...
	66	__fir ...
	67	
0.46%	68	...
0.12%	69	...
0.04%	70	...
	71	
0.04%	72	...
	73	
0.04%	74	...
	75	

Total Samples: 2598

Elapsed Time: 00:00:03

Enabled

Figure 2-38. Linear Profiling Results, FIR Program Performance Analysis

The field values in the left pane are defined as follows.

Histogram

A graphical representation of the percentage of time spent in a particular execution unit. This percentage is based on the total time that the program spent running, so longer bars denote more time spent in a particular execution unit.The **Linear Profiling Results** window always sorts the data with the most time-consuming (expensive) execution units at the top.

%

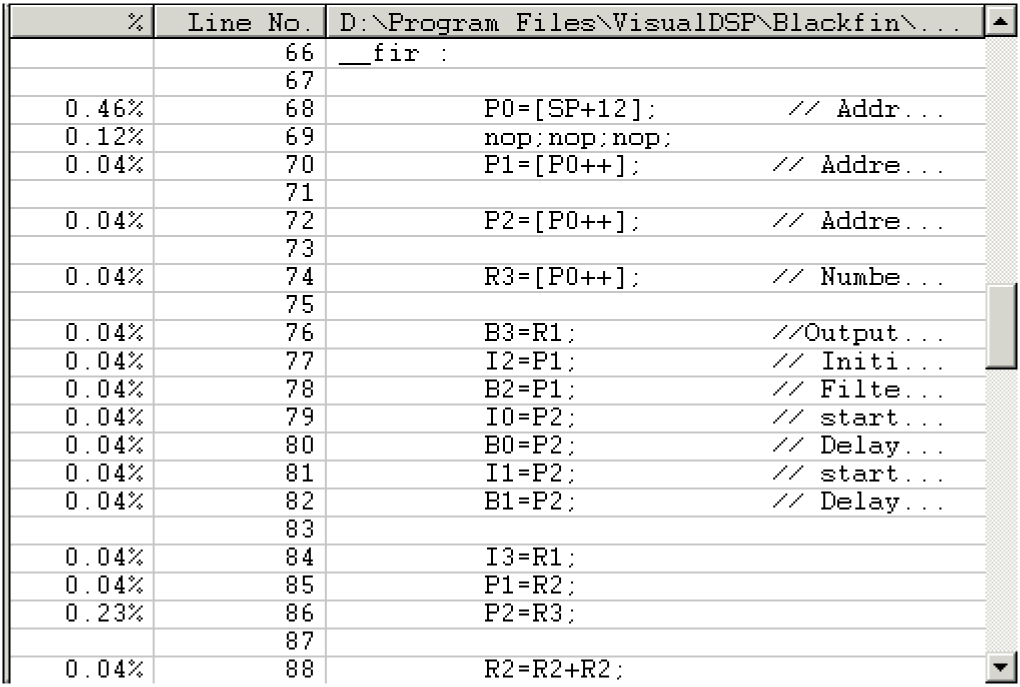
The numerical percent of the same data found in the Histogram column. You can view this value as an absolute number of samples by right-clicking in the **Linear Profiling Results** window and by selecting **View Sample Count** from the popup menu.

Execution Unit

The program location to which the samples belong. If the instructions are inside a C function or a C++ method, the execution unit is the name of the function or method. For instructions that have no corresponding symbolic names, such as hand-coded assembly or source files compiled without debugging information, this value is an address in the form of `PC[xxx]`, where `xxx` is the address of the instruction.

If the instructions are part of an assembly file, the execution unit is either an assembly function or the assembly file followed by the line number in parentheses.

In Figure 2-38 on page 2-53 the left pane shows that the `fir` function consumes over 93% of the total execution time. The right (source) pane, shown in Figure 2-39, displays the percentage that each line in the `fir` function consumes.



%	Line No.	D:\Program Files\VisualDSP\Blackfin\...
	66	`__fir :`
	67	
0.46%	68	`P0=[SP+12];          // Addr...`
0.12%	69	`nop;nop;nop;`
0.04%	70	`P1=[P0++];          // Addre...`
	71	
0.04%	72	`P2=[P0++];          // Addre...`
	73	
0.04%	74	`R3=[P0++];          // Numbe...`
	75	
0.04%	76	`B3=R1;              //Output...`
0.04%	77	`I2=P1;              // Initi...`
0.04%	78	`B2=P1;              // Filte...`
0.04%	79	`I0=P2;              // start...`
0.04%	80	`B0=P2;              // Delay...`
0.04%	81	`I1=P2;              // start...`
0.04%	82	`B1=P2;              // Delay...`
	83	
0.04%	84	`I3=R1;`
0.04%	85	`P1=R2;`
0.23%	86	`P2=R3;`
	87	
0.04%	88	`R2=R2+R2;`

Figure 2-39. Linear Profile Data for fir.asm

You are now ready to begin Exercise Five.

Exercise Five: Installing and Using a VCSE Component

In this exercise, you will:

- Start up the VisualDSP++ environment (if it is not running)
- Open an existing project
- Install a VCSE component on your system
- Add the component to the project
- Build and run the program with the component

The sources for the exercise are in the `vcse_component` folder. The default installation path is:

```
Program Files\Analog Devices\VisualDSP 3.5 16-Bit\Blackfin\
Examples\Tutorial\vcse_component
```

Step 1: Start VisualDSP++ and Open the Project

If VisualDSP++ is already running, proceed to step 2 on the next page.

To start VisualDSP++ and open the project:

1. Click the Windows **Start** button and select **Programs**, **VisualDSP**, and **VisualDSP++ Environment**.

   The VisualDSP++ main window appears.

   If you have already run VisualDSP++ and the **Reload last project at startup** option is selected (on the **Project** page in the **Settings**, **Preferences** dialog box), VisualDSP++ opens the last project that you worked on.

To close this project, choose **Close** from the **Project** menu and then click **No** when prompted to save the project. Since you have made no changes to the project, you do not have to save it.

2. From the **Project** menu, choose **Open**.

   VisualDSP++ displays the **Open Project** dialog box.

3. In the **Look in** box, open the following folders.

   `Program Files\Analog Devices\VisualDSP 3.5 16-Bit`

   Then double-click the following sub-folders in succession.

   `Blackfin\Examples\Tutorial\vcse_component`

This path is based on the default installation.

4. Double-click the `useg711.dpj` project file.

   VisualDSP++ loads the project. The environment displays messages in the **Output** window as it processes the project settings.

   The useg711 project contains a single C language source file `useg711.c`, which contains the code needed to create an instance of the CULawc component and to invoke the methods of the IG711 interface.

Step 2: Install the EXAMPLES::CULawc Component on Your System

The EXAMPLES::CULawc component is distributed as part of VisualDSP++ and is ready to be installed on your system:

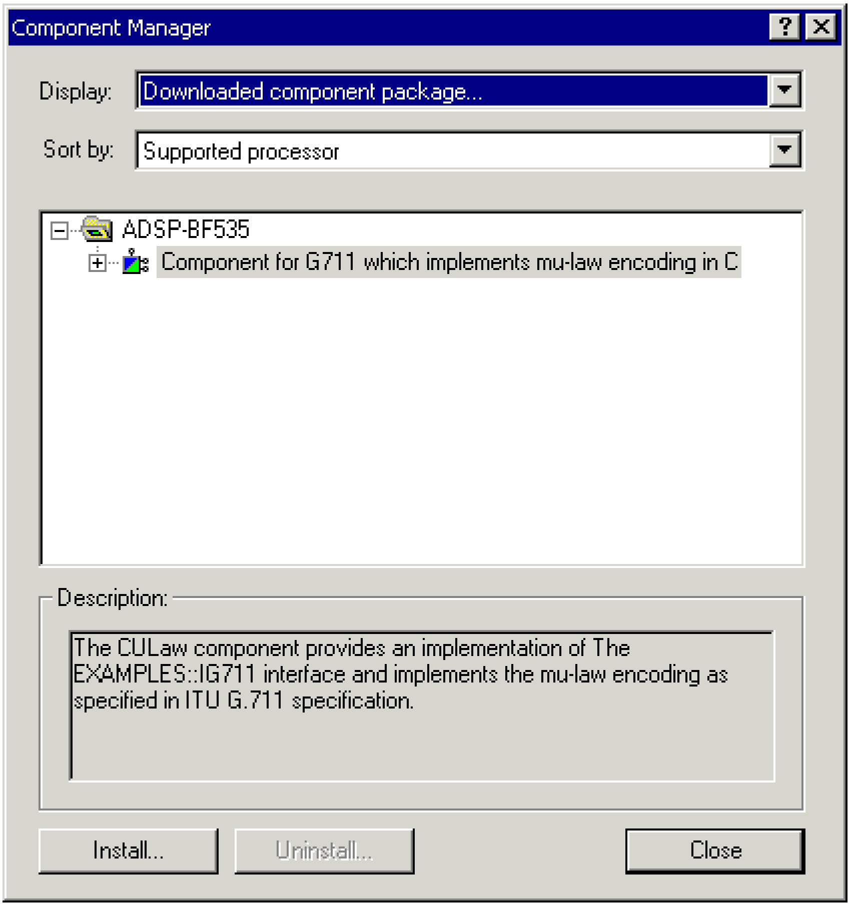
1. From the **Tools** menu, select the **VCSE** submenu and then choose **Manage Components**.
2. In the **Display** field, select **Downloaded component package…** from the drop-down list.

   The **Open** dialog box is displayed.
3. In the **Look in** box, open the `Program Files\Analog Devices` folder and double-click the following subfolders in succession.

   ```
   VisualDSP 3.5 16-Bit\Blackfin\Examples\Tutorial\
   vcse_component
   ```

   This path is based on the default installation.
4. Double-click the `examples_culawc_BF535.vcp` file.

   VisualDSP++ opens the file, extracts the information about the component, and shows it as a downloaded component in the **Component Manager** dialog box (Figure 2-40 on page 2-59).



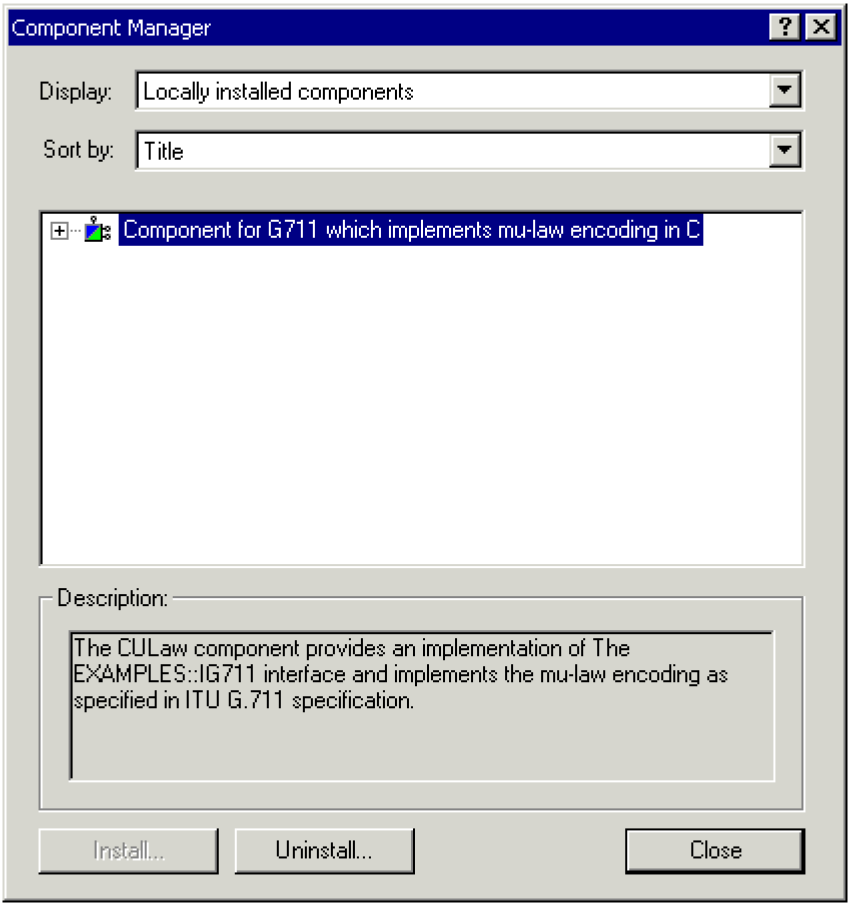
The CULaw component provides an implementation of The EXAMPLES::IG711 interface and implements the mu-law encoding as specified in ITU G.711 specification.

