3 TUTORIAL

In This Chapter

This chapter contains the following topics:

Overview

This tutorial demonstrates key features and capabilities of the VisualDSP++ Integrated Development and Debugging Environment (IDDE). The exercises use sample programs written in C, C++, and assembly for SHARC DSPs. For these exercises, you will use the ADSP-2106*x* simulator for the ADSP-21065L target, the ADSP-21065 (MP,2) target, and the ADSP-21061 (MP,6) target.

You can use different SHARC processors with only minor changes to the Linker Description Files (`.LDFs`) included with each project.

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

```
Analog Devices\VisualDSP\21k\ldf folder
```

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

The tutorial contains five exercises:

- In **Exercise One**, you will start up VisualDSP++, build a project containing C source code, and profile the performance of a C function.
- In **Exercise Two**, you will create a new project, modify sources to call an assembly routine, rebuild the project, and profile the performance of the assembly language routine.
- In **Exercise Three**, you will plot the various waveforms produced by a Convolution algorithm.
- In **Exercise Four**, you will use statistical profiling to examine the efficiency of the Convolution algorithm used in Exercise Three. Using the collected statistical profile data, you will pinpoint the most time-consuming areas of the algorithm, which are likely to require hand tuning in the assembly language.

- In **Exercise Five**, you will explore the multiprocessor debugging capabilities of VisualDSP++, including synchronous control of multiple targets, window pinning, and the use of multiprocessor groups to control complex systems.

Tip: Become familiar with the VisualDSP++ toolbar buttons, shown in Figure 3-1. 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.

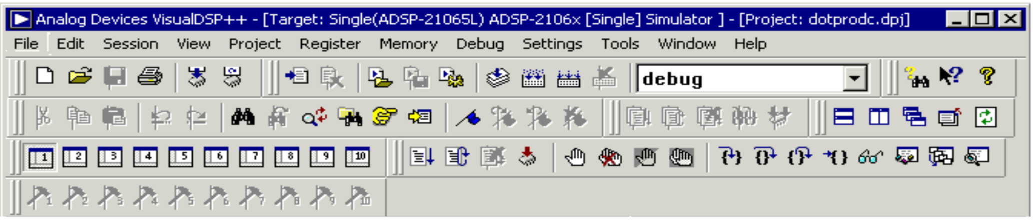

Analog Devices VisualDSP++ - [Target: Single(ADSP-21065L) ADSP-2106x [Single] Simulator] - [Project: dotprodc.dpj]
File
Edit
Session
View
Project
Register
Memory
Debug
Settings
Tools
Window
Help
debug
1
2
3
4
5
6
7
8
9
10

Figure 3-1. VisualDSP++ Toolbar Buttons

Exercise One: Building and Running a C Program

In this exercise, you will complete the following tasks:

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

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

```
Analog Devices\VisualDSP\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++ for SHARC**.

   If you are running VisualDSP++ for the first time, the **New Session** dialog box opens to enable you to set up a session. Select the values shown in the table under step 2 on page 3-10. Then click **OK**. The VisualDSP++ main window appears.

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

2. From the **Project** menu, choose **Open**. VisualDSP++ displays the **Open Project** dialog box.

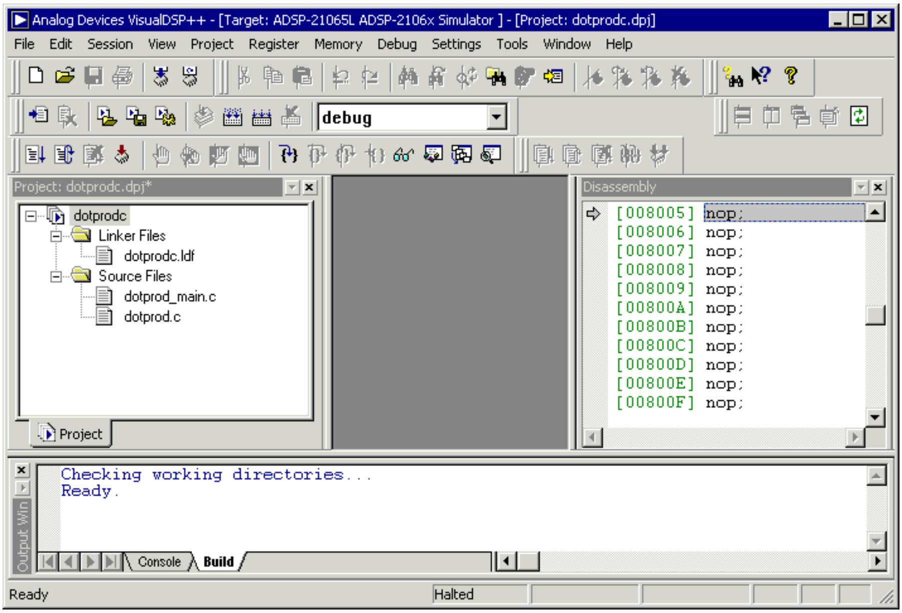
3. In the **Look in** box, open the `Program Files\Analog Devices` folder and double-click the following sub-folders in succession:

   `VisualDSP\21k\Examples\Tutorial\dot_product_c`

   Note: This path is based on the default installation.

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

   VisualDSP++ loads the project, and generates dependencies for the source files. The environment also displays the project files in the Project window and displays messages on the **Build** tab in the Output window as it processes the project settings and file dependencies. See Figure 3-2.



```
[008005] nop;
[008006] nop;
[008007] nop;
[008008] nop;
[008009] nop;
[00800A] nop;
[00800B] nop;
[00800C] nop;
[00800D] nop;
[00800E] nop;
[00800F] nop;
```

```
Checking working directories...
Ready.
```

Figure 3-2. The `dotprodc` Project Files in the Project Window

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

5. From the **Settings** menu, choose **Preferences** to open the **Preferences** dialog box.
6. On the **General** tab page, under **General Preferences**, make sure that the following options are selected:
    - **Run to main after load**
    - **Load executable after build**
7. Click **OK** to close the **Preferences** dialog box.

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**.

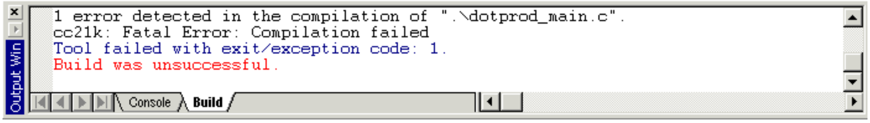
   VisualDSP++ builds the project by using the project source files.

   As the build progresses, the Output window’s **Build** tab 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 on the **Build** tab.

   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 select **Rebuild All**. 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, notice that the compiler detects an undefined identifier and issues the following error in the Output window:

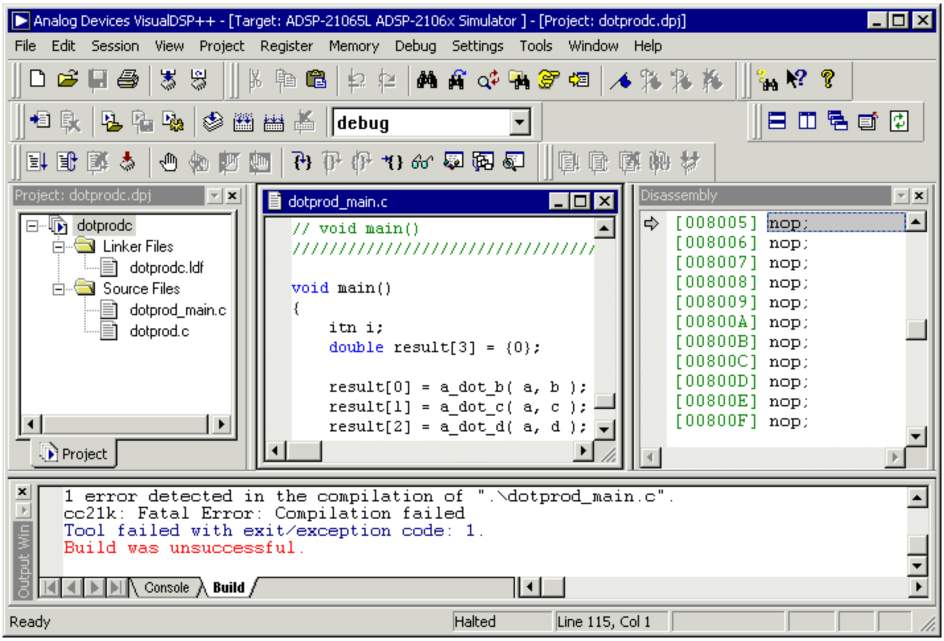


```
1 error detected in the compilation of ".\dotprod_main.c".
cc21k: Fatal Error: Compilation failed
Tool failed with exit/exception code: 1.
Build was unsuccessful.
```

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

   VisualDSP++ opens the C source file `dotprod_main.c` in an Editor window, and places the cursor on the line that contains the error (see Figure 3-3).

The Editor window in Figure 3-3 shows that the integer variable declaration `int` has been misspelled as `itn`.


Analog Devices VisualDSP++ - [Target: ADSP-21065L ADSP-2106x Simulator] - [Project: dotprodc.dpj]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
debug
Project: dotprodc.dpj
dotprodc
Linker Files
dotprodc.ldf
Source Files
dotprod_main.c
dotprod.c
Project
dotprod_main.c
Disassembly
Output Win
Console
Build
Ready
Halted
Line 115, Col 1

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

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

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

```
[008005] nop;
[008006] nop;
[008007] nop;
[008008] nop;
[008009] nop;
[00800A] nop;
[00800B] nop;
[00800C] nop;
[00800D] nop;
[00800E] nop;
[00800F] nop;
```

```
1 error detected in the compilation of ".\dotprod_main.c".
cc21k: Fatal Error: Compilation failed
Tool failed with exit/exception code: 1.
Build was unsuccessful.
```

Figure 3-3. Output Window and Editor Window

3. In the Editor window, click on `itn` and change it to `int`.
4. Save the source file by choosing **Save** from the **File** menu. Notice that `int` is now color-coded to signify that it is a valid C keyword.
5. Build the project again by choosing **Build Project** from the **Project** menu. The project is now built without any errors, as reported on the **Build** tab page in the Output window.

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

Step 3: Run the Program

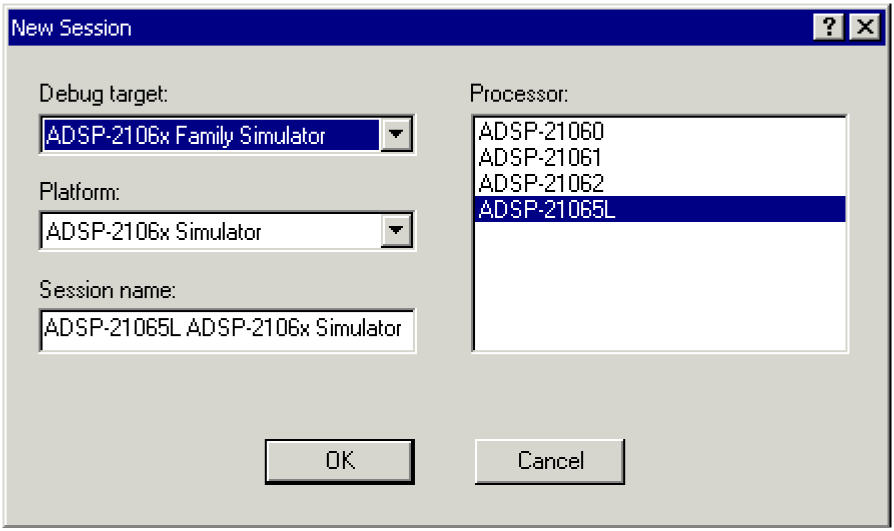
In this procedure, you will complete these tasks:

- 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** tab page in the **Preferences** dialog box, the executable file `dotprodc.dxe` is automatically downloaded to the target. If the debug session's processor does not match the project's build target, VisualDSP++ reports the discrepancy and asks if you want to select another session before downloading the executable to the target. Click **Yes** to continue. If VisualDSP++ does not open the **Session List** dialog box, skip steps 1–5.

To set up the debug session:

1. From the **Session List** dialog box, click **New Session** to open the **New Session** dialog box, shown in Figure 3-4.



Debug target: ADSP-2106x Family Simulator

Platform: ADSP-2106x Simulator

Session name: ADSP-21065L ADSP-2106x Simulator

Processor: ADSP-21065L

Figure 3-4. New Session Dialog Box

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

2. Specify the target and processor information listed in the following table:

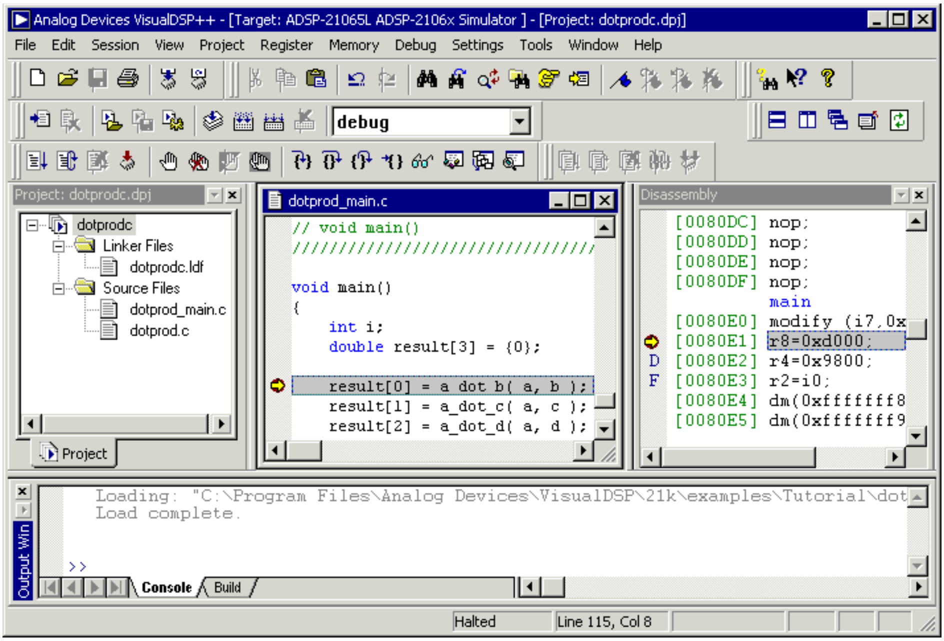
Box	**Value**
Debug Target	ADSP-2106*x* Family Simulator
Platform	ADSP-2106*x* Simulator
Session Name	ADSP-21065L ADSP-2106*x* Simulator
Processor	ADSP-21065L

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**.

   Note: If you do not click **Activate**, the session mismatch message is displayed 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 3-5).


Analog Devices VisualDSP++ - [Target: ADSP-21065L ADSP-2106x Simulator] - [Project: dotprodc.dpj]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
debug
Project: dotprodc.dpj
dotprodc
Linker Files
dotprodc.ldf
Source Files
dotprod_main.c
dotprod.c
Project
dotprod_main.c
Disassembly
Output Win
Console
Build
Halted
Line 115, Col 8

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

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

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

```
  [0080DC] nop;
  [0080DD] nop;
  [0080DE] nop;
  [0080DF] nop;
           main
  [0080E0] modify (i7,0x
  [0080E1] r8=0xd000;
D [0080E2] r4=0x9800;
F [0080E3] r2=i0;
  [0080E4] dm(0xfffffff8
  [0080E5] dm(0xfffffff9
```

```
Loading: "C:\Program Files\Analog Devices\VisualDSP\21k\examples\Tutorial\dot
Load complete.

