

VISUALDSP++[®] 3.5

Getting Started Guide

for 32-Bit Processors

Revision 1.0, March 2004

Part Number
82-000035-10

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



Copyright Information

© 2004 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

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

The Analog Devices logo, the CROSSCORE logo, TigerSHARC, SHARC, and VisualDSP++ are registered trademarks of Analog Devices, Inc.

EZ-KIT Lite is a trademark of Analog Devices, Inc.

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

CONTENTS

PREFACE

Purpose of This Manual	vii
Intended Audience	vii
Manual Contents	viii
What's New in This Manual	viii
Technical or Customer Support	ix
Supported Processors	x
Product Information	x
MyAnalog.com	xi
Processor Product Information	xi
Related Documents	xii
Online Technical Documentation	xiii
From VisualDSP++	xiv
From Windows	xiv
From the Web	xv
Printed Manuals	xv
VisualDSP++ Documentation Set	xv
Hardware Tools Manuals	xv
Processor Manuals	xv

CONTENTS

Data Sheets	xvi
Notation Conventions	xvi

FEATURES AND TOOLS

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

BASIC TUTORIAL

Overview	2-1
Exercise 1: Building a C Program	2-3
Step 1: Start VisualDSP++ and Open a Project	2-4
Step 2: Build the dotprodc Project	2-8
Step 3: Set Up the Debug Session	2-11
Step 4: Run dotprodc	2-16
Exercise 2: Calling an Assembly Routine	2-17
Step 1: Create a New Project	2-17
Step 2: Add Source Files to dot_product_asm	2-21
Step 3: Create a Linker Description File	2-22
Step 4: Modify the Project Source Files	2-26
Step 5: Modify dot_prod_asm.ldf	2-29
Step 6: Rebuild and Run dot_product_asm	2-32
Exercise 3: Plotting Data	2-33
Step 1: Load the Convolution Program	2-33
Step 2: Open a Plot Window	2-35

Step 3: Run the Convolution Program	2-40
Exercise 4: Linear Profiling	2-45
Step 1: Load the Convolution Program	2-45
Step 2: Open the Profiling Window	2-46
Step 3: Collect and Examine Linear Profile Data	2-49
Exercise 5: Using a VCSE Component	2-52
Step 1: Start VisualDSP++ and Open the Project	2-52
Step 2: Install the EXAMPLES::CULawc Component	2-54
Step 3: Add the Component to Your Project	2-57
Step 4: Build and Run the Program	2-59

ADVANCED TUTORIAL

Overview	3-1
Exercise 1: Using PGO	3-2
Step 1: Load the Project	3-4
Step 2: Configure a Data Set	3-6
Step 3: Attach an Input Stream	3-10
Step 4: Configure Additional Data Sets	3-14
Step 5: Create PGO Files, Optimize the Program	3-16
Step 6: Compare Execution Times	3-17
Exercise 2: Using BTC	3-21
Adding BTC to Your DSP Application	3-21
Running the BTC Assembly Demo	3-23
Step 1: Load the Btc_AsmDemo Project	3-24
Step 2: Examine the BTC Commands	3-25

CONTENTS

Step 3: Set Up the BTC Memory Window and View Data	3-28
Running the BTC FFT Demo	3-36
Step 1: Build the FFT Demo	3-37
Step 2: Plot BTC Data	3-38
Step 3: Record and Analyze BTC Data	3-43

INDEX

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 32-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 DSP 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 DSP architecture and instruction set.

Manual Contents

The manual's contents are summarized as follows.

- 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 Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at:
<http://www.analog.com/processors/technicalSupport>
- E-mail tools questions to:
dsptools.support@analog.com
- E-mail processor questions to:
dsp.support@analog.com
- Phone questions to: **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor.
- Send questions by mail to:

Analog Devices, Inc.
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA

Supported Processors

The names “*SHARC*®” and “*TigerSHARC*®” refer to the family of Analog Devices 32-bit, digital signal processors. VisualDSP++ currently supports the following SHARC processors.

ADSP-21020	ADSP-21261
ADSP-21060	ADSP-21262
ADSP-21061	ADSP-21266
ADSP-21062	ADSP-21267
ADSP-21065L	ADSP-21363
ADSP-21160	ADSP-21364
ADSP-21161	ADSP-21365

VisualDSP++ currently supports the following TigerSHARC processors.

ADSP-TS101	ADSP-TS202
ADSP-TS201	ADSP-TS203

Product Information

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

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

MyAnalog.com

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

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

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

Processor Product Information

For information about embedded processors and DSPs, visit our Web site at www.analog.com/processors. This site 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 the following ways.

- E-mail 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 FTP Web site at:
[ftp ftp.analog.com](ftp://ftp.analog.com) or [ftp 137.71.23.21](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 32-Bit Processors

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

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

VisualDSP++ 3.5 Linker and Utilities Manual for 32-Bit Processors

VisualDSP++ 3.5 Assembler and Preprocessor Manual for SHARC Processors

VisualDSP++ 3.5 Assembler and Preprocessor Manual for TigerSHARC Processors

VisualDSP++ 3.5 Loader Manual for 32-Bit Processors

VisualDSP++ 3.5 Product Release Bulletin for 32-Bit Processors

VisualDSP++ 3.5 Kernel (VDK) User's Guide for 32-Bit Processors

VisualDSP++ 3.5 Component Software Engineering User's Guide for 32-Bit Processors

Quick Installation Reference Card

For hardware information, refer to your processors's hardware reference, programming reference, or data sheet. All documentation is available online. Most documentation is available in printed form.

To access all processor and tools manuals and data sheets, visit the Technical Library Web site at:

<http://www.analog.com/processors/resources/technicalLibrary>

Online Technical Documentation

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

Each documentation file type is described in the following table.

File	Description
.CHM	Help system files and manuals in Help format
.HTM or .HTML	Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the .HTML files 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 running the Tools installation. Access the online documentation from the VisualDSP++ environment, Windows[®] Explorer, or the Analog Devices Web site.

Product Information

From VisualDSP++

From VisualDSP++ environment:

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

Using Windows Explorer

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

Using the Windows Start Button

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

From the Web

Download manuals at the following Web site:

<http://www.analog.com/processors/resources/technicalLibrary/manuals>

Select a processor family and book title. Download archive (.ZIP) files, one for each manual. Use any archive management software, such as WinZip, to decompress downloaded files.

Printed Manuals

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

VisualDSP++ Documentation Set

To purchase VisualDSP++ manuals, call 1-603-883-2430. The manuals may be purchased only as a kit.

If you do not have an account with Analog Devices, you are referred to Analog Devices distributors. For information on our distributors, log onto <http://www.analog.com/salesdir/continent.asp>.

Hardware Tools Manuals

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

Processor Manuals

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

Notation Conventions

Data Sheets

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

To have a data sheet faxed to you, call **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.

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. Code has been formatted to fit this manual's page width.

Example	Description
Close command (File menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the Close command appears on the File menu).
{this that}	Alternative items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as <i>this</i> or <i>that</i> . One or the other is required.
[this that]	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional <i>this</i> or <i>that</i> .
[this,...]	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of <i>this</i> .
.SECTION	Commands, directives, keywords, and feature names are in text with letter gothic font.

Example	Description
	This symbol indicates a note that provides supplementary information on a related topic. In the online version of this book, the word Note appears instead of this symbol.
	This symbol indicates a warning that advises on an inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word Warning appears instead of this symbol.

Notation Conventions

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.

VisualDSP++ Features

- **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.

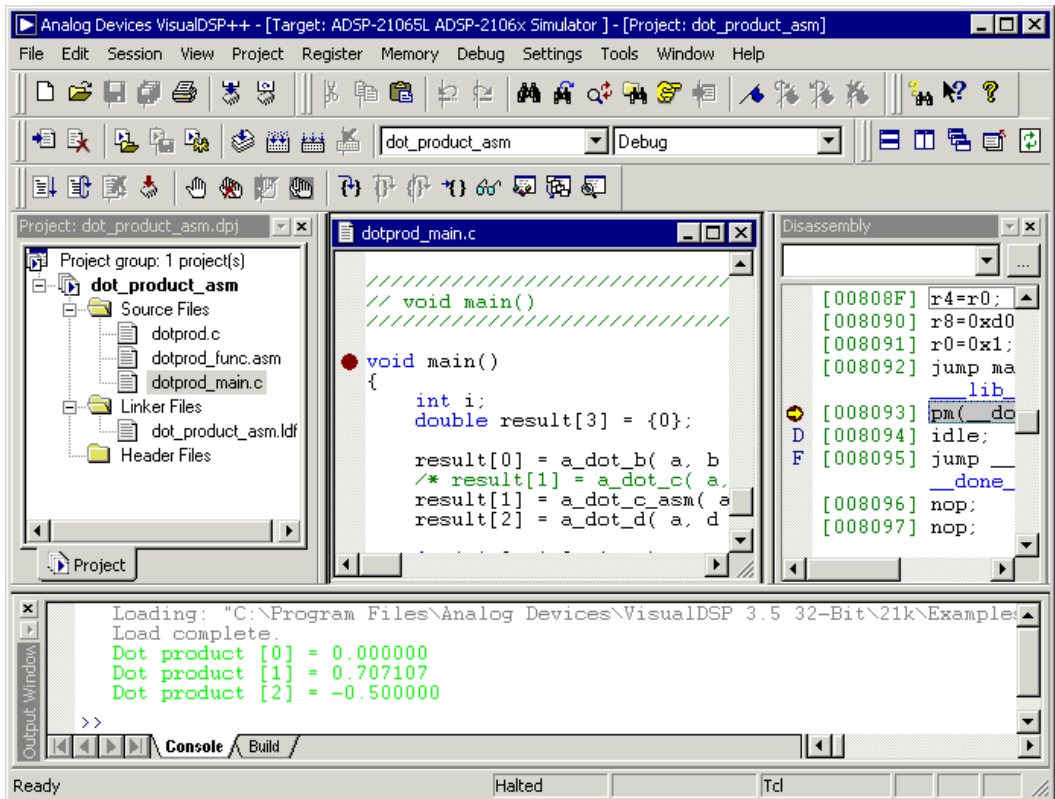


Figure 1-1. The VisualDSP++ IDDE

VisualDSP++ Features

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.** These new SHARC processors are supported: ADSP-21261, ADSP-21262, ADSP-21266, ADSP-21267, ADSP-21363, ADSP-21364, and ADSP-21365. Thsee new TigerSHARC processors are supported: ADSP-TS202 and ADSP-TS203. Refer to the processor's 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.
- **Menus with Icons.** Icons now appear beside menu commands that have corresponding toolbar buttons.

New Features in Release 3.5

- **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.
- **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.
- **Program flow manipulation for the TigerSHARC cycle-accurate simulator.** You can change the program counter value in the **PC Register** window to specify which instruction the simulator executes next. When you change a PC value, the simulator automatically flushes the pipeline and aborts all its instructions. Normal execution then resumes from a memory address indicated by the new PC value.

- **New data format support in Memory and Register windows.** Data in a **Register** or **Memory** window can be displayed in 8-bit or 32-bit character format. When compiled and loaded into memory, a code `char string[]` can be displayed in a **Memory** window as defined. Select the memory address of the symbol string and choose the display format for which the code was compiled.
- **New Select Loader Program command added to the Settings→Simulator menu for TigerSHARC processors.** You can specify a custom loader program that will run automatically every time before a user program is loaded.
- **Splitter support for TigerSHARC processors.** VisualDSP++ 3.5 includes a TigerSHARC splitter.

Code Development Tools

Code development tools for 32-bit processors include:

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

Code Development Tools

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” on page 2-1](#)
- [“Exercise 1: Building a C Program” on page 2-3](#)
- [“Exercise 2: Calling an Assembly Routine” on page 2-17](#)
- [“Exercise 3: Plotting Data” on page 2-33](#)
- [“Exercise 4: Linear Profiling” on page 2-45](#)
- [“Exercise 5: Using a VCSE Component” on page 2-52](#)

Overview

This tutorial demonstrates some of the 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 ADSP-21xxx processors. For these exercises, you will use the ADSP-2106x simulator for the ADSP-21065L target.

You can use a different ADSP-21xxx processor with only minor changes to the Linker Description File (.LDF) included with each project.

Overview

VisualDSP++ includes basic Linker Description Files for each processor type in the `ldf` folder. The default installation path for this folder is:

```
Analog Devices\VisualDSP++ 3.5\21k\ldf
```

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

The tutorial contains five exercises.

- In **Exercise 1**, you start up VisualDSP++, build a project containing C source code, set up a debug session, and run the program.
- In **Exercise 2**, you create a new project, use Expert Linker to create a Linker Description File for the project, modify sources to call an assembly routine, use Expert Linker to modify the `.LDF` file, and rebuild the project.
- In **Exercise 3**, you apply a simple convolution algorithm to a buffer of data. You use the VisualDSP++ plotting engine to view the different data arrays graphically.
- In **Exercise 4**, you use linear profiling to examine the efficiency of the convolution algorithm used in Exercise 3. Using the collected linear profile data, you pinpoint the most time-consuming areas of the algorithm, which are likely to require hand tuning in the assembly language.
- In **Exercise 5**, you install a VCSE component on your system and add the component to the project. Then you build and run the program with the component.

Tip: Become familiar with the VisualDSP++ toolbar buttons, shown in [Figure 2-1](#). They are shortcuts for menu commands such as **File**, **Open**. Toolbar buttons and menu commands that are not available for the task that you are performing are disabled and displayed in gray.

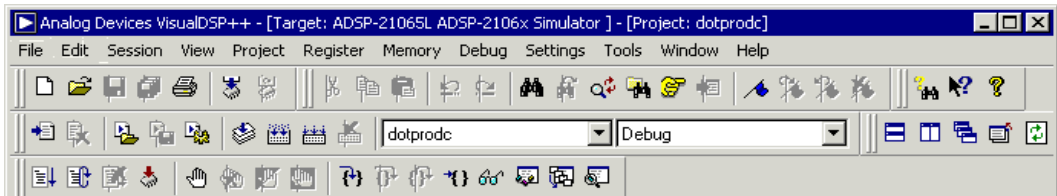


Figure 2-1. VisualDSP++ Toolbar Buttons

Exercise 1: Building a C Program

In this exercise, you:

- Start up the VisualDSP++ environment
- Open and build an existing project
- Set up the debug session and examine windows and dialog boxes
- Run the program

Exercise 1: Building a C 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\21k\  
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, VisualDSP,** and **VisualDSP++ Environment**.

If you are running VisualDSP++ for the first time, the **New Session** dialog box ([Figure 2-6 on page 2-12](#)) 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-2106x Family Simulator
Platform	ADSP-2106x Simulator
Session Name	ADSP-21065L ADSP-2106x Simulator
Processor	ADSP-21065L

- 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\21k\Examples\tutorial\dot_product_c`



This path is based on the default installation.

Exercise 1: Building a C Program

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

VisualDSP++ loads the project in the **Project** window, as shown in [Figure 2-2](#).



Figure 2-2. Project Loaded in the Project Window

The environment displays messages in the **Output** window as it processes the project settings and file dependencies.

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](#).

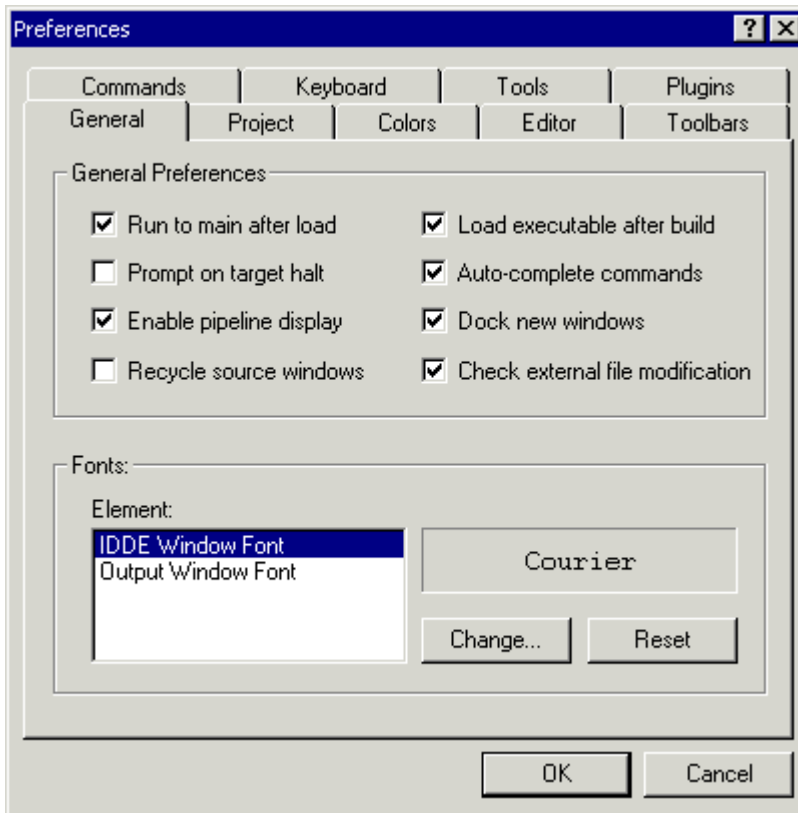


Figure 2-3. Preferences Dialog Box

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**

Exercise 1: Building a C Program

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

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 message in the **Output** window.