Figure 2-40. Component Manager Dialog Box – Downloaded Component

5. Click the **Install…** button. Component Manager installs the component on your system. Once the component is installed, click **OK**.

6. In the **Display** field select **Locally installed components** from the drop-down list, and in the **Sort by** field select **Title**.

   Select **Component for G711 which implements the mu-law encoding in C**, as shown in Figure 2-41.


Component Manager
?
Display:
Locally installed components
Sort by:
Title
Component for G711 which implements mu-law encoding in C
Description:
Install...
Uninstall...
Close

The CULaw component provides an implementation of The EXAMPLES::IG711 interface and implements the mu-law encoding as specified in ITU G.711 specification.

Figure 2-41. Component Manager Dialog Box – Selected Component

7. Click **Close** to close Component Manager.

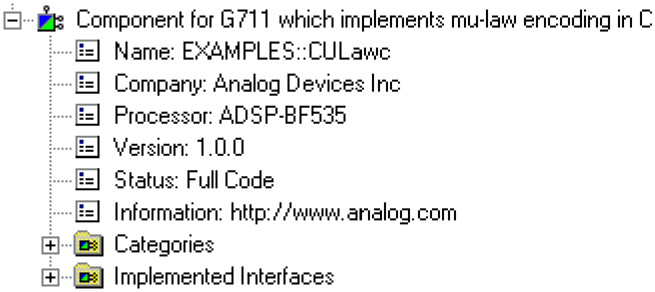
Step 3: Add the Component to Your Project

To add the newly installed component to the project:

1. From the **Tools** menu, select the **VCSE** submenu and then choose **Add Component**.

2. Click **Component for G711 which implements the mu-law encoding in C** to select it.

If you have multiple components on your system and you are not sure which one to add, click the expand button ⊞ to display the component information (shown in Figure 2-42). Make sure that **ADSP-BF535** is designated as the **Processor** for the component that you are adding to your project.

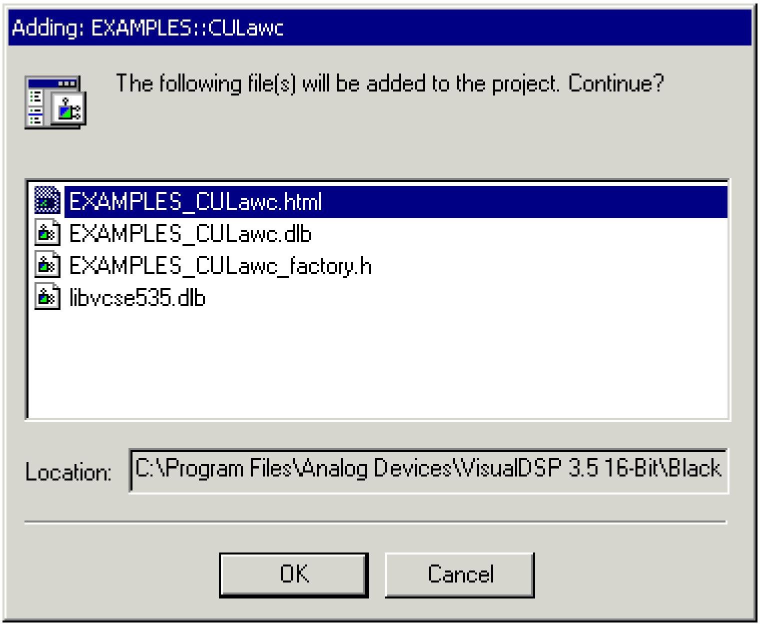


Component for G711 which implements mu-law encoding in C
Name: EXAMPLES::CULawc
Company: Analog Devices Inc
Processor: ADSP-BF535
Version: 1.0.0
Status: Full Code
Information: http://www.analog.com
Categories
Implemented Interfaces

Figure 2-42. Expanded View of Component Information

3. Click **Add** to indicate that you want to add the component to the project.

   Component Manager displays the dialog box shown in Figure 2-43 on page 2-62.



Adding: EXAMPLES::CULawc

The following file(s) will be added to the project. Continue?

- EXAMPLES_CULawc.html
- EXAMPLES_CULawc.dlb
- EXAMPLES_CULawc_factory.h
- libvcse535.dlb

Location: C:\Program Files\Analog Devices\VisualDSP 3.5 16-Bit\Black

Figure 2-43. Adding Files to the Project

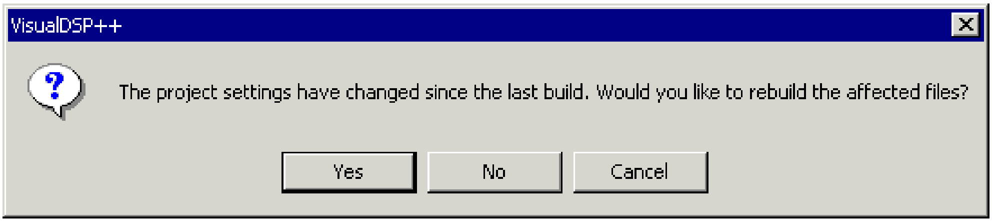
4. Click **OK** to add the component files to the project.

Step 4: Build and Run the Program

To build and run the program:

1. From the **Project** menu, choose **Build Project**.

   VisualDSP++ displays the message shown in Figure 2-44 on page 2-63.



The project settings have changed since the last build. Would you like to rebuild the affected files?

Figure 2-44. Rebuilding Files Affected by Changes to Project Settings

2. Click **Yes**. VisualDSP++ compiles the source files and creates the program.

3. From the **Debug** menu, choose **Run** to execute the program. The program generates the following output.

```
Harness to test component code generated by
EXAMPLES_CULawc.idl
Testing EXAMPLES::IG711
Test Completed result = MR_OK
```

You have now completed the Basic Tutorial.

3 ADVANCED TUTORIAL

This chapter contains the following topics.

Overview

This tutorial demonstrates the more advanced features and techniques that you can use in the VisualDSP++ Integrated Development and Debugging Environment (IDDE). The exercises use sample programs written in C, C++, and assembly for Blackfin processors.

- In **Exercise One: Using Profile-Guided Optimization**, you will build a project with PGO support, create PGO files, compile the project without using the information in the PGO files, re-compile the project by using the PGO files to optimize the build, check the PGO results, and compare execution times.
- In **Exercise Two: Using Background Telemetry Channel**, you will add BTC to your DSP application and then run two demos that demonstrate BTC functionality.

The ADSP-BF53x Family Simulator and ADSP-BF532 processor are used for Exercise One. The EZ-KIT Lite USB emulator, HP PCI ICE, and ADSP-BF535 processor are used for Exercise Two.

Exercise One: Using Profile-Guided Optimization

Profile-guided optimization (PGO) is an optimization technique that uses previously-collected information to guide the compiler optimizer's decisions.

Traditionally, a compiler compiles each function only once and attempts to produce generated code that will perform well in most cases. The compiler has to make decisions about the best code to generate. For example, given an if…then…else construct, the compiler has to decide whether the most common case is the "then" or the "else." You can offer crude guidelines—compile for speed or compile for space—but, usually, the compiler has to make a default decision.

With PGO, the compiler makes these decisions based on data collected during previous executions of the generated code. This process involves the following steps.

1. Compile the application to collect profile information.
2. Run the application in a simulator session by using representative data sets. The simulator accumulates profile data indicating where the application spends most of its time.
3. Re-compile the application by using the collected profile data. The compiler uses the collected information rather than the application's default behavior to make decisions about the relative importance of parts of the application.

The profile data collected from a simulator run is stored in a file with a `.PGO` suffix. You can process multiple data sets to cover the spectrum of potential data and create a separate `.PGO` file for each data set. The re-compilation stage can accept multiple `.PGO` files as input.

The basic steps required to use PGO are:

1. Build the application with PGO support.
2. Set up one or more streams in the simulator to provide a set of data inputs that represent what the application would see in a real target environment.
3. Tell the simulator to produce a `.PGO` file with a specified filename.
4. Load and run the application to produce the `.PGO` file.
5. Rebuild the application and pass all `.PGO` files to the compiler, which uses the generated PGO results to optimize the application.

In this exercise, you will:

- Load the PGO example project in the VisualDSP++ environment.
- Create data sets for profile-guided optimization.
- Attach input streams to the data sets.
- Create `.PGO` files by executing the project with the data sets as input.
- Re-compile the project by using the `.PGO` files to optimize the build.
- Run the optimized version of the project with the same data sets as input.
- Compare the execution times of all three executions.

The files used for this exercise are in the `pgo` folder. The default installation path of this folder is:

```
Program Files\Analog Devices\VisualDSP++ 3.5 for 16-Bit
Processors\Blackfin\Examples\Tutorial\pgo
```

Step 1: Load the Project

To open a VisualDSP++ project:

1. Start VisualDSP++ and connect to an ADSP-BF532 simulator session. For information about connecting to a session, refer to "Step 1: Start VisualDSP++ and Open a Project" on page 2-4.

2. Open the project PgoExample.dpj. For details about opening projects, see "Step 1: Start VisualDSP++ and Open a Project" on page 2-4.

   This project contains a C file, PgoExample.c. When you run the program, it reads data from an address and counts the number of even and odd values. This counting is done with an if…then…else statement. If the majority of values read are odd, the program will spend most of its time in the then… branch. If the majority of values read are even, the program will spend most of its time in the else… branch. Normally, the compiler has no way of knowing which branch will be taken more often. By using PGO, the compiler can determine which branch is most often used and optimize the next build.

   ⓘ This project also contains a Visual Basic script that demonstrates how to use the VisualDSP++ Automation API to perform PGO. The automation functionality is beyond the scope of this tutorial. Refer to the online Help for more information about automation.

   Three data files are used as input to the C program. These simple text files contain lists of values.

   - Dataset_1.dat has 128 even values (50%) and 128 odd values (50%).
   - Dataset_2.dat has 192 even values (75%) and 64 odd values (25%).
   - Dataset_3.dat has 256 even values (100%) and 0 odd values (0%).

To view these files, use the **Open** command on the **File** menu in VisualDSP++. The two possible values in all three files are either `0x01` or `0x02`. Each file contains 256 values.