>>
```

Figure 3-5. Disassembly, Editor, Output Windows: Load dotprodc.dxe

By default, VisualDSP++ opens an Output window, a Disassembly window, and an Editor window that displays the source file containing the project's `main` function, `dotprod_main.c`.

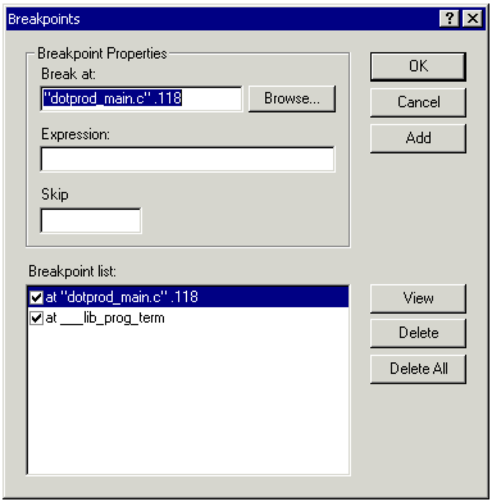
5. Look at the information in the open windows.

   The Output window's **Console** tab 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 containing a yellow arrow appears at address 0x80E1.

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.

6. From the **Settings** menu, choose **Breakpoints** to view the breakpoints set in your program. VisualDSP++ displays the **Breakpoints** dialog box, shown in Figure 3-6.



Break at: "dotprod_main.c".118

☑ at "dotprod_main.c".118
☑ at ___lib_prog_term

Figure 3-6. Breakpoints Dialog Box

The breakpoints are set at these C program labels:

- `“dotprod_main.c” 118`
- `__lib_prog_term`

The **Breakpoints** dialog box enables you to view, add, and delete breakpoints, as well as 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. Use these tool buttons to set or clear breakpoints:

Toggles a breakpoint for the current line

Clears all breakpoints

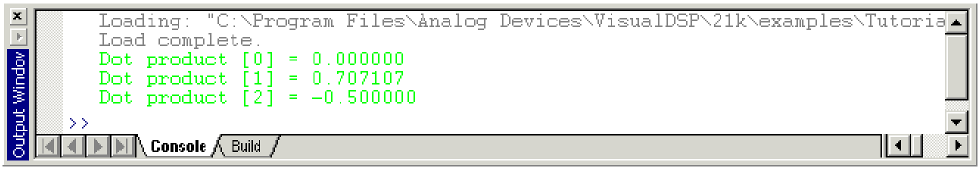
7. Click **OK** or **Cancel** to exit the **Breakpoints** dialog box.

Step 4: Run dotprodc

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

VisualDSP++ computes the dot products and displays the following results on the **Console** tab page (Figure 3-7) in the Output window:

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



```
Loading: "C:\Program Files\Analog Devices\VisualDSP\21k\examples\Tutoria
Load complete.
Dot product [0] = 0.000000
Dot product [1] = 0.707107
Dot product [2] = -0.500000
>>
```

Figure 3-7. Results of the dotprodc Program

Step 5: Profile a_dot_c

You can examine program execution within a selected range of code by using a profile to determine the following information:

- Percentage of time spent executing instructions
- Number of clock cycles spent executing instructions
- Number of instructions executed
- Number of times memory is read from or written to

In this procedure, you will complete the following tasks to profile the `a_dot_c` function:

- Enable profiling to collect execution information during the next program execution
- Specify a start and end address for the code segment to be profiled
- Run the `a_dot_c` program to collect the profile information

To set up profiling for the `a_dot_c` function:

1. From the **Tools** menu, choose **Profile** and then choose **Enable Profiling**, as shown in Figure 3-8.

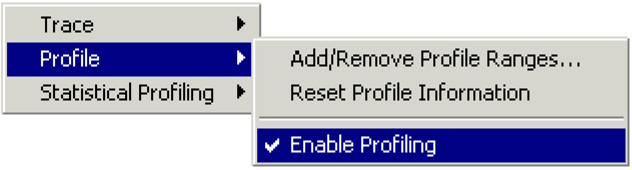
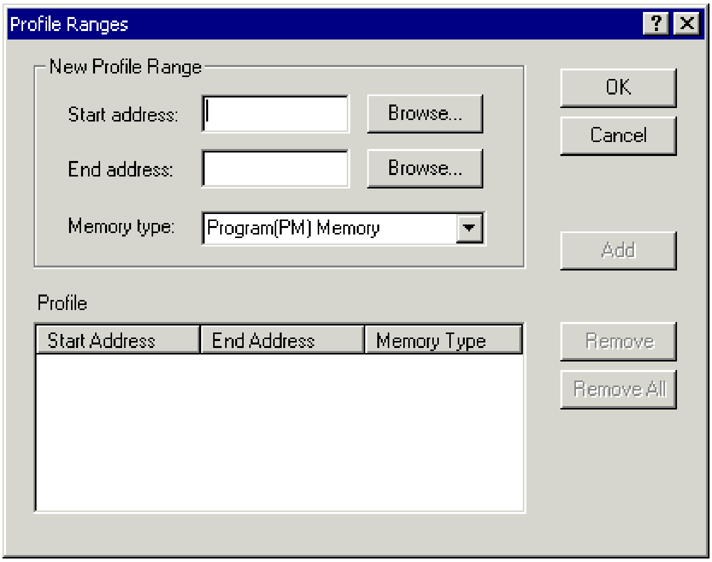


Figure 3-8. Tools Menu: Enable Profiling for a_dot_c

Profiling will be performed when you run the `a_dot_c` program. When the profile is enabled, a check mark appears beside the **Enable Profiling** command on the **Tools** menu. By default, profiling is disabled.

2. From the **Tools** menu, choose **Profile**. Then choose **Add/Remove Profile Ranges** to open the **Profile Ranges** dialog box, shown in Figure 3-9.



Memory type: Program(PM) Memory

Figure 3-9. Profile Ranges Dialog Box

3. In the **New Profile Range** group box, complete the address boxes as follows:

Start address
Click the **Browse** button next to this box to open the **Browse for Symbol** dialog box. Select `a_dot_c`, the label identifying the start of the function, and then click **OK** to enter the label in the **Start address** box.

End address
Use the **Browse** button to select `a_dot_c_end`, the label identifying the last instruction in the function `a_dot_c`.

4. Click **Add**.

   The profile defaults to **Memory type Program(PM) Memory**, which is required for this example.

5. Click **OK** to exit the **Profile Ranges** dialog box and return to the Disassembly window.

6. From the **View** menu, choose **Debug Windows**, and then choose **Profile**. The Profile window displays the results of the profiling session. You can drag and dock the window to the top of the main window (directly under the toolbars) to improve the column view.

   The Profile window shows the address range that you just specified. To collect profile information, however, you must run the program.

7. From the **Debug** menu, choose **Restart**. From the **Tools** menu, choose **Profile**, and then choose **Reset Profile Information**.

8. Press **F5** to run the program. The program runs to the breakpoint set at `main()`.

9. Clear the text displayed on the **Console** tab page as follows:

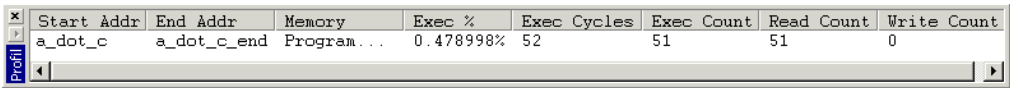
   a. Right-click on the **Console** tab page in the Output window.

   b. Choose **Clear** from the popup menu.

   Press **F5** to continue running the program.

   The program computes the dot products and writes the results to the **Console** tab page. When the program is finished running, the message “`Halted`” appears in the status bar at the bottom of the main window.

The Profile window (Figure 3-10) now contains the results of running the C program. Later in this tutorial, you will compare these results with a profile of the assembly language function `a_dot_c_asm`.



Start Addr	End Addr	Memory	Exec %	Exec Cycles	Exec Count	Read Count	Write Count
a_dot_c	a_dot_c_end	Program...	0.478998%	52	51	51	0

Figure 3-10. Profile Window: Results of the C Function Profile

The fields in the Profile window are described as follows:

Exec %
The total number of clock cycles spent executing instructions in the specified range compared to the total number of cycles spent executing instructions

Exec Cycles
The total number of clock cycles, including all wait states and extra cycles, spent while executing instructions within the profile range

Exec Count
Number of instructions executed in the profile range. This value is **0** when the profiled memory space does not contain instructions. Profiling data memory would result in a **0** Exec count.

Read Count
Number of memory reads from any address in the profile range. The read count includes instruction fetches.

Write Count
Number of memory writes to any address in the profile range

You are now ready to start Exercise Two.

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

In Exercise One, you built and ran a C program. In this exercise, you will modify this program to call an assembly language routine, rebuild the project, and profile the assembly function. The project files are largely identical to those of Exercise One. Minor modifications illustrate the changes needed to call an assembly language routine from C source code.

Step 1: Create a New Project

To create a new project:

1. From the **Project** menu, choose **Close** to close the dotprodc project. Click **Yes** when prompted to close all open source windows.

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

2. From the **Project** menu, choose **New** to open the **Save New Project As** dialog box, shown in Figure 3-11.

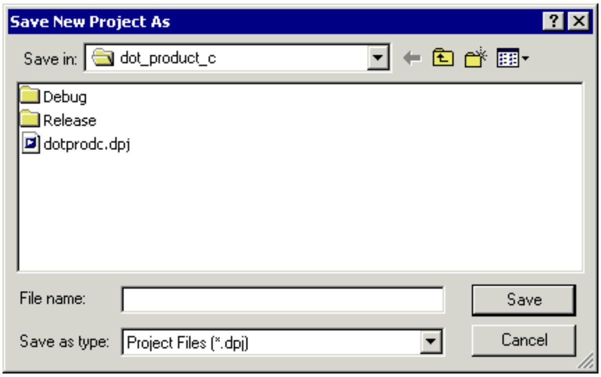
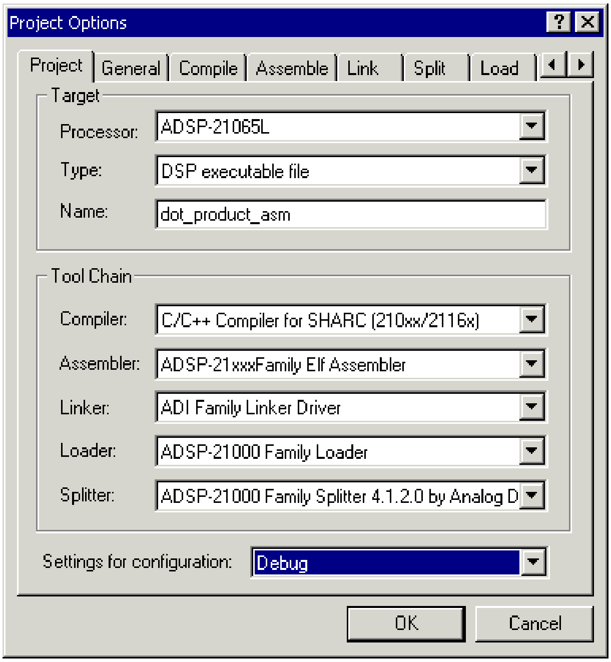


Figure 3-11. Save New Project As Dialog Box

3. Click the up-one-level button until you locate the `dot_product_asm` folder, and then double-click this folder.
4. In the **File name** box, type `dot_product_asm`, and click **Save**.

   The **Project Options** dialog box (Figure 3-12) appears.



Processor: ADSP-21065L

Type: DSP executable file

Name: dot_product_asm

Compiler: C/C++ Compiler for SHARC (210xx/2116x)

Assembler: ADSP-21xxxFamily Elf Assembler

Linker: ADI Family Linker Driver

Loader: ADSP-21000 Family Loader

Splitter: ADSP-21000 Family Splitter 4.1.2.0 by Analog D

Settings for configuration: Debug

Figure 3-12. Project Options Dialog Box: Project Tab Page

This dialog box enables you to specify project build information.

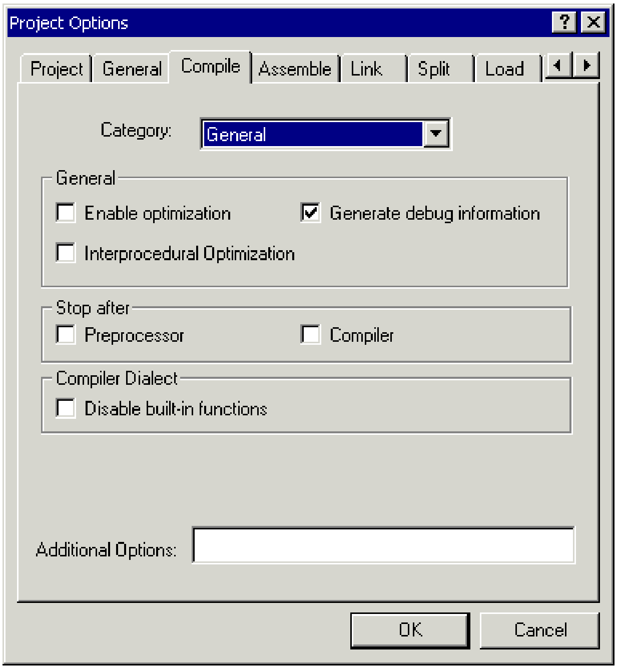
5. Take a moment to view the tab pages in the Project Options window: **Project**, **General**, **Compile**, **Assemble**, **Link**, **Split**, **Load**, and **Post Build**. On each tab page, you specify the tool options used to build the project.

6. On the **Project** tab page (Figure 3-12), specify the following values:

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. The executable contains debug information, so you can examine program execution.

7. Click the **Compile** tab to display the **Compile** tab page, shown in Figure 3-13.



Category: General

General

☐ Enable optimization
☑ Generate debug information
☐ Interprocedural Optimization

Stop after

☐ Preprocessor
☐ Compiler

Compiler Dialect

☐ Disable built-in functions

Additional Options:

Figure 3-13. Project Options Dialog Box: Compile Tab Page

Complete the **General** group box as follows:

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

These settings direct the C compiler to optimize code for the ADSP-21065L DSP. Because the optimization takes advantage of DSP architecture and assembly language features, some of the C debug information is not saved. Debugging is, therefore, performed through debug information at the assembly language level.

8. Click **OK** to apply changes to the project options and to close the **Project Options** dialog box. You are asked if you want to add VisualDSP++ kernel support to the project. Click **No**.

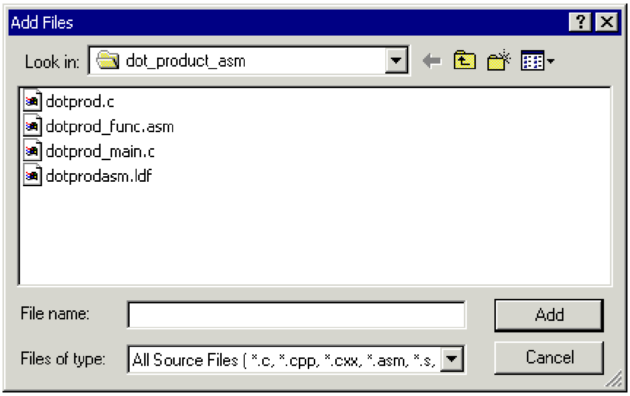
You are now ready to add the source files to the project.

Step 2: Add Source Files to dot_product_asm

To add the source files and Linker Description File 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 3-14) appears.



Look in: dot_product_asm

dotprod.c
dotprod_func.asm
dotprod_main.c
dotprodasm.ldf

File name:

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

Figure 3-14. 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. Press and hold down the **Ctrl** key and click on `dotprod.c`, `dotprod_func.asm`, `dotprod_main.c`, and `dotprodasm.ldf`. Then click **Add**.

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

You are now ready to modify the sources to call the assembly function.

Step 3: Modify the Project Source Files

In this procedure, you will:

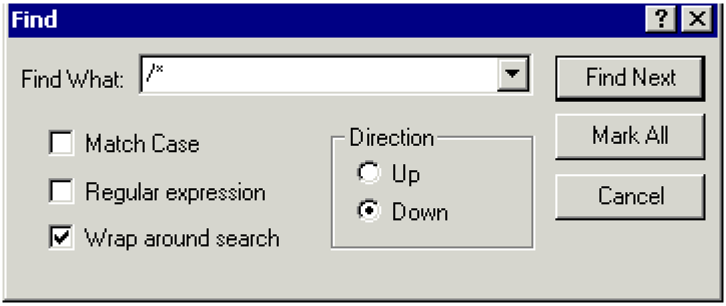
- Modify `dotprod_main.c` to call `a_dot_c_asm` instead of `a_dot_c`
- Save the modified file

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

1. In the Project window, double-click `dotprod_main.c`.

   The C source file opens in an Editor window. Resize or maximize the window for better viewing.

2. From the **Edit** menu, choose **Find** to open the **Find** dialog box, shown in Figure 3-15.



Find What: /*

☐ Match Case

☐ Regular expression

☑ Wrap around search

Direction
○ Up
● Down

Figure 3-15. Find Dialog Box: Locating All 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 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, you will see the code change color as you cut and paste the comment characters.

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

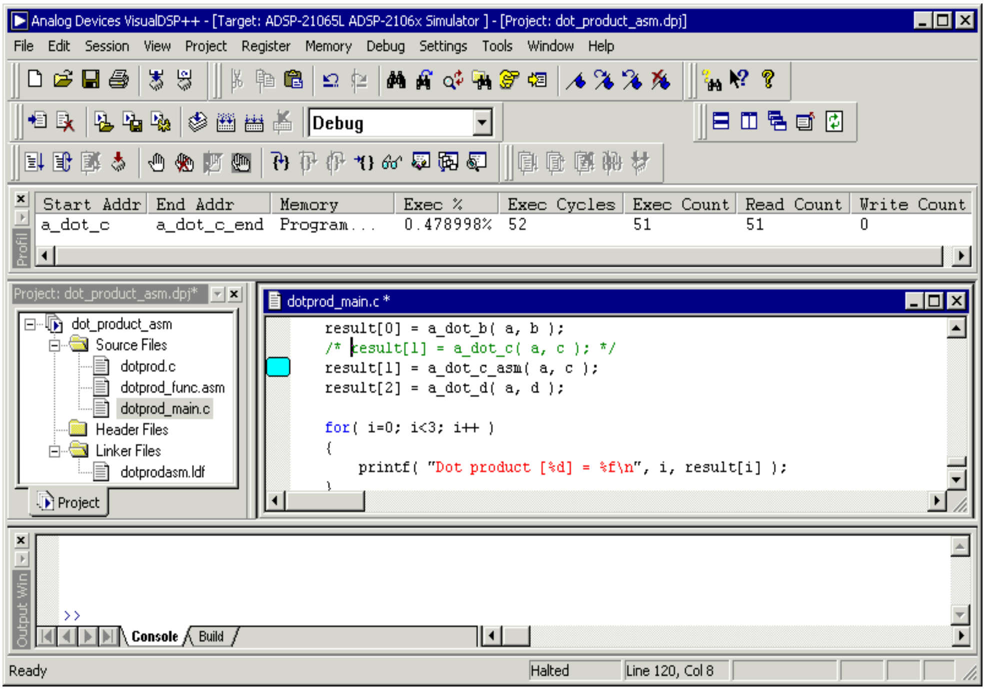
5. Press **F3** to move to the next bookmark.

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

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

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

   Figure 3-16 shows the changes made in step 6.


Analog Devices VisualDSP++ - [Target: ADSP-21065L ADSP-2106x Simulator] - [Project: dot_product_asm.dpj]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
Debug
Profil
Project: dot_product_asm.dpj*
dot_product_asm
Source Files
dotprod.c
dotprod_func.asm
dotprod_main.c
Header Files
Linker Files
dotprodasm.ldf
Project
dotprod_main.c *
Output Win
>>
Console
Build
Ready
Halted
Line 120, Col 8

Start Addr	End Addr	Memory	Exec %	Exec Cycles	Exec Count	Read Count	Write Count
a_dot_c	a_dot_c_end	Program...	0.478998%	52	51	51	0

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

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

Figure 3-16. Editor Window: Modifying dotprod_main.c to Call a_dot_c_asm

7. From the **File** menu, choose **Save** to save the changes that you just made 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 4: Modify dotprodasm.ldf

In this procedure you will:

- View the Linker Description File to see what information and specifications are included
- Modify the Linker Description File to link against the `a_dot_c_asm` assembly routine

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

1. In the Project window, double-click `dotprodasm.ldf` to open the file for editing.

   Use the scroll bar on the right side of the window to scroll through the `.ldf` file. The beginning of the file contains the ADSP-21065L memory map, which describes the processor's physical memory.

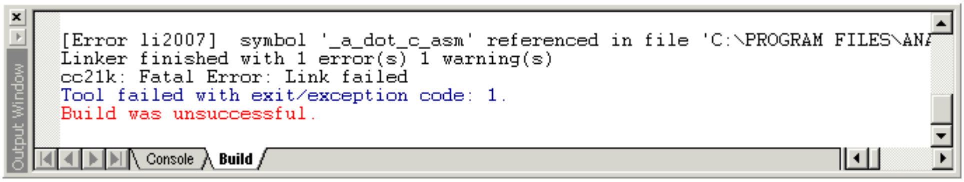
   Following the memory map are commands (`SEARCH_DIR` and `$OBJECTS`) used to define the path names that the linker uses to search and resolve references in the input files.

   Next is the `MEMORY` command, which defines the system's physical memory and assigns labels to logical segments within it. These logical segments define program, data, and stack memory types.

Following the `MEMORY` command is the `SECTIONS` command. The `SECTIONS` command defines the placement of code in physical memory by mapping the sections specified in program files to the sections declared in the `MEMORY` command. The `INPUT_SECTIONS` statement specifies the object file that the linker uses to resolve the mapping.

2. 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 3-17.

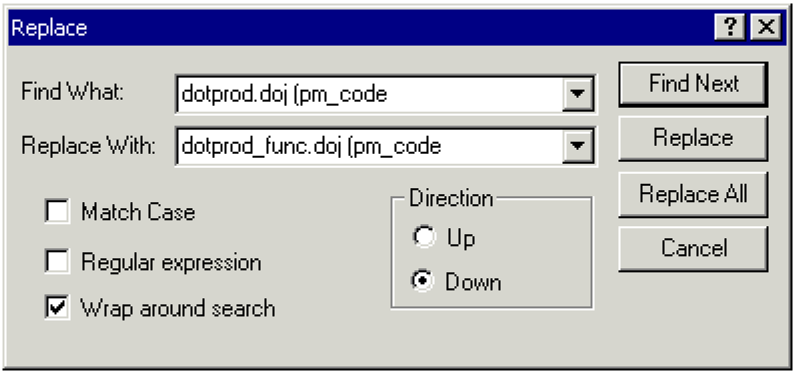