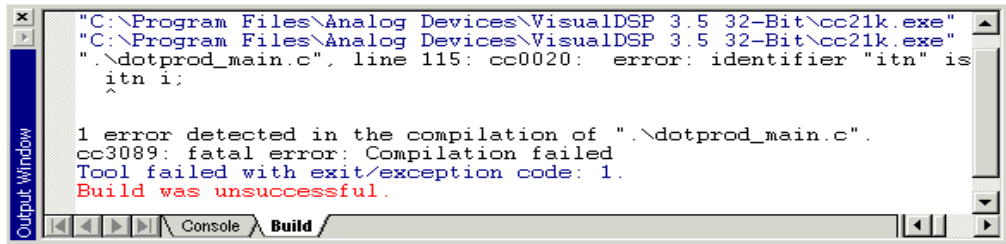


Figure 2-4. Example of Error Message

2. Double-click the error message (black) 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).

Exercise 1: Building a C Program

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

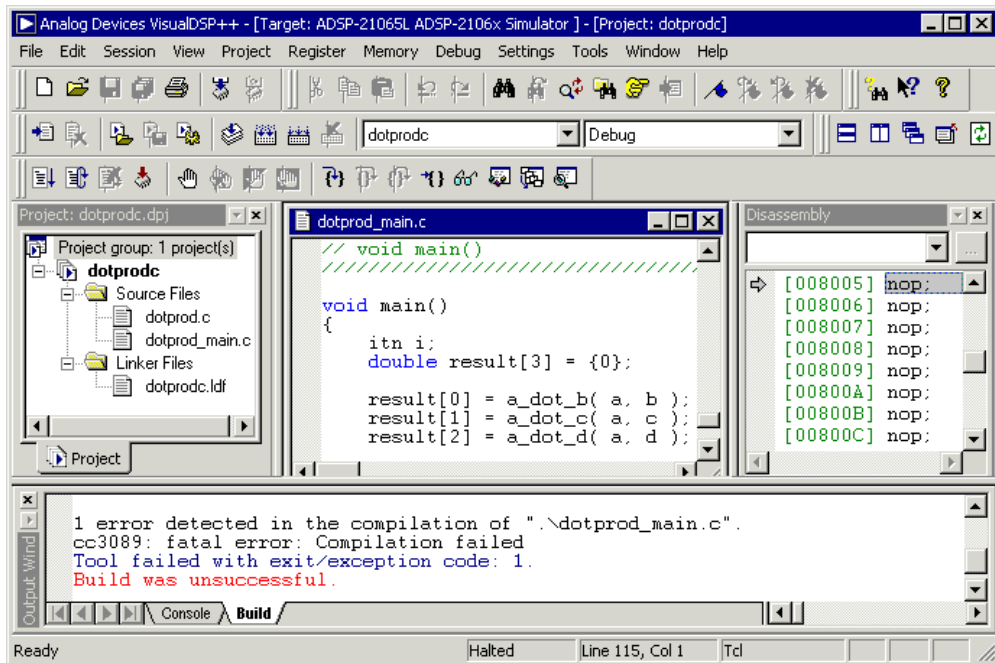


Figure 2-5. Output Window and Editor Window

3. In the editor window, click on `int` 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 in the **Build** view in the **Output** window.

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

Step 3: Set Up the Debug Session

In this procedure, you:

- 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 selected processor in the debug session 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.

Exercise 1: Building a C Program

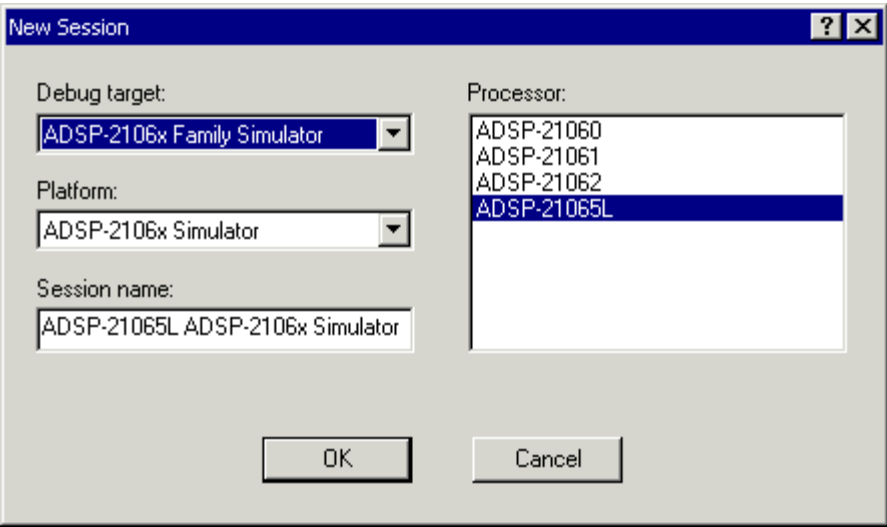


Figure 2-6. New Session Dialog Box

- 2. Specify the target and processor information listed in [Table 2-2](#).

Table 2-2. Session Specification

Box	Value
Debug Target	ADSP-2106x Family Simulator
Platform	ADSP-2106x Simulator
Session Name	ADSP-21065L ADSP-2106x Simulator
Processor	ADSP-21065L

- 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](#)).

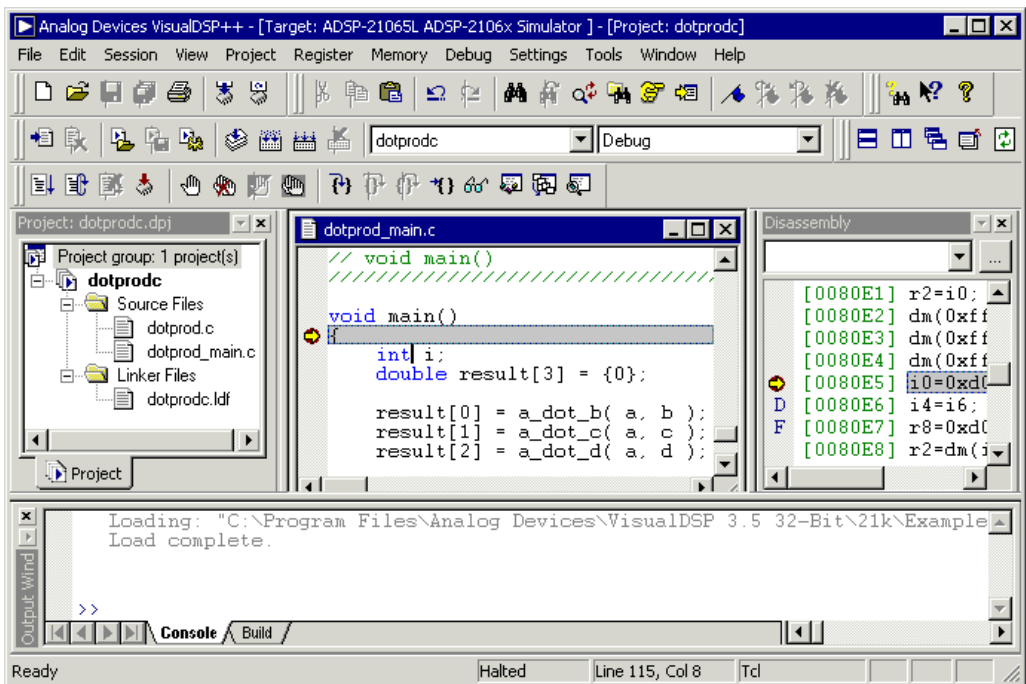


Figure 2-7. Loading dotprodc.dxe

Exercise 1: Building a C Program

5. Look at the information in the open windows.

The **Output** window's **Console** view 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 machine 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.

- 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](#).

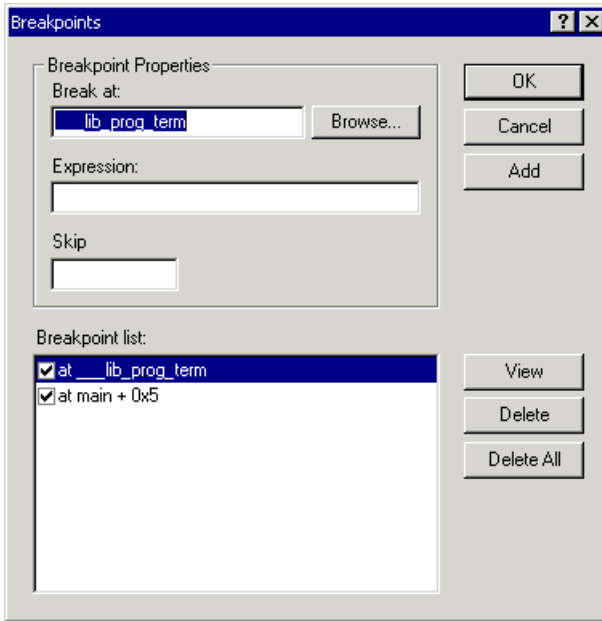


Figure 2-8. Breakpoints Dialog Box

The breakpoints are set at these C program labels:

- `__lib_prog_term`
- `"main + 0x5"`

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 cursor in the gutter before double-clicking.

Exercise 1: Building a C Program

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 in the **Console** view ([Figure 2-9](#)) in the **Output** window.

```
Dot product [0] = 0.000000
Dot product [1] = 0.707107
Dot product [2] = -0.500000
```

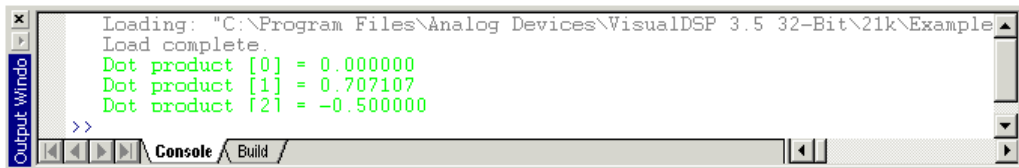


Figure 2-9. Results of the `dotprodc` Program

You are now ready to begin Exercise 2.

Exercise 2: Calling an Assembly Routine

In Exercise 1, you built and ran a C program. In this exercise, you modify this program to call an assembly language routine, create a Linker Description File to link with the assembly routine, and rebuild the project. The project files are largely identical to those of Exercise 1. 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 editor 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](#).

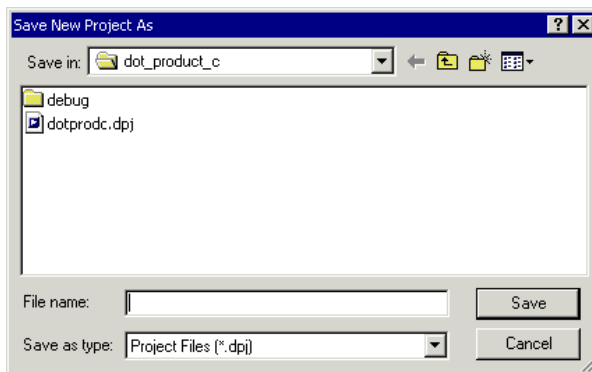


Figure 2-10. Save New Project As Dialog Box

Exercise 2: Calling an Assembly Routine

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.

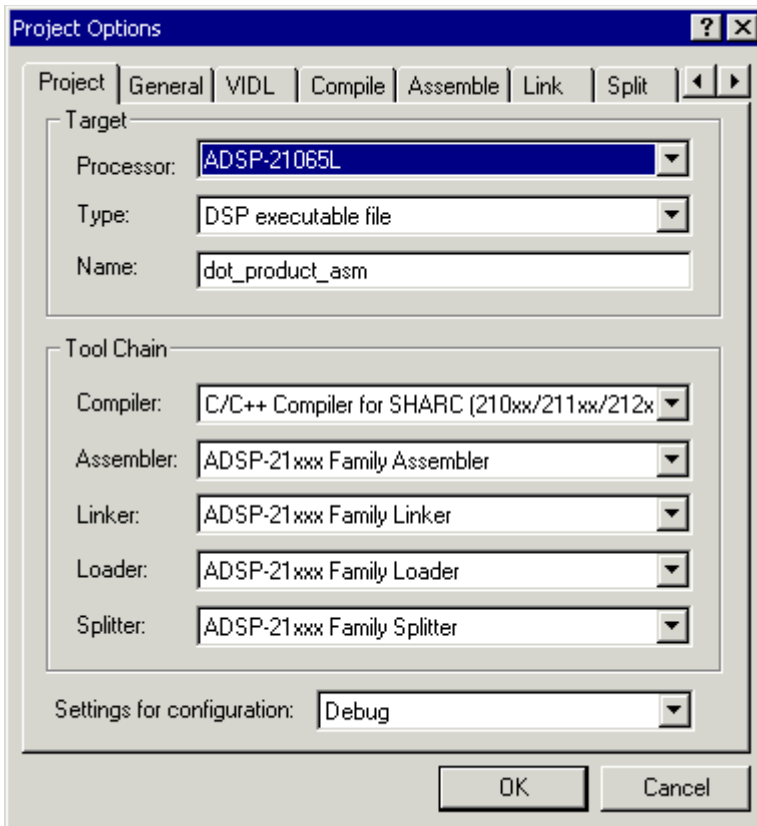


Figure 2-11. Project Options Dialog Box: Project Page

This dialog box enables you to specify project build information.

5. Take a moment to examine the tabbed pages in the **Project Options** window: **Project**, **General**, **VIDL**, **Compile**, **Assemble**, **Link**, **Split**, **Load**, and **Post Build**. On each page, you specify the tool options used to build the project.
6. On the **Project** page ([Figure 2-11](#)), specify the following values.

Table 2-3. Completing the Project Page

Box	Value
Processor	ADSP-21065L
Type	DSP executable file
Name	dot_product_asm
Settings for configuration	Debug

These settings specify information for building an executable file for the ADSP-21065L DSP. 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](#).

Exercise 2: Calling an Assembly Routine

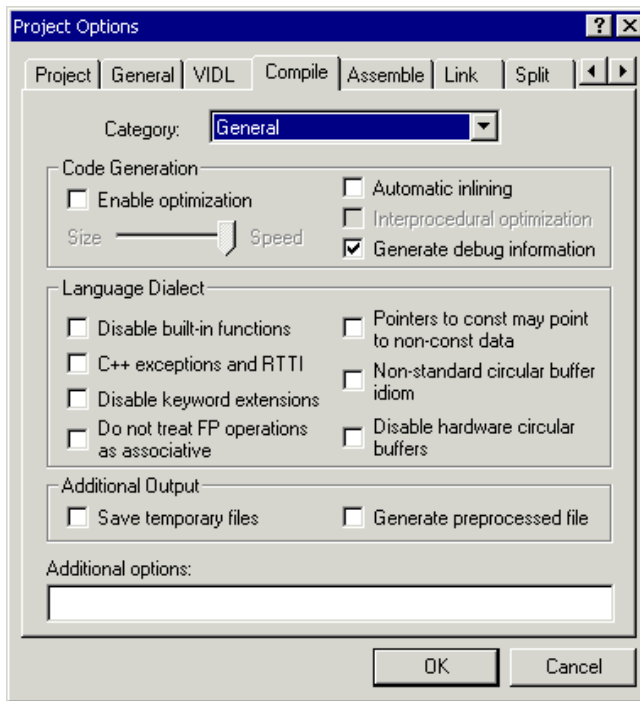


Figure 2-12. Project Options Dialog Box: Compile Page

8. In the **Code Generation** group box, select the **Generate debug information** check box, if it is not already selected, to enable debug information for the C source.
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**. Once added, kernel support cannot be removed.

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.

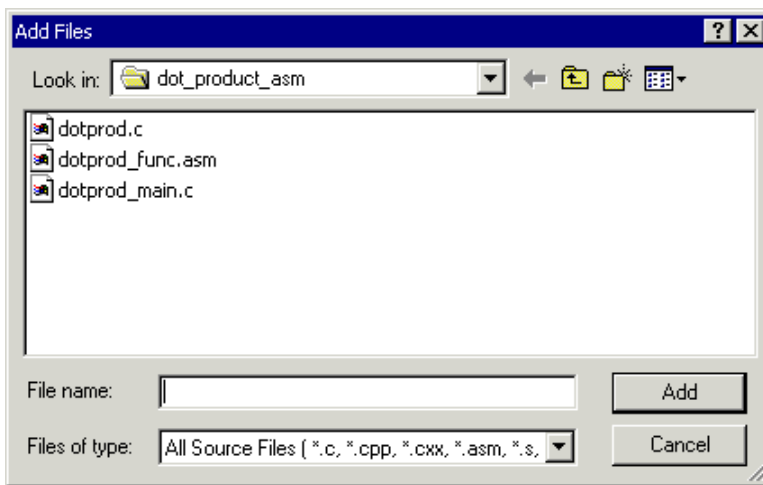


Figure 2-13. Add Files Dialog Box: Adding Source Files to the Project

2. In the **Look in** box, locate the project folder, `dot_product_asm`.
3. In the **Files of type** box, select **All Source Files**.
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.

Exercise 2: Calling an Assembly Routine

Step 3: Create a Linker Description File

In this procedure, you 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](#).

