

# 3 TUTORIAL

## In This Chapter

This chapter contains the following topics:

- [“Overview” on page 3-2](#)
- [“Exercise One: Building and Running a C Program” on page 3-4](#)
- [“Exercise Two: Modifying a C Program to Call an Assembly Routine” on page 3-17](#)
- [“Exercise Three: Plotting Data” on page 3-34](#)
- [“Exercise Four: Linear Profiling” on page 3-44](#)
- [“Exercise Five: Multiprocessor Debugging” on page 3-50](#)
- [“What’s Next” on page 3-70](#)

# 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 ADSP-21xx DSPs. The ADSP-219x simulator will be used for all exercises.

You can use different ADSP-21xx processors with only minor changes to the Linker Definition Files (.LDFs) included with each project.

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

```
Analog Devices\VisualDSP\21xx\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 Finite Impulse Response (FIR) algorithm.
- In Exercise Four, you will use linear profiling to examine the efficiency of the FIR algorithm used in Exercise Three. Using the collected linear profile data, you will pinpoint the most time-consuming areas of the algorithm, which are likely to require hand tuning in the assembly language.

- In Exercise Five, you will explore the multiprocessor debugging capabilities of VisualDSP++ (for ADSP-219x DSPs only), 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.

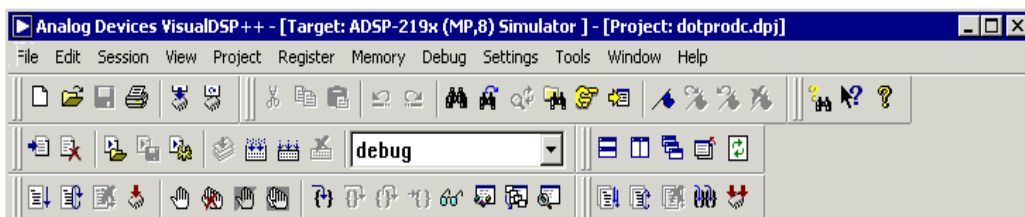


Figure 3-1. VisualDSP++ Toolbar Buttons

# 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 and profile a C function

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

`Analog Devices\VisualDSP\219x\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 21xx.

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\219x\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 displays messages in the Output window as it processes the project settings and file dependencies.

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.

## Exercise One: Building and Running a C Program

### 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 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 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".
cc219x: 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 3-2](#)).