```
[Error li2007]  symbol '_a_dot_c_asm' referenced in file 'C:\PROGRAM FILES\AN
Linker finished with 1 error(s) 1 warning(s)
cc21k: Fatal Error: Link failed
Tool failed with exit/exception code: 1.
Build was unsuccessful.
```

Figure 3-17. Output Window: Linker Error

The reference to `_a_dot_c_asm` could not be resolved because that code has not been placed into the target program's memory. You must change the `SECTIONS` command in the `.ldf` file to inform the linker where to place the code.

3. From the Edit menu, choose Replace to open the **Replace** dialog box, shown in Figure 3-18.



Find What: dotprod.doj (pm_code

Replace With: dotprod_func.doj (pm_code

☐ Match Case

☐ Regular expression

☑ Wrap around search

Direction

○ Up

● Down

Figure 3-18. Replace Dialog Window

4. In the **Find What** box, type: `dotprod.doj(pm_code`
5. In the **Replace With** box, type: `dotprod_func.doj(pm_code`
6. Click **Replace All**.

   Now the `.LDF` links by using `dotprod_func.doj` for the program code segments `pm_code1`, `pm_code2`, and `pm_code3`. The edited statement should look like this:

   ```
   INPUT_SECTIONS ( dotprod.doj (seg_pmco) dotprod_func.doj
   (pm_code1) dotprod_func.doj (pm_code2) dotprod_func.doj
   (pm_code3)
   ```

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

   As shown in Figure 3-19, your `dot_product_asm.dpj` project should now contain the files `dotprod.c`, `dotprod_func.asm`, `dotprod_main.c`, and `dotprodasm.ldf`.

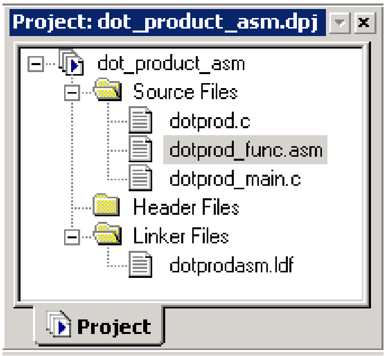


Figure 3-19. Files in the dot_product_asm Project

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

Step 5: Rebuild and Run dot_product_asm

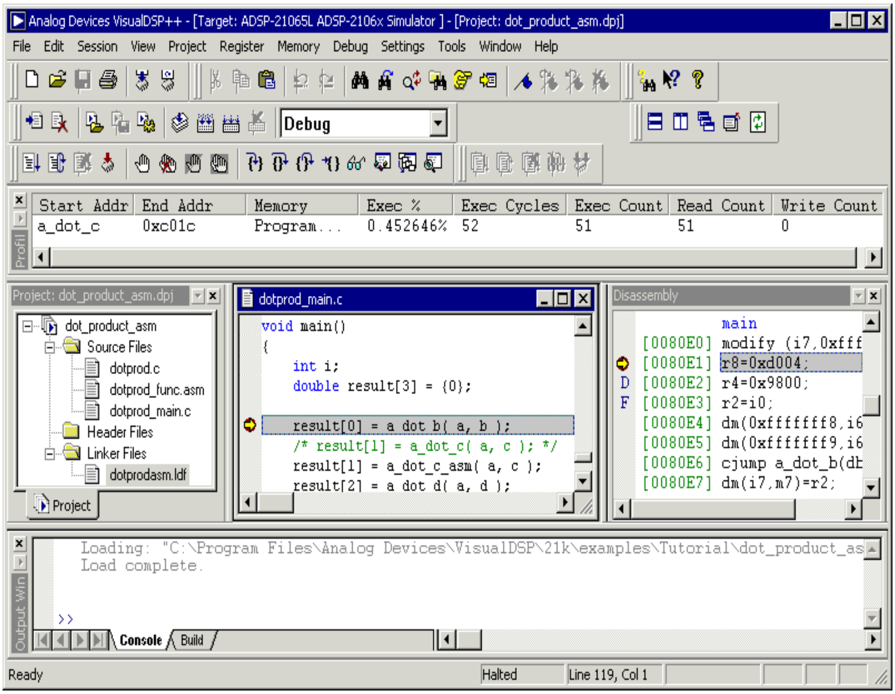
To run `dot_product`:

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

   At the end of the build, the Output window displays the following message on the **Console** tab page:

   “`Load complete.`”

   VisualDSP++ loads the program, runs to main, and displays the Project, Output, Disassembly, and Profile windows (shown in Figure 3-20).


Analog Devices VisualDSP++ - [Target: ADSP-21065L ADSP-2106x Simulator] - [Project: dot_product_asm.dpj]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
Debug
Profil
Project: dot_product_asm.dpj
dot_product_asm
Source Files
dotprod.c
dotprod_func.asm
dotprod_main.c
Header Files
Linker Files
dotprodasm.ldf
Project
dotprod_main.c
Disassembly
D
F
Output Win
Console
Build
Ready
Halted
Line 119, Col 1

Start Addr	End Addr	Memory	Exec %	Exec Cycles	Exec Count	Read Count	Write Count
a_dot_c	0xc01c	Program...	0.452646%	52	51	51	0

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

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

```
          main
[0080E0] modify (i7,0xfff
[0080E1] r8=0xd004;
[0080E2] r4=0x9800;
[0080E3] r2=i0;
[0080E4] dm(0xfffffff8,i6
[0080E5] dm(0xfffffff9,i6
[0080E6] cjump a_dot_b(dt
[0080E7] dm(i7,m7)=r2;
```

```
Loading: "C:\Program Files\Analog Devices\VisualDSP\21k\examples\Tutorial\dot_product_as
Load complete.