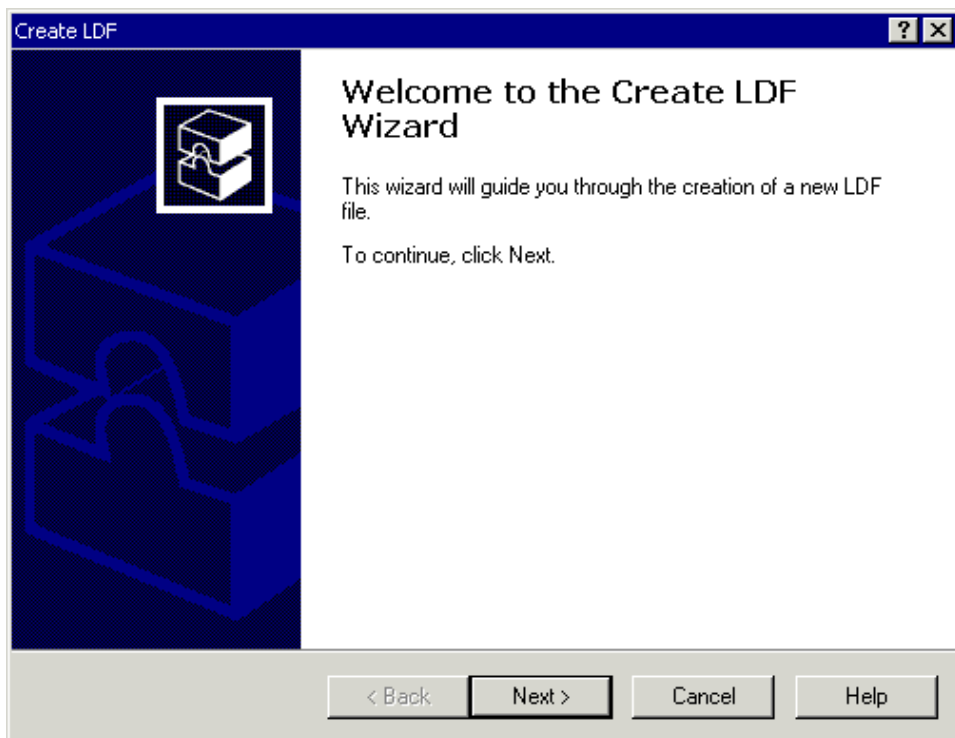


Figure 2-14. Create LDF Wizard

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

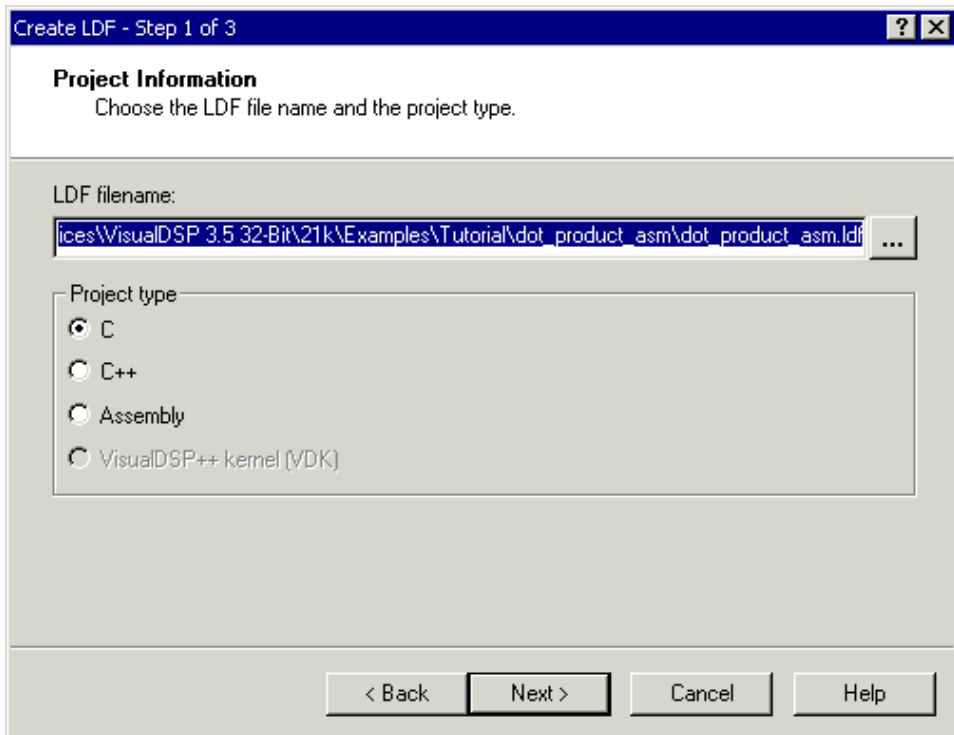


Figure 2-15. Create LDF – Step 1 of 3 Page

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](#).

Exercise 2: Calling an Assembly Routine

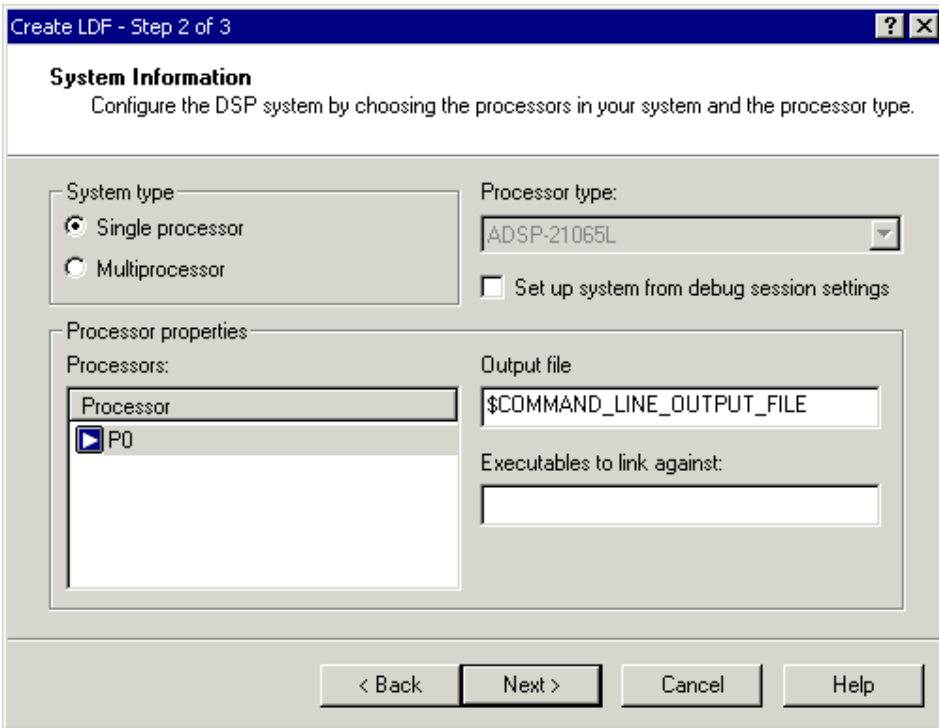


Figure 2-16. Create LDF – Step 2 of 3 Page

This page enables you to set the **System type** (defaulted to **Single processor**), the **Processor type** (defaulted to **ADSP-21065L** 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 next page (**Create LDF – Step 3 of 3**), shown in [Figure 2-17](#).

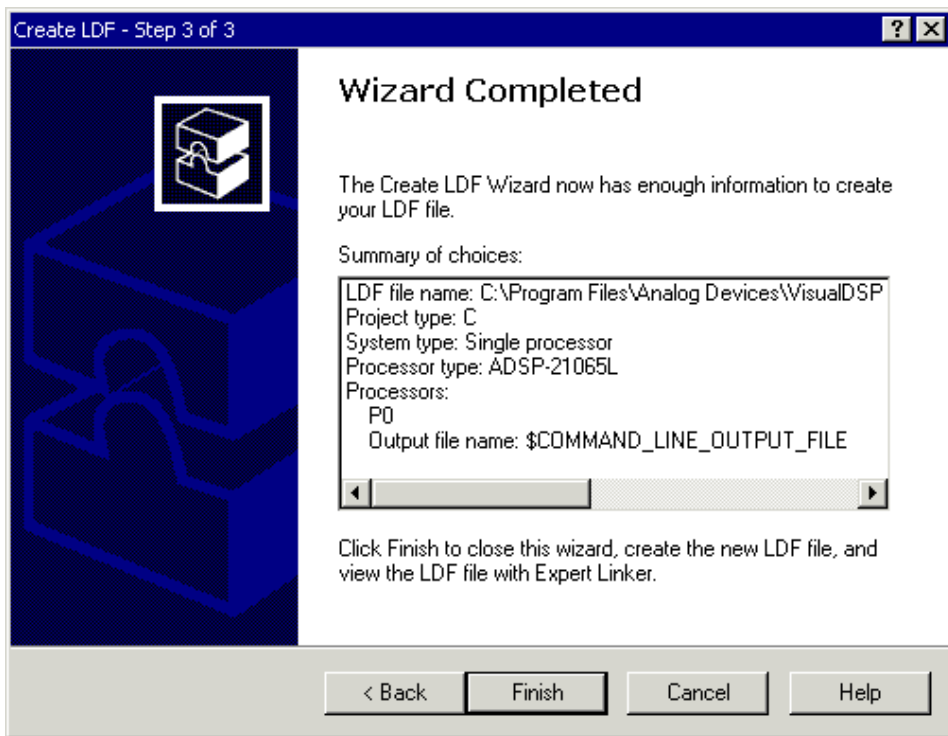


Figure 2-17. Create LDF – Step 3 of 3 Page

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 new .LDF file. This file is complete for this project. Close the **Expert Linker** window.

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

Exercise 2: Calling an Assembly Routine

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:

- 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.

If an editor window is not open, double-click `dotprod_main.c` in the **Project** window.

2. From the **Edit** menu, choose **Find** to open the **Find** dialog box, shown in [Figure 2-18](#).

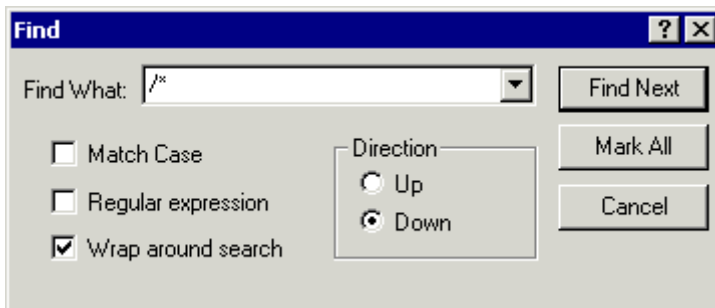


Figure 2-18. Find Dialog Box: Locating Occurrences of /*

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 double 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 1.

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

Exercise 2: Calling an Assembly Routine

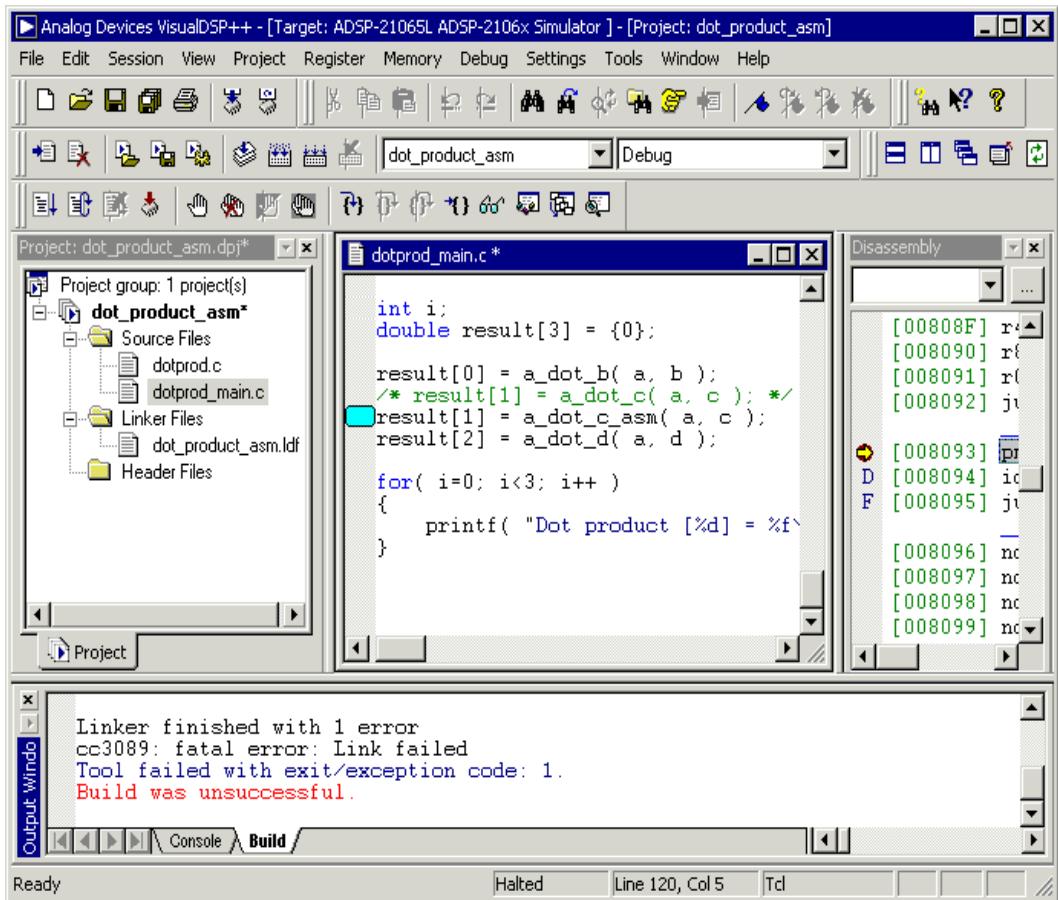


Figure 2-19. Editor Window: Modifying dotprod_main.c to Call a_dot_c_asm

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: Modify dot_prod_asm.ldf

In this procedure you:

- 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](#).

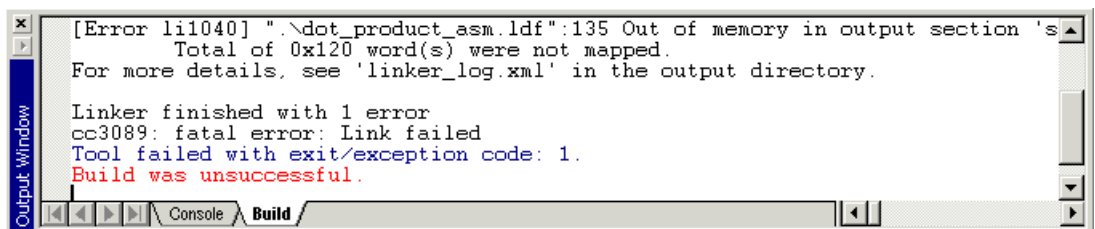


Figure 2-20. Output Window: Linker Error

Exercise 2: Calling an Assembly Routine

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

Resize the window to expand the 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**.

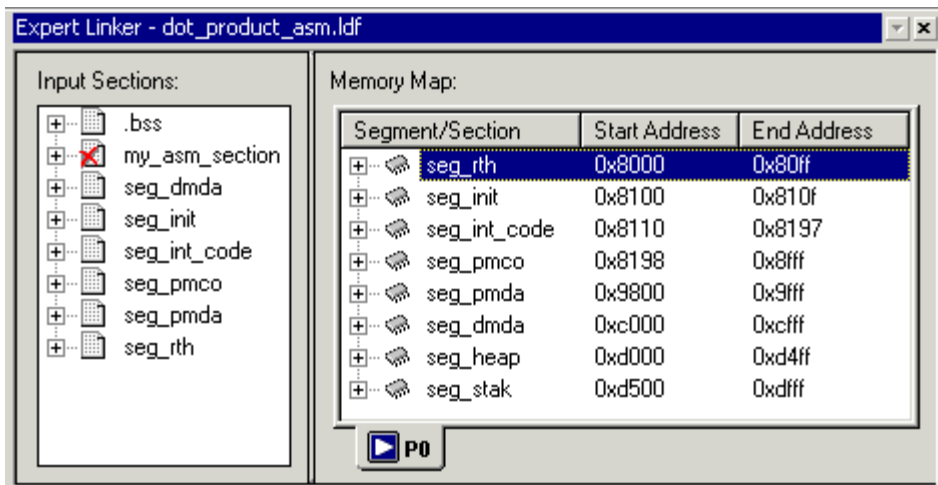


Figure 2-21. Expert Linker Window

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 `seg_pmco` as follows.

In the **Input Sections** pane, open the `my_asm_section` by clicking 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.

Drag the icon in front of `$OBJECTS` from the **Input Sections** pane onto `seg_pmco` in the **Memory Map** pane. Then expand the `seg_pmco` folder. As shown in Figure 2-22, the red X no longer appears because the section `my_asm_section` has been mapped.

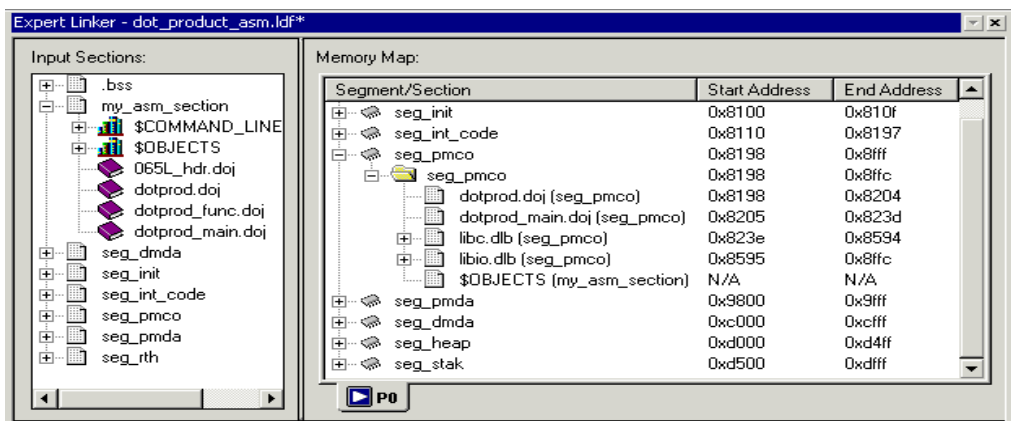


Figure 2-22. Dragging `$OBJECTS` onto Program Output Section

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

If you rebuild the project without saving the file, VisualDSP++ sees that you modified the file and automatically saves it.

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

Exercise 2: Calling an Assembly Routine

Step 6: Rebuild and Run dot_product_asm

To run dot_product_asm:

1. Build the project by clicking the **Build Project** button  or by choosing **Build Project** from the **Project** menu.

At the end of the build, the **Output** window displays “Build completed successfully” in the **Build** view. VisualDSP++ loads the program, runs to main, and displays the **Output**, **Disassembly**, and editor windows (shown in Figure 2-23).