For the purposes of this exercise, assume that this program will be used in the real world, and that you can expect a similar distribution of values as input from the real world.

By looking at the C code and the potential input, you can easily see that the executed program will spend more time in the else… branch than in the then… branch. Without using PGO, the compiler cannot make this same conclusion. By default, it will expect the then… branch to be executed most frequently and will compile the code without optimizing execution time.

Since the example program and input are very simple, you could fix the problem by making a few minor changes to the code. Manually tweaking a large program to speed up execution time, however, would take far too long, and you would have to analyze sample input on your own. PGO provides a quick and easy way to enable the compiler to make these adjustments for you.

Step 2: Configure a Data Set

The first step in the PGO process is to create a *data set*—a collection of sample input for the program being optimized. A data set feeds the input into the executing program, and this input causes the program to be executed along certain paths. Some paths will be used more often than others. This information is recorded by the simulator and is stored in a `.PGO` file for the compiler to use later for optimization. The most commonly used paths will be optimized to run quickly, while less common paths will run more slowly.

Complete the following steps to create the first of three data sets for this exercise.

1. From the **Tools** menu, choose **PGO** and then **Manage Data Sets**, as shown in Figure 3-1.

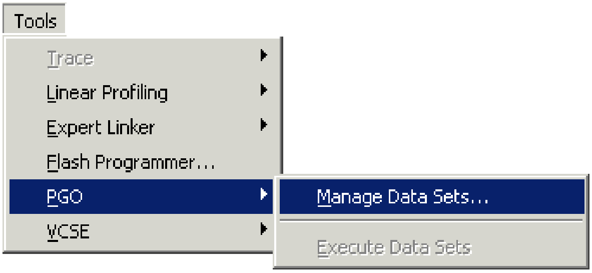


Figure 3-1. Manage Data Sets Menu Option

The **Manage Data Sets** dialog box (Figure 3-2) is displayed.

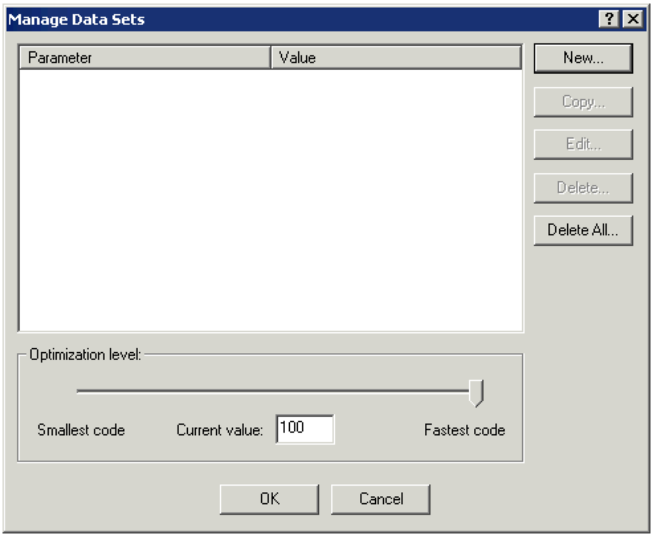
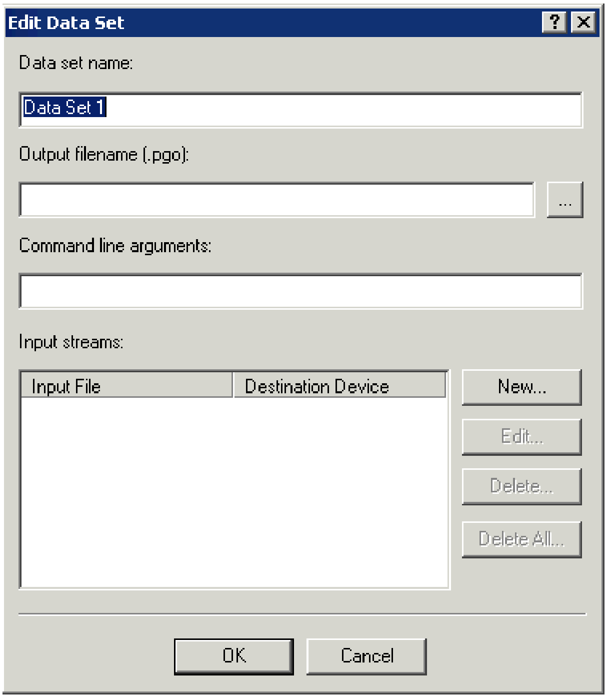


Figure 3-2. Manage Data Sets Dialog Box

This dialog box is where you manage your data sets. Note the slider bar near the bottom of the window. This control allows you to customize your optimization. Moving the slider all the way to the left enables you to build as small an executable as possible, but may sacrifice execution speed. Moving the slider all the way to the right enables you to build a fast executable, with a potential space trade-off. Placing the slider between the two extremes provides varying ratios of space vs. speed optimization. For this exercise, move the slider all the way to the right.

2. Click the **New** button to open the **Edit Data Set** dialog box, shown in Figure 3-3.



Data set name: Data Set 1

Figure 3-3. Edit Data Set Dialog Box

3. Replace the default **Data set name** with a more descriptive name. Since the first data file contains an equal number of even and odd values, use a name such as `50% Even - 50% Odd`.

4. Specify the **Output filename** (where the optimization information produced by this data set will be saved). Optimization information is saved in files with the `.PGO` suffix.

   Type in a file name such as `dataset_1.pgo`. The file will be saved in the project directory. To save the files elsewhere, type in a full path name. You can use **Command line arguments** for more advanced control of the data set, but they are not covered in this tutorial.

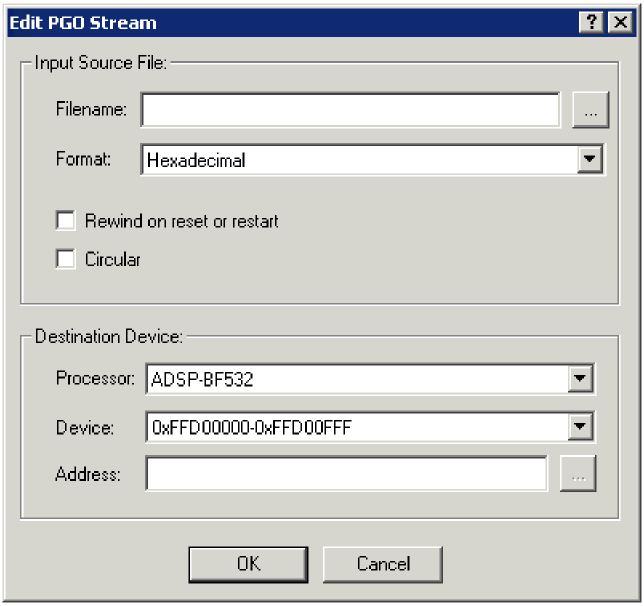
   For more information about command line arguments, see the *VisualDSP++ 3.5 C/C++ Compiler and Library Manual for Blackfin Processors*.

Now you have to attach an input stream to this data set.

Step 3: Attach an Input Stream

In this step you will attach an input stream to the data set.

1. Click the **New** button to open the **Edit PGO Stream** dialog box, shown in Figure 3-4.



Filename:

Format: Hexadecimal

□ Rewind on reset or restart

□ Circular

Processor: ADSP-BF532

Device: 0xFFD00000-0xFFD00FFF

Address:

Figure 3-4. Edit PGO Stream Dialog Box

An input stream maps a data file to a destination device. In this exercise, the input streams map the three data files to the simulator. The input stream provides the program with input as needed during execution.

For more information about streams, see the “Debugging” chapter in the *VisualDSP++ 3.5 User's Guide for 16-Bit Processors*.

2. Complete the **Input Source File** group box as described in Table 3-1.

Table 3-1. Input Source File Group Box Settings

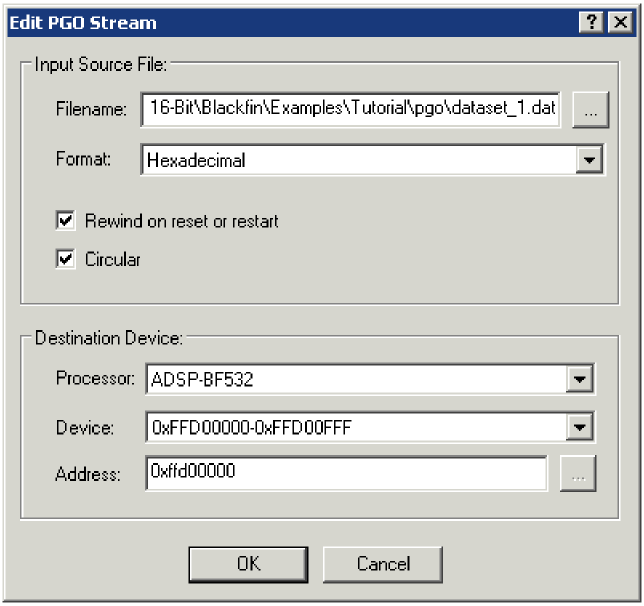
Field/Control	Action/Value
Filename	Specify a file name by clicking the file browse button and selecting the input source file `dataset_1.dat` from the `pgo` directory.
Format	The data in this file is in hexadecimal format, so leave the format setting as is.
Rewind on reset or restart	Select this option. When you run a program with an input stream, the program may or may not work through all of the data in the stream. If the program encounters a reset or restart event before working through the entire data stream and this option is enabled, the next execution will start at the beginning of the input stream. Otherwise, execution will continue where it left off.
Circular	Select this option. It allows the program to read through an input stream many times during a single execution.

3. In the **Destination Device** group box, specify where the data from the input stream is sent. Refer to Table 3-2.

Table 3-2. Destination Device Group Box Settings

Field/Control	Action/Value
Processor	This field is used to specify a peripheral in another processor as the destination device. For this tutorial, you are connected to a single processor session, so this field is disabled.
Device	This field allows you to choose any stream device supported by the simulator target as the destination. Devices can include a memory address or various peripherals. Available devices depend on which processor you are using. For more information on devices, see the hardware manual for your processor. The program reads the input streams from memory, so leave this field as it is.
Address	Specify where in memory the input will be sent. Since the program in this exercise reads data from address `0xFFD00000` (refer to `PgoExample.c`), enter this value.

The dialog box should now look like the one in Figure 3-5.



Input Source File:

Filename: 16-Bit\Blackfin\Examples\Tutorial\pgo\dataset_1.dat

Format: Hexadecimal

☑ Rewind on reset or restart

☑ Circular

Destination Device:

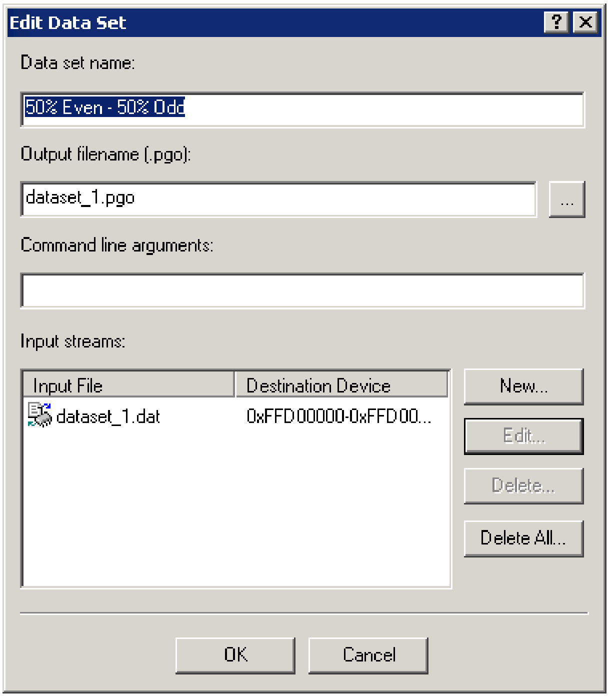
Processor: ADSP-BF532

Device: 0xFFD00000-0xFFD00FFF

Address: 0xffd00000

Figure 3-5. A Configured PGO Stream

4. Click **OK** to return to the **Edit Data Set** dialog box. Your configured data set should match the one in Figure 3-6.