>>
```

Figure 3-20. Windows Left Open at the End of the Previous Debugger Session

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

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

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

You are now ready to profile `a_dot_c_asm`.

Step 6: Set Up the Profile dot_product_asm

In this procedure, you will set up the profile of the `a_dot_c_asm` function. You will:

- Enable profiling to collect execution information during the next program execution
- Define the starting address and ending address of the code segment to be profiled

To set up the profile of the `a_dot_c_asm` function:

1. From the **Tools** menu, choose **Profile** and then choose **Enable Profiling** if it is not checked (➽).

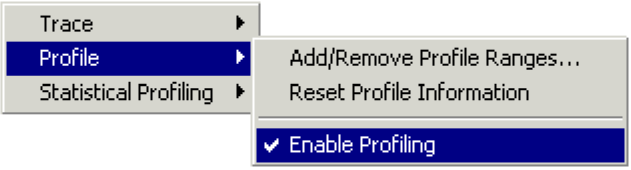
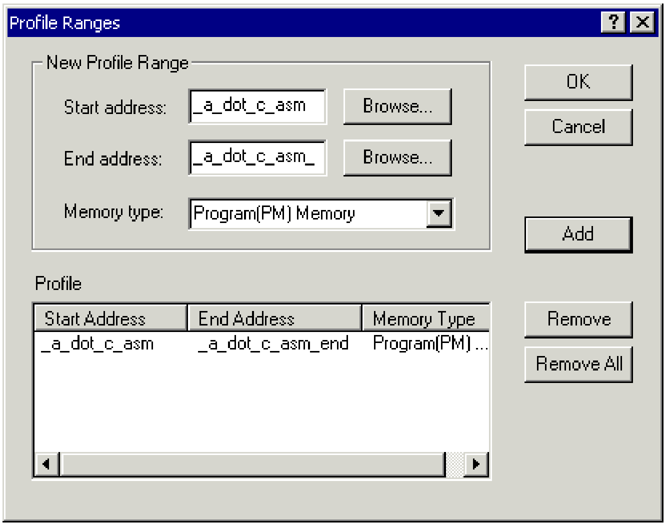


Figure 3-21. Tools Menu: Enable Profiling for a_dot_c_asm

2. From the **Tools** menu, choose **Profile**. Then choose **Add/Remove Profile Ranges** to open the **Profile Ranges** dialog box, shown in Figure 3-22.



Start address: _a_dot_c_asm

End address: _a_dot_c_asm_

Memory type: Program(PM) Memory

Start Address	End Address	Memory Type
_a_dot_c_asm	_a_dot_c_asm_end	Program(PM) ...

Figure 3-22. Profile Ranges Dialog Box

3. Click **Remove All** to remove the profile ranges that you set up in Exercise One.

4. In the **New Profile Range** group box, complete the address boxes as follows:

   Start address
   Click the **Browse** button next to this box to open the **Browse Program Symbols** dialog box. Select `_a_dot_c_asm`, the label identifying the start of the function, and then click **OK** to enter the label in the **Start address** box.

   End address
   Use the **Browse** button to select `_a_dot_c_asm_end`, the label identifying the last instruction in the function `a_dot_c`.

5. Click **Add** to add this range to the Profile list.

The profile **Memory type** defaults to **Program (PM) Memory**, which is required for this example.

6. Click **OK** to exit the **Profile Ranges** dialog box and return to the Disassembly window.
7. From the **View** menu, choose **Debug Windows**, and then choose **Profile**. The Profile window displays the results of the profiling session. You can drag and dock the window to the top of the main window (directly under the toolbars) to improve the column view.

   The Profile window shows the address range that you just specified. To collect profile information, however, you must run the program.

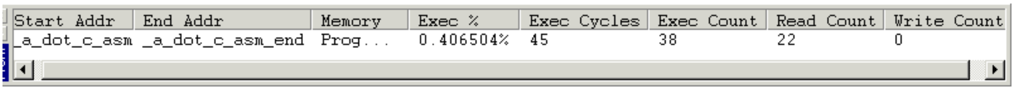
You are now ready to run the `dot_product_asm` program.

Step 7: Run dot_product_asm

To run `dot_product_asm`:

1. From the **Debug** menu, choose **Restart**. From the **Tools** menu, choose **Profile**, and then choose **Reset Profile Information**.
2. Press **F5** to run the program. The program runs to the breakpoint set at `main()`.
3. Clear the text displayed on the **Console** tab page as follows:
   a. Right-click on the **Console** tab page in the Output window.
   b. Choose **Clear** from the popup menu.
4. Press **F5** to continue running the program.

The program computes the dot products and writes the results to the **Console** tab page. The program halts at `a_dot_c_asm_end` and displays the profile results in the Profile window, as shown in Figure 3-23.



Start Addr	End Addr	Memory	Exec %	Exec Cycles	Exec Count	Read Count	Write Count
_a_dot_c_asm	_a_dot_c_asm_end	Prog...	0.406504%	45	38	22	0

Figure 3-23. Profile Window: Results of the Assembly Language Function a_dot_c_asm

The fields in the Profile window are described on page 3-17.

You are now ready to compare the profile results.

Step 8: Compare the Profile Results

You have now profiled two functions that solve the same problem (each computes a dot product). One function is written in C and one is written in assembly. Table 3-1 compares the results of the two versions. The hand-coded assembly version shows a significant performance boost.

Note: Your actual values may differ slightly from those shown in the following table, depending on the compiler version that you are using.

Table 3-1. Profile Results: a_dot_c vs. a_dot_c_asm

Function	Exec %	Exec Cycles	Exec Count	Read Count	Write Count
`a_dot_c`	0.478998%	52	51	51	0
`_a_dot_c_asm`	0.406504%	45	38	22	0

Assembly language routines have two advantages over C code. First, the assembly instruction set is optimized for the processor. The instruction set supports multifunction, parallel operation instructions that provide highly specialized optimizations and perform multiple operations that complete in a single cycle.

Second, programs coded in assembly also have low overhead, compared to programs coded in C. When you code in assembly, you save only the used registers. When you code in C, the compiler saves and restores reserved registers. The C code operates in a run-time environment, defined for the DSP, which also adds some overhead to the code.

You are now ready to start Exercise Three.

Exercise Three: Plotting Data

In this exercise, you will load and debug a pre-built program that applies a simple convolution algorithm to a buffer of data. You will use VisualDSP++'s 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. Keep the Disassembly window and **Console** tab page (in the Output window) open, but close all other windows still open from the previous exercises.

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 your **Analog Devices** folder and double-click the following sub-folders in succession:

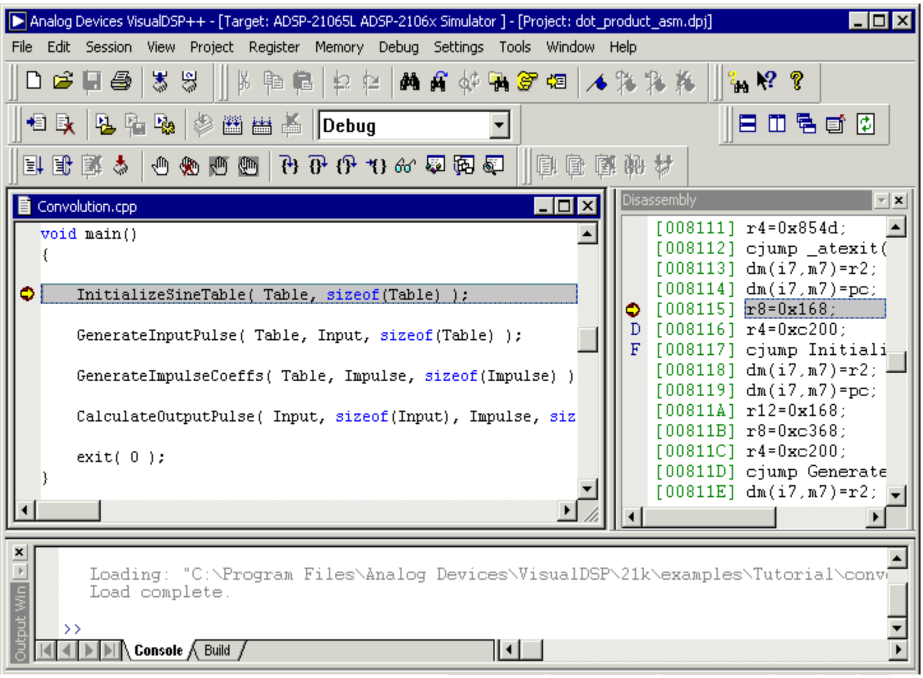
      `VisualDSP\21k\Examples\Tutorial\convolution\debug`

   b. Double-click `Convolution.dxe` to load the program.

   c. When prompted to look for `Convolution.cpp`, click **Yes** to open the **Find** dialog box.

   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 3-24.


Analog Devices VisualDSP++ - [Target: ADSP-21065L ADSP-2106x Simulator] - [Project: dot_product_asm.dpj]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
Debug
Convolution.cpp
Disassembly
Output Win
Console
Build

```
void main()
{

    InitializeSineTable( Table, sizeof(Table) );

    GenerateInputPulse( Table, Input, sizeof(Table) );

    GenerateImpulseCoeffs( Table, Impulse, sizeof(Impulse) )

    CalculateOutputPulse( Input, sizeof(Input), Impulse, siz

    exit( 0 );
}
```

```
  [008111] r4=0x854d;
  [008112] cjump _atexit(
  [008113] dm(i7,m7)=r2;
  [008114] dm(i7,m7)=pc;
  [008115] r8=0x168;
D [008116] r4=0xc200;
F [008117] cjump Initiali
  [008118] dm(i7,m7)=r2;
  [008119] dm(i7,m7)=pc;
  [00811A] r12=0x168;
  [00811B] r8=0xc368;
  [00811C] r4=0xc200;
  [00811D] cjump Generate
  [00811E] dm(i7,m7)=r2;
```

```
Loading: "C:\Program Files\Analog Devices\VisualDSP\21k\examples\Tutorial\conv
Load complete.

>>
```

Figure 3-24. Loading the Convolution Program

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

   You can see four global data arrays: `Table`, `Input`, `Output`, and `Impulse`.

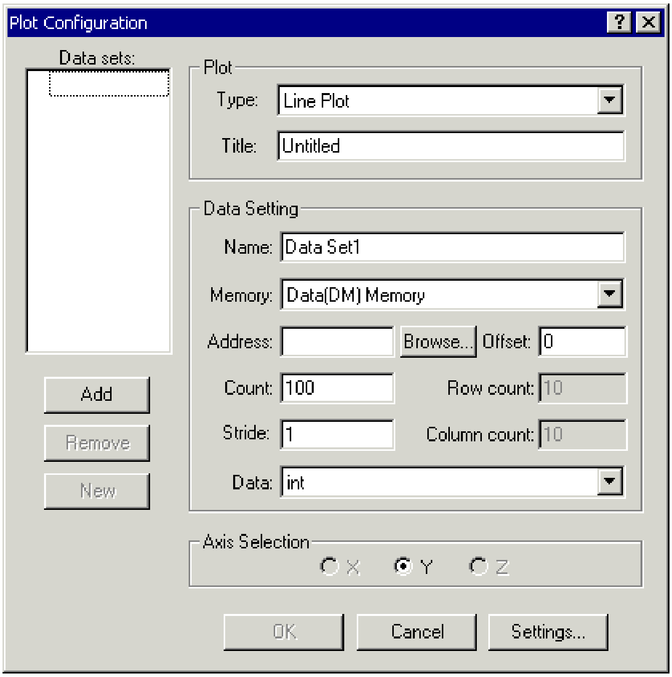
   You can also see four functions that operate on these arrays: `InitializeSineTable()`, `GenerateInputPulse()`, `GenerateImpulseCoeffs()`, and `CalculateOutputPulse()`. Stepping over each function will enable you to see the data being calculated in a plot window.

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 3-25.



Plot Configuration

Data sets:

Plot
Type: Line Plot
Title: Untitled

Data Setting
Name: Data Set1
Memory: Data(DM) Memory
Address:
Offset: 0
Count: 100
Row count: 10
Stride: 1
Column count: 10
Data: int

Axis Selection
○ X ● Y ○ Z

Figure 3-25. Plot Configuration Dialog Box: Specifying Data Sets to Be Plotted

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

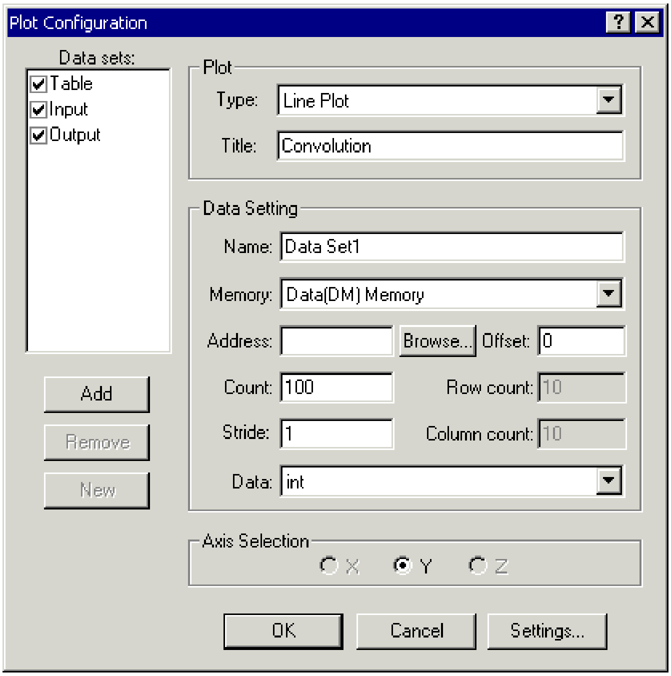
2. In the **Plot** group box, specify the following values:
    - In the **Type** box, select **Line Plot** from the drop-down menu.
    - In the **Title** box, type **convolution**.
3. Enter three data sets to plot by using the values in Table 3-2.

    After entering each data set, click **Add** to add the data set to the **Data Sets** list, as shown in Figure 3-26.

Table 3-2. 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.

The **Plot Configuration** dialog box should now look like this:



Plot Configuration

Data sets:

☑ Table
☑ Input
☑ Output

Plot

Type: Line Plot
Title: Convolution

Data Setting

Name: Data Set1
Memory: Data(DM) Memory
Address:
Offset: 0
Count: 100
Row count: 10
Stride: 1
Column count: 10
Data: int

Axis Selection

○ X ● Y ○ Z

Figure 3-26. Plot Configuration Dialog Box: Entering the Table, Input, and Output Data Sets

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

   The Plot window now displays the three arrays. Since, by default, the Simulator initializes memory to zero, the three data sets appear as one horizontal line, shown in Figure 3-27.

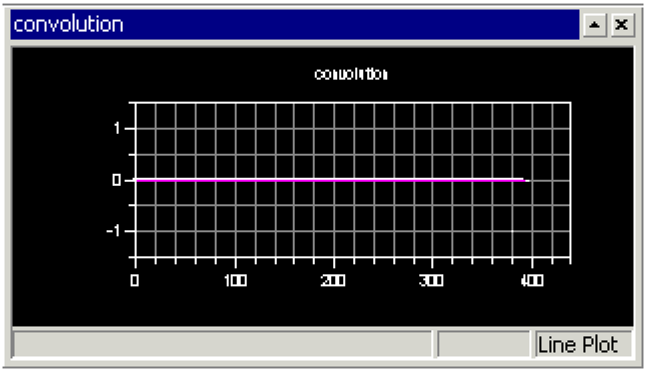


Figure 3-27. Plot Window: Before Running the Convolution Program

Step 3: Run the Convolution Program and View the Data

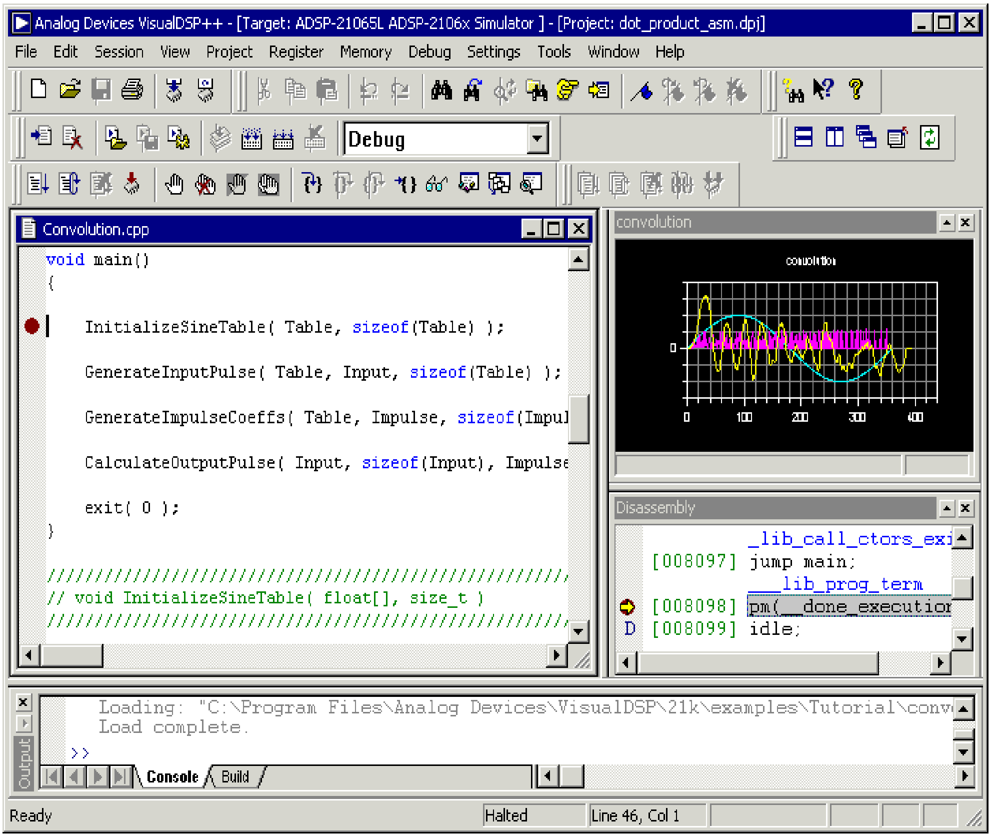
To run the `Convolution` program and view the data:

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

   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.

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.

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 3-28.


Analog Devices VisualDSP++ - [Target: ADSP-21065L ADSP-2106x Simulator] - [Project: dot_product_asm.dpj]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
Debug
Convolution.cpp
convolution
convolution
0
0
100
200
300
400
Disassembly
Output
Console
Build
Ready
Halted
Line 46, Col 1