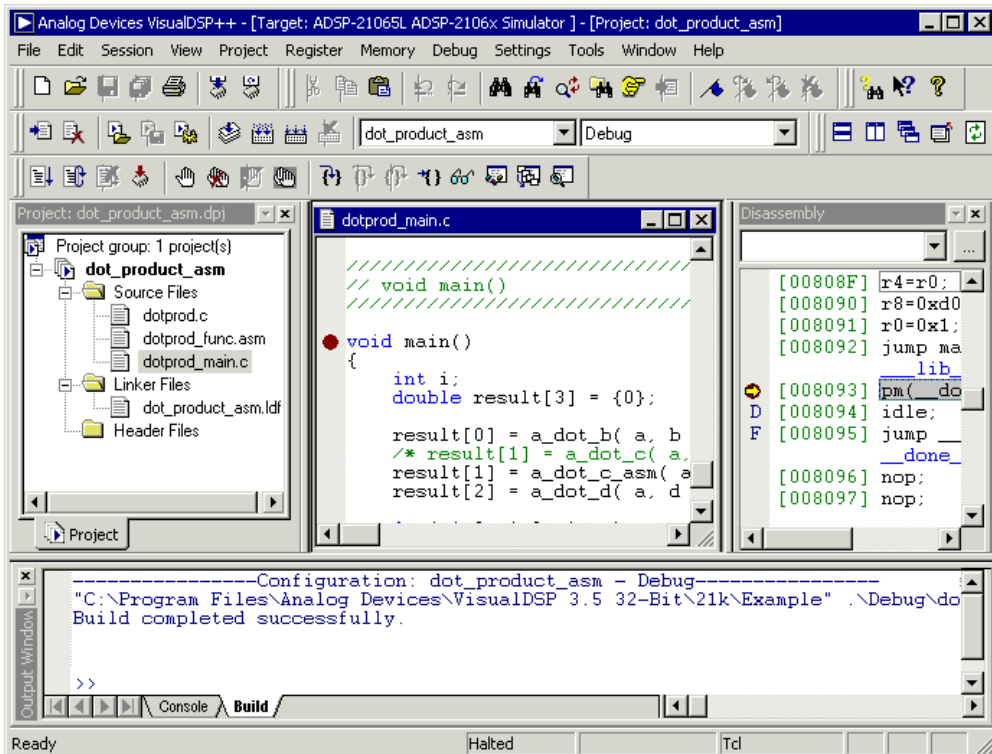


Figure 2-23. dot_product_asm Successfully Built and Loaded

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

The program calculates the three dot products and displays the results in the **Console** view 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 1.

```
Dot product [0] = 0.000000
Dot product [1] = 0.707107
Dot product [2] = -0.500000
```

You are now ready to begin Exercise 3.

Exercise 3: Plotting Data

In this exercise, you load and debug a prebuilt program that applies a simple convolution algorithm to a buffer of data. You use the VisualDSP++ plotting engine to view the different data arrays graphically, both before and after running the program.

Step 1: Load the Convolution Program

To load the Convolution program:

1. Close the `dot_product_asm` project, but keep the **Disassembly** window and **Output** window (in the **Console** view) open.
2. From the **File** menu, choose **Load Program** or click . The **Open a Processor Program** dialog box appears.
3. Select the `convolution.dxe` program to load as follows.
 - a. Open the **Analog Devices** folder and double-click the subfolder `VisualDSP++ 3.5\21k\Examples\Tutorial\convolution\Debug`.

Exercise 3: Plotting Data

- b. Double-click `Convolution.dxe` to load the program in an editor window.
- c. If you are prompted to look for `convolution.cpp`, click **Yes** to open the **Find** dialog box and proceed to step d. If VisualDSP++ opens an editor window, proceed to [step 4 on page 2-35](#).
- d. Click the up-one-level button  to access the `convolution` folder.
- e. Double-click `convolution.cpp` to display the file in an editor window, as shown in [Figure 2-24](#).

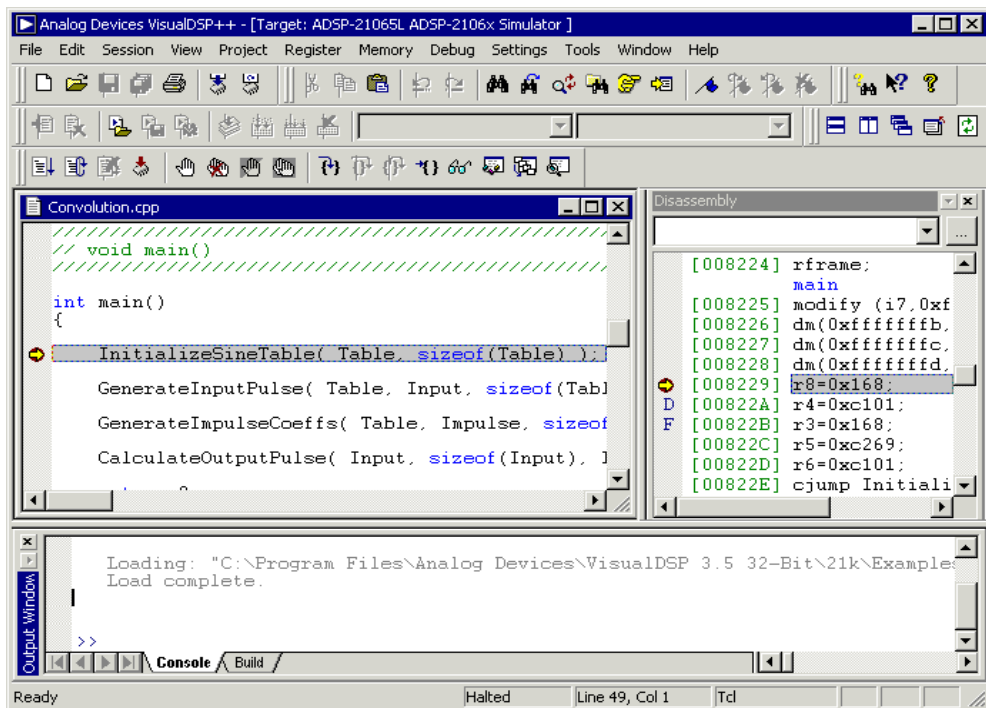


Figure 2-24. Loading the Convolution Program

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

You can see four global data arrays:

Table

Input

Output

Impulse

You can also see four functions that operate on these arrays:

InitializeSineTable()

GenerateInputPulse()

GenerateImpulseCoeffs()

CalculateOutputPulse()

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](#).

Exercise 3: Plotting Data

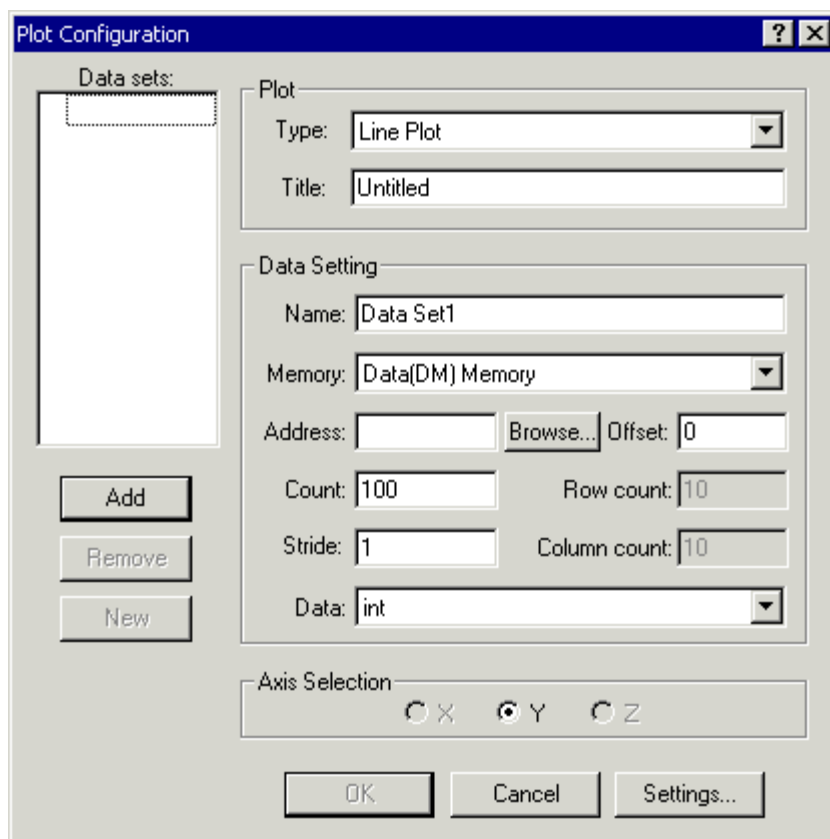


Figure 2-25. Plot Configuration Dialog Box

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

2. In the **Plot** group box, specify the following values.
 - In the **Type** box, select **Line Plot** from the drop-down menu.
 - In the **Title** box, type **convolution**.

3. Enter three data sets to plot by using the values in [Table 2-4](#).

Table 2-4. Three Data Sets: Table, Input, and Output

Data Setting Field	Table Data Set	Input Data Set	Output Data Set	Description
Name	Table	Input	Output	Data set
Memory	Data(DM) Memory	Data(DM) Memory	Data(DM) Memory	Data memory
Address	Table	Input	Output	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	360	360	396	The arrays are 360 and 396 elements long.
Stride	1	1	1	The data is contiguous in memory.
Data	float	float	float	Input and Output are arrays of float values.
Offset	0	0	0	Use zero, the default value.

After entering each data set, click **Add** to add the data set to the **Data Sets** list. The completed **Plot Configuration** dialog box should look like the one in [Figure 2-26](#).

Exercise 3: Plotting Data

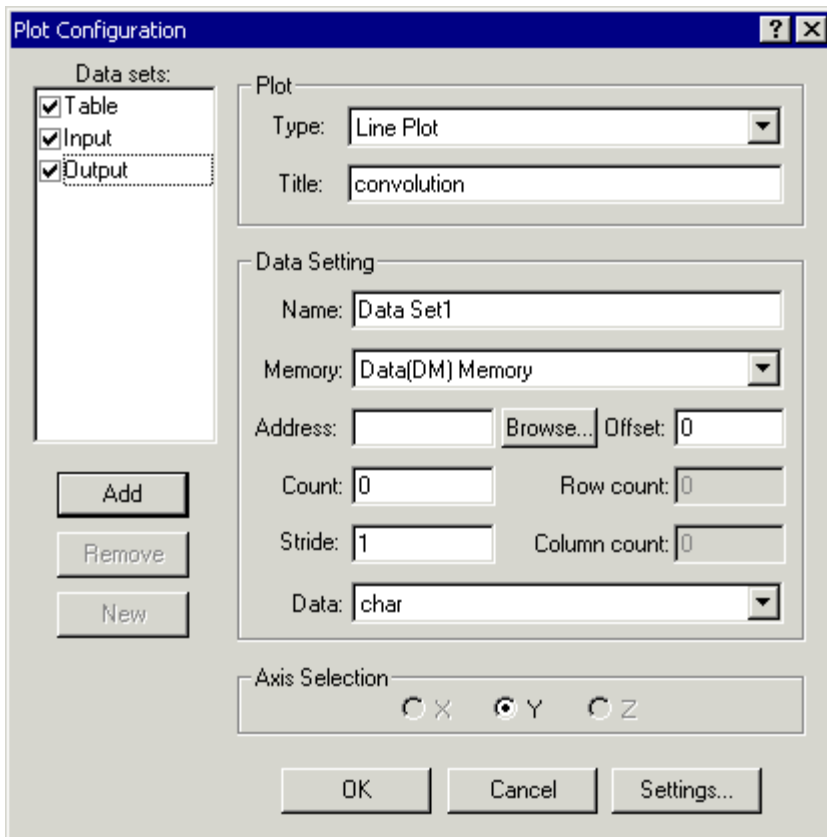


Figure 2-26. Plot Configuration Data Sets (Table, Input, and Output)

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

The plot window now displays the three arrays. Since, by default, the simulator initializes memory to zero, the data sets appear as one horizontal line, shown in [Figure 2-27](#).

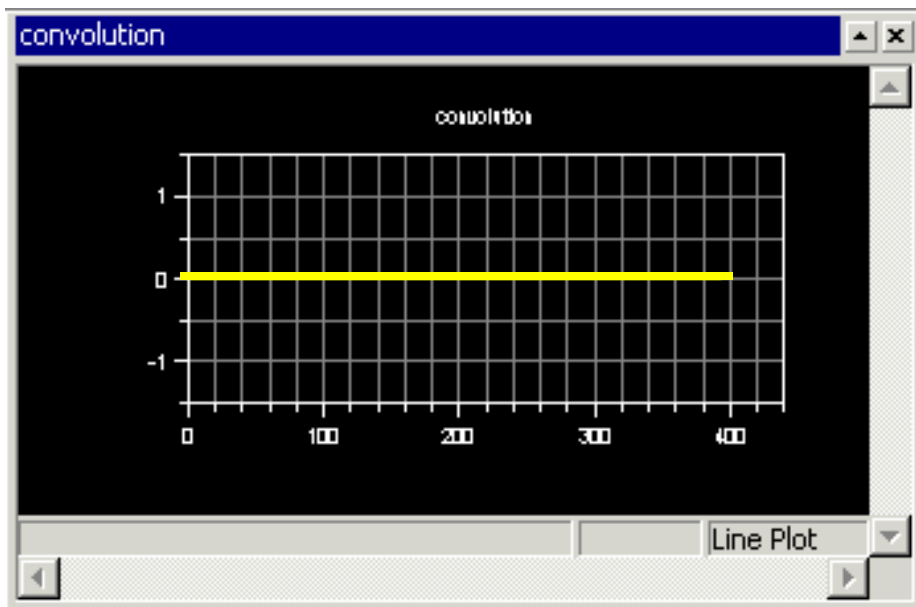


Figure 2-27. Plot Window Before Running the Convolution Program

To display the legend box in the plot window, right-click in the plot window and choose **Modify Settings**. Then, on the **General** page, select **Legend** in the **Options** group box.



The legend box is shown only in [Figure 2-28](#) in this tutorial.

Exercise 3: Plotting Data

Step 3: Run the Convolution Program

To run the Convolution program and view the data:

1. Press **F10** or click the **Step Over** button  to step over the first line in main that calls the `InitializeSineTable()` function.

Stepping over each function enables you to see the data being calculated in a plot window.

2. Step over the call to `GenerateInputPulse()` by using the **Step Over** command as you did in the previous step. The plot window now displays the data for both the Input array and the Table array.

Once you finish stepping over the function, the word “Halted” appears in the status bar at the bottom of the screen. The plot window should now show the sine wave data in the Table array.

3. 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 `convolution` algorithm in the Output array. All three data sets are now visible in the plot window, as shown in [Figure 2-28](#).

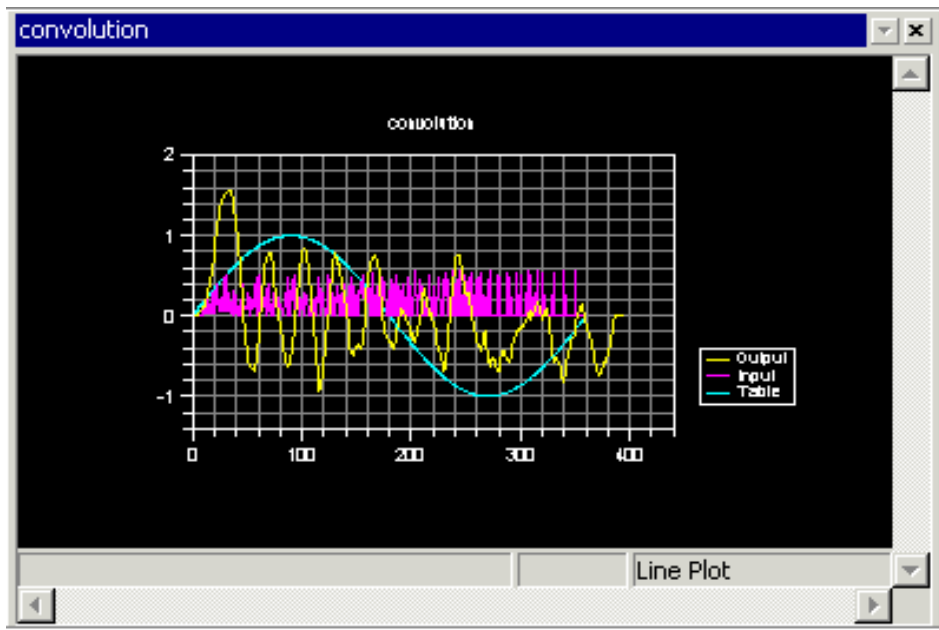


Figure 2-28. Plot Window After Running the Convolution Program to Completion

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

Exercise 3: Plotting Data

4. 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.

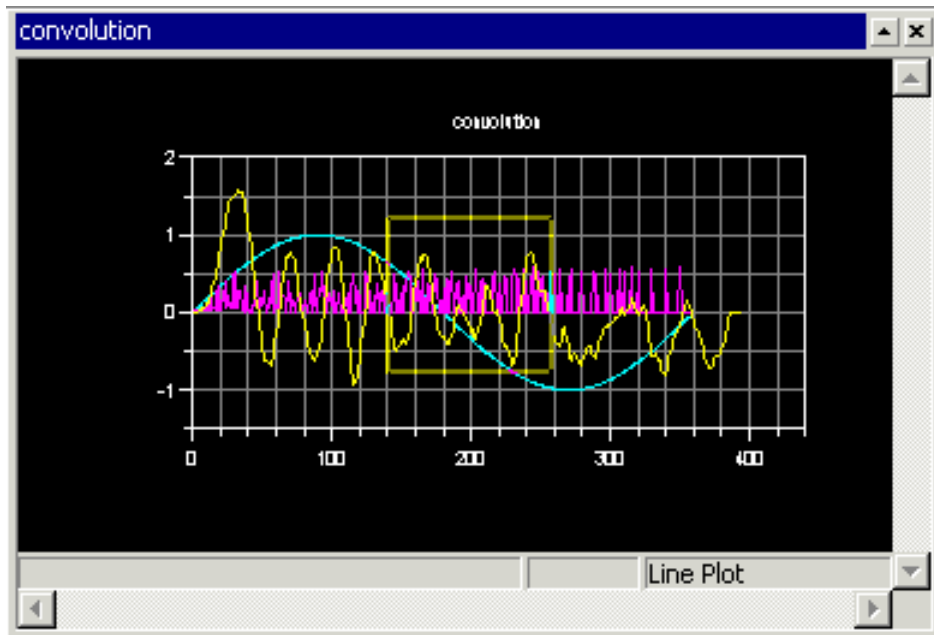


Figure 2-29. Plot Window: Selecting a Region to Magnify

Figure 2-30 shows the magnified results.