Data set name:

50% Even - 50% Odd

Output filename (.pgo):

dataset_1.pgo

Command line arguments:

Input streams:

Input File	Destination Device
dataset_1.dat	0xFFD00000-0xFFD00...

Figure 3-6. A Configured Data Set

5. Click **OK** to save the data set and close the dialog box.

You now have to create the remaining two data sets.

Step 4: Configure Additional Data Sets

To create the remaining two data sets, you can repeat the steps used to create the first data set and substitute the appropriate files, or you can use the **Copy** button.

The following steps explain how to use the **Copy** button to create a data set.

1. Highlight the `50% Even - 50% Odd` data set, and click the **Copy** button.

   The **Edit Data Set** dialog box opens with the information for the `50% Even - 50% Odd data set`. Clicking the **OK** button makes a copy of the `50% Even - 50% Odd data set`. For this exercise, however, you will edit the data set.

2. In the **Data set name** field, specify an appropriate name for the new data set.

   The second input source file contains three times as many even values as odd values, so use a name such as `75% Even - 25% Odd`.

3. In the **Output filename** field, type the name `dataset_2.pgo` to save the `.PGO` file in the project directory.

   To save the file elsewhere, use the file browse button ... to specify a full path.

4. In the **Input streams** list, highlight the `dataset_1.dat` **Input File** and click the **Edit** button.

5. Use the file browse button ... to change the **Input Source File** from `dataset_1.dat` to d`ataset_2.dat.`

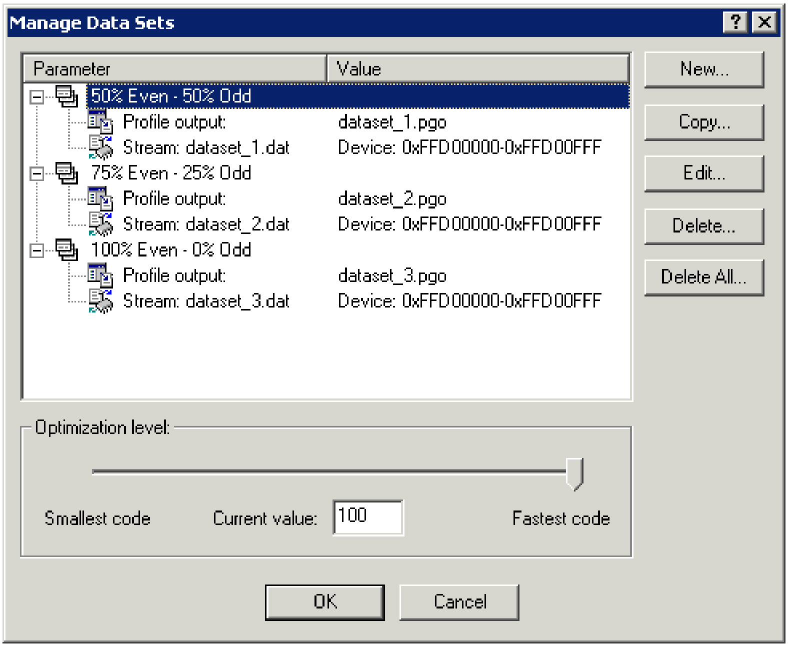
6. Click the **OK** button to return to the **Edit Data Set** dialog box.

   The second data set is now complete.

7. Click the **OK** button to return the **Manage Data Sets** dialog box.

8. Create the third data set from scratch or modify a copy of one of the existing data sets.

   Make sure that you use the `dataset_3.pgo` and `dataset_3.dat` files. The third data set contains all even values, so give it a name such as `100% Even - 0% Odd`. When you are finished, expand the three data sets listed in the **Manage Data Sets** dialog box and compare them with the data sets in Figure 3-7.



Parameter	Value
50% Even - 50% Odd	
Profile output:	dataset_1.pgo
Stream: dataset_1.dat	Device: 0xFFD00000-0xFFD00FFF
75% Even - 25% Odd	
Profile output:	dataset_2.pgo
Stream: dataset_2.dat	Device: 0xFFD00000-0xFFD00FFF
100% Even - 0% Odd	
Profile output:	dataset_3.pgo
Stream: dataset_3.dat	Device: 0xFFD00000-0xFFD00FFF

Optimization level:

Smallest code   Current value: 100   Fastest code

Figure 3-7. Expanded Data Sets

   If your data sets match those in Figure 3-7, you have the data sets needed to optimize the program.

9. Click **OK** to save the data sets and close the dialog box.

You are now ready to create `.PGO` files.

Step 5: Create PGO Files and Optimize Your Program

Now that you have configured the data sets, you are ready to optimize your program.

From the **Tools** menu, choose **PGO** and then **Execute Data Sets**, as shown in Figure 3-8.

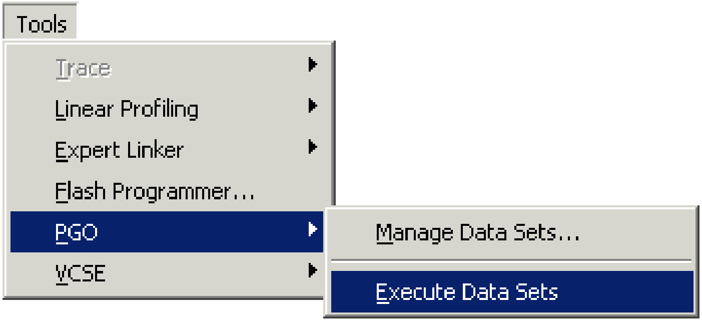


Figure 3-8. Execute Data Sets Menu Option

Several things happen during the execute process. First, the project is built with the `-pguide` switch which enables the collection of the PGO data that will later be fed back into the compiler. The compiler makes default assumptions about which sections of code will be most commonly executed. Next, the resulting executable is run once with each data set. While the program is running, the simulator monitors the paths of execution through the program, and the number of cycles used in the execution. As stated before, this information is stored in the `.PGO` file that you specified when creating each data set.

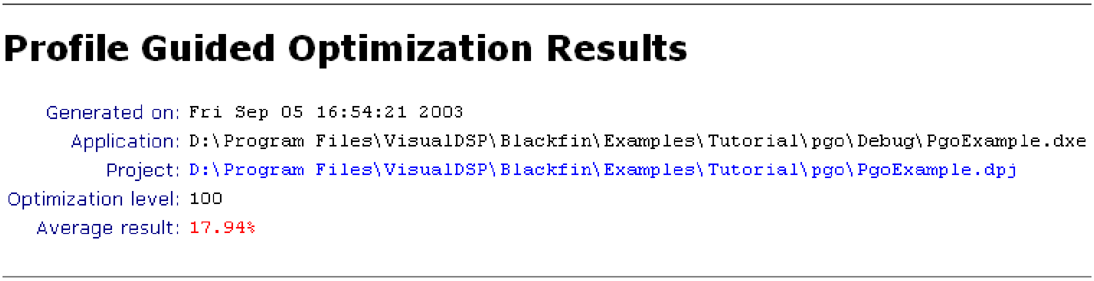
Once the program has been run with each data set, the project is re-compiled. This time, however, the compiler uses the information found in the `.PGO` files to optimize the resulting executable. This optimized executable is then run with the input provided by each data set, and, again, the simulator monitors each execution.

You are now ready to examine the results of the optimization.

Step 6: Compare Execution Times

When the execution is completed, an XML report of the PGO optimization results is generated and displayed in a browser window. This file is in the `pgo\output` folder and is named `PgoReport.`*date and time*`.xml` (for example, `PgoReport.20031027145428.xml`).

At the top of the report is a header, shown in Figure 3-9.



Profile Guided Optimization Results

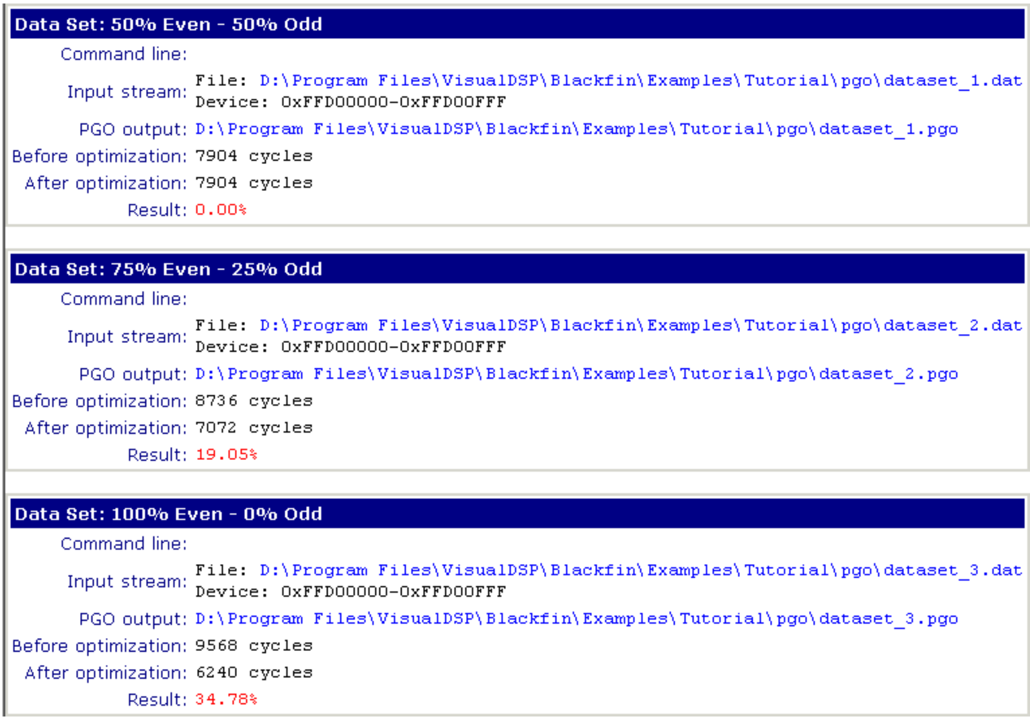
```
     Generated on: Fri Sep 05 16:54:21 2003
      Application: D:\Program Files\VisualDSP\Blackfin\Examples\Tutorial\pgo\Debug\PgoExample.dxe
          Project: D:\Program Files\VisualDSP\Blackfin\Examples\Tutorial\pgo\PgoExample.dpj
Optimization level: 100
   Average result: 17.94%
```

Figure 3-9. PGO Results – Report Header

The header provides basic information such as the project name, location, and the time when the report was generated. Also listed is the optimization level (which you specified with the slider bar in the **Manage Data Sets** dialog box, Figure 3-2 on page 3-6), and an average result. The **Average result** is the difference in total cycle counts on all executions from before and after optimization.