```
void main()
{
    InitializeSineTable( Table, sizeof(Table) );

    GenerateInputPulse( Table, Input, sizeof(Table) );

    GenerateImpulseCoeffs( Table, Impulse, sizeof(Impul

    CalculateOutputPulse( Input, sizeof(Input), Impulse

    exit( 0 );
}

///////////////////////////////////////////////////////
// void InitializeSineTable( float[], size_t )
///////////////////////////////////////////////////////
```

```
          _lib_call_ctors_exi
[008097] jump main;
          ___lib_prog_term
[008098] pm(___done_executior
[008099] idle;
```

```
Loading: "C:\Program Files\Analog Devices\VisualDSP\21k\examples\Tutorial\conv
Load complete.
>>
```

Figure 3-28. Plot Window: After Running the Convolution Program to Completion

Next you will zoom in on a particular region of interest in the Plot window to focus in on the 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 3-29 shows the selected region, and Figure 3-30 shows the magnified result.

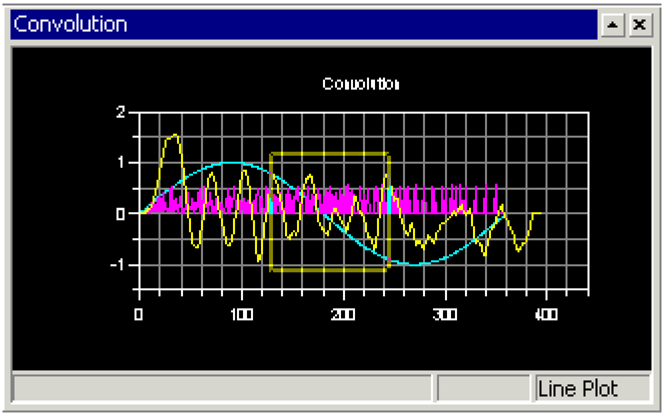


Figure 3-29. Plot Window: Selecting a Region to Magnify

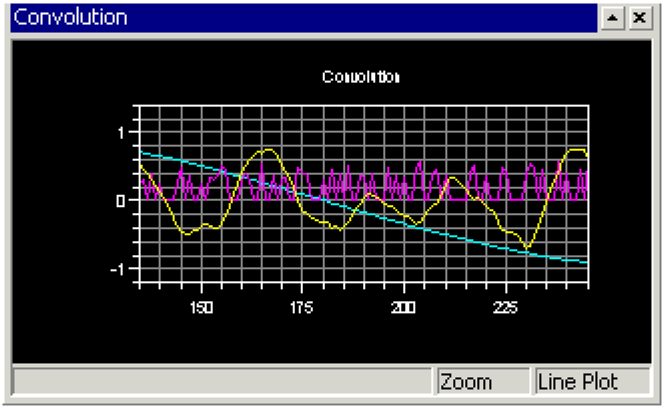
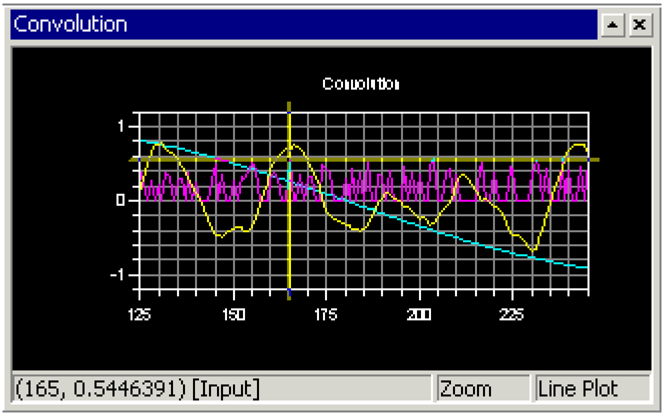


Figure 3-30. Plot Window: Magnified Result

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

You can view individual data points in the **Data Cursor** from the popup 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 3-31.



(165, 0.5446391) [Input]

Figure 3-31. Plot Window: Viewing Individual Data Points by 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 popup menu.

You are now ready to start Exercise Four.

Exercise Four: Statistical Profiling

In this exercise, you will load and debug the `Convolution` program from the previous exercise. You will use statistical 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.

Tip: 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 following sub-folders in succession:

      `VisualDSP\21k\Examples\Tutorial\convolution\debug`

   b. Double-click `Convolution.dxe` to load and run the `Convolution` program.

c. When prompted to look for `Convolution.cpp`, click **Yes** to open the **Find** dialog box.

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 enable statistical profiling.

Step 2: Enable Statistical Profiling

To enable statistical profiling:

1. From the **Tools** menu, choose **Statistical Profiling**, and then choose **Enable Profiling**.

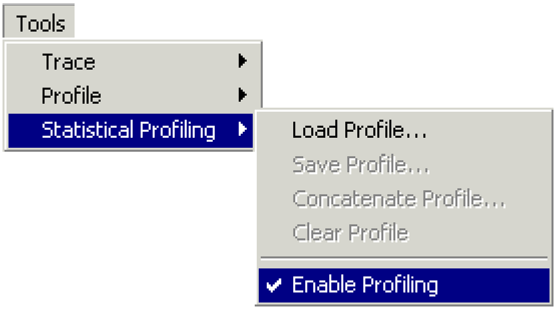
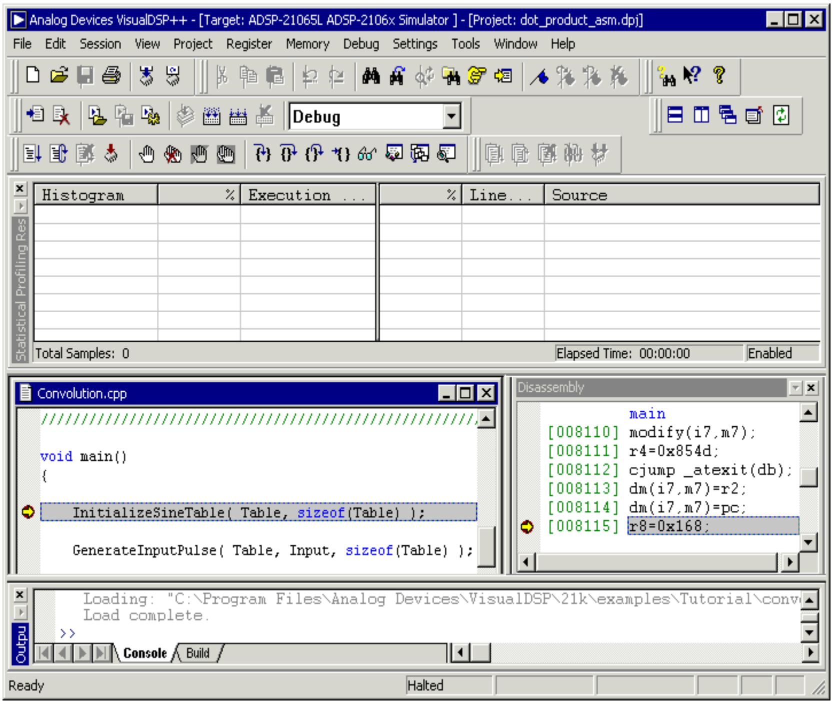


Figure 3-32. Enabling Statistical Profiling for the Convolution Program

A check mark appears beside **Enable Profiling** to indicate that statistical profiling is enabled.

Statistical profiling will be performed when you next run the `Convolution` program. By default, statistical profiling is disabled.

2. From the **View** menu, choose **Debug Windows**. Then choose **Statistical Profiling Results** to open the **Statistical Profiling Results** window, shown in Figure 3-33.


Analog Devices VisualDSP++ - [Target: ADSP-21065L ADSP-2106x Simulator] - [Project: dot_product_asm.dpj]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
Debug
Statistical Profiling Res
Histogram
%
Execution ...
%
Line...
Source
Total Samples: 0
Elapsed Time: 00:00:00
Enabled
Convolution.cpp
Disassembly
Outpu
Console
Build
Ready
Halted

```
////////////////////////////////////////////////////////////

void main()
{

    InitializeSineTable( Table, sizeof(Table) );

    GenerateInputPulse( Table, Input, sizeof(Table) );
```

```
          main
[008110] modify(i7,m7);
[008111] r4=0x854d;
[008112] cjump _atexit(db);
[008113] dm(i7,m7)=r2;
[008114] dm(i7,m7)=pc;
[008115] r8=0x168;
```

```
  Loading: "C:\Program Files\Analog Devices\VisualDSP\21k\examples\Tutorial\conv
  Load complete.
>>
```

Figure 3-33. Statistical Profiling Results Window (Empty)

The Statistical Profiling Results window is intially empty. After you run the program and collect data, this window displays the results of the profiling session.

For a better view of the data, use the window's title bar to drag and dock the window to the top of the VisualDSP++ main window.

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

Step 3: Collect and Examine the Statistical Profile Data

To collect and examine the statistical profile data:

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

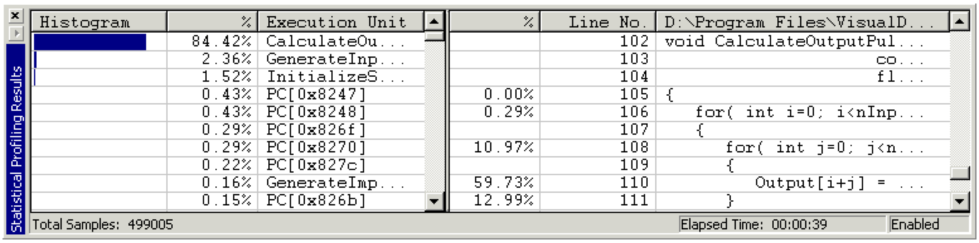
   When the program halts, the results of the linear profile appear in the left pane of the Statistical Profiling Results window (see Figure 3-34).

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

2. Double-click on a line in the left pane to display the corresponding source code for the profile data in the right pane.

3. When prompted to look for the `Convolution.cpp` file, click **Yes**.

   The source code for the profile data appears in the right pane, as shown in Figure 3-34.



Histogram	%	Execution Unit
	84.42%	CalculateOu...
	2.36%	GenerateInp...
	1.52%	InitializeS...
	0.43%	PC[0x8247]
	0.43%	PC[0x8248]
	0.29%	PC[0x826f]
	0.29%	PC[0x8270]
	0.22%	PC[0x827c]
	0.16%	GenerateImp...
	0.15%	PC[0x826b]

%	Line No.	D:\Program Files\VisualD...
	102	`void CalculateOutputPul...`
	103	`co...`
	104	`fl...`
0.00%	105	`{`
0.29%	106	`for( int i=0; i<nInp...`
	107	`{`
10.97%	108	`for( int j=0; j<n...`
	109	`{`
59.73%	110	`Output[i+j] = ...`
12.99%	111	`}`

Total Samples: 499005

Elapsed Time: 00:00:39

Enabled

Figure 3-34. Statistical Profiling Results of Analyzing the Performance of the Convolution Program

4. Examine the results of your statistical profiling session.

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

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

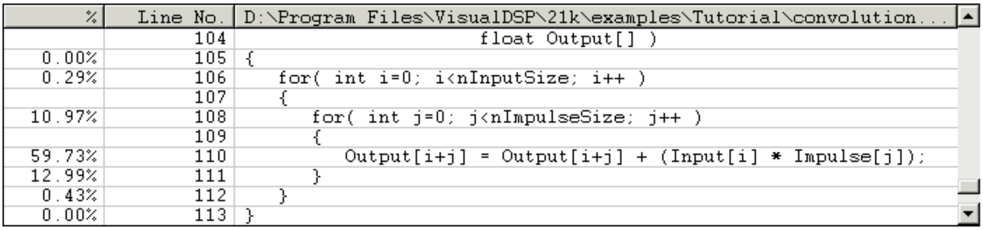
%
The numerical percent of the same data found in the Histogram column. You can view this value as an absolute number of samples by right-clicking in the Statistical Profiling Results window and by selecting **View Sample Count** from the popup menu.

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

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

The left pane in Figure 3-34 shows that the function `CalculateOutputPulse()` has consumed over 84% of the total execution time. Double-clicking one of these lines displays the source file, `Convolution.cpp`, in the right (source) pane. The source pane displays data for each line of executable code in the file for which statistical profile data has been collected.

Double-clicking the line with the `CalculateOutputPulse()` function in the left pane displays the statistical profile data shown in Figure 3-35 in the right pane.