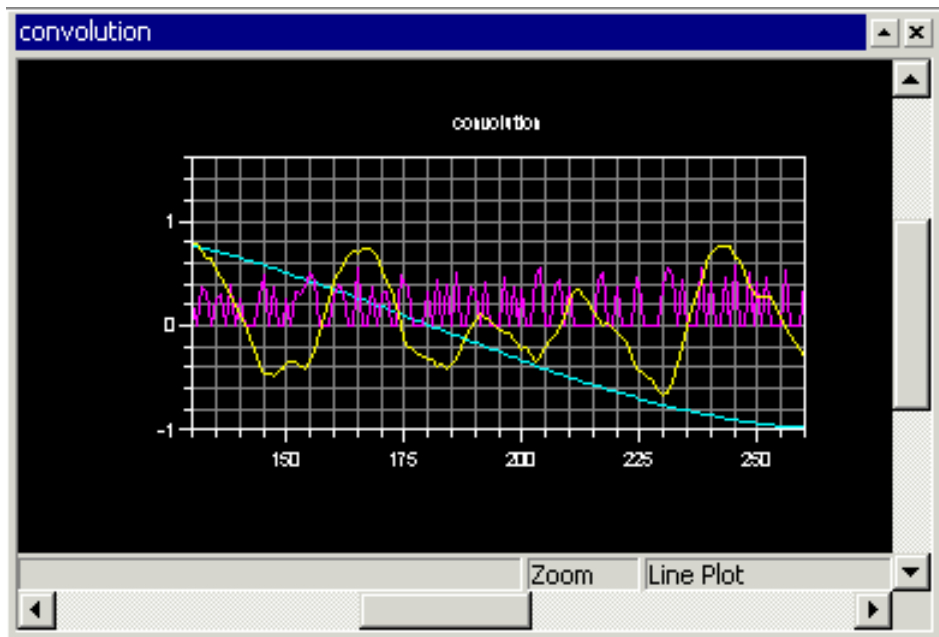


Figure 2-30. Plot Window: Magnified Result

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

5. Right-click inside the plot window and choose **Data Cursor** from the pop-up menu. Then move through the individual data points in the current data set by pressing and holding the Left (←) and Right (→) arrow keys on the keyboard. The value of the current data point appears in the lower-left corner of the plot window, as shown in [Figure 2-31](#).

Exercise 3: Plotting Data

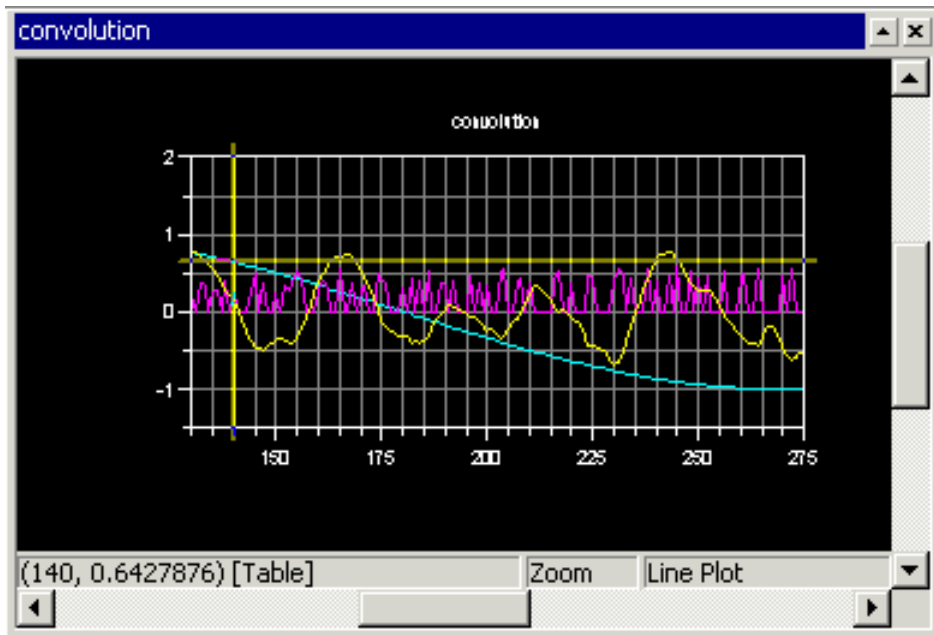


Figure 2-31. Plot Window: Using the Data Cursor Feature

To switch data sets, press the Up (↑) and Down (↓) arrow key.

To disable the data cursor, right-click in the plot window and choose (de-select) **Data Cursor**.

To return to the previous view (before magnification), right-click in the plot window and choose **Reset Zoom** from the pop-up menu.

You are now ready to begin Exercise 4.

Exercise 4: Linear Profiling

In this exercise, you load and debug the Convolution program from the previous exercise. You use linear profiling, however, 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 DSP target. The count in the **Statistical Profiling Results** window is based on random sampling.

Step 1: Load the Convolution Program

To load the Convolution 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 the subfolder `VisualDSP++ 3.5\21k\Examples\Tutorial\convolution\Debug`.
 - b. Double-click `Convolution.dxe` to load and run the Convolution program.

Exercise 4: Linear Profiling

- c. If you are prompted to look for `convolution.cpp`, click **Yes** to open the **Find** dialog box and proceed to step d. If VisualDSP++ opens an editor window, proceed to “[Step 2: Open the Profiling Window.](#)”
- d. Click the up-one-level button  to access the `convolution` folder.
- e. Double-click `convolution.cpp` to display the file in an editor window.

You are now ready to set up linear profiling.

Step 2: Open the Profiling Window

To open the **Linear Profiling Results** window:

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

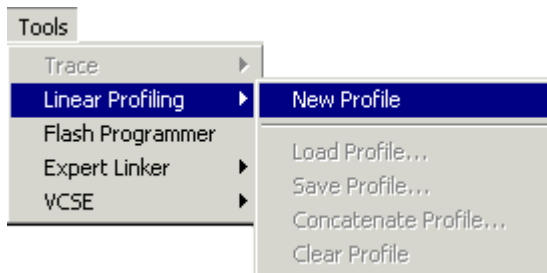


Figure 2-32. Setting Up Linear Profiling for the Convolution Program

The **Linear Profiling Results** window opens.

2. For a better view of the data, drag and dock the window's title bar to the top of the VisualDSP++ main window, as shown in [Figure 2-33](#).

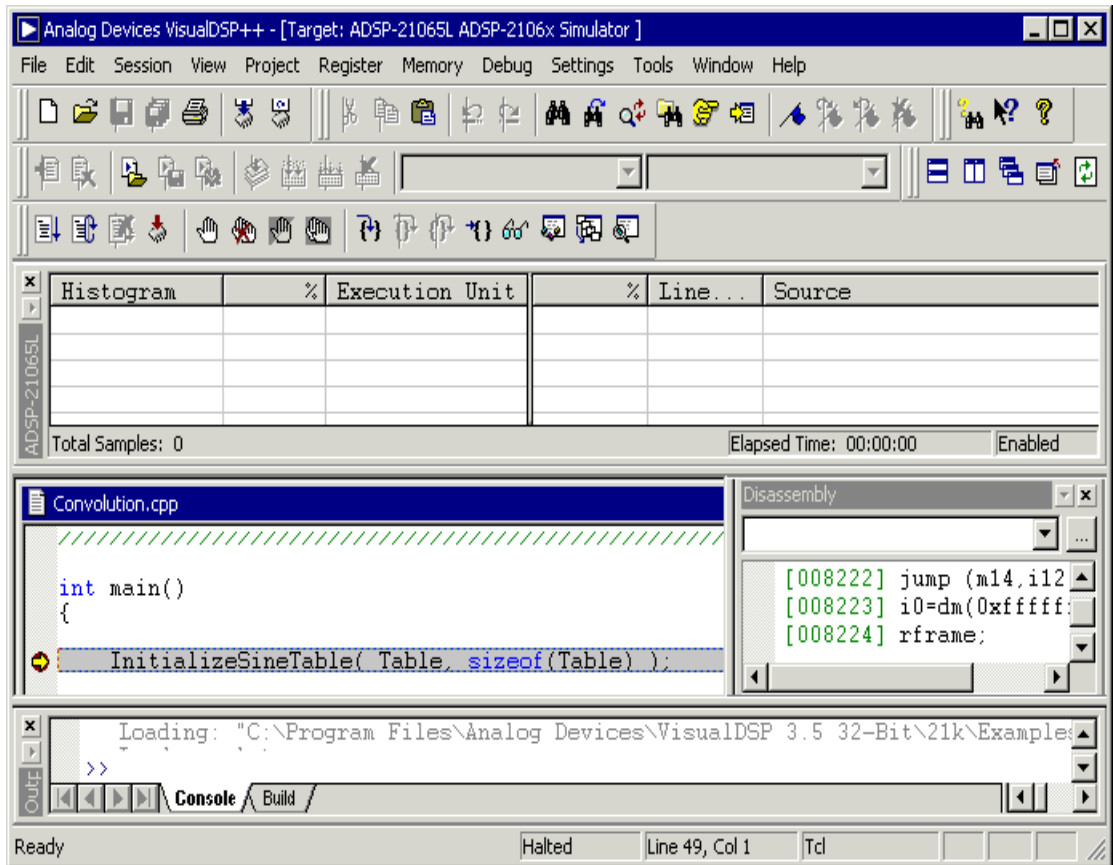


Figure 2-33. Linear Profiling Results Window (Empty)

The **Linear Profiling Results** window is initially empty. Linear profiling is performed when you run the Convolution program.

Exercise 4: Linear Profiling

After you run the program and collect data, this window displays the results of the profiling session. Since we are interested only in high-level source code at this point, we will filter out any samples that do not map directly to our source code.

3. Right-click in the **Linear Profiling Results** window and choose **Properties** to open the **Profile Window Properties** dialog box. Then click the **Filter** tab to display the **Filter** page, shown in [Figure 2-34](#).

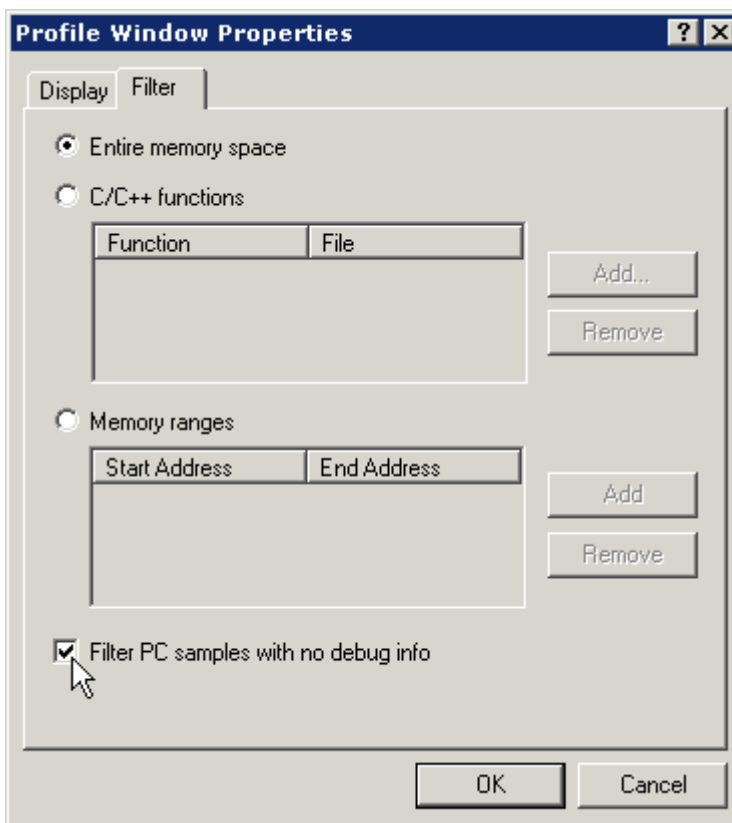


Figure 2-34. Filtering Samples With No Debug Information

4. Click in the **Filter PC samples with no debug info** check box to enable the filter.
5. Click **OK** to close the dialog box.

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

Step 3: Collect and Examine 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 displays the results of the profile data, as shown in [Figure 2-35](#).

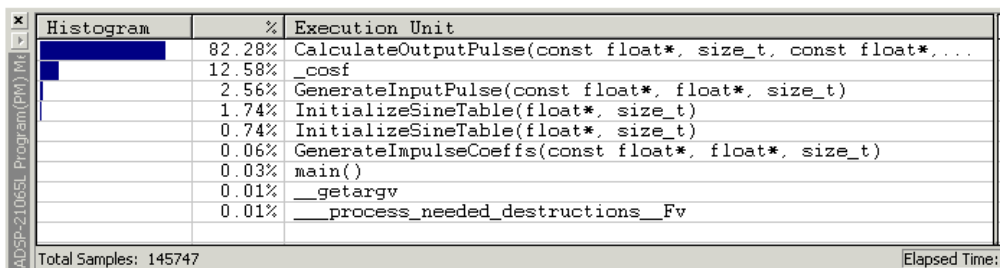


Figure 2-35. Convolution Program Performance Analysis – Left Pane

The field values in the left pane are defined on the next page.

Exercise 4: Linear Profiling

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 pop-up 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 <code>PC[xxx]</code> , where <code>xxx</code> is the address of the instruction.

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

Double-clicking on a line in the left pane displays the source file, `convolution.cpp`, in the right (source) pane and the corresponding source code for the profile data (see [Figure 2-36](#)). The source pane displays data for each line of executable code in the file for which linear profile data has been collected.

If you are prompted to look for `convolution.cpp`, complete these steps:

1. Click **Yes** to open the **Find** dialog box.
2. Click the up-one-level button  to access the `convolution` folder.
3. Double-click `convolution.cpp` to display the file in an editor window.

In [Figure 2-35](#) the left pane shows that the `CalculateOutputPulse()` function has consumed over 82% of the total execution time.

Double-clicking on the `CalculateOutputPulse()` line in the left pane displays the data shown in [Figure 2-36](#) in the right pane.

%	Line...	C:\Program Files\Analog Devices\VisualDSP 3.5 32-Bit\21k\Examp...
	101	////////////////////////////////////
	102	// void CalculateOutputPulse(const float[], size_t, const fl...
	103	////////////////////////////////////
	104	
0.00%	105	void CalculateOutputPulse(const float Input[], size_t nInput...
	106	const float Impulse[], size_t nImpulseSize,
	107	float Output[])
0.01%	108	{
0.01%	109	for(int i=0; i<nInputSize; i++)
	110	{
2.22%	111	for(int j=0; j<nImpulseSize; j++)
	112	{
80.03%	113	Output[i+j] = Output[i+j] + (Input[i] * Impulse[j]);
	114	}
	115	}

Elapsed Time: 00:00:05 Enabled

Figure 2-36. Linear Profiling Data for `Convolution.cpp` – Right Pane

The details of the `CalculateOutputPulse()` function show that 80.03% of the time spent running the entire `Convolution` program is spent inside the nested `for` loop, calculating the convolution.

The data suggests that you should rewrite this function in hand-tuned assembly language to decrease the total running time of the algorithm and improve performance.

Exercise 5: Using a VCSE Component

You are now ready to begin Exercise 5.

Exercise 5: Using a VCSE Component

In this exercise, you complete the following tasks.

- Start up the VisualDSP++ environment and select a new session
- 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\21k\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 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 **Sessions** menu, choose **New Session**. The **New Session** dialog box appears.
3. From the **Processor** list, choose the **ADSP-21060** processor and click **OK**.
4. From the **Project** menu, choose **Open**.

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

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

`VisualDSP++ 3.5\21k\Examples\Tutorial\vcse_component`

Note: This path is based on the default installation.

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

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

Note: The first time that you open projects installed from the software kit, VisualDSP++ may detect that files, folders, or both have moved. If you receive a “Project has been moved” message, click **OK** to continue.

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.

Exercise 5: Using a VCSE Component

Step 2: Install the EXAMPLES::CULawc Component

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\21k\Examples\Tutorial\vcse_component`

Note: This path is based on the default installation.

4. Double-click the `examples_culawc_ADSP-21K.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 2-37](#)).

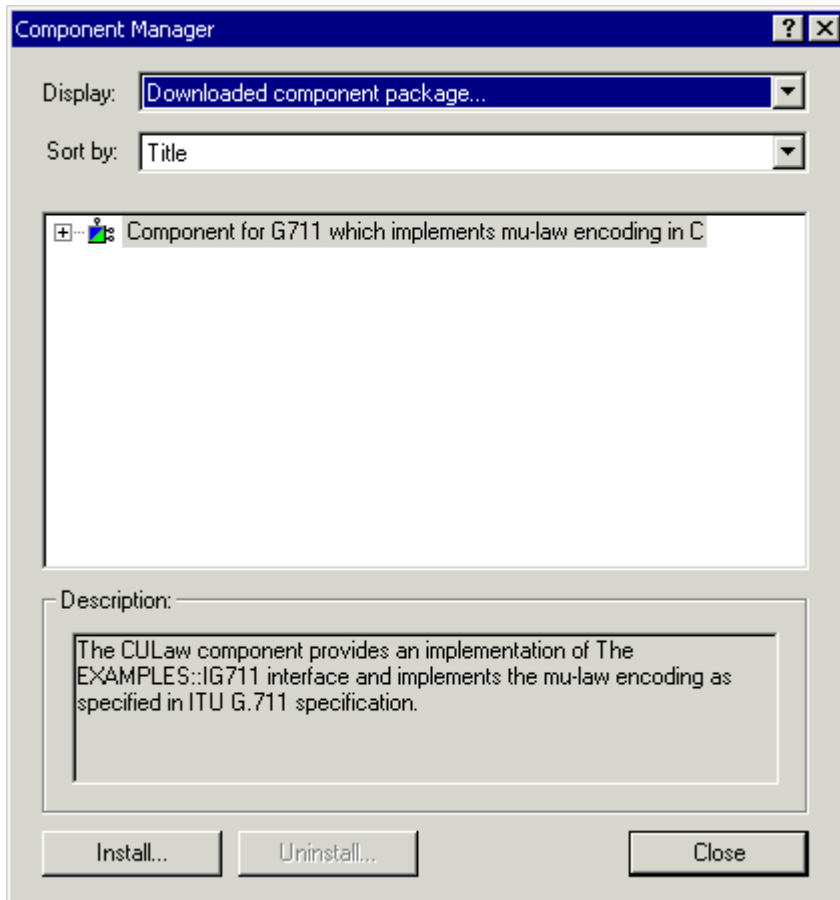


Figure 2-37. Component Manager Dialog Box – Downloaded Component

5. Click the **Install...** button to install the component on your system. Once the component is installed, click **OK**.