The **Average result** obtained on your machine may vary slightly from the result shown in Figure 3-9.

The header is followed by information about each data set (see Figure 3-10).



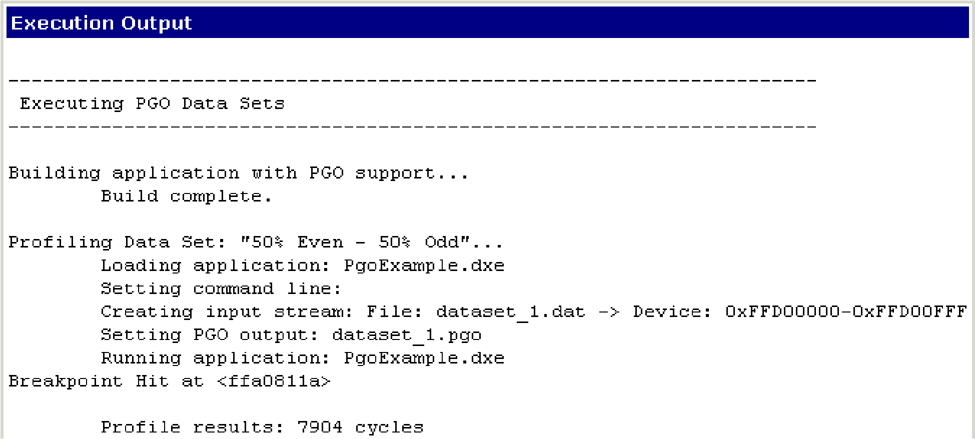
```
Data Set: 50% Even - 50% Odd
        Command line:
         Input stream: File: D:\Program Files\VisualDSP\Blackfin\Examples\Tutorial\pgo\dataset_1.dat
                       Device: 0xFFD00000-0xFFD00FFF
           PGO output: D:\Program Files\VisualDSP\Blackfin\Examples\Tutorial\pgo\dataset_1.pgo
  Before optimization: 7904 cycles
   After optimization: 7904 cycles
               Result: 0.00%

Data Set: 75% Even - 25% Odd
        Command line:
         Input stream: File: D:\Program Files\VisualDSP\Blackfin\Examples\Tutorial\pgo\dataset_2.dat
                       Device: 0xFFD00000-0xFFD00FFF
           PGO output: D:\Program Files\VisualDSP\Blackfin\Examples\Tutorial\pgo\dataset_2.pgo
  Before optimization: 8736 cycles
   After optimization: 7072 cycles
               Result: 19.05%

Data Set: 100% Even - 0% Odd
        Command line:
         Input stream: File: D:\Program Files\VisualDSP\Blackfin\Examples\Tutorial\pgo\dataset_3.dat
                       Device: 0xFFD00000-0xFFD00FFF
           PGO output: D:\Program Files\VisualDSP\Blackfin\Examples\Tutorial\pgo\dataset_3.pgo
  Before optimization: 9568 cycles
   After optimization: 6240 cycles
               Result: 34.78%
```

Figure 3-10. PGO Results – Data Sets

The file information, including the **Data Set** file name, **Input stream** file name, and **PGO output** file name, is listed first. Then the results of optimization are shown. The number of cycles needed to run the original build with this data set (**Before optimization**) is followed by the number of cycles needed to run this data set on the optimized build (**After optimization**). Note that the number of cycles may vary on different machines.

Finally, the percent difference between the two (**Result**) is listed. A positive percentage indicates that the optimized build ran faster than the original build.

The **Execution Output** section of the log appears first. Figure 3-11 shows some selections from the execution output.



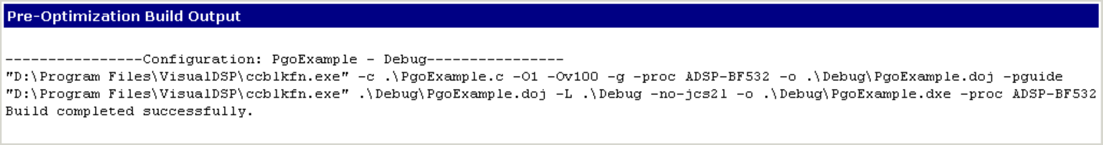
Execution Output

```
----------------------------------------------------------------------
 Executing PGO Data Sets
----------------------------------------------------------------------

Building application with PGO support...
        Build complete.

Profiling Data Set: "50% Even - 50% Odd"...
        Loading application: PgoExample.dxe
        Setting command line:
        Creating input stream: File: dataset_1.dat -> Device: 0xFFD00000-0xFFD00FFF
        Setting PGO output: dataset_1.pgo
        Running application: PgoExample.dxe
Breakpoint Hit at <ffa0811a>

        Profile results: 7904 cycles
```

Figure 3-11. PGO Results – Execution Output Sample

This information is the output that was displayed in the **Console** window while the PGO was running. The output includes the basic events that occurred during execution.

The **Build Output** section appears next at the bottom of the report. This section contains build output for each build. Figure 3-12 shows a build output sample.



Pre-Optimization Build Output

```
----------------Configuration: PgoExample - Debug----------------
"D:\Program Files\VisualDSP\ccblkfn.exe" -c .\PgoExample.c -O1 -Ov100 -g -proc ADSP-BF532 -o .\Debug\PgoExample.doj -pguide
"D:\Program Files\VisualDSP\ccblkfn.exe" .\Debug\PgoExample.doj -L .\Debug -no-jcs2l -o .\Debug\PgoExample.dxe -proc ADSP-BF532
Build completed successfully.
```

Figure 3-12. PGO Results – Build Output Sample

This information is the output that was displayed in the **Build** window while the PGO was running. **Build Output** includes the command-line build options and switches used by the various build tools. See the *VisualDSP++ 3.5 C/C++ Compiler and Library Manual for Blackfin Processors* for details about build options.

This output information shows how effective PGO can be. As shown in Figure 3-9 on page 3-16, the optimized executions used roughly 17% fewer cycles than the original executions. The gain in performance is significant, especially given the ease with which it was accomplished.

You are now ready to begin Exercise Two.

Exercise Two: Using Background Telemetry Channel

A background telemetry channel (BTC) enables you to exchange data between a host and target application without halting the processor. This mechanism provides real-time visibility into a running program. Uses for a BTC include:

- Monitoring program status without halting the processor
- Viewing algorithm output in real time
- Injecting data into a program
- Streaming real-time data to a file (data logging)
- Providing IO, either standard or user-defined

In this exercise, you will:

- Add BTC to your DSP application
- Run the BTC Assembly demo, designed to demonstrate the basic functionality of BTC
- Run the BTC FFT demo, which demonstrates the transfer of data from the Blackfin EZ-KIT Lite over background telemetry channels

Adding BTC to Your DSP Application

To add BTC to your DSP application, follow these steps:

1. Add the `btc.h` header file to your source code.

   The `btc.h` file contains the macros necessary to use BTC in your programs, including those macros necessary to define a channel.

2. Define one or more channels in the DSP source code.

   Channels are used to identify an address range for data transfer. Each channel consists of a name, a start address, and a length. All channels are defined inside a single BTC map block. A sample BTC channel definition is shown below:

   ```
   #include “btc.h”
   short AudioData[12000]//audio capture array
   BTC_MAP_BEGIN
   BTC_MAP_ENTRY( “Audio Data Channel”, (long)&AudioData,
   sizeof(AudioData))
   BTC_MAP_END
   ```

3. Define the BTC polling loop with the `btc_poll()` command.

   The polling loop checks for incoming commands from the host. The location of the polling loop determines the polling priority, which affects application performance. You can place the polling loop in a high priority interrupt, a low priority interrupt, or in the main program loop.

4. Initialize BTC by using the `btc_init()` command.

5. Add the BTC library, `libbtc532.dlb` or `libbtc535.dlb`, to the project.

   The BTC library contains the functions that transfer data to and from the host.

Several example programs included with VisualDSP++ 3.5 can help you become familiar with BTC. The following demos walk you step-by-step through two of the Blackfin ADSP-BF535 examples. Analog Devices highly recommends that you be connected to an ADSP-BF535 EZ-KIT Lite board via HP PCI ICE when running these examples.

The low data rate required enables you to run the first example by using the EZ-KIT Lite USB interface. The second example, however, requires a data rate beyond what the EZ-KIT Lite USB interface can provide. Refer to the manual included with your HP PCI ICE or EZ-KIT Lite if you are unsure of how to establish a connection.

ⓘ BTC is not currently supported by simulation targets.

The default installation path for these examples is:

```
Program Files\Analog Devices\VisualDSP 3.5 16-Bit\BlackFin\
Examples\BTC Examples\BF535
```

You are now ready to run the first BTC demo.

Running the BTC Assembly Demo

The BTC Assembly demo is designed to demonstrate the basic functionality of BTC. The program defines several BTCs to allow the transfer of data over the BTC interface while the DSP is running. For example, one channel counts the number of interrupts that have occurred, while another counts the number of times a push button is pressed. See the header of `Btc_AsmDemo.asm` for more details. You will use the **BTC Memory** window in the IDDE to view the data in each channel.

Figure 3-13 provides an overview of data transfer over the BTC interface in the BTC Assembly demo.

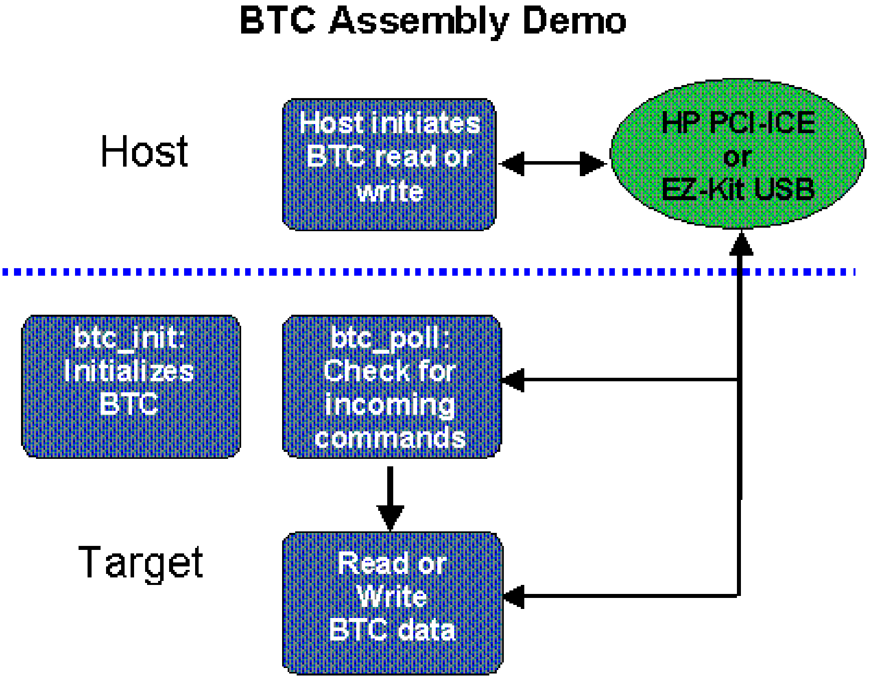

BTC Assembly Demo
Host
Host initiates BTC read or write
HP PCI-ICE or EZ-Kit USB
btc_init: Initializes BTC
btc_poll: Check for incoming commands
Target
Read or Write BTC data

Figure 3-13. Data Transfer in the Assembly Demo

Step 1: Load the Btc_AsmDemo Project

1. Start VisualDSP++ and connect to an ADSP-BF535 HP PCI ICE or EZ-KIT Lite USB emulator session.

   For information about connecting to a session, refer to “Step 1: Start VisualDSP++ and Open a Project” on page 2-4 in Exercise One.