```
      %   Line No. D:\Program Files\VisualDSP\21k\examples\Tutorial\convolution...
               104                        float Output[] )
  0.00%        105 {
  0.29%        106    for( int i=0; i<nInputSize; i++ )
               107    {
 10.97%        108       for( int j=0; j<nImpulseSize; j++ )
               109       {
 59.73%        110          Output[i+j] = Output[i+j] + (Input[i] * Impulse[j]);
 12.99%        111       }
  0.43%        112    }
  0.00%        113 }
```

Figure 3-35. Statistical Profile Data for Convolution.cpp

The details of the `CalculateOutputPulse()` function show that 59% 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.

You are now ready to start Exercise Five.

Exercise Five: Multiprocessor Debugging

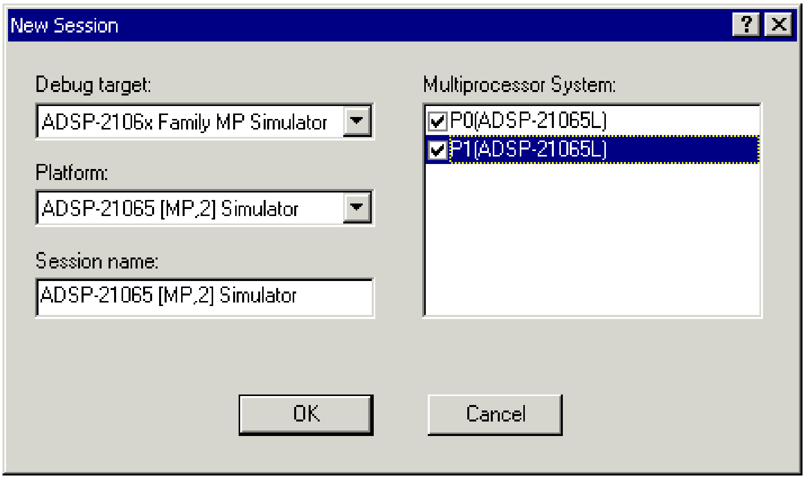
In this exercise you will create a multiprocessor (MP) simulator session and explore the various features for debugging complex multiprocessor systems. You will use synchronous MP commands to control multiple targets, pin windows to specific processors, and use MP groups to control multiple subsets of processors in an MP system.

Step 1: Create a Multiprocessor Simulator Session

To create a multiprocessor simulator session:

1. If necessary, start VisualDSP++. (VisualDSP++ automatically connects to the last session that was open.)

2. From the **Session** menu, choose **New Session** to open the **New Session** dialog box, shown in Figure 3-36.



New Session

Debug target: ADSP-2106x Family MP Simulator

Platform: ADSP-21065 [MP,2] Simulator

Session name: ADSP-21065 [MP,2] Simulator

Multiprocessor System:
☑ P0(ADSP-21065L)
☑ P1(ADSP-21065L)

Figure 3-36. New Session Dialog Box: Setting up a Multiprocessor Simulator Session

3. Create a new multiprocessor simulator session by specifying the values listed in the following table:

Box	Value
Debug Target	ADSP-2106*x* Family MP Simulator
Platform	ADSP-21065 (MP,2) Simulator
Session Name	ADSP-21065L (MP,2) Simulator

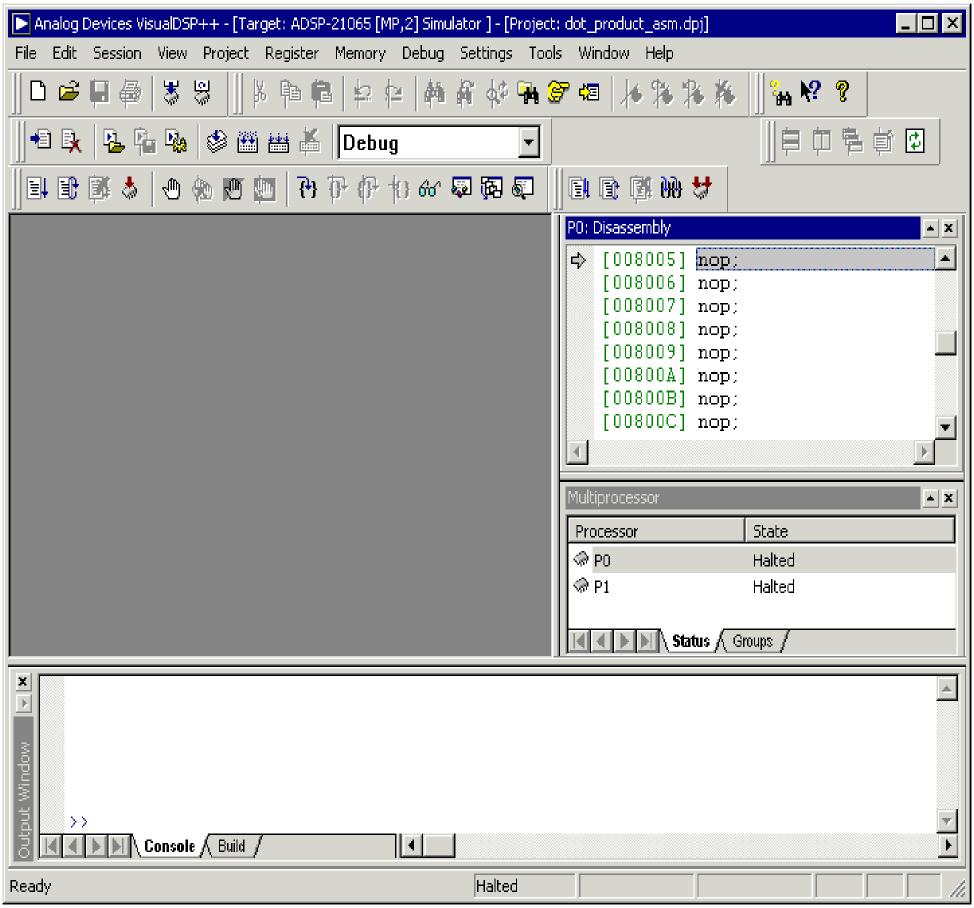
4. Under **Multiprocessor System**, select the check boxes next to processors **P0** and **P1**, as shown in Figure 3-36.

   The check marks (✓) indicate that these processors belong to the default multiprocessor group created when VisualDSP++ attaches to the session. Multiprocessor groups are described in greater detail later in this exercise.

5. Click **OK** to create the session.

   VisualDSP++ closes the current session and attaches to the new multiprocessor session specified above.

   Figure 3-37 shows the windows in the new multiprocessor session.



```
[008005] nop;
[008006] nop;
[008007] nop;
[008008] nop;
[008009] nop;
[00800A] nop;
[00800B] nop;
[00800C] nop;
```

Processor	State
P0	Halted
P1	Halted

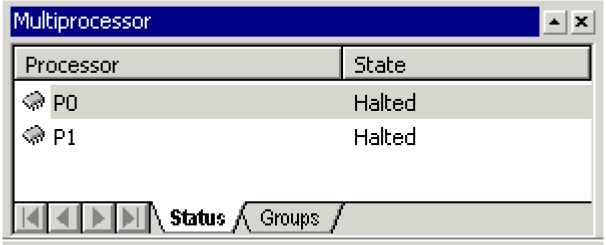
Figure 3-37. VisualDSP++ Windows: After Creating a Multiprocessor Session

6. Take a moment to examine the session windows. Except for a few minor changes, notice that a multiprocessor session looks nearly identical to a single-processor session.

The main differences are the:

- Multiprocessor window, with the tabs **Status** and **Groups**
- Name of the processor (such as **P0**) in the title bar of each debug window displaying processor information
- Multiprocessor toolbar buttons to run five MP commands

In Figure 3-37, the Multiprocessor window (Figure 3-38) appears under the Disassembly window in the lower-right portion of the VisualDSP++ main window.



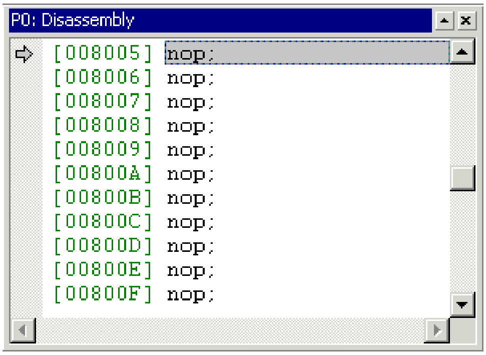
Processor	State
P0	Halted
P1	Halted

Figure 3-38. Multiprocessor Window: Status Tab Page

The **Status** tab page lists each processor in the session and its current status: **Running**, **Halted**, **Stepping**, or **Unknown**.

The title bar of each debug window (such as Disassembly, Memory, or Registers) includes the name of the processor from which information has been collected.

For example, the title bar of the Disassembly window shown in Figure 3-37 and in Figure 3-39 indicates that information was collected from processor **P0**.



```
[008005] nop;
[008006] nop;
[008007] nop;
[008008] nop;
[008009] nop;
[00800A] nop;
[00800B] nop;
[00800C] nop;
[00800D] nop;
[00800E] nop;
[00800F] nop;
```

Figure 3-39. Disassembly Window: Disassembled Instructions in Processor P0

Figure 3-40 shows the **Multiprocessor** toolbar.

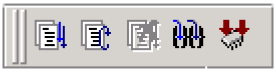


Figure 3-40. Multiprocessor Toolbar: Provides Access to Common Multiprocessor Commands

The buttons on this toolbar provide easy access to the following commands:

Multiprocessor Run

Multiprocessor Restart

Multiprocessor Halt

Multiprocessor Step

Multiprocessor Reset

You can also access these commands from the **Debug** menu by choosing **Multiprocessor**.

You are now ready to change focus and pin windows.

Step 2: Changing Focus and Pinning Windows

You can easily view data from any processor in the current session. Displaying data from a different processor in the same window is called *changing focus*. On the Multiprocessor window's **Status** tab page (Figure 3-38), the first processor listed, **P0**, is highlighted and currently *has focus* by default.

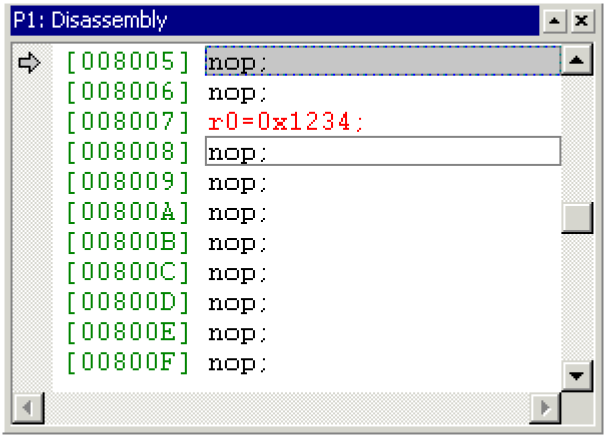
During a debug session, you may want a debugger window to display the data from only one particular processor and to maintain that focus as new processors are selected. This feature is known as *pinning* a window.

To change focus and pin windows:

1. On the Multiprocessor window's **Status** tab page (Figure 3-38), click **P1**.

   Notice that the title bar of the Disassembly window changes to "**P1: Disassembly**" to indicate that this window is now focused on processor P1. The window displays P1's disassembled instructions.

2. Click on the text field to the right of address `0x8007` in the Disassembly window. A gray box appears around the field.
3. Type the instruction `r0=0x1234` and press **Enter**. The instruction is written into P1's memory, and the Disassembly window is updated as shown in Figure 3-41 to reflect this new instruction.

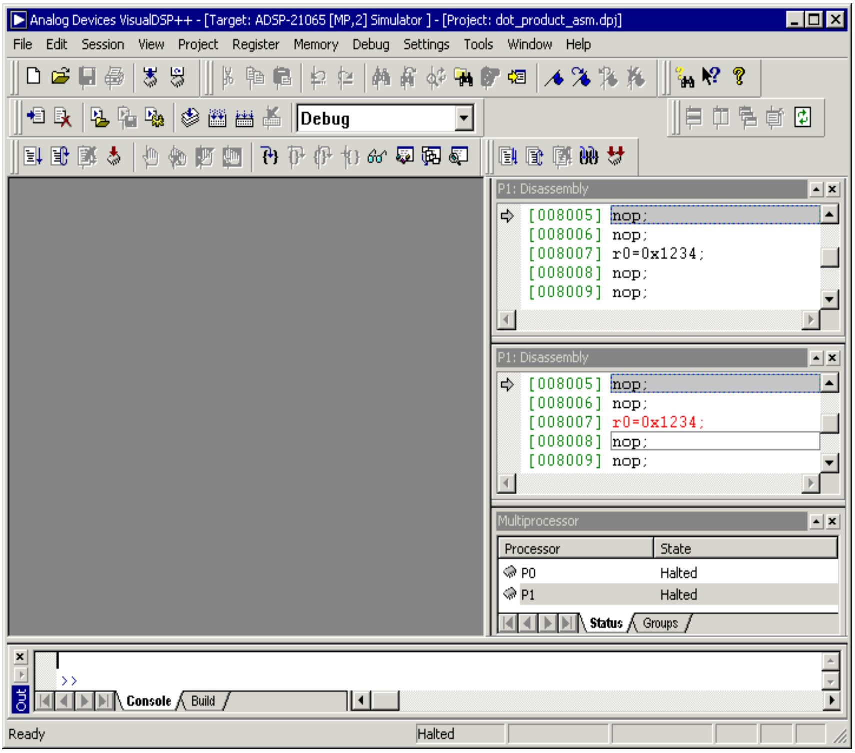