Exercise 5: Using a VCSE Component

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**. Component Manager displays the dialog box shown in [Figure 2-38](#).

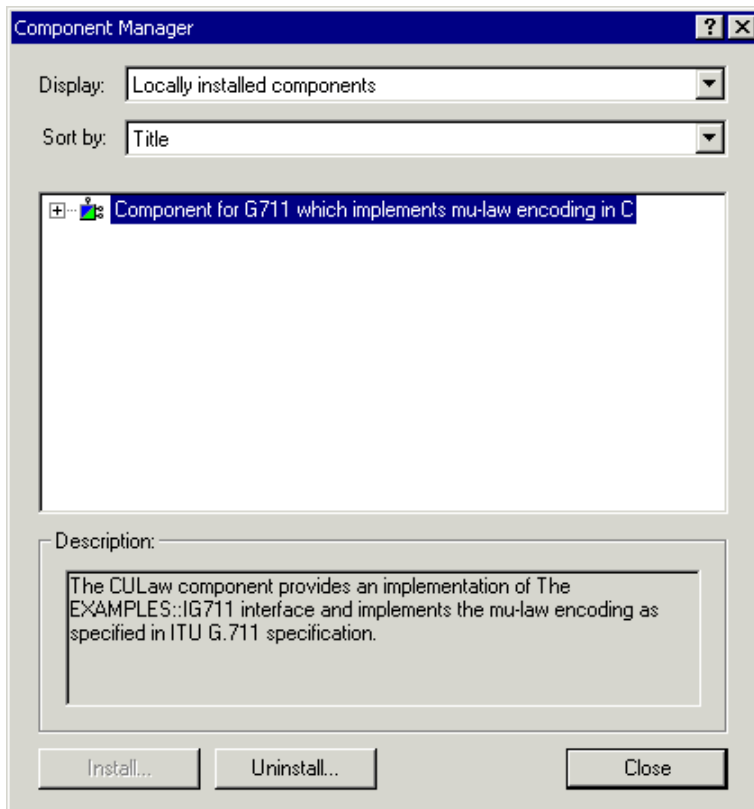


Figure 2-38. Component Manager Dialog Box – Selected Component

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, as shown in [Figure 2-39](#).

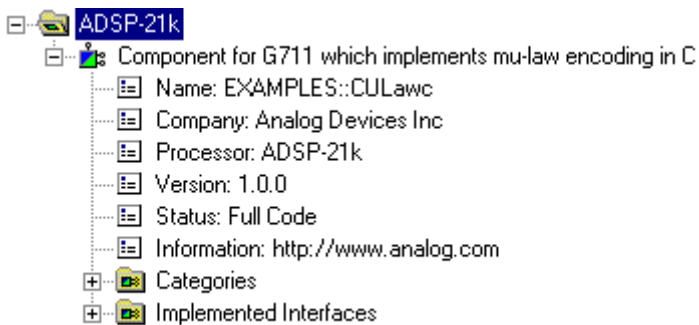


Figure 2-39. Expanded View of Component Information

Make sure that **Processor: ADSP-21k** is listed for the component that you are adding to your project.

Exercise 5: Using a VCSE Component

3. Click **Add** to indicate that you want to add the component to the project. Component Manager displays the dialog box shown in [Figure 2-40](#).

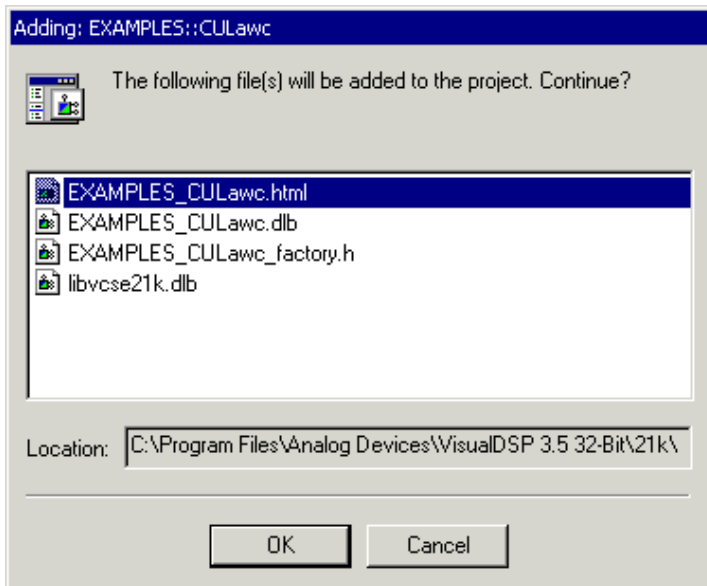


Figure 2-40. Adding Files to the Project

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 2-41](#).

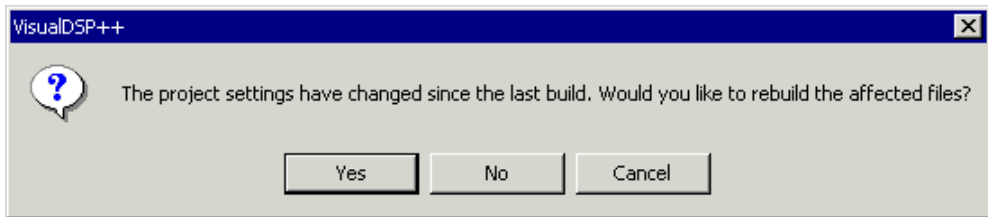


Figure 2-41. Rebuilding Files Affected by Changes to Project Settings

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 this exercise and the Basic Tutorial.

Exercise 5: Using a VCSE Component

3 ADVANCED TUTORIAL

This chapter contains the following topics.

- [“Overview” on page 3-1](#)
- [“Exercise 1: Using PGO” on page 3-2](#)
- [“Exercise 2: Using BTC” on page 3-21](#)

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 32-bit processors.

- In **Exercise 1: Using Profile-Guided Optimization**, you build a project with PGO support, create PGO files, compile the project without using the information in the PGO files, recompile the project by using the PGO files to optimize the build, check the PGO results, and compare execution times.
- In **Exercise 2: Using Background Telemetry Channel**, you add BTC to your DSP application and then run two demos that demonstrate BTC functionality.

The ADSP-2106x Family Simulator debug target and ADSP-21065L processor are used for Exercise 1. The ADSP-21262 processor, EZ-KIT Lite™ USB evaluation system, and HP-PCI ICE or HP-USB ICE emulator are used for Exercise 2.

Exercise 1: Using PGO

Profile-guided optimization (PGO) is an optimization technique that uses information collected previously 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. Recompile 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 recompilation 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 file name.
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:

- 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.
- Recompile 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 32-Bit\21k\  
Examples\Tutorial\pgo
```

Step 1: Load the Project

To open a VisualDSP++ project:

1. Start VisualDSP++ and connect to an ADSP-21065L 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 spends most of its time in the `then...` branch. If the majority of values read are even, the program spends most of its time in the `else...` branch. Normally, the compiler has no way of knowing which branch is 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.

Exercise 1: Using PGO

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 are 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 are optimized to run faster than less commonly used paths.

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](#).

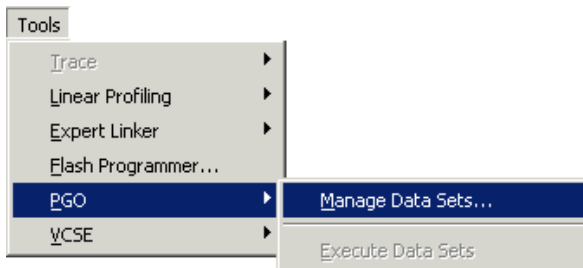


Figure 3-1. Manage Data Sets Menu Option

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

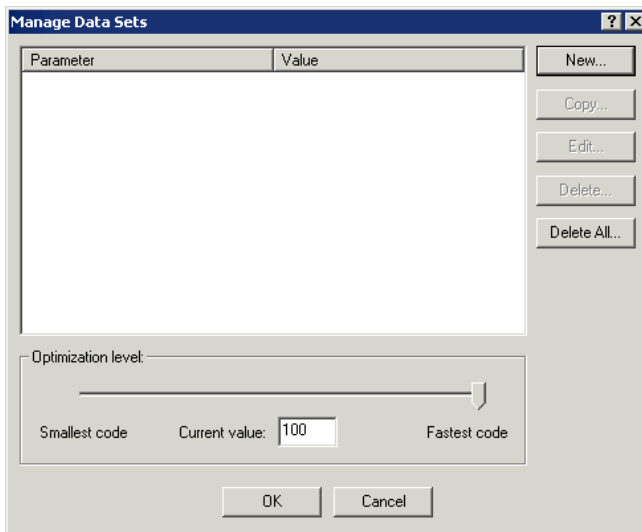


Figure 3-2. Manage Data Sets Dialog Box

This dialog box is where you manage 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 tradeoff. Placing the slider between the two extremes provides varying ratios of space versus speed optimization. For this exercise, leave the slider all the way to the right.

Exercise 1: Using PGO

- Click the **New** button to open the **Edit Data Set** dialog box, shown in [Figure 3-3](#).

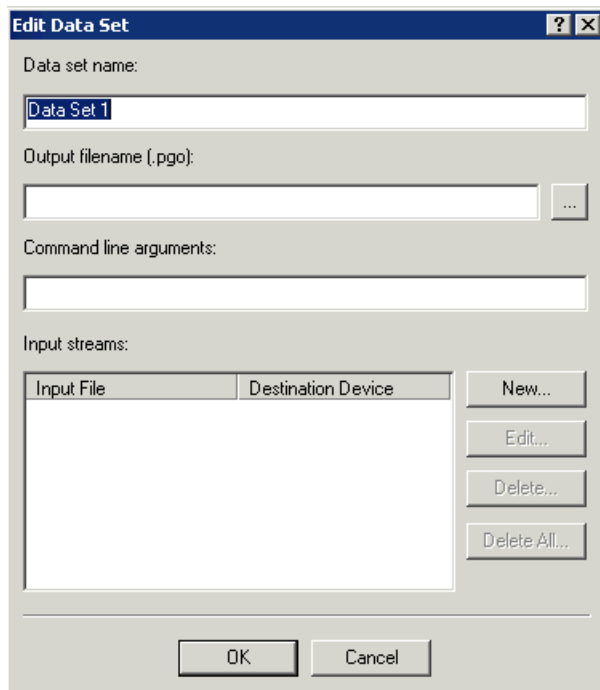


Figure 3-3. Edit Data Set Dialog Box

- 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 is 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 SHARC Processors*.

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

Exercise 1: Using PGO

Step 3: Attach an Input Stream

In this step you 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](#).

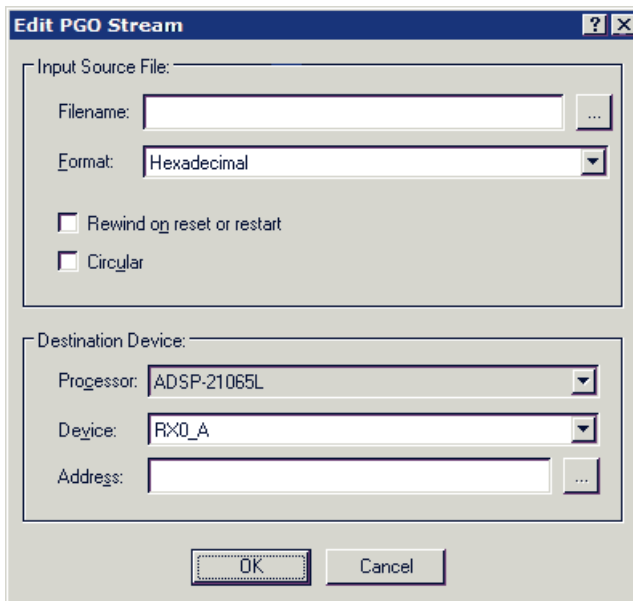


Figure 3-4. Edit PGO Stream Dialog Box

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 32-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, <code>dataset_1.dat</code> , from the <code>pgo</code> 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 starts at the beginning of the input stream. Otherwise, execution continues 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	In a multi-processor session, this field enables you to select the processor to which the stream will apply. Since we are connected to a single processor session in this exercise, 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. Since the program in this exercise reads the input streams from 32-bit memory, select 32-Bit Mem Map Port I/O from the list of available devices.
Address	Specify where in memory the input will be sent. Since the program in this exercise reads data from a global variable called <code>input</code> , type input in this field.

Exercise 1: Using PGO

The completed dialog box should look like the one in [Figure 3-5](#).

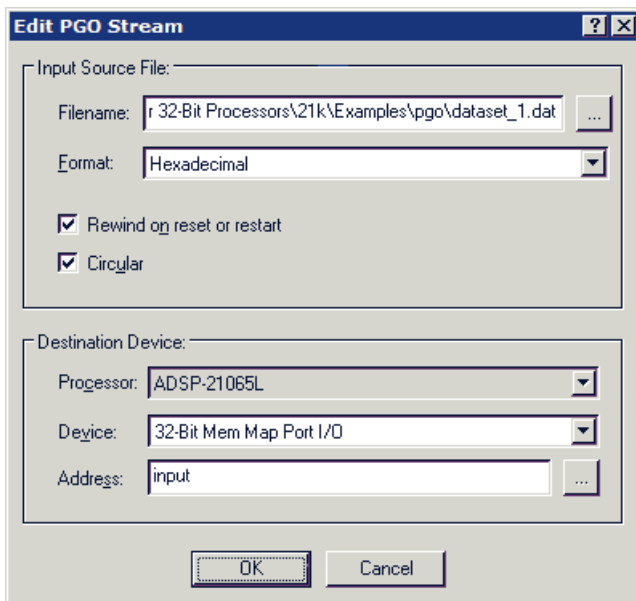


Figure 3-5. A Configured PGO Stream

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

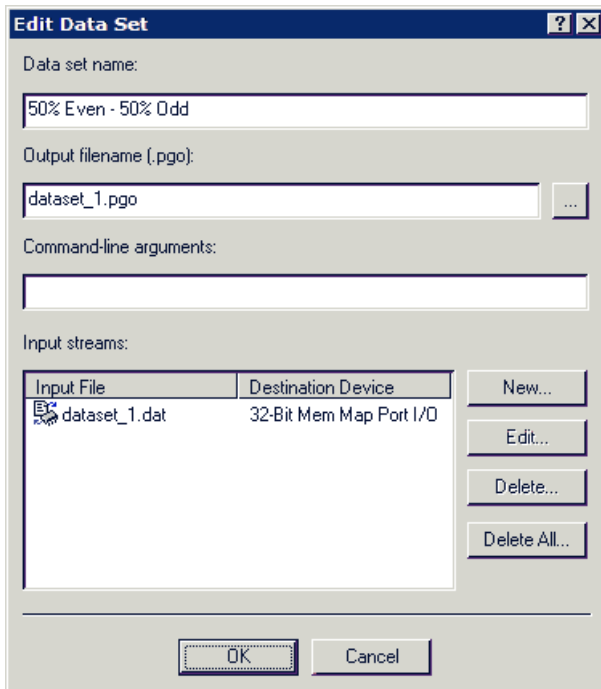


Figure 3-6. A Configured Data Set

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

You now have to create the remaining two data sets.

Exercise 1: Using PGO

Step 4: Configure Additional Data Sets

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

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 `dataset_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 to 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](#).

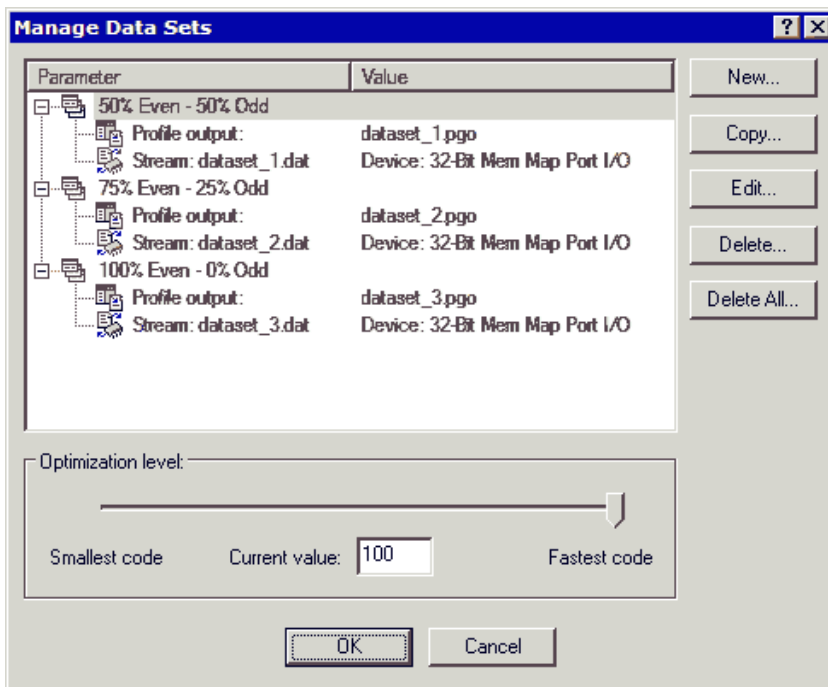


Figure 3-7. Expanded Data Sets

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.

Exercise 1: Using PGO

You are now ready to create .PGO files.

Step 5: Create PGO Files, Optimize the 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](#).

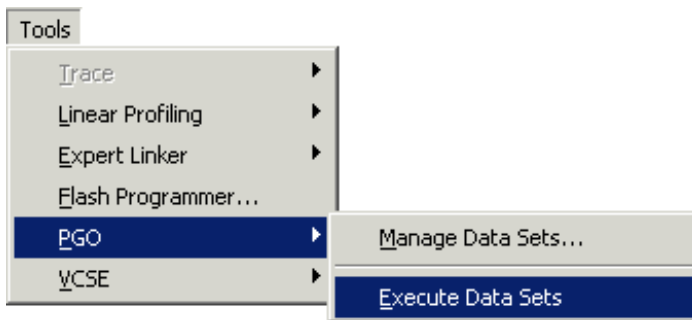


Figure 3-8. Execute Data Sets Menu Option

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 is fed back later into the compiler. The compiler makes default assumptions about which sections of code are most commonly executed. Next, the resulting executable file 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 recompiled. This time, however, the compiler uses the information found in the .PGO files to optimize the resulting executable file. This optimized file 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: Mon Jan 12 15:14:17 2004
Application: D:\Program Files\VisualDSP++ 3.5 for 32-Bit Processors\21k\Examples\pgo\Debug\PgoExample.dxe
Project: D:\Program Files\VisualDSP++ 3.5 for 32-Bit Processors\21k\Examples\pgo\PgoExample.dpj
Optimization level: 100
Average cycle reduction: 26.28%
```

Figure 3-9. PGO Results – Report Header

Exercise 1: Using PGO

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 3-7](#)), 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](#)).