The Editor window in [Figure 3-2](#) shows that the integer variable declaration `int` has been misspelled as `itn`.

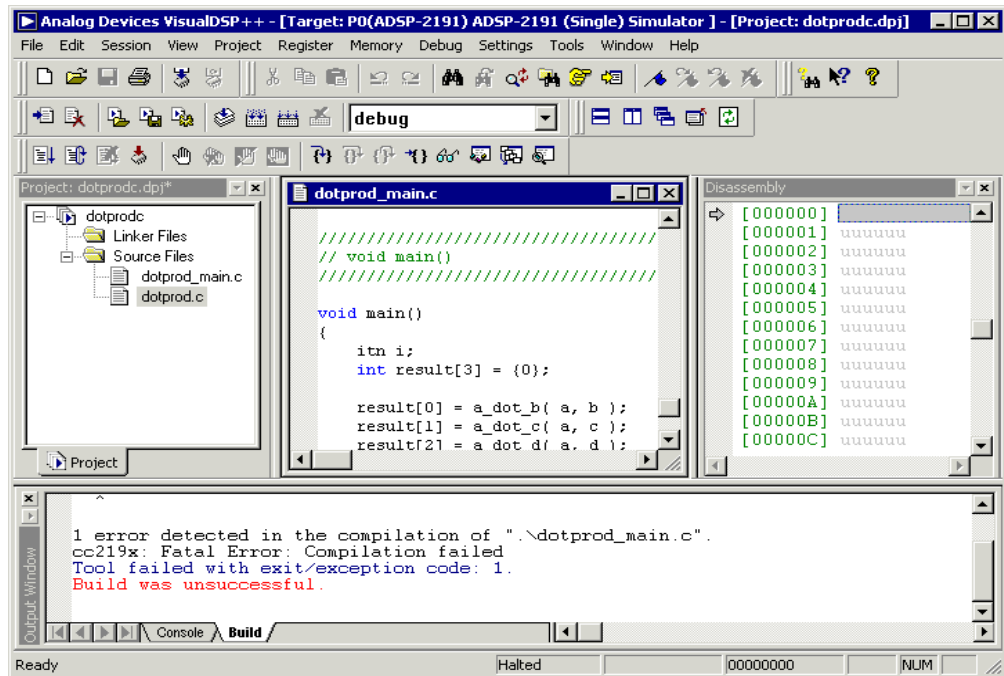


Figure 3-2. Output Window and Editor Window

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.

## Exercise One: Building and Running a C 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. Skip steps 1–3 if VisualDSP++ does not open the Session List dialog box.

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 3-3](#).

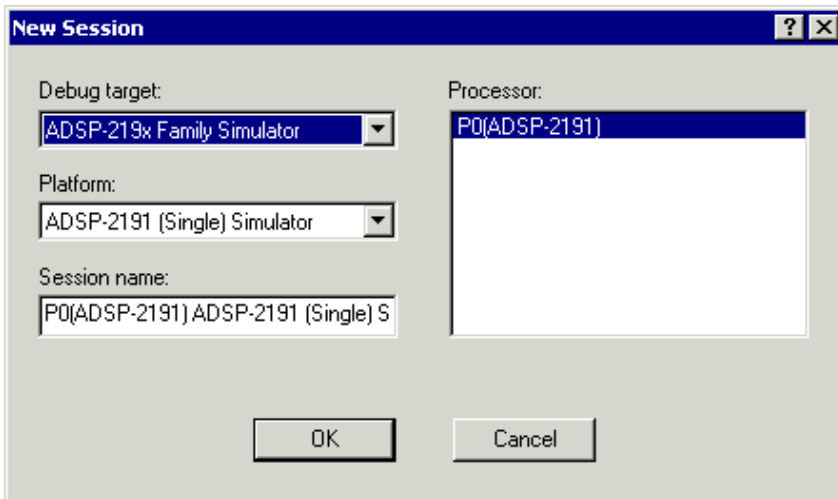


Figure 3-3. New Session Dialog Box

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-219x Family Simulator                 |
| Platform     | ADSP-2191 (Single) Simulator               |
| Session Name | PO(ADSP-2191) ADSP-2191 (Single) Simulator |
| Processor    | PO(ADSP-2191)                              |

3. Click OK to close the New Session dialog box. Then click OK to close the Session List dialog box.

VisualDSP++ automatically loads your project's executable file (dotprodc.dxe) and advances to the `main` function of your code (see [Figure 3-4](#)).

## Exercise One: Building and Running a C Program

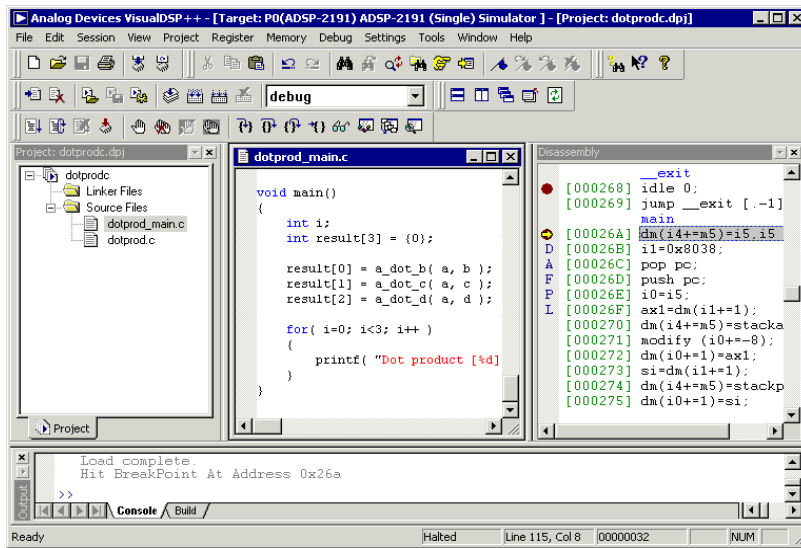


Figure 3-4. Disassembly, Editor, Output Windows: Load dotprod.dxe

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`.

Note: VisualDSP++ opens all the windows left open at the end of its previous session.

#### 4. Look at the information in the open windows.

The Output window contains messages about the status of the debug session. In this case, VisualDSP++ reports that the `dotprod.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 appears at address 0x268 and at address 0x26A. A yellow arrow appears at address 0x26A.

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.

5. From the Settings menu, choose Breakpoints to view the breakpoints set in your program. VisualDSP++ displays the Breakpoints dialog box, shown in [Figure 3-4](#).

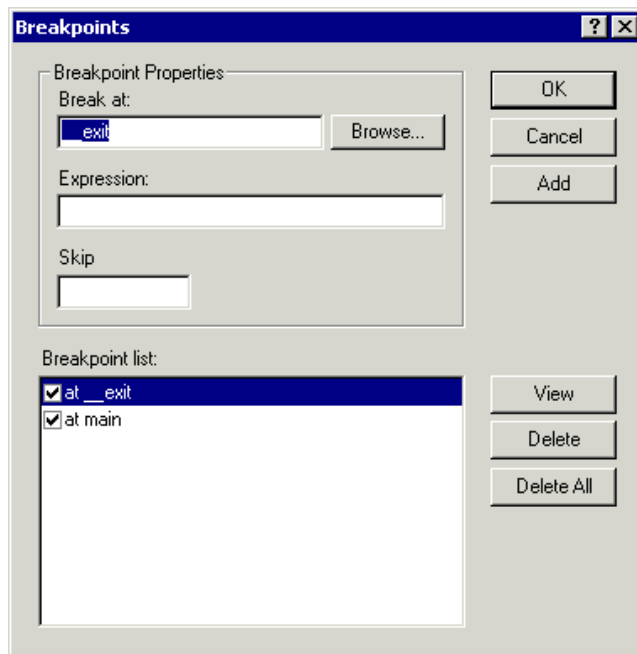


Figure 3-5. Breakpoints Dialog Box

The breakpoints are set at these C program labels:

- `at __exit`
- `at main`

## Exercise One: Building and Running a C Program

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

6. 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 3-6](#)) in the Output window:

```
Dot product [0] = -30213
Dot product [1] = -17646
Dot product [2] = 24326
```

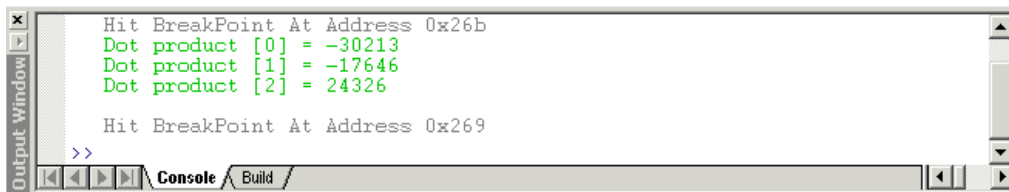


Figure 3-6. Results of the dotprodc Program

You are now ready to profile the `a_dot_c` function's performance.

## 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 3-7](#).

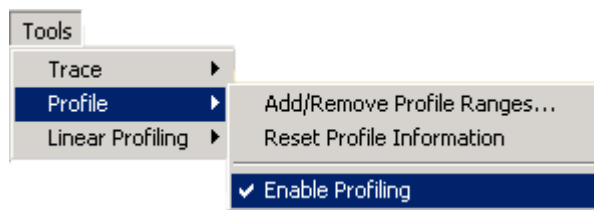


Figure 3-7. Tools Menu: Enable Profiling for `a_dot_c`

## Exercise One: Building and Running a C Program

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 3-8](#).

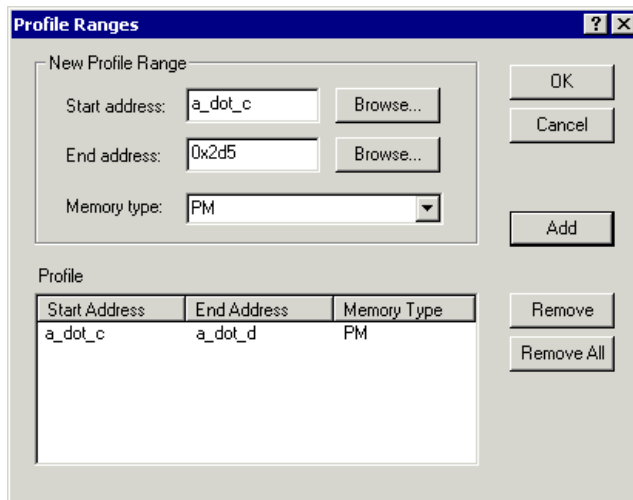


Figure 3-8. Profile Ranges Dialog Box

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 Program Symbols 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

Type `0x2d5`, the address of the last instruction in the function `a_dot_c`.

4. Click Add.

The profile defaults to Memory type PM (Program 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:

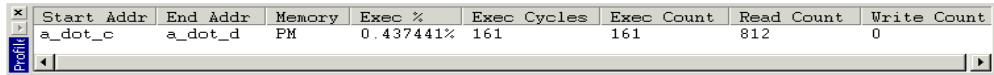
- a. Right-click on the Console tab page in the Output window.
- b. Choose Clear from the popup menu.

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.

## Exercise One: Building and Running a C Program

The Profile window (Figure 3-9) 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_d  | PM     | 0.437441% | 161         | 161        | 812        | 0           |

Figure 3-9. 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 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 3-10](#).

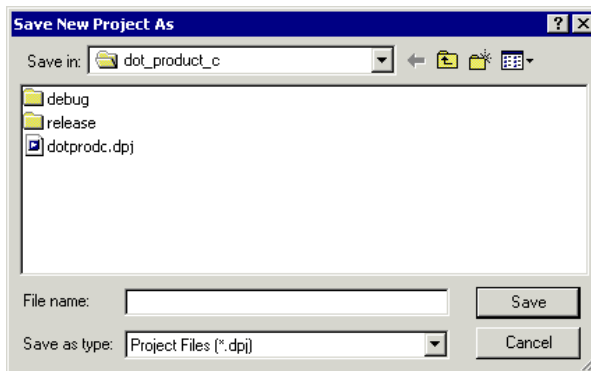


Figure 3-10. Save New Project As Dialog Box

## Exercise Two: Modifying a C Program to Call 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 3-11) appears.

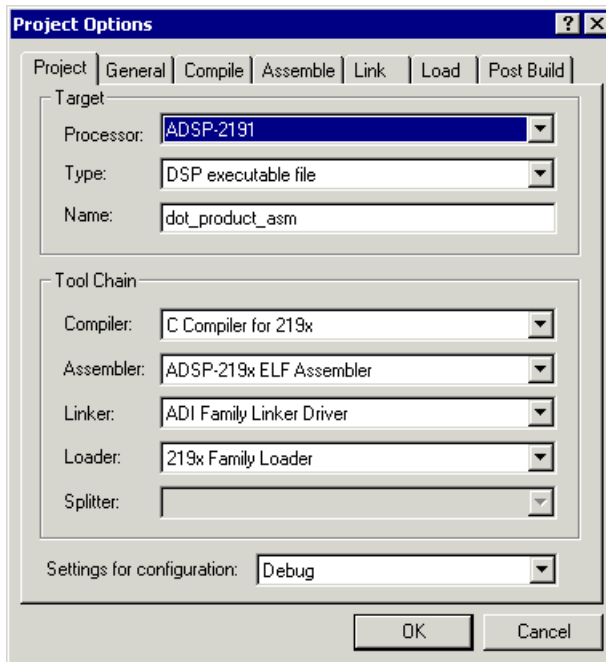


Figure 3-11. Project Options Dialog Box: Project Tab Page

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, 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 3-11](#)), specify the following values:

| Box                        | Value               |
|----------------------------|---------------------|
| Processor                  | ADSP-2191           |
| Type                       | DSP executable file |
| Name                       | dot_product_asm     |
| Settings for configuration | Debug               |

These settings specify information for building an executable file for the ADSP-2191. 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 3-12](#).

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

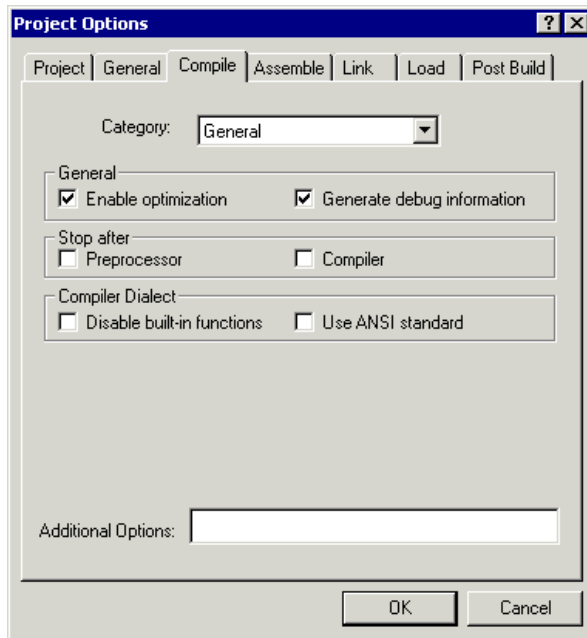


Figure 3-12. Project Options Dialog Box: Compile Tab Page

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-2191 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 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 3-13](#)) appears. The four files are added to the project.

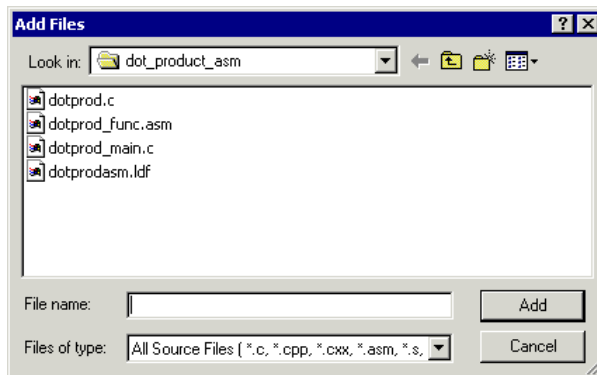


Figure 3-13. Add Files Dialog Box: Adding Source Files to the Project

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. Press and hold down the Ctrl key and click on dotprod.c, dotprod\_func.asm, dotprod\_main.c, and dotprodasm.ldf. Then click Add.

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

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 3-14](#).

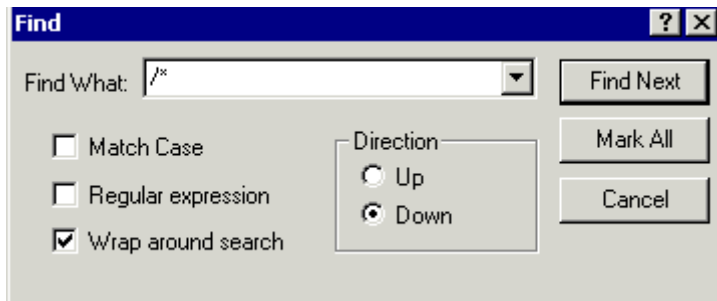


Figure 3-14. Find Dialog Box: Locating All Occurrences of `/*`

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 3-15 shows the changes made in step 6.

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

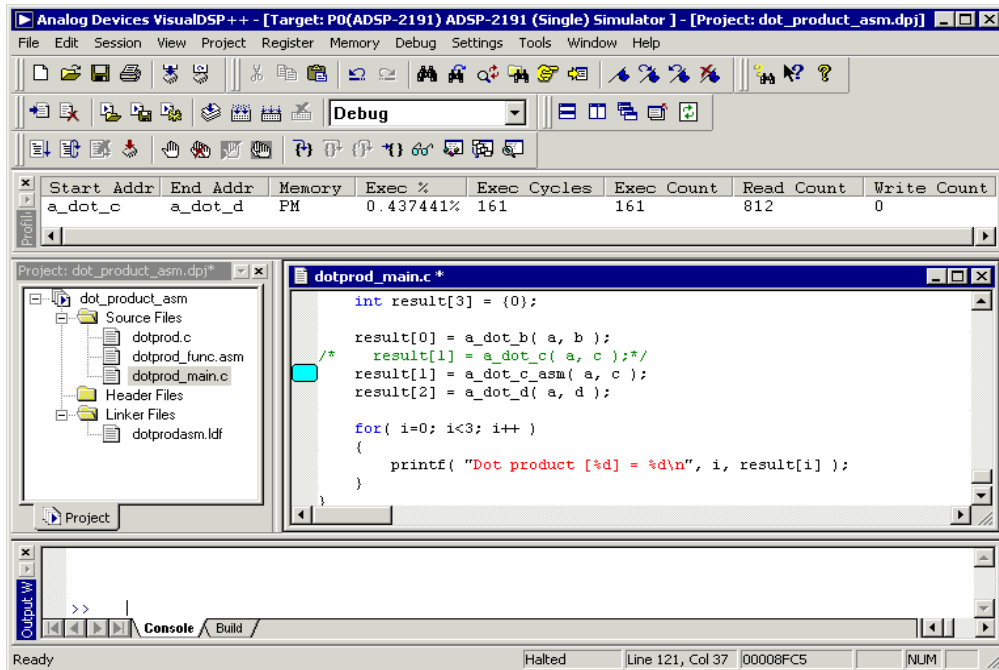


Figure 3-15. Editor Window: Modifying dotprod\_main.c to Call a\_dot\_c\_asm

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-2191 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.

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

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 3-16](#).

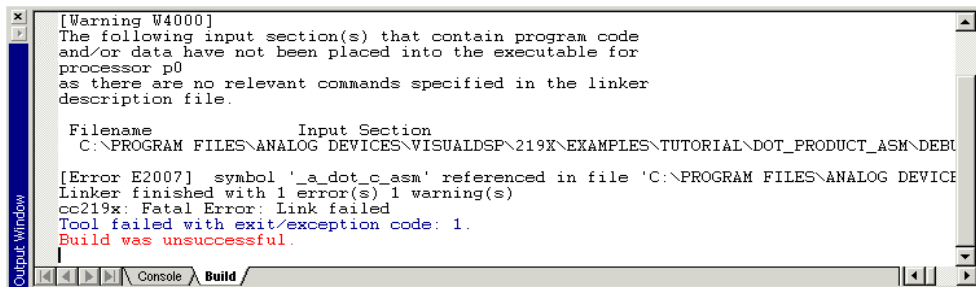


Figure 3-16. Output Window: Linker Error

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. In the `PROCESSOR` section, change the `SECTIONS` command as follows:

| Change this code...                                                                                                      | To this code...                                                                                                                                                                |
|--------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>sec_code {     INPUT_SECTIONS ( \$OBJECTS(program)                      \$LIBRARIES(program) ) } &gt;mem_code</pre> | <pre>sec_code {     INPUT_SECTIONS ( \$OBJECTS(program)                      \$LIBRARIES(program)                      dotprod_func.doj(my_asm_section) ) } &gt;mem_code</pre> |

When you re-link the project, `my_asm_section` will be placed in the `mem_code` memory segment.

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

As shown in [Figure 3-17](#), your project `dot_product_asm.dpj` should now contain the files `dotprod.c`, `dotprod_main.c`, `dotprod_func.asm`, and `dotprodasm.ldf`.

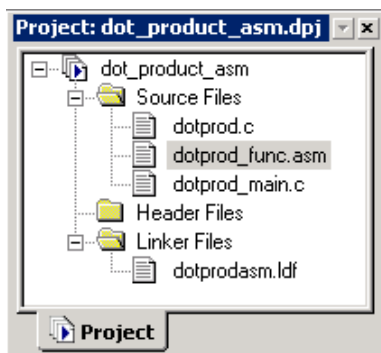


Figure 3-17. Files in the `dot_product_asm` Project

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

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

### 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 Build tab page:

```
"Build completed successfully."
```

VisualDSP++ loads the program, runs to main, and displays the Output, Disassembly, and Profile windows (shown in [Figure 3-18](#)).

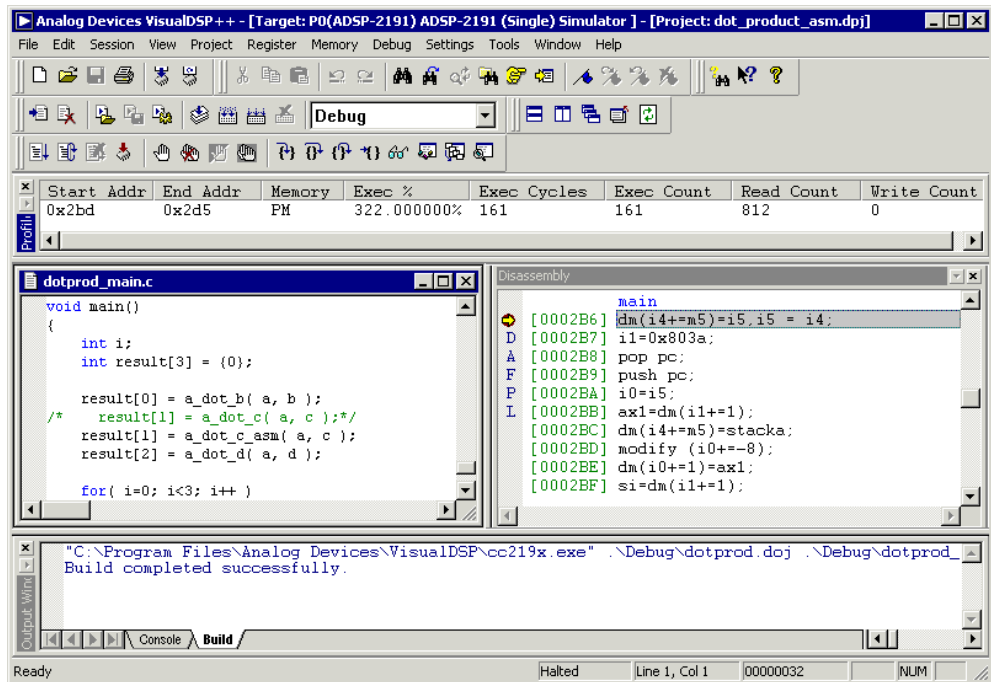


Figure 3-18. Windows Left Open at the End of the Previous Debugger Session

- 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] = -30213
Dot product [1] = -17646
Dot product [2] = 24326

```

You are now ready to profile the a\_dot\_c\_asm assembly function.

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

### Step 6: Set up the Profile of a\_dot\_c\_asm

In this procedure, you will profile the `a_dot_c_asm` function as follows:

- Enable profiling to collect execution information during the next run of the program
- Define the start address and end address of the code segment to be profiled

To profile the `a_dot_c_asm` function:

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



Figure 3-19. Tools Menu: Enable Profiling for `a_dot_c`

2. From the Tools menu, choose Profile and then choose Add/Remove Profile Ranges.

The Profile Ranges dialog box ([Figure 3-20](#)) appears.

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

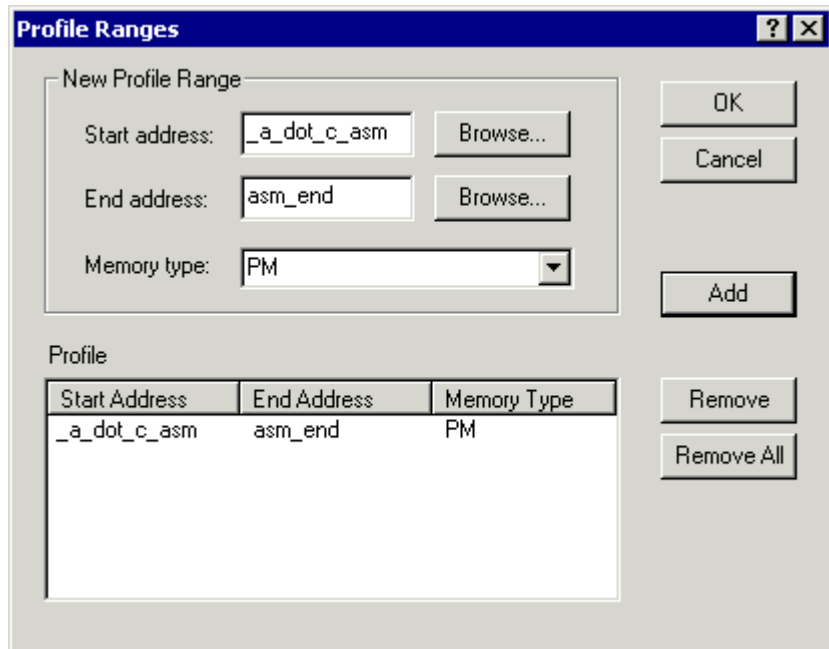


Figure 3-20. Profile Ranges Dialog Box: Set the Start and End Addresses for Profiling the `a_dot_c_asm` Function

4. In the Start Address box, type `_a_dot_c_asm`, the starting address of the `a_dot_c_asm` function. You can also use the Browse button to locate and enter this label.
5. In the End Address box, type (or browse for) `asm_end`, the address of the instruction that returns to `main()`.
6. Click Add to add this range to the Profile list, and then click OK to exit the Profile Ranges dialog box.

The Profile window contains the profile range that you just defined.

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

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

### Step 7: Run dot\_product\_asm

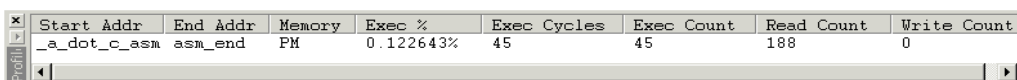
To run `dot_product`:

1. Click the Restart button , or from the Debug menu, choose Restart.
2. From the Tools menu, choose Profile, and then choose Reset Profile Information to reset the profile information collected in a previous run.
3. Click the Run button  (or press F5) to run the program.

The program halts at the first executable instruction in the `main` function.

4. Right-click on the Console tab page and choose Clear from the popup menu to clear the text on the Console tab page.
5. Press F5 to continue running the program.

After computing the dot products, the program writes the results to the Console tab page in the Output window. The program halts at `__lib_prog_term` and displays the profile results in the Profile window, as shown in [Figure 3-21](#).



| Start Addr   | End Addr | Memory | Exec %    | Exec Cycles | Exec Count | Read Count | Write Count |
|--------------|----------|--------|-----------|-------------|------------|------------|-------------|
| _a_dot_c_asm | asm_end  | PM     | 0.122643% | 45          | 45         | 188        | 0           |

Figure 3-21. Results of the Profile of the Assembly Language Function `a_dot_c_asm`.

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      | 0.437441% | 161         | 161        | 812        | 0           |
| _a_dot_c_asm | 0.122643% | 45          | 45         | 188        | 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 Finite Impulse Response (FIR) filter 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 FIR Program

To load the FIR program:

1. Keep the Disassembly window and Console tab page (in the Output window) open, but close all other windows (such as the Profile window) 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 FIR program to load as follows:
  - a. Open your Analog Devices folder and double-click the following sub-folders in succession:  
`VisualDSP\219x\Examples\Tutorial\fir\debug`
  - b. Double-click `fir.dxe` to load the program in an Editor window ([Figure 3-22](#)).

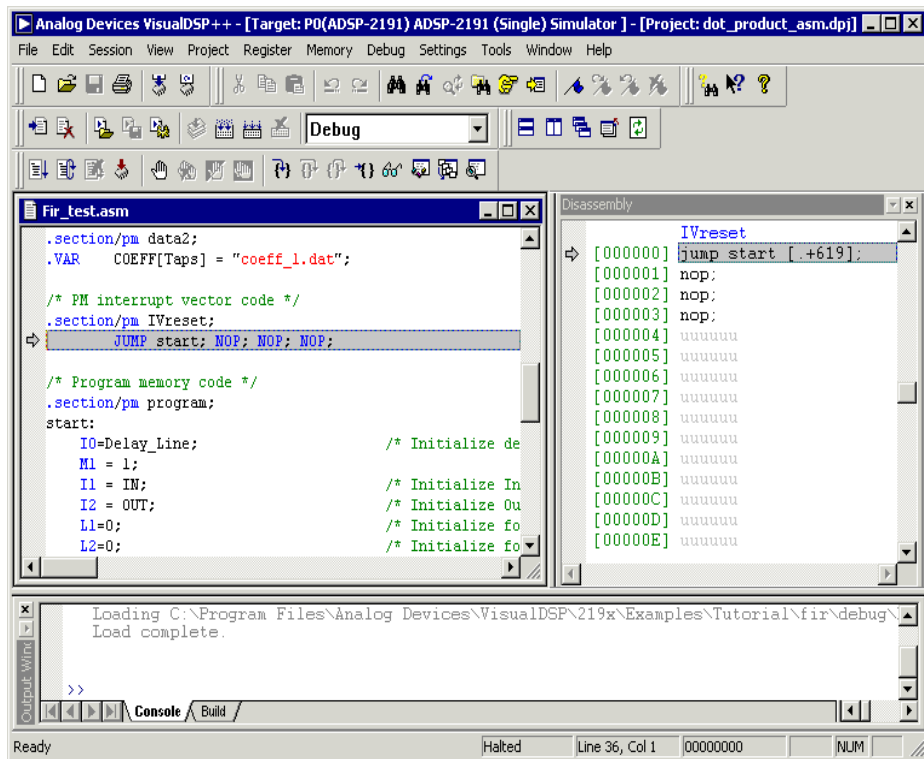


Figure 3-22. Loading the FIR Program

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

You can see two global data arrays:

- IN
- OUT

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

You are now ready to open a Plot window.

## Exercise Three: Plotting Data

### 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 3-23](#).

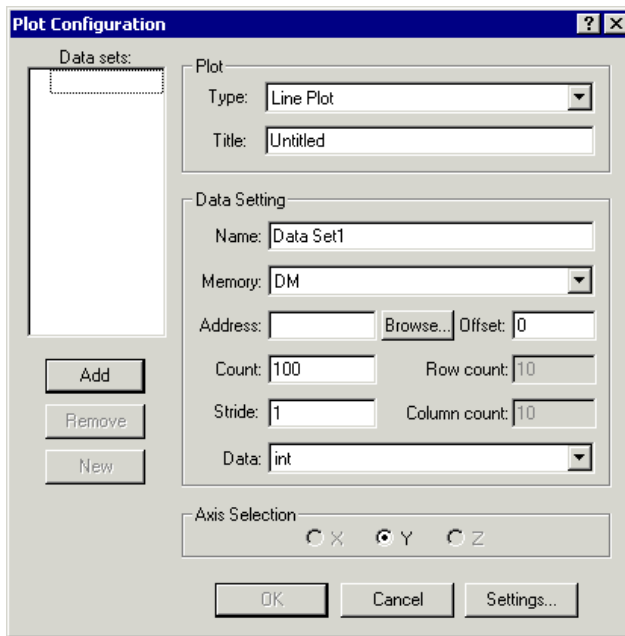


Figure 3-23. Plot Configuration Dialog Box: Specifying Data Sets to Be Plotted

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 fir.
3. Enter two 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 3-24](#).

Table 3-2. Two Data Sets: Input and Output

| Box     | Input Data Set | Output Data Set | Description                                                                                                                                 |
|---------|----------------|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| Name    | Input          | Output          | Data set                                                                                                                                    |
| Memory  | DM             | DM              | Data memory                                                                                                                                 |
| Address | IN             | OUT             | The address of this data set is that of the Input or Output array.<br><br>Click Browse to select the value from the list of loaded symbols. |
| Count   | 128            | 128             | The array is 512 elements long, but you are plotting the first 128 elements.                                                                |
| Stride  | 1              | 1               | The data is contiguous in memory.                                                                                                           |
| Data    | int            | int             | Input and Output are arrays of int values.                                                                                                  |

## Exercise Three: Plotting Data

The Plot Configuration dialog box should now look like this:

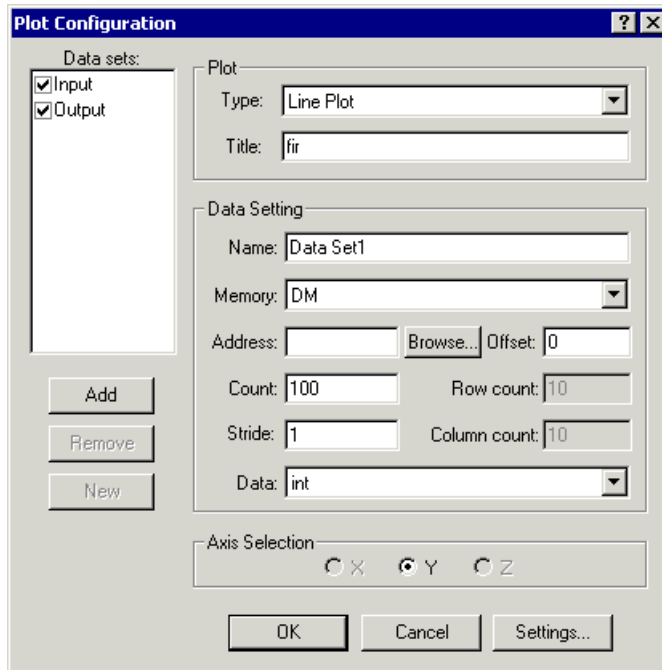


Figure 3-24. Plot Configuration Dialog Box: Entering the Input and Output Data Sets

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

The Plot window now displays the two arrays. Since, by default, the Simulator initializes memory to zero, the Output data set appears as one horizontal line, shown in [Figure 3-25](#).

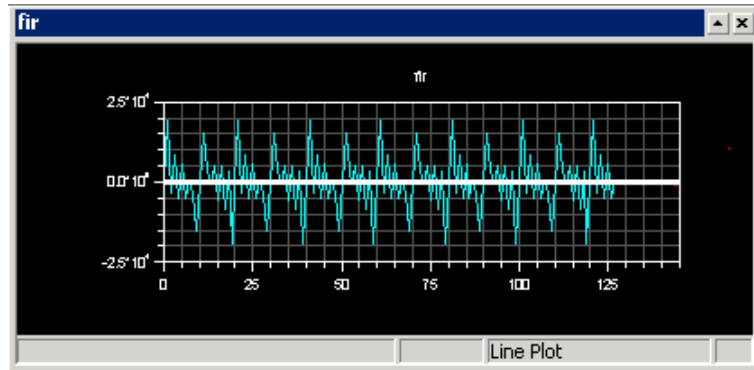


Figure 3-25. Plot Window: Before Running the FIR Program

### Step 3: Run the FIR Program and View the Data

To run the FIR program and view the data:

1. From the Settings menu, choose Breakpoints to open the Breakpoints dialog box ([Figure 3-5 on page 3-11](#)). Then set two breakpoints as follows:
  - a. In the Break at box, type or browse for `per_bin` and click OK. Then click Add. A breakpoint at “`Fir_test.asm`” .71 appears in the Breakpoint list box.
  - b. In the Break at box, type or browse for `looping` and click OK. Then click Add. A breakpoint at “`Fir_test.asm`” .73 appears in the Breakpoint list box.
  - c. Click OK to exit the dialog box.
2. Perform one of these actions to run to the breakpoint at `per_bin`:
  - Click the Run button .
  - From the Debug menu, choose Run.

## Exercise Three: Plotting Data

Once the program finishes running to `per_bin`, the message “Halted” appears in the status bar at the bottom of the screen. The Plot window should now show the partial output of the FIR filter. Run the program a few more times to the breakpoint at `per_bin` to see the output generated.

3. From the Settings menu, choose Breakpoints to open the Breakpoints dialog box (Figure 3-5 on page 3-11). Then select the breakpoint at “`Fir_test.asm`” .71 (`per_bin`) and click Delete. Click OK to exit the dialog box.
4. Press F5 or click the Run button  to run to the end of the program. When the program halts, you see the results of the FIR filter in the Output array. The two data sets are now visible in the Plot window, as shown in Figure 3-26.

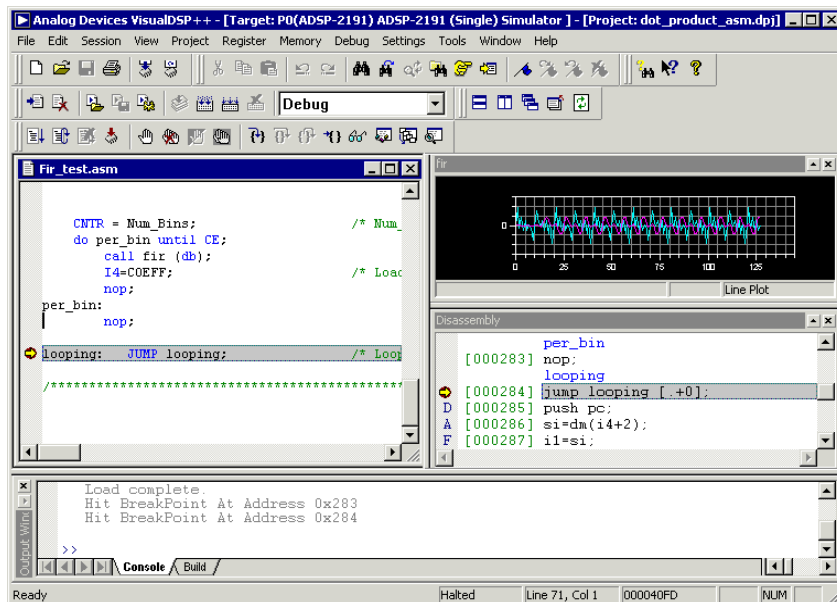


Figure 3-26. Plot Window: After Running the FIR Program to Completion

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

5. 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 3-27](#) shows the selected region, and [Figure 3-28](#) shows the magnified result.

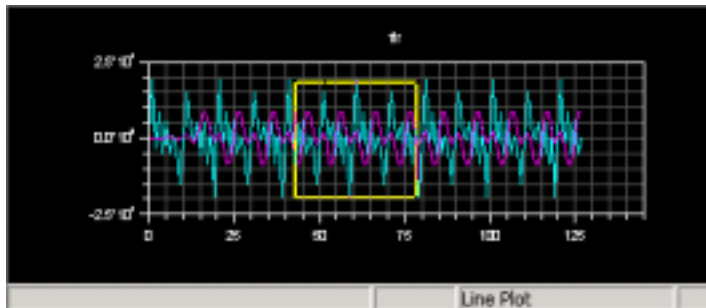


Figure 3-27. Plot Window: Selecting a Region to Magnify

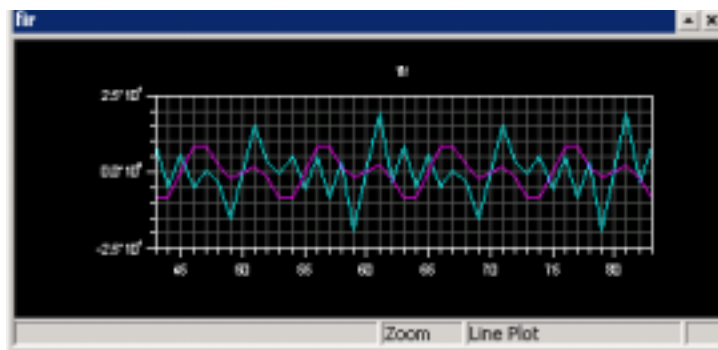


Figure 3-28. Plot Window: Magnified Result

## Exercise Three: Plotting Data

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 Plot window by enabling the data cursor, as explained in the next step.

6. Right-click inside the Plot window and choose **Data Cursor** from the popup menu. Then move through the individual data points in the current data set by pressing and holding the Left ( $\leftarrow$ ) and Right ( $\rightarrow$ ) 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 3-29](#).

To switch data sets, press the Up ( $\uparrow$ ) and Down ( $\downarrow$ ) arrow keys.