```
[008005] nop;
[008006] nop;
[008007] r0=0x1234;
[008008] nop;
[008009] nop;
[00800A] nop;
[00800B] nop;
[00800C] nop;
[00800D] nop;
[00800E] nop;
[00800F] nop;
```

Figure 3-41. Disassembly Window: Patching a New Instruction into Processor P1’s Memory

4. Open a second Disassembly window from the **View** menu by choosing **Debug Windows** and **Disassembly**.

   Two Disassembly windows focus on processor P1, as shown in Figure 3-42.


Analog Devices VisualDSP++ - [Target: ADSP-21065 [MP,2] Simulator] - [Project: dot_product_asm.dpj]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
Debug
P1: Disassembly
P1: Disassembly
Multiprocessor
Status
Groups
Out
Console
Build
Ready
Halted

```
[008005] nop;
[008006] nop;
[008007] r0=0x1234;
[008008] nop;
[008009] nop;
```

```
[008005] nop;
[008006] nop;
[008007] r0=0x1234;
[008008] nop;
[008009] nop;
```

Processor	State
P0	Halted
P1	Halted

Figure 3-42. Two Open Disassembly Windows Focused on Processor P1

5. Select different processors from the Multiprocessor window's **Status** tab page (Figure 3-38) and notice that both Disassembly windows change focus to the currently selected processor. Also note that the `r0=0x1234` instruction is present only when the Disassembly windows are focused on P1.

6. Select processor **P0**.

7. Right-click in the top Disassembly window and choose **Pin to Processor** from the popup menu, shown in Figure 3-43.

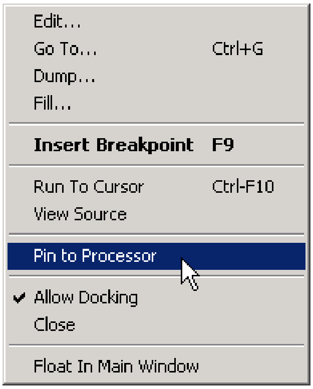
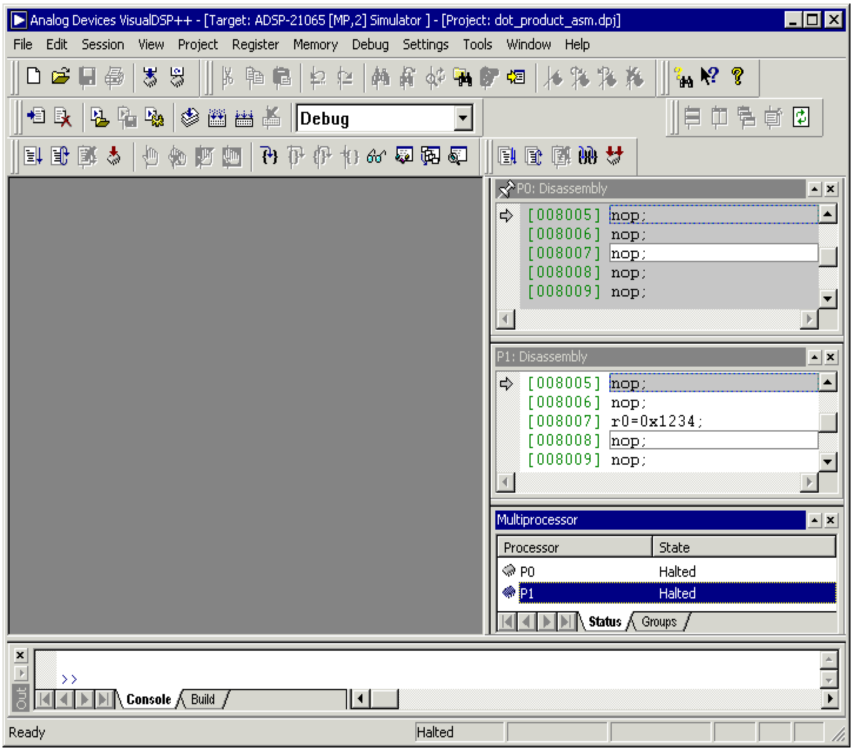


Figure 3-43. Selecting Pin to Processor

A small pin icon 📌 appears in the title bar of the Disassembly window. This icon indicates that the window is now “locked” onto processor P0, and will always display data from P0, regardless of the currently focused processor.

8. On the Multiprocessor window’s **Status** tab page (Figure 3-38), select processor **P1**. Notice that the bottom Disassembly window changes focus to processor P1 and that the top Disassembly window stays focused on processor P0.

   Figure 3-44 shows the top (P0) and bottom (P1) Disassembly windows.


Analog Devices VisualDSP++ - [Target: ADSP-21065 [MP,2] Simulator] - [Project: dot_product_asm.dpj]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
Debug
P0: Disassembly
P1: Disassembly
Multiprocessor
Status Groups
Console Build
Ready
Halted

```
[008005] nop;
[008006] nop;
[008007] nop;
[008008] nop;
[008009] nop;
```

```
[008005] nop;
[008006] nop;
[008007] r0=0x1234;
[008008] nop;
[008009] nop;
```

Processor	State
P0	Halted
P1	Halted

Figure 3-44. Pinning a Disassembly Window

The top Disassembly window is shaded gray to indicate that it is not displaying data from the currently focused processor. When many windows are open simultaneously during a debug session, shading helps you see which windows are focused on the current processor.

9. Right-click in the bottom Disassembly window and choose **Pin to Processor** from the popup menu.

   The bottom Disassembly window is now pinned to P1.

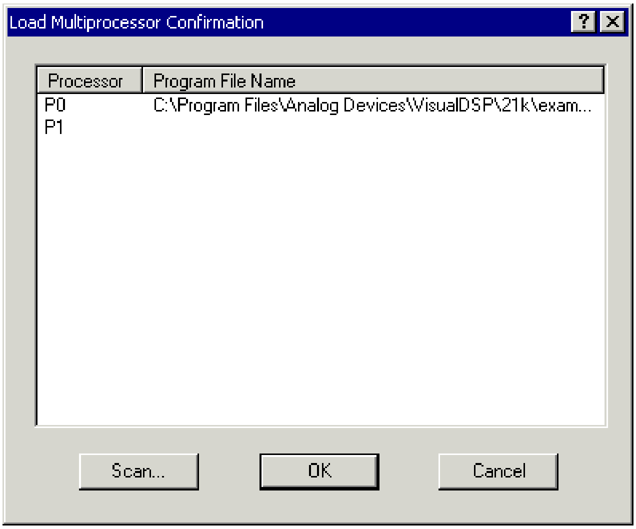
You are now ready to load programs in a multiprocessor session.

Step 3: Loading Programs in a Multiprocessor Session

To load programs to all the processors in a multiprocessor session:

1. On the Multiprocessor window's **Status** tab page (Figure 3-38), select **P0** to set focus to processor P0.
2. Click the **Load Program** button, or choose **Load Program** from the **File** menu. The **Open a Processor Program** dialog box appears.
3. Open your `Analog Devices` folder and double-click the following sub-folders in succession:

   `VisualDSP\21k\examples\tutorial\dot_product_c\debug`
4. Double-click `dotprodc.dxe` to open the **Load Multiprocessor Confirmation** dialog box, shown in Figure 3-45.



Processor	Program File Name
P0	C:\Program Files\Analog Devices\VisualDSP\21k\exam...
P1	

Figure 3-45. Load Multiprocessor Confirmation Dialog Box: Specifying Load dotprodc.dxe into Processor P0

Since P0 is the currently selected processor, the path to `dotprodc.dxe` is added to the P0 row.

5. Click the left mouse button in the **Program File Name** column next to P1.

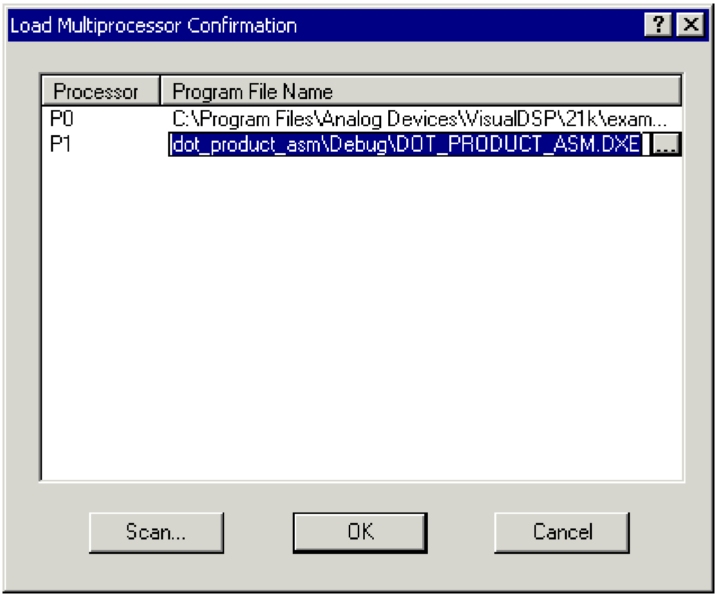
   An edit box appears to enable you to enter the name of the program to load into processor P1.

6. Load the `dot_product_asm.dxe` program as follows:

   a. Click the **Browse** button ... to open the **Select a Processor Program** dialog box, which enables you to browse for the program to load.

   b. Click the up-one-level button 📁 until you locate the `dot_product_asm` folder. Then double-click the `dot_product_asm` and `debug` folders in succession.

   c. Double-click `dot_product_asm.dxe` to place the path to this program into the P1 row of the **Load Multiprocessor Confirmation** dialog box (see Figure 3-46).



Processor	Program File Name
P0	C:\Program Files\Analog Devices\VisualDSP\21k\exam...
P1	dot_product_asm\Debug\DOT_PRODUCT_ASM.DXE

Figure 3-46. Load Multiprocessor Confirmation Dialog Box: Specifying Load dot_product_asm.dxe into Processor P1

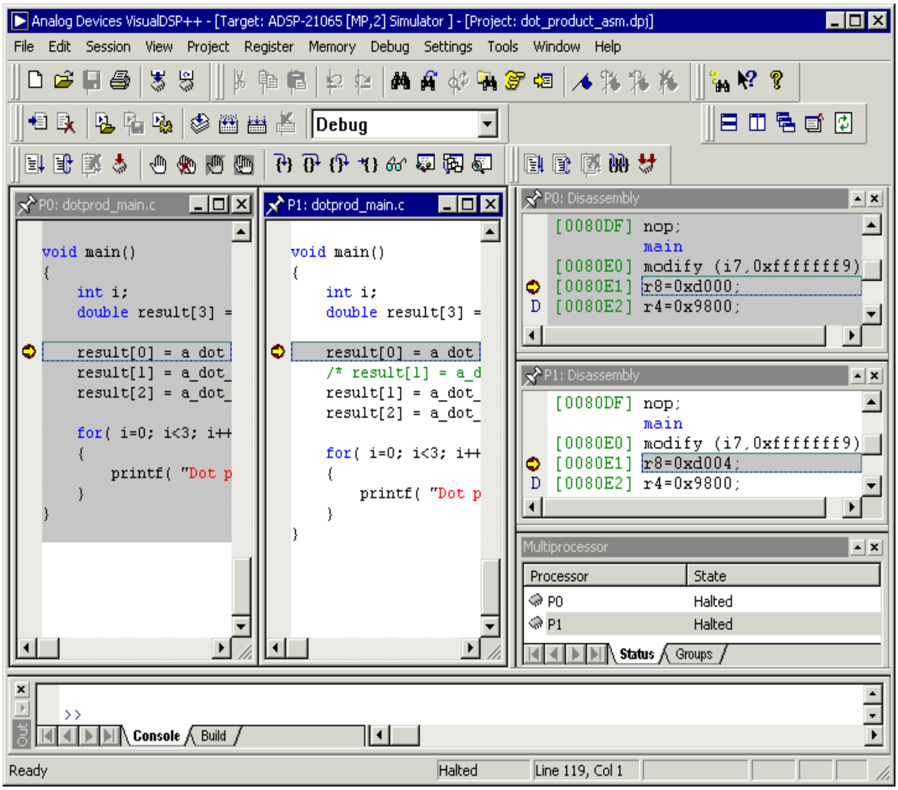
d. Click **OK** to load each program into its respective processor.

   The source files for each program open in Editor windows, and both processors run to the first executable line in `main()`. If the source files do not appear, right-click in each Disassembly window and choose **View Source**.

7. Perform one of the following actions to rearrange the Editor windows for easier viewing:

   - From the **Window** menu, choose **Tile Vertically**.
   - Click the **Tile Vertically** button .

   Figure 3-47 shows the Editor windows tiled vertically.


Analog Devices VisualDSP++ - [Target: ADSP-21065 [MP,2] Simulator] - [Project: dot_product_asm.dpj]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
Debug
P0: dotprod_main.c
P1: dotprod_main.c
P0: Disassembly
P1: Disassembly
Multiprocessor
Status Groups
Out
Console Build
Ready
Halted
Line 119, Col 1

P0: dotprod_main.c

```
void main()
{
    int i;
    double result[3] =

    result[0] = a_dot
    result[1] = a_dot_
    result[2] = a_dot_

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

P1: dotprod_main.c

```
void main()
{
    int i;
    double result[3] =

    result[0] = a_dot
    /* result[1] = a_d
    result[1] = a_dot_
    result[2] = a_dot_

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

P0: Disassembly

```
  [0080DF] nop;
           main
  [0080E0] modify (i7,0xfffffff9)
  [0080E1] r8=0xd000;
D [0080E2] r4=0x9800;
```

P1: Disassembly