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: 32-Bit Mem Map Port I/O
  Setting PGO output: dataset_1.pgo
  Running application: PgoExample.dxe
  Profile results: 3248 cycles

Profiling Data Set: "75% Even - 25% Odd"...
  Loading application: PgoExample.dxe
  Setting command line:
  Creating input stream: File: dataset_2.dat -> Device: 32-Bit Mem Map Port I/O
  Setting PGO output: dataset_2.pgo
  Running application: PgoExample.dxe
  Profile results: 3376 cycles

Profiling Data Set: "100% Even - 0% Odd"...
  Loading application: PgoExample.dxe
  Setting command line:
  Creating input stream: File: dataset_3.dat -> Device: 32-Bit Mem Map Port I/O
  Setting PGO output: dataset_3.pgo
  Running application: PgoExample.dxe
  Profile results: 3504 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.

Exercise 1: Using PGO

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\cc21k.exe" -c .\PgoExample.c -pguide -O1 -Ov100 -g -double-size-32 -proc ADSP-21065L -o .\Debug\PgoExample.doj
"D:\Program Files\VisualDSP\cc21k.exe" .\Debug\PgoExample.doj -L .\Debug -flags-link -od,. \Debug -o .\Debug\PgoExample.dxe -proc ADSP-21065L
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 SHARC Processors* for details about build options.

This output information shows how effective PGO can be. As shown in [Figure 3-9 on page 3-17](#), the optimized executions used roughly 26% 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 2.

Exercise 2: Using BTC

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.



Currently, TigerSHARC processors do not support BTC.

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 I/O, either standard or user-defined

In this exercise, you:

- 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 SHARC 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.

Exercise 2: Using BTC

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. Enable the EMULI interrupt.

This interrupt invokes the `btc_isr` function.

4. Initialize BTC by using the `btc_init()` command.
5. Add the BTC library, `libbtc26x.dlb` or `libcbtc26x.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 SHARC ADSP-21262 examples. Analog Devices highly recommends that you be connected to a SHARC ADSP-21262 EZ-KIT Lite board via an HP-PCI ICE or HP-USB ICE emulator 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. If you are unsure of how to establish a connection, refer to the manual included with your HP-PCI ICE, HP-USB ICE, or EZ-KIT Lite.



BTC is not currently supported by simulation targets.

The default installation paths for these examples are:

```
Program Files\Analog Devices\VisualDSP 3.5 32-Bit\212xx\  
Examples\BTC Examples\ASM\Background Telemetry Channel
```

```
Program Files\Analog Devices\VisualDSP 3.5 32-Bit\212xx\  
Examples\BTC Examples\C\Background Telemetry Channel
```

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.

Exercise 2: Using BTC

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

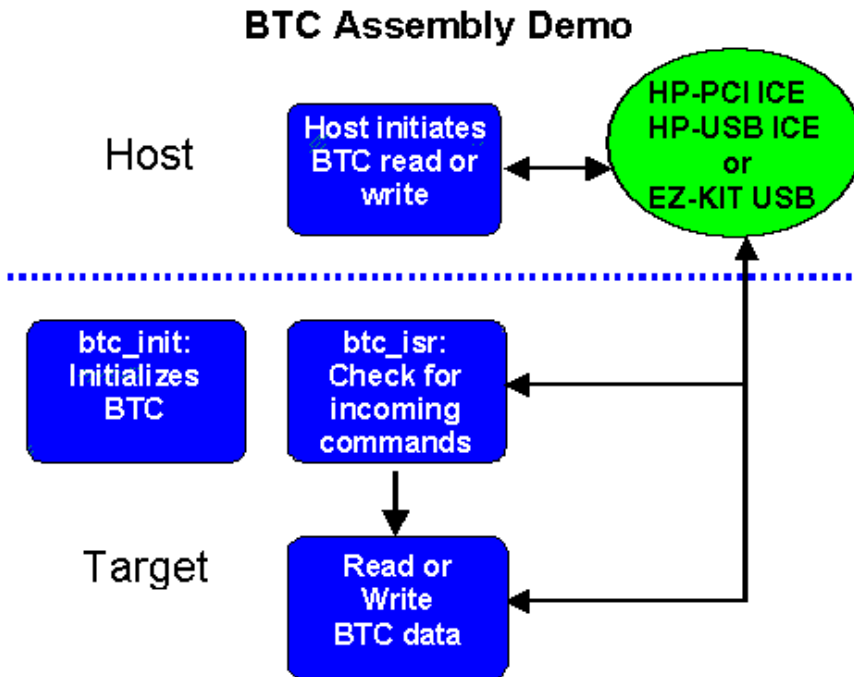


Figure 3-13. Data Transfer in the Assembly Demo

Step 1: Load the Btc_AsmDemo Project

1. Start VisualDSP++ and connect to an ADSP-21262 HP-PCI ICE emulator, HP-USB ICE emulator, 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 1.

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

VisualDSP 3.5 32-Bit\21k\Examples\212xx\BTC Examples\ASM\
Background Telemetry Channel

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 four channels are defined as described in step 2 of [“Adding BTC to Your DSP Application” on page 3-21](#).

```

20
21 // Define the channel map
22 BTC_MAP_BEGIN
23 //           Channel Name,           Starting Address, Length
24 BTC_MAP_ENTRY('Timer Interrupt Counter', timerCounter, 0x0001)
25 BTC_MAP_ENTRY('Data Value', dataVal, 0x0001)
26 BTC_MAP_ENTRY('Data Buffer (13w)', dataBuf, DATA_BUF_SIZE)
27 BTC_MAP_ENTRY('Data Array (8kw)', array1, ARRAY_SIZE)
28 BTC_MAP_END

```

Figure 3-14. BTC Channel Definitions

Exercise 2: Using BTC

3. Scroll down to the `main` program.

On line 55 the command to initialize BTC (`_btc_init;`) is called, as shown in Figure 3-15.

```
39 .global _main;
40 _main:
41
42 // init the stack...this can go away once I start using the canned LDFs
43 b7 = ldf_stack_space;
44 i7 = ldf_stack_space + ldf_stack_length - 2;
45 m7 = -1;
46 l7 = ldf_stack_length - 1;
47
48 nop;
49 nop;
50 //BTC_CHANNEL_ADDR(0,r0);
51 //BTC_CHANNEL_LEN(0, r1);
52 nop;
53
54 ustat1 = 0x55555555;
55 call _btc_init;
56
57
58 call initLEDs;
59 call initInterrupts;
60 call initTimer;
```

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 32-Bit\Help\btc.chm`.

The EMULI interrupt is enabled, as shown in [Figure 3-16](#).

```

197 //////////////////////////////////////////////////
198 //
199 //////////////////////////////////////////////////
200 initInterrupts:
201     // setup imask
202     ustat1 = imask;
203     bit set ustat1 GPTMROI | EMULI;
204     imask = ustat1;
205
206     // enable interrupts
207     ustat1 = model;
208     bit set ustat1 IRPTEN;
209     model = ustat1;
210
211     rts;

```

Figure 3-16. EMULI Interrupt Enabled

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 1: Building a C Program”](#) on page 2-3.

Exercise 2: Using BTC

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](#).

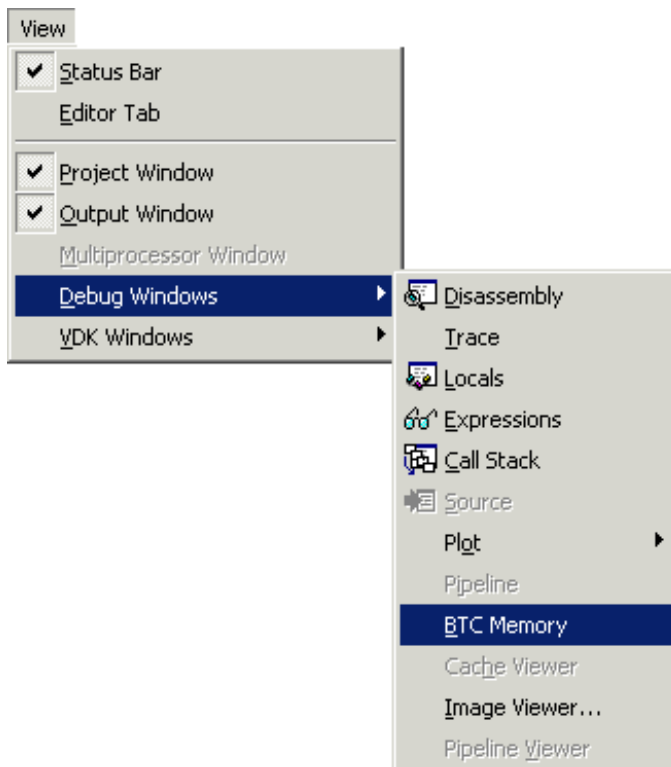


Figure 3-17. BTC Memory Menu Option

The **BTC Memory** window, shown in [Figure 3-18](#), is displayed.

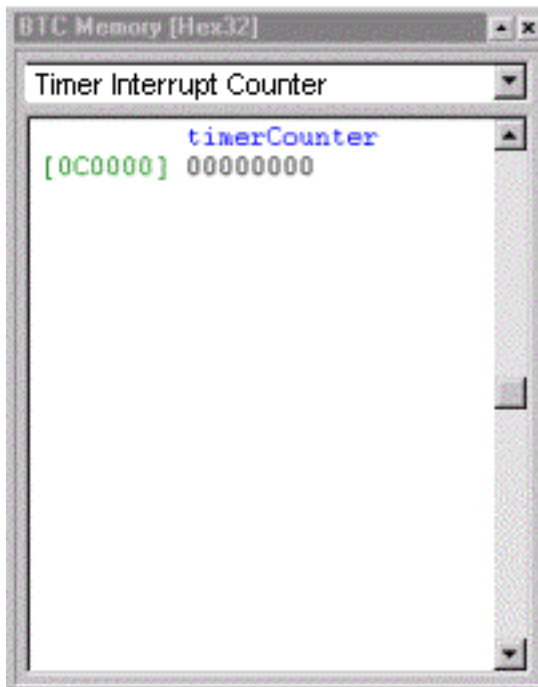


Figure 3-18. BTC Memory Window

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](#).

Exercise 2: Using BTC

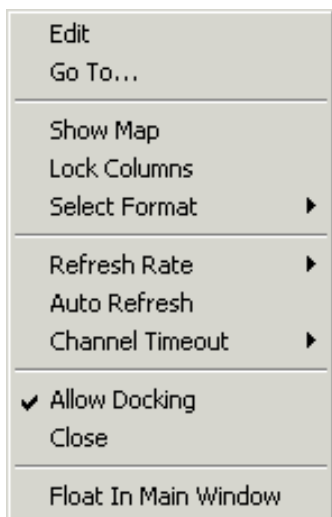


Figure 3-19. BTC Memory Window Right-Click Menu

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](#).

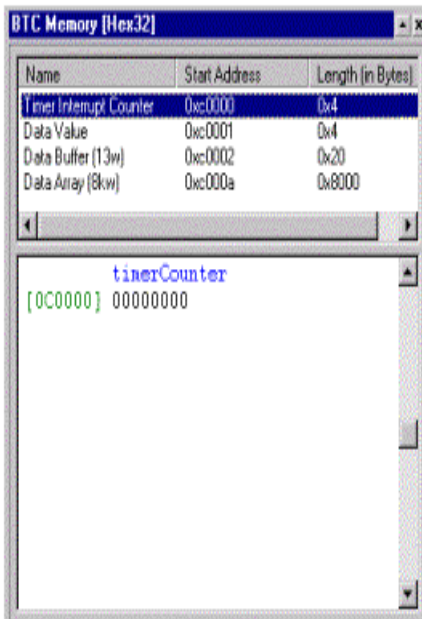


Figure 3-20. BTC Memory Window With Map

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).

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.

Exercise 2: Using BTC

See [Figure 3-21](#), [Figure 3-22](#), and [Figure 3-23](#) for comparisons.

[Figure 3-21](#) shows the original window.

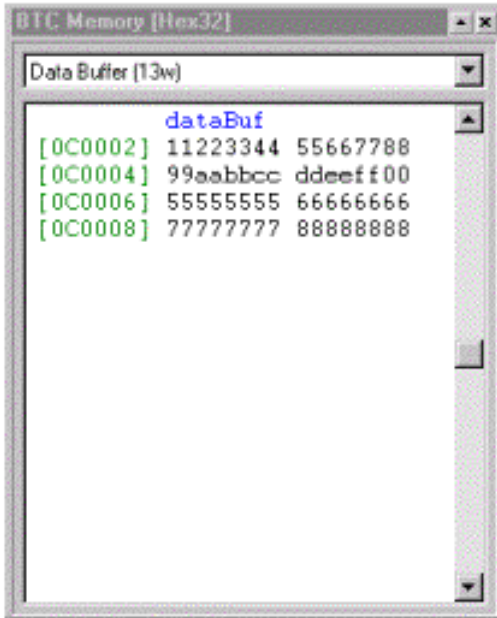


Figure 3-21. Original Window Width

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

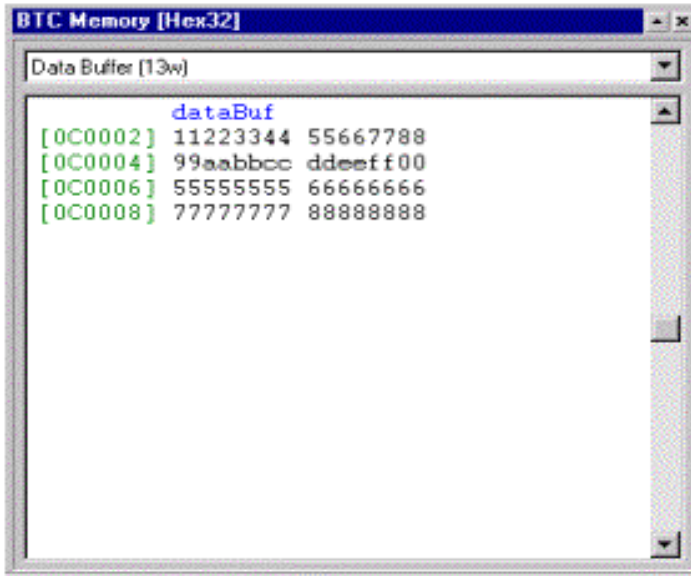


Figure 3-22. Expanded Window With Lock Columns Enabled

Exercise 2: Using BTC

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

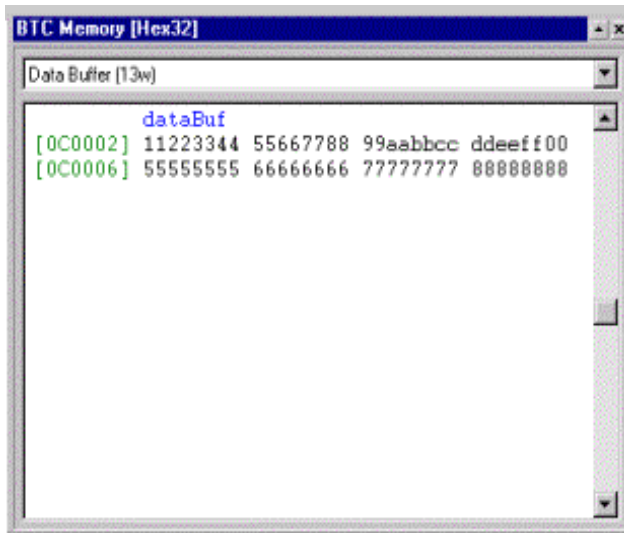


Figure 3-23. Expanded Window With Lock Columns Disabled

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++ waits 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 does not dock. If you move it to a location shared by a docked window, the docked window sits 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.

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

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

You are now ready to run the BTC FFT Demo.

Exercise 2: Using BTC

Running the BTC FFT Demo

The BTC FFT Demo demonstrates the transfer of data from the SHARC ADSP-21262 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.

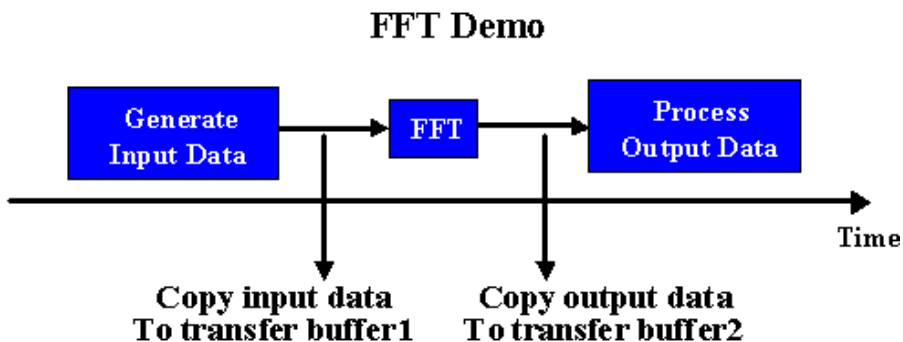


Figure 3-24. Data Transfer in the BTC FFT Demo

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



Make sure that you are in an ADSP-21262 HP-PCI or HP-USB emulator session.