2. Open the `Btc_AsmDemo.dpj` project, under `Analog Devices` in:

   ```
   VisualDSP 3.5 16-Bit\BlackFin\Examples\BTC Examples\BF535
   ```

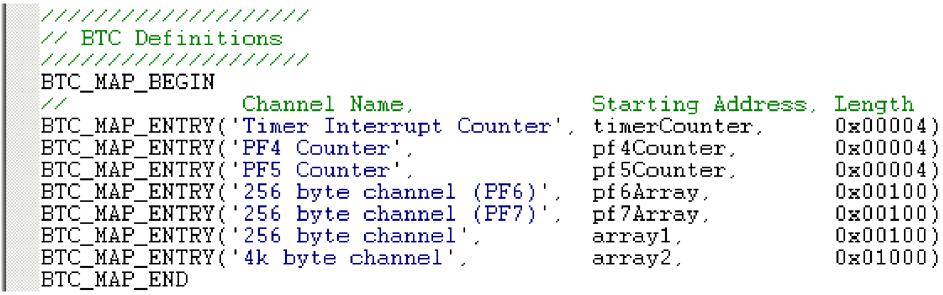
   For details about loading a project, see “Step 1: Start VisualDSP++ and Open a Project” on page 2-4.

You are now ready to examine BTC commands.

Step 2: Examine the BTC Commands

1. Open the `Btc_AsmDemo.asm` file by double-clicking on it in the **Project** window on the left.

2. Scroll down to the section labeled `BTC Definitions` in the comments (see Figure 3-14). Notice how the seven channels are defined as described in step 2 of “Adding BTC to Your DSP Application” on page 3-20.

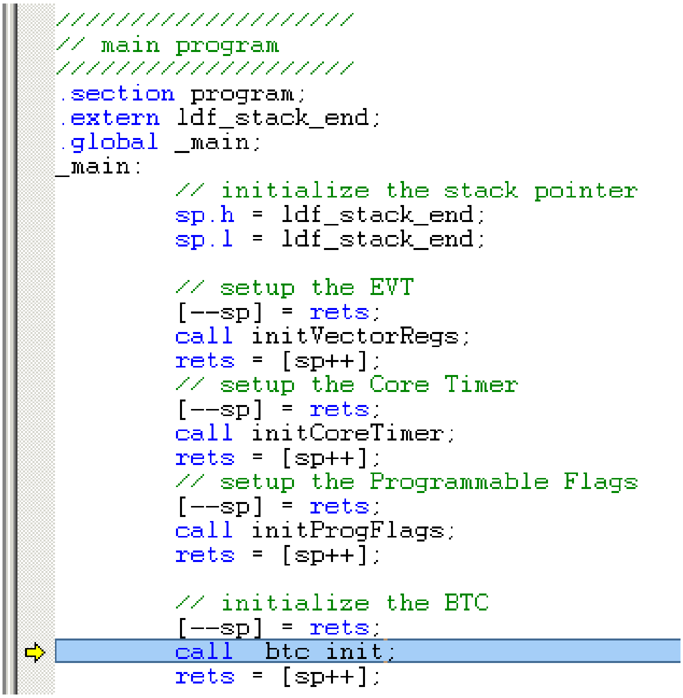


```
////////////////////
// BTC Definitions
////////////////////
BTC_MAP_BEGIN
//             Channel Name,               Starting Address,  Length
BTC_MAP_ENTRY('Timer Interrupt Counter', timerCounter,      0x00004)
BTC_MAP_ENTRY('PF4 Counter',             pf4Counter,        0x00004)
BTC_MAP_ENTRY('PF5 Counter',             pf5Counter,        0x00004)
BTC_MAP_ENTRY('256 byte channel (PF6)',  pf6Array,          0x00100)
BTC_MAP_ENTRY('256 byte channel (PF7)',  pf7Array,          0x00100)
BTC_MAP_ENTRY('256 byte channel',        array1,            0x00100)
BTC_MAP_ENTRY('4k byte channel',         array2,            0x01000)
BTC_MAP_END
```

Figure 3-14. BTC Channel Definitions

3. Scroll down to the `main program`.

   Twenty lines below the `_main:` label is the command to initialize BTC, `_btc_init;` (shown in Figure 3-15).

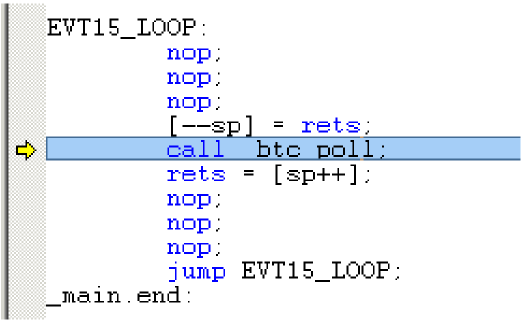


```
///////////////////////
// main program
///////////////////////
.section program;
.extern ldf_stack_end;
.global _main;
_main:
        // initialize the stack pointer
        sp.h = ldf_stack_end;
        sp.l = ldf_stack_end;

        // setup the EVT
        [--sp] = rets;
        call initVectorRegs;
        rets = [sp++];
        // setup the Core Timer
        [--sp] = rets;
        call initCoreTimer;
        rets = [sp++];
        // setup the Programmable Flags
        [--sp] = rets;
        call initProgFlags;
        rets = [sp++];

        // initialize the BTC
        [--sp] = rets;
        call _btc_init;
        rets = [sp++];
```

Figure 3-15. BTC Initialize Command

For more information about `_btc_init`, refer to the BTC Help included with VisualDSP++. The help file is in the installation directory at `VisualDSP++ 3.5 16-Bit\Help\btc.chm`.

In this example, the _btc_poll command is placed in the EVT15_LOOP, defined below main and shown in Figure 3-16.



```
EVT15_LOOP:
        nop;
        nop;
        nop;
        [--sp] = rets;
        call  btc poll;
        rets = [sp++];
        nop;
        nop;
        nop;
        jump EVT15_LOOP;
_main.end:
```

Figure 3-16. BTC Polling Loop

Refer to the BTC Help for more information about _btc_poll. This function is called when the evt15 interrupt is triggered. This interrupt has the lowest priority on this particular processor.

Now that you have seen how BTC has been added to this example, it is time to build the project.

4. On the toolbar click **Rebuild All** or select **Rebuild All** from the **Project** menu.

   This command builds the project and automatically downloads the application to the target. For details about building projects, refer to "Exercise One: Building and Running a C Program" on page 2-3.

Step 3: Set Up the BTC Memory Window and View Data

1. From the **View** menu, choose **Debug Windows** and **BTC Memory** as shown in Figure 3-17.

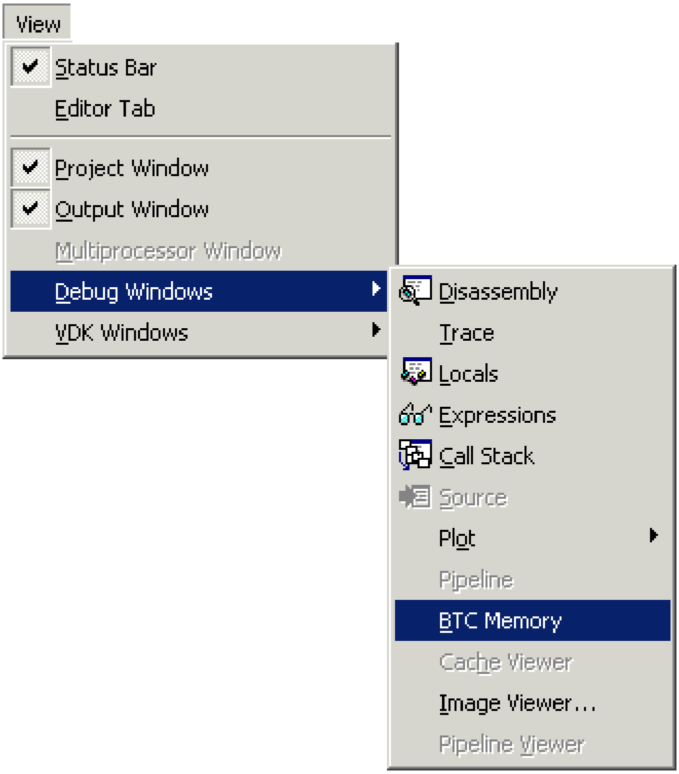


Figure 3-17. BTC Memory Menu Option

The **BTC Memory** window, shown in Figure 3-18, is displayed.

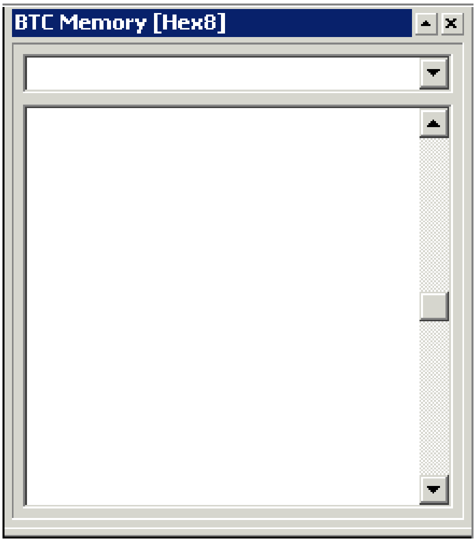


Figure 3-18. BTC Memory Window

The **BTC Memory** window is used to display BTC data in real time. Data is read from the target when the IDDE issues a read request, and is written when a value is edited in the **BTC Memory** window. You can adjust the rate at which the IDDE requests data by changing the refresh rate.

2. Right-click in the **BTC Memory** window to display a menu of features, shown in Figure 3-19 on page 3-28.

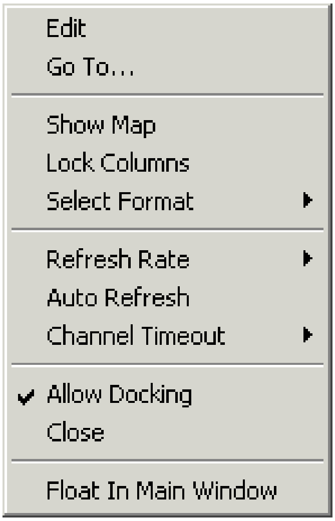


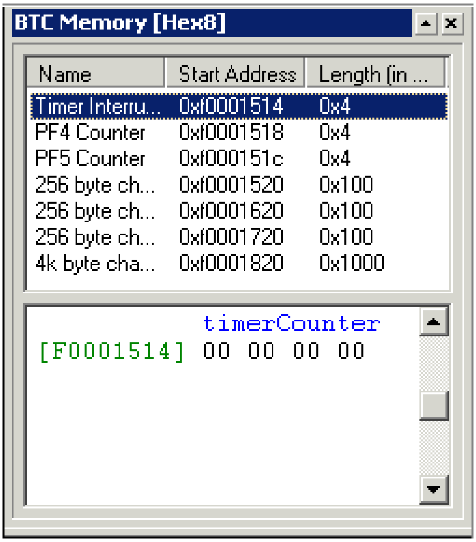
Figure 3-19. BTC Memory Window Right-Click Menu

Each menu option is described as follows.

Edit – Right click on a memory location in the **BTC Memory** window and select **Edit** to modify the value in that location. You can also edit by left-clicking on a memory value in the window and typing in a new value.

Go To – Enables you to enter an address or browse for a symbol, and displays memory starting at that address in the BTC Memory window. If the address entered is outside the range of the defined BTC channels, an error message is displayed.

Show Map – When this option is enabled, a map of all the defined channels is displayed, as shown in Figure 3-20 on page 3-29.



Name	Start Address	Length (in ...
Timer Interru...	0xf0001514	0x4
PF4 Counter	0xf0001518	0x4
PF5 Counter	0xf000151c	0x4
256 byte ch...	0xf0001520	0x100
256 byte ch...	0xf0001620	0x100
256 byte ch...	0xf0001720	0x100
4k byte cha...	0xf0001820	0x1000

```
           timerCounter
[F0001514] 00 00 00 00
```

Figure 3-20. BTC Memory Window with Map

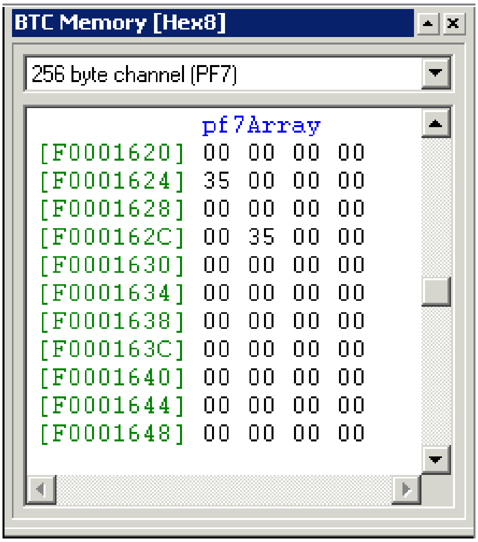
Double-clicking on a channel displays the corresponding memory in the **BTC Memory** window. When **Show Map** is disabled, you can choose a channel from a drop-down list selected from the **BTC Memory** window right-click menu (Figure 3-19 on page 3-28).

Lock Columns – Locks the number of columns displayed in the **BTC Memory** window.

- If this option is not enabled, VisualDSP++ displays as many columns as the window's width can accommodate.

- If this option is enabled, the number of columns does not change, regardless of the window's width. For example, if four columns are displayed when the option is enabled, four columns are displayed, regardless of the window's width. See Figure 3-21, Figure 3-22, and Figure 3-23 for comparisons.

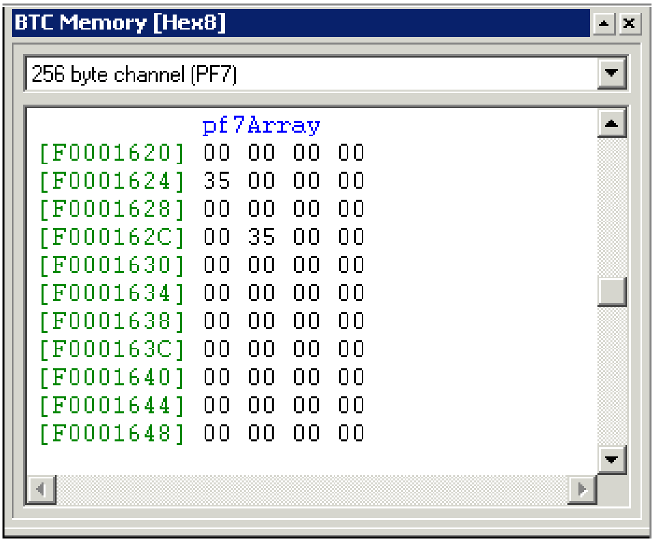
Figure 3-21 shows the original window.



```
           pf7Array
[F0001620] 00  00  00  00
[F0001624] 35  00  00  00
[F0001628] 00  00  00  00
[F000162C] 00  35  00  00
[F0001630] 00  00  00  00
[F0001634] 00  00  00  00
[F0001638] 00  00  00  00
[F000163C] 00  00  00  00
[F0001640] 00  00  00  00
[F0001644] 00  00  00  00
[F0001648] 00  00  00  00
```

Figure 3-21. Original Window Width

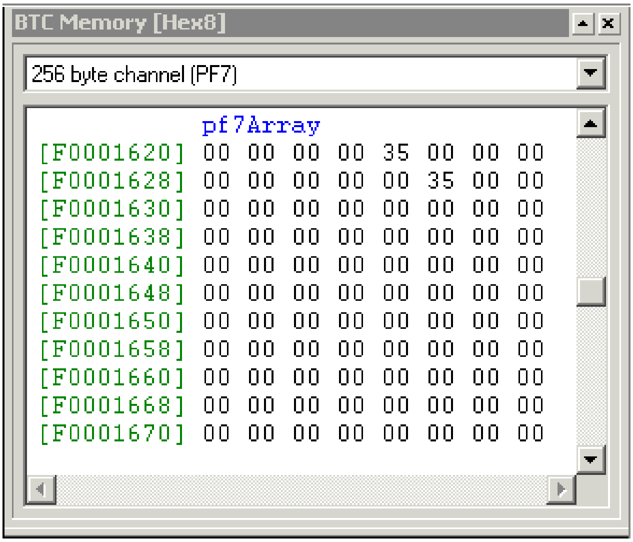
Figure 3-22 shows the original window expanded with **Lock Columns** enabled.



```
             pf7Array
[F0001620] 00 00 00 00
[F0001624] 35 00 00 00
[F0001628] 00 00 00 00
[F000162C] 00 35 00 00
[F0001630] 00 00 00 00
[F0001634] 00 00 00 00
[F0001638] 00 00 00 00
[F000163C] 00 00 00 00
[F0001640] 00 00 00 00
[F0001644] 00 00 00 00
[F0001648] 00 00 00 00
```

Figure 3-22. Expanded Window with Lock Columns Enabled

Figure 3-23 shows the original window expanded with **Lock Columns** disabled.



256 byte channel (PF7)

```
           pf7Array
[F0001620] 00 00 00 00 35 00 00 00
[F0001628] 00 00 00 00 00 35 00 00
[F0001630] 00 00 00 00 00 00 00 00
[F0001638] 00 00 00 00 00 00 00 00
[F0001640] 00 00 00 00 00 00 00 00
[F0001648] 00 00 00 00 00 00 00 00
[F0001650] 00 00 00 00 00 00 00 00
[F0001658] 00 00 00 00 00 00 00 00
[F0001660] 00 00 00 00 00 00 00 00
[F0001668] 00 00 00 00 00 00 00 00
[F0001670] 00 00 00 00 00 00 00 00
```

Figure 3-23. Expanded Window with Lock Columns Disabled

Select Format – Enables you to select how memory is displayed. The choices are 8-bit, 16-bit, and 32-bit hex.

Refresh Rate – Enables you to choose the rate at which the **BTC Memory** window is refreshed. The IDDE issues a read request based on the rate you select. You may choose one of the preexisting options (1, 5, 10, or 15 seconds), or use a custom refresh rate. The custom rate is specified in milliseconds.

Auto Refresh – Automatically refreshes the **BTC Memory** window, based on the refresh rate you select. If this option is disabled, the **BTC Memory** window is not refreshed until the program is halted.

Channel Timeout – The amount of time that VisualDSP++ will wait for a memory request to the target. After this time, the IDDE stops polling the BTC to prevent a hang.

Allow Docking – Docking locks the **BTC Memory** window to a fixed location (for example, the right side of the workspace). Disabling docking enables you to position the window anywhere in the workspace, including on top of docked windows.

Close – Closes the **BTC Memory** window

Float In Main Window – Disables docking and centers the **BTC Memory** window in the center of the workspace. You can then move it to any location, but it will not dock. If you move it to a location shared by a docked window, the docked window will sit on top.

3. Select the **Timer Interrupt Counter** channel from the drop-down list in the **BTC Memory** window. Set the **Refresh Rate** to `1` second, and enable **Auto Refresh**.

4. Run the program. Notice how the values in the **BTC Memory** window are updated each second.

5. Select the **PF4 Counter** channel. This channel counts the number of times that the **PF4** button on the ADSP-BF535 EZ-KIT Lite board is pressed. Press this button and watch the **PF4 Counter** increment in the **BTC Memory** window.

   You have now seen some of the basic functionality of BTC.

6. Halt the program and close the `Btc_AsmDemo` project.

You are now ready to run the BTC FFT demo.

Running the BTC FFT Demo

The BTC FFT demo demonstrates the transfer of data from the Blackfin EZ-KIT Lite over background telemetry channels. The program generates an input sine wave that increases in frequency over time, and performs a Fast Fourier Transform (FFT) on this input signal. The input and output data are transferred to the IDDE over BTC.

Figure 3-24 provides an overview of the data transfer in the BTC FFT demo.

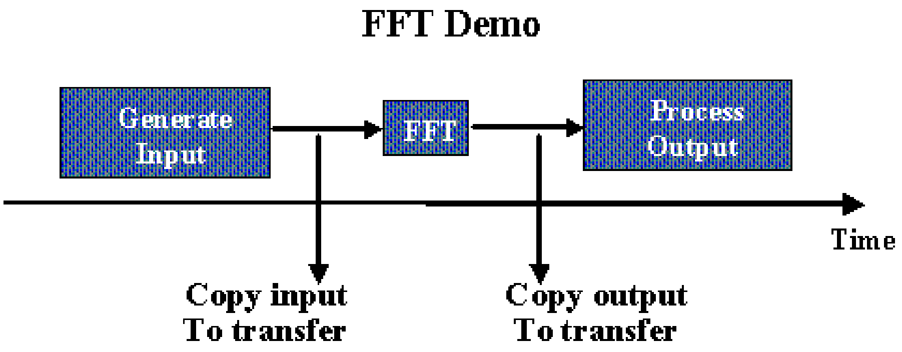


Figure 3-24. Data Transfer in the BTC FFT Demo

For more information, see the included `readme.txt` file.

Make sure that you are in a BF535 emulator HP PCI session.

Step 1: Build the FFT Demo

1. Open the FFT demo, located in the following folder.

```
\Program Files\Analog Devices\VisualDSP 3.5 16-Bit\
Blackfin\Examples\BTC Examples\BF535\BTC_fft
```

2. Build the FFT project by clicking **Rebuild All** on the toolbar or by selecting **Rebuild All** from the **Project** menu.

   This command builds the project and automatically downloads the application to the target. For more details about building projects, refer to "Exercise One: Building and Running a C Program" on page 2-3.

Step 2: Plot BTC Data

1. Open the **BTC Memory** window if it is not already open.
2. From the **View** menu, select **Debug Windows**, **Plot**, and then **Restore**, as shown in Figure 3-25.

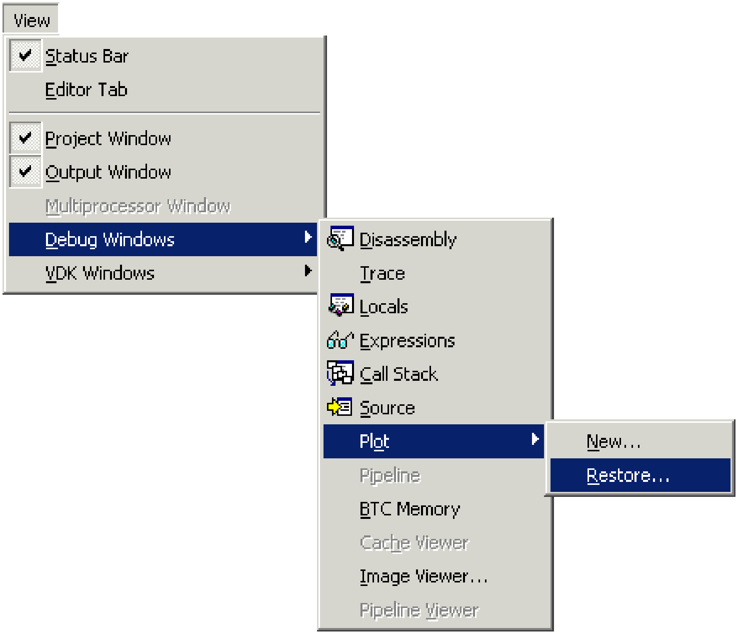
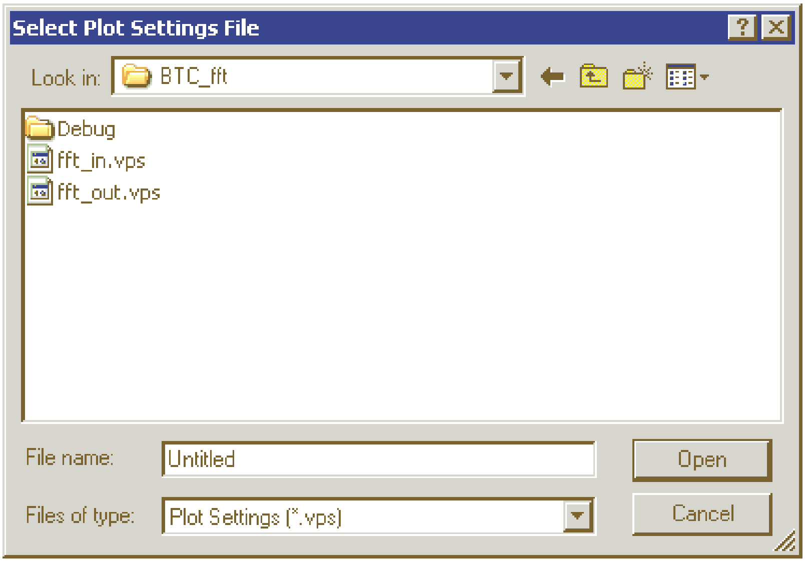


Figure 3-25. Plot Restore Menu Option

The **Restore** command opens the **Select Plot Settings File** window, shown in Figure 3-26.


Select Plot Settings File

Look in: BTC_fft

Debug
fft_in.vps
fft_out.vps

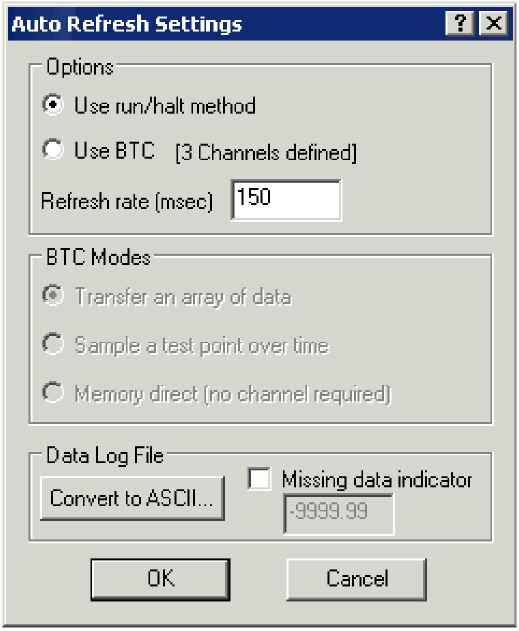
File name: Untitled

Files of type: Plot Settings (*.vps)

Figure 3-26. Select Plot Settings File Window

Select the `fft_in.vps` file and open it. A plot window appears. Follow the same procedure to restore the `fft_out.vps` file.

3. Right-click in the **FFT In** plot window and select **Auto Refresh Settings** to open the **Auto Refresh Settings** dialog box, shown in Figure 3-27.



Auto Refresh Settings

Options

● Use run/halt method

○ Use BTC [3 Channels defined]

Refresh rate (msec) 150

BTC Modes

● Transfer an array of data

○ Sample a test point over time

○ Memory direct (no channel required)

Data Log File

□ Missing data indicator -9999.99

Figure 3-27. Auto Refresh Settings Dialog Box

This dialog box enables you to configure the plotting tool to plot the BTC data in real-time.

4. Complete the dialog box as follows.

In the **Options** group box, select the **Use BTC** option.

The **Use run/halt method** option plots the data, but refreshes the plot window only when the program is halted.

The **Refresh rate** enables you to choose the interval between plot window refreshes. Use the default setting of `150` milliseconds.

The **BTC Modes** group box includes two methods of transferring data to the plot window:

- **Transfer an array of data** – This method uses the `btc_write_array` function. Data is captured at a specific point in the DSP application, is copied to a transfer buffer, and is held until the host reads the data.
- **Sample a test point over time** – This method uses a data buffer in the DSP program and the `btc_write_value` function.The sampled input data value is copied to the data transfer buffer, and is read according to the plot refresh rate. The minimum size of the transfer buffer is the product of the plot refresh rate and the data sampling rate (PRR * DSR).

Use the default setting, **Transfer an array of data**.

In the **Data Log File** group box, the **Convert to ASCII** button enables you to convert log data to ASCII format. This subject will be discussed in more detail shortly.

5. Click **OK** to close the **Auto Refresh Settings** dialog box.
6. Enable the **Use BTC** option in the **FFT Out** window as you did in step 4.

7. Right-click in both plot windows and enable **Auto Refresh**.

   A toolbar appears at the top of each plot window, as shown in Figure 3-28. This toolbar enables you to record BTC data to a file and play back BTC data from a file.

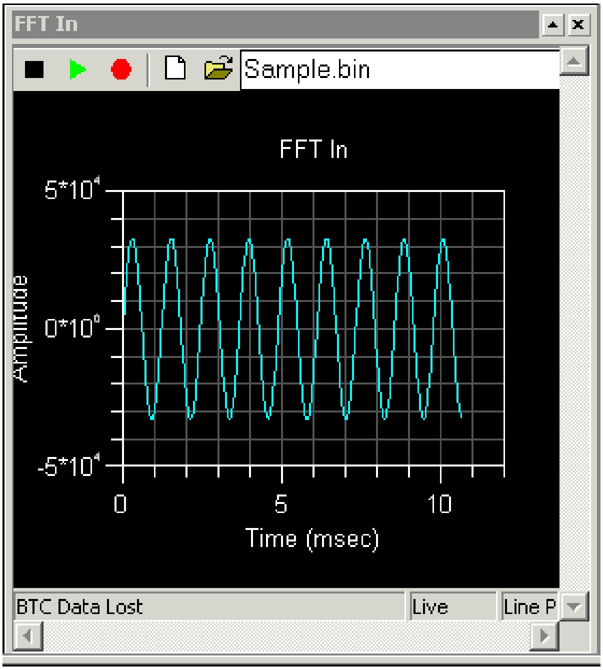


Figure 3-28. Plot Window with Toolbar

8. In the **FFT In** plot window, enter a file name, such as `Sample.bin`, in the text box.

9. Run the program.

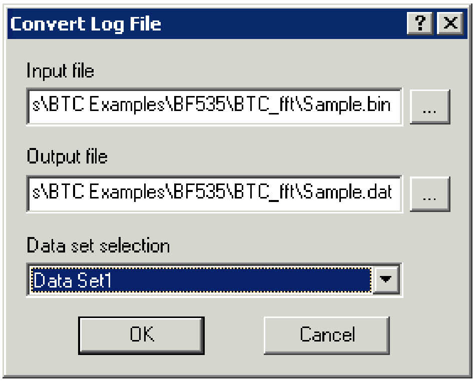
   Both plot windows should display data being plotted in real-time.

Step 3: Record and Analyze BTC Data

1. In the **FFT In** plot window toolbar, click **Record** ● . All data in the `FFT_Input` channel is logged to a file until you stop recording.
2. In the **BTC Memory** window, select the `FREQ STEP SIZE` channel.

   First, right-click and change the format to `Hex32`. Then change the value in memory from `100` to `200`, and notice its effect on the plots. If you would like, try using other values.
3. In the **FFT In** plot window toolbar, click **Stop** ■ to stop logging BTC data.
4. Halt the program.
5. In the **FFT In** plot window toolbar, click **Play** ▶ .

   The plot window displays the data that was logged. The window should appear as if the FFT program is still running.
6. Right-click in the **FFT In** plot window, open the **Auto Refresh Settings** dialog box, and click the **Convert to ASCII** button. The **Convert Log File** dialog box, shown in Figure 3-29, is displayed.



Input file: s\BTC Examples\BF535\BTC_fft\Sample.bin

Output file: s\BTC Examples\BF535\BTC_fft\Sample.dat

Data set selection: Data Set1

Figure 3-29. Convert Log File Dialog Box

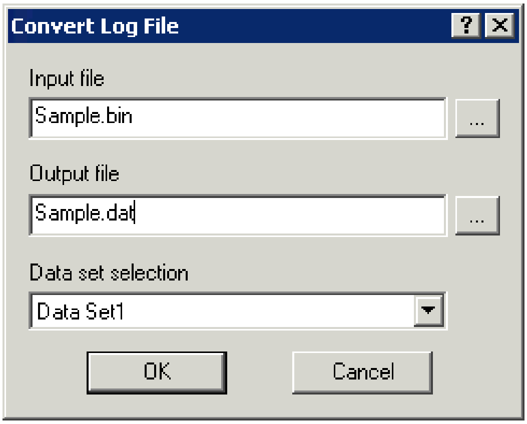
7. Complete the dialog box as follows.

   In the **Input file** text box, use the browse button ... to select `Sample.bin`.

   If `Sample.bin` contained more than one data set, you would be able to choose from among them in the **Data set selection** drop-down list. `Sample.bin` has only one data set, which is selected when you enter the **Input file** name.

   Next, click the file browse button ... next to the **Output file** text box. The **Select Log Output File** dialog box that appears should have the file name `Sample.dat` already in the **File name** text box. Click **Save**.

   If your window matches the one shown in Figure 3-30, click **OK**. The log file is converted from binary to ASCII, which is readable by other programs.


Convert Log File

Input file: Sample.bin

Output file: Sample.dat

Data set selection: Data Set1

Figure 3-30. Completed Convert Log File Dialog Box

8. Launch Microsoft Excel. Then open the `Sample.dat` file and follow the instructions in the **Text Import Wizard**.

   The `.DAT` file is a tab delimited file. Importing the file into Excel or another program, such as MatLab, enables you to analyze or modify the log file.

You have now completed the BTC FFT demo and the Advanced Tutorial.

I INDEX