```
  [0080DF] nop;
           main
  [0080E0] modify (i7,0xfffffff9)
  [0080E1] r8=0xd004;
D [0080E2] r4=0x9800;
```

Processor	State
P0	Halted
P1	Halted

```
>>
```

Figure 3-47. Two Programs Loaded onto Two Separate Processors

Notice that the Editor windows are pinned to their respective processors.

8. Click the left mouse button in the Disassembly window pinned to P0. Then click the left mouse button in the bottom Disassembly window pinned to P1.

   Notice that for all pinnable windows, including Editor windows, the focus follows the currently active window to facilitate navigation between processors.

Step 4: Stepping in a Multiprocessor Session

When debugging a multiprocessor session, you can still use all the standard single-processor commands such as **Run**, **Step**, **Halt**, **Reset**, and **Restart**. Each command operates only on the currently focused processor. All other processors are unaffected.

The multiprocessor commands **MP Run**, **MP Step**, and **MP Halt** operate *synchronously*. All three operations execute on exactly the same clock cycle in all processors. This feature is helpful when critical multiprocessor interaction issues arise during the development process. **MP Load**, **MP Reset**, and **MP Restart** are **not** synchronous operations. They are executed in order, one after the other.

To step instructions in a multiprocessor session:

1. Set the focus to processor **P0**.
2. Step P0 two assembly instructions as follows:
    a. Click the left mouse button in the Disassembly window pinned to P0.
    b. Click the **Step Into** button twice.

    Notice that the P0 Disassembly window is now at address `0x80E3`, and the P1 Disassembly window is still at address `0x80E1`.

    You can issue Multiprocessor commands to both processors at the same time.
3. Click the **Multiprocessor Step** button twice.

    Notice that both processors P0 and P1 have stepped instructions. P0 is now at address `0x80E5`, and P1 is at address `0x80E3`.

Step 5: Configuring and Using Multiprocessor Groups

Often, in large multiprocessor systems, individual processors are organized into a group of processors that work together to perform a related task. Using multiprocessor groups is helpful for debugging complex systems. For example, you can send multiprocessor commands to all the groups at one time. By changing the active group, you can send debugging commands to only the processors in one particular group.

When you create a multiprocessor session, a group named "**Default**" is created. The processors that you select in the **New Session** dialog box are added to the **Default** group. (See "Step 1: Create a Multiprocessor Simulator Session" on page 3-51.) Up to this point, you have worked with only two processors in a multiprocessor session. You will now work with six processors to configure and use multiprocessor groups.

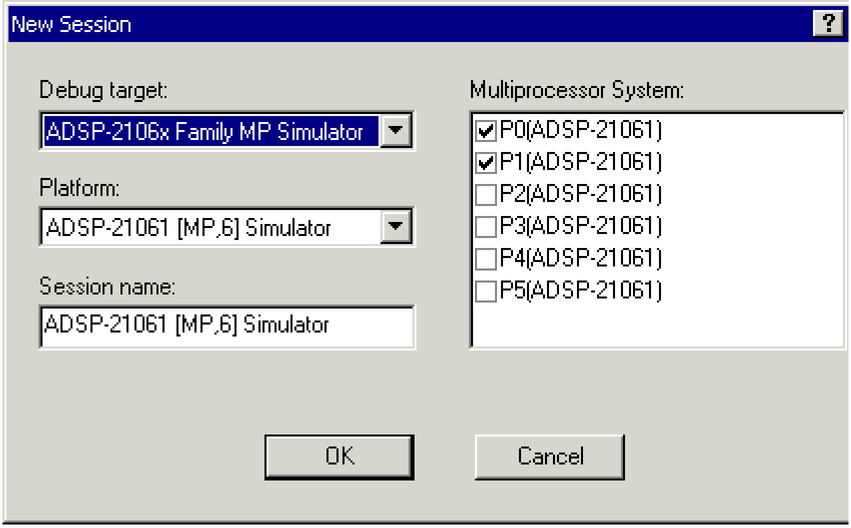
In this step you will complete the following tasks:

- Create a new multiprocessor session that uses six processors
- Add processors to the **Default** group and issue an **MP Reset** command to all the processors
- Create new groups and add processors to them
- Make different groups active

Creating a New Multiprocessor Simulator Session

To create a multiprocessor simulator session:

1. If necessary, start VisualDSP++. (VisualDSP++ automatically connects to the last session that was open.)
2. From the **Session** menu, choose **New Session** to open the **New Session** dialog box, shown in Figure 3-48.



Debug target: ADSP-2106x Family MP Simulator

Platform: ADSP-21061 [MP,6] Simulator

Session name: ADSP-21061 [MP,6] Simulator

Multiprocessor System:

☑ P0(ADSP-21061)
☑ P1(ADSP-21061)
☐ P2(ADSP-21061)
☐ P3(ADSP-21061)
☐ P4(ADSP-21061)
☐ P5(ADSP-21061)

Figure 3-48. New Session Dialog Box: Setting up a Multiprocessor Simulator Session

3. Create a new multiprocessor simulator session by specifying the values listed in the following table:

Box	**Value**
Debug Target	ADSP-2106*x* Family MP Simulator
Platform	ADSP-21061 (MP,6) Simulator
Session Name	ADSP-21061 (MP,6) Simulator

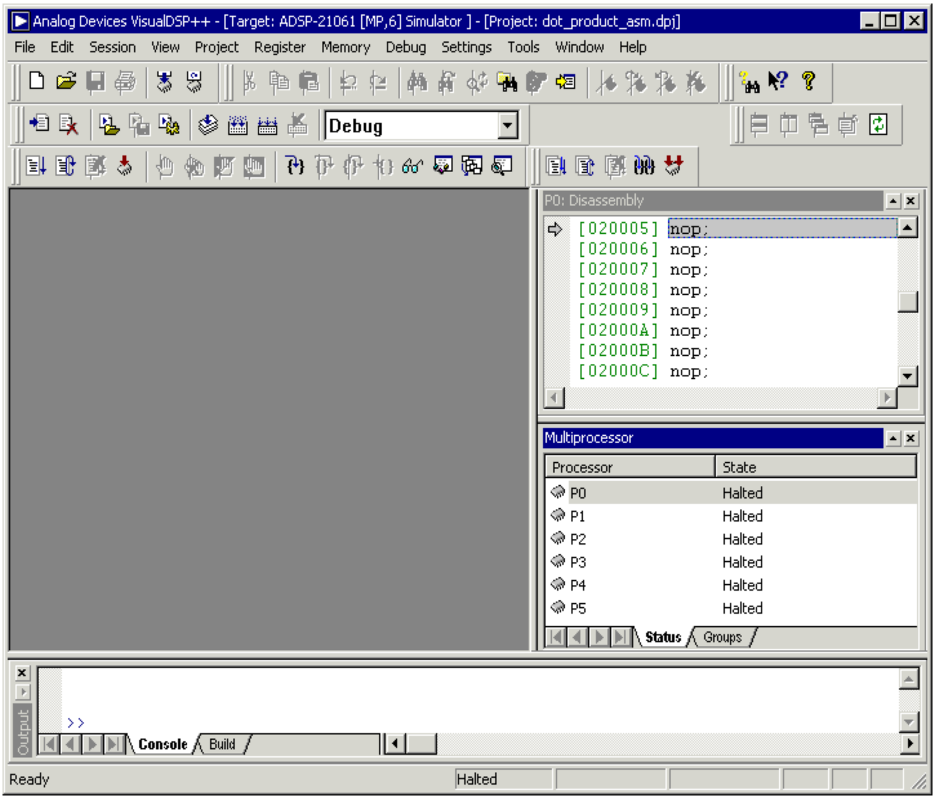
4. Under **Multiprocessor System**, select the check boxes next to processors **P0** and **P1**, as shown in Figure 3-48.

The check marks (✓) indicate that these processors belong to the default multiprocessor group created when VisualDSP++ attaches to the session. Multiprocessor groups are described in greater detail later in this exercise.

5. Click **OK** to create the session.

   VisualDSP++ closes the current session and attaches to the new multiprocessor session specified above.

   Figure 3-49 shows the windows in the new multiprocessor session.



```
[020005] nop;
[020006] nop;
[020007] nop;
[020008] nop;
[020009] nop;
[02000A] nop;
[02000B] nop;
[02000C] nop;
```

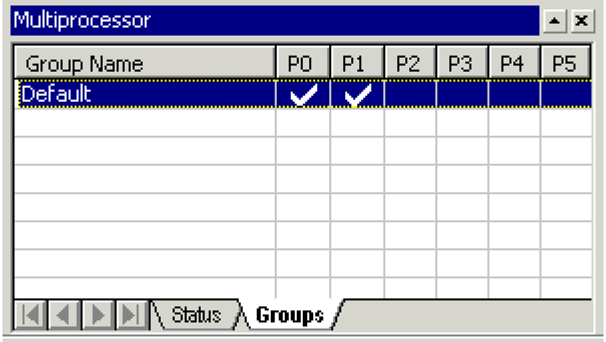
Processor	State
P0	Halted
P1	Halted
P2	Halted
P3	Halted
P4	Halted
P5	Halted

Figure 3-49. VisualDSP++ Windows in the New Multiprocessor Session

Adding Processors to the Default Group and Issuing MP Reset

To add processors to the **Default** group and issue an **MP Reset** command:

1. In the Multiprocessor window, click the **Groups** tab to display the **Groups** tab page, shown in Figure 3-50.



Group Name	P0	P1	P2	P3	P4	P5
Default	✓	✓				

Figure 3-50. Multiprocessor Window: Groups Tab Page

Notice that processors P0 and P1 are checked to show that they are in the **Default** group, and processors P2–P5 are not. All the MP commands (**MP Run**, **MP Step**, **MP Halt**, **MP Reset**, and **MP Restart**) are sent only to the processors in this **Default** group. The remaining processors are unaffected by these commands.

2. Right-click in the Multiprocessor Group window.

3. Choose **Select All Processors** from the popup menu, shown in Figure 3-51.

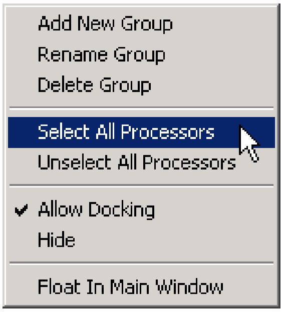
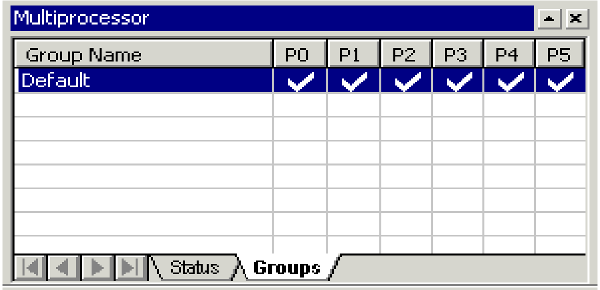


Figure 3-51. Moving All Processors into the Default Group

In the Multiprocessor Group window, a check mark appears for all six processors in the **Default** group (see Figure 3-52).



Group Name	P0	P1	P2	P3	P4	P5
Default	✓	✓	✓	✓	✓	✓

Figure 3-52. All Processors Moved into the Default Group

4. Issue an **MP Reset** command to all six processors by performing one of these actions:

   - Click the **Multiprocessor Reset** button .
   - From the **Debug** menu, choose **Multiprocessor** and **Reset**.

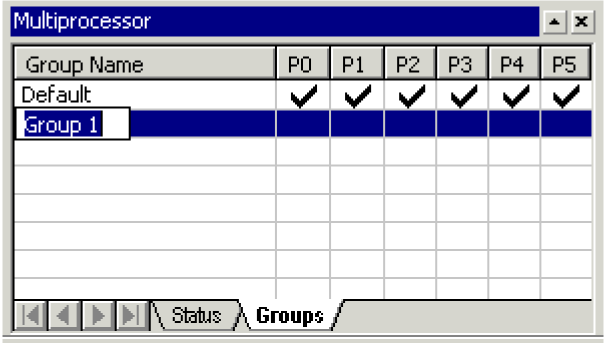
   Because all six processors are in the selected group, the reset is applied to all of them.

Creating and Configuring New Multiprocessor Groups

To create new groups and add processors to them:

1. Right-click in the Multiprocessor Group window.
2. Choose **Add New Group** from the popup menu.

   This command creates an empty group (**Group 1**), as shown in Figure 3-53. You can change the group's name. For this exercise, however, you will use the default.

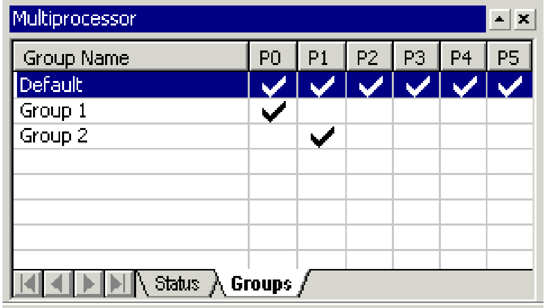


Group Name	P0	P1	P2	P3	P4	P5
Default	✓	✓	✓	✓	✓	✓
Group 1						

Figure 3-53. Multiprocessor Window, Groups Tab Page: Creating a New Group

3. Press **Enter** to accept the default name, **Group 1**.
4. Place processor P0 into this new group by clicking the left mouse button in the **P0** column of the **Group 1** row.
5. Create a second new group as explained above and accept the default name **Group 2**.
6. Place processor P1 into this new group by clicking the left mouse button in the **P1** column of the **Group 2** row.

   Figure 3-54 shows the two new multiprocessor groups.



Group Name	P0	P1	P2	P3	P4	P5
Default	✓	✓	✓	✓	✓	✓
Group 1	✓					
Group 2		✓				

Figure 3-54. Multiprocessor Window, Groups Tab Page: Creating Two New Groups

Activating New Multiprocessor Groups

To make the new groups active:

1. Open a second Disassembly window and pin one window to **P0** and the other window to **P1**. For details, see "Step 2: Changing Focus and Pinning Windows" on page 3-56.
2. Make **Group 1** active by clicking the left mouse button on the **Group 1** label.
3. Click the **Multiprocessor Step** button a few times and notice that only processor P0 advances.

   Because P0 is the only processor in the active group, only the P0 processor is affected by multiprocessor commands.
4. Make **Group 2** active by clicking the left mouse button on the **Group 2** label.
5. Click the **Multiprocessor Step** button a few times and notice that only processor P1 advances.

You have now completed this exercise and the tutorial.

What's Next

After completing the tutorial, you can begin building your own project or you can look at some of the VDK examples included with your VisualDSP++ software.

These example programs are in the following folder:

```
VisualDSP\21k\Examples\VDK Examples
```