Step 1: Build the FFT Demo

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

```
\Program Files\Analog Devices\VisualDSP 3.5 32-Bit\  
212xx\Examples\C\Background Telemetry Channel\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, see [“Exercise 1: Building a C Program” on page 2-3](#).

Exercise 2: Using BTC

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](#).

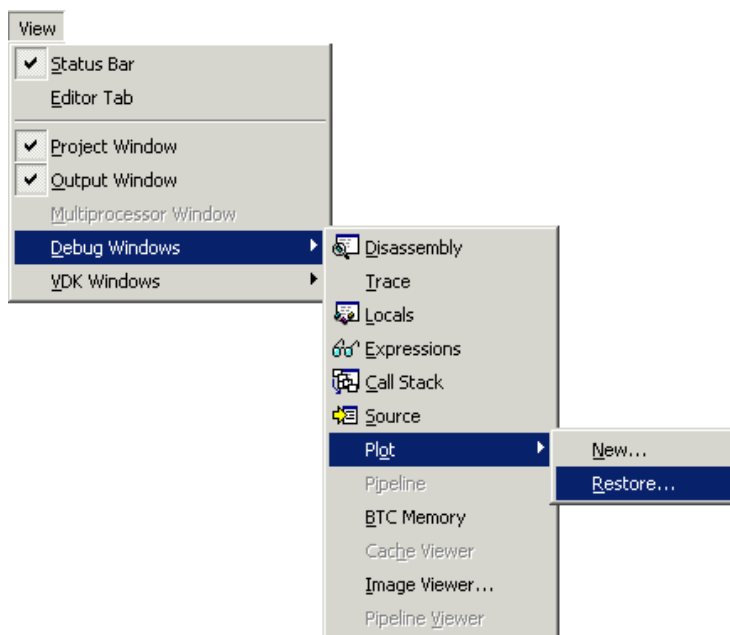


Figure 3-25. Plot Restore Menu Option

The **Restore** command opens the **Select Plot Settings File** window, shown in [Figure 3-26](#).

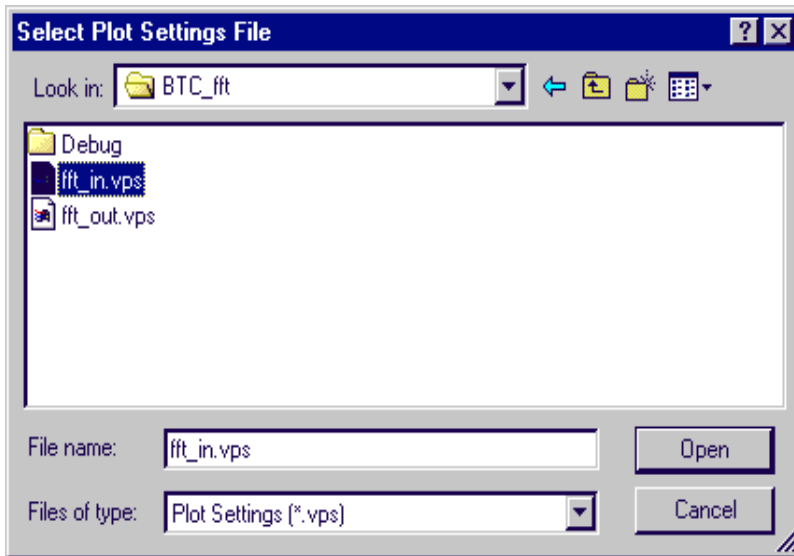


Figure 3-26. Select Plot Settings File Window

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](#).

Exercise 2: Using BTC

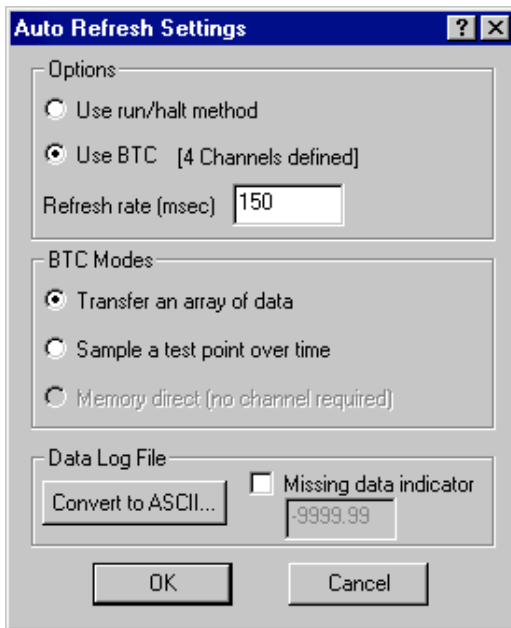


Figure 3-27. Auto Refresh Settings Dialog Box

This dialog box enables you to configure the plotting tool to plot the BTC data in realtime.

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** (default) – This method uses the `btc_plot_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_plot_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 ($\text{PRR} * \text{DSR}$).

Use the default method, **Transfer an array of data**, for transferring data.

In the **Data Log File** group box, the **Convert to ASCII** button enables you to convert log data to ASCII format. This subject is 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.

Exercise 2: Using BTC

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.

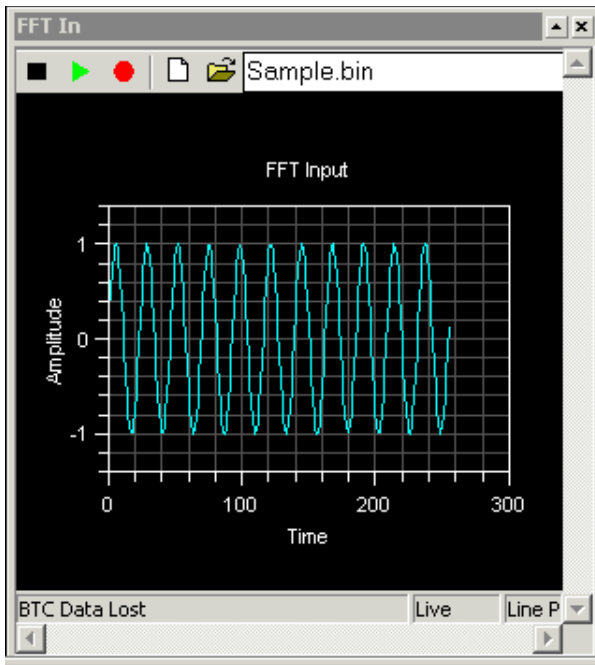


Figure 3-28. Plot Window with Toolbar

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 display data being plotted in realtime.

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 1 to 10, and notice its effect on the plots. 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.

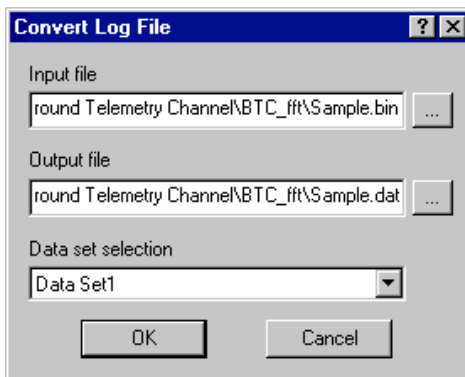


Figure 3-29. Convert Log File Dialog Box

Exercise 2: Using BTC

7. 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.

8. Click **OK** to convert the log file from binary to ASCII, which is readable by other programs.
9. 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

Symbols

% of Histogram data, defined, [2-50](#)
.PGO files, creating, [3-16](#)
/*, [2-27](#)

A

Add Files dialog box, [2-21](#), [3-24](#)
adding project source files, [2-21](#)
Advanced Tutorial
 overview, [3-1](#)
 using background telemetry
 channel (BTC), [3-21](#)
 using profile-guided optimization,
 [3-2](#)
Auto Refresh command, [3-42](#)
auto refresh rate, setting (BTC),
 [3-34](#)
Auto Refresh Settings dialog box,
 [3-39](#)

B

background telemetry channel, [3-21](#)
 adding to your application, [3-21](#)
 channel definitions, [3-25](#)
 commands, [3-25](#)
 converting BTC log data to
 ASCII, [3-41](#)

 map of defined channels, [3-30](#)
 modes of transferring data, [3-41](#)
Basic Tutorial
 building and running C programs,
 [2-3](#)
 installing and using a VCSE
 component, [2-52](#)
 linear profiling, [2-45](#)
 modifying a C program to call an
 assembly routine, [2-17](#)
 overview, [2-1](#)
 plotting data, [2-33](#)
bookmarks, adding to source files,
 [2-27](#)
breakpoint symbols
 red circle, [2-14](#)
 yellow arrow, [2-14](#)
BTC assembly demo, running, [3-23](#)
BTC FFT demo
 building, [3-37](#)
 plotting BTC data, [3-38](#)
 running, [3-36](#)
BTC Memory window, [3-29](#)
BTC polling loop command, [3-27](#)
BTC, *See* background telemetry
 channel
Build Project command, [2-8](#)

INDEX

C

C programs

- building and running, [2-3](#)
- modifying to call assembly routines, [2-17](#)

channel timeout, setting (BTC), [3-35](#)

channels, BTC, [3-25](#)

code development tools

- features, [1-7](#)
- overview, [1-2](#)

commands

- BTC, [3-25](#)
- BTC polling loop, [3-27](#)
- Build Project, [2-8](#)
- Execut Data Sets, [3-16](#)
- initialize BTC, [3-26](#)
- Lock Columns (BTC), [3-31](#)
- Rebuild All, [2-8](#), [3-27](#), [3-37](#)
- Restore, [3-39](#)
- Select Format (BTC), [3-34](#)
- Show Map (BTC), [3-30](#)

comment characters, moving in source files, [2-27](#)

Compile page, [2-19](#)

configuring PGO data sets, [3-6](#)

Console page, [2-16](#)

conventions used in this manual, [xvi](#)

Convert Log File dialog box, [3-43](#)

converting BTC log data to ASCII, [3-41](#)

Convolution program

- data arrays, [2-35](#)
- global data arrays, [2-35](#)

loading, [2-33](#)

running, [2-40](#)

viewing the results, [2-41](#)

convolution.cpp source file, [2-34](#), [2-50](#), [2-51](#)

convolution.dxe

loading, [2-33](#)

See also Convolution program

creating

- .PGO files, [3-16](#)
- debug sessions, [2-11](#)
- new projects, [2-17](#)

customer support, [ix](#)

D

Data Cursor command, [2-43](#), [2-44](#)

Data Log file (BTC), [3-41](#)

data sets

- attaching to input streams, [3-10](#)
- configuring (PGO), [3-6](#)
- configuring with the Copy command, [3-14](#)
- input and output, [2-37](#)
- plotting, [2-35](#)

debug features, [1-4](#)

debug sessions

- setting up new sessions, [2-11](#)
- setting up subsequent sessions, [2-11](#)

demos, running

- BTC assemble, [3-23](#)
- BTC FFT, [3-36](#)

dialog boxes

- Add Files, [2-21](#), [3-24](#)

- Adding (VCSE files), [2-58](#)
- Auto Refresh Settings, [3-39](#)
- Breakpoints, [2-15](#)
- Component Manager (VCSE),
[2-56](#)
- Convert Log File, [3-43](#)
- Edit Data Set, [3-8](#)
- Edit PGO Stream, [3-10](#)
- Find, [2-26](#)
- Manage Data Sets, [3-7](#), [3-15](#)
- New Session, [2-11](#)
- Plot Configuration, [2-35](#)
- Profile Window Properties, [2-48](#)
- Project Options, [2-18](#), [2-19](#)
- Save New Project As, [2-17](#)
- Disassembly window
 - adding and deleting breakpoints,
[2-15](#)
 - information displayed, [2-14](#)
- dot_product
 - rebuilding, [2-32](#)
 - running, [2-33](#)
- dot_product_asm, building the
 project, [2-18](#)
- dot_product_c, folder location, [2-5](#)
- dotprod_main.c, [2-9](#)
 - modifying to call a_dot_c_asm,
[2-26](#)
 - opening, [2-9](#)
- dotprodasm.ldf
 - modifying, [2-29](#)
 - viewing, [2-29](#)
- dotprodc, running, [2-16](#)

- dotprodc.dxe, automatically
 loading, [2-13](#)

E

- Edit Data Set dialog box, [3-8](#)
- Edit PGO Stream dialog box, [3-10](#)
- editing project source files, [2-26](#),
[3-28](#)
- editor windows, [2-10](#), [2-27](#), [3-35](#)
- Execut Data Sets command, [3-16](#)
- execution units, definition of, [2-50](#)
- exercises
 - building and running C programs,
[2-3](#)
 - installing and using a VCSE
 component, [2-52](#)
 - linear profiling, [2-45](#)
 - modifying a C program to call an
 assembly routine, [2-17](#)
 - plotting data, [2-33](#)
 - using background telemetry
 channel (BTC), [3-21](#)
 - using profile-guided optimization,
[3-2](#)
- Expert Linker
 - creating a Linker Description File,
[2-22](#)
 - modifying an .LDF file, [2-29](#)

F

- files
 - .PGO, [3-16](#)
 - Data Log file (BTC), [3-41](#)
- Find dialog box, [2-23](#), [2-26](#)

INDEX

G

General page, [2-7](#)
Generate debug information check
box, [2-20](#)

H

histogram, defined, [2-50](#)

I

initialize BTC command, [3-26](#)
input data sets, entering, [2-37](#)
input streams, attaching to data sets,
[3-10](#)
Integrated Development and
Debugging Environment
(IDDE), [2-1](#)

J

JTAG emulators, [2-45](#)

L

linear profiling, [2-45](#)
collecting and examining profile
data, [2-49](#)
Linear Profiling Results window,
[2-49](#)
loading the Convolution
program, [2-45](#)
opening the Profiling window,
[2-46](#)
results of analyzing the
Convolution program, [2-49](#)

Linker Description File (LDF)

creating, [2-22](#)
folder, [2-2](#)
linker errors, viewing in Output
window, [2-29](#)
Load executable after build
command, [2-11](#)
loading
PGO projects, [3-4](#)
Lock Columns command (BTC),
[3-31](#)

M

magnifying selected regions, [2-42](#)
Manage Data Sets dialog box, [3-7](#),
[3-15](#)
map of defined BTC channels, [3-30](#)
messages
Output window, [2-14](#)
project is up to date, build
completed successfully, [2-8](#)
modifying C programs to call
assembly routines, [2-17](#)
MyAnalog.com, [xi](#)

N

New Session dialog box, [2-11](#)

O

opening projects, [2-4](#), [3-4](#)
optimizing your program, [3-16](#)
optimizing your program with
PGO, [3-16](#)
output data sets, entering, [2-37](#)

Output window, [2-9](#)
 Console view, [2-16](#)
 information displayed, [2-14](#)
 viewing linker errors, [2-29](#)

overview

Advanced Tutorial, [3-1](#)
 Basic Tutorial, [2-1](#)

P

PGO, *See* profile-guided
 optimization

Plot Configuration dialog box,
[2-35](#), [2-37](#)

plot windows

after running the Convolution
 program, [2-41](#)
 before running the Convolution
 program, [2-39](#)
 FFT In (BTC), [3-42](#)
 magnified result, [2-42](#)
 magnifying data points, [2-43](#)
 opening, [2-35](#)
 selecting a region to magnify, [2-42](#)
 toolbar (BTC), [3-42](#)
 viewing data points, [2-44](#)
 zooming in on a region, [2-41](#)

plotting

BTC data, [3-38](#)
 data, [2-33](#)
 specifying data sets, [2-35](#)

preferences, selecting, [2-7](#)

Profile Window Properties dialog
 box, [2-48](#)

profile-guided optimization
 attaching an input stream, [3-10](#)
 configuring a data set, [3-6](#)
 configuring data sets with the
 Copy command, [3-14](#)
 creating .PGO files, [3-16](#)
 loading PGO projects, [3-4](#)
 using, [3-2](#)

profile-guided optimization results
 report

build output, [3-20](#)
 data set information, [3-18](#)
 execution output, [3-19](#)
 header, [3-17](#)

programs, running in a debug
 session, [2-11](#)

Project Options dialog box, [2-18](#),
[2-19](#)

Project page, [2-18](#)

projects

adding files to dot_product_asm,
[2-21](#)
 building dot_product, [2-32](#)
 building dotprodc, [2-8](#)
 creating an .LDF file, [2-22](#)
 creating new projects, [2-17](#)
 dot_product_asm, building, [2-18](#)
 dotprodc files, [2-6](#)
 managing overview, [1-1](#)
 modifying source files, [2-26](#)
 opening, [2-4](#)
 opening (PGO), [3-4](#)
 options, [2-19](#)

INDEX

R

Rebuild All command, [2-8](#), [3-27](#),
[3-37](#)
recording BTC data, [3-43](#)
refresh rate, setting for BTC
 Memory window, [3-34](#)
Reset Zoom command, [2-43](#), [2-44](#)
Restore command, [3-39](#)

S

Save New Project As dialog box,
[2-17](#)
Select Format command (BTC),
[3-34](#)
Select Plot Settings File window,
[3-39](#)
Show Map command (BTC), [3-30](#)
source files
 adding to projects, [2-21](#)
 modifying, [2-26](#)
statistical profiling, [2-45](#)

T

technical support, [ix](#)
Text Import Wizard, [3-44](#)
timer interrupt counter, setting
 (BTC), [3-35](#)
toolbar buttons, [2-3](#)
toolbars
 plot windows (BTC), [3-42](#)
 VisualDSP++ main window, [2-3](#)

transferring data, modes of (BTC),
[3-41](#)

V

VCSE
 Adding (files) dialog box, [2-58](#)
 adding a VCSE component to a
 project, [2-57](#)
 building and running the
 program, [2-59](#)
 Component Manager dialog
 boxes, [2-55](#), [2-56](#)
 installing a VCSE component,
 [2-54](#)
View Sample Count command,
[2-50](#)
VisualDSP++
 features, [1-1](#)
 starting, [2-4](#)
 tool buttons, [2-3](#)

W

windows
 BTC Memory, [3-29](#)
 Disassembly, [2-14](#), [2-15](#)
 editor, [2-10](#), [2-27](#), [3-35](#)
 Linear Profiling Results, [2-47](#),
 [2-49](#)
 Output, [2-9](#), [2-10](#), [2-14](#)
 plot, [2-39](#), [2-41](#), [2-42](#), [2-44](#)
 Select Plot Settings File, [3-39](#)
wizards
 Text Import, [3-44](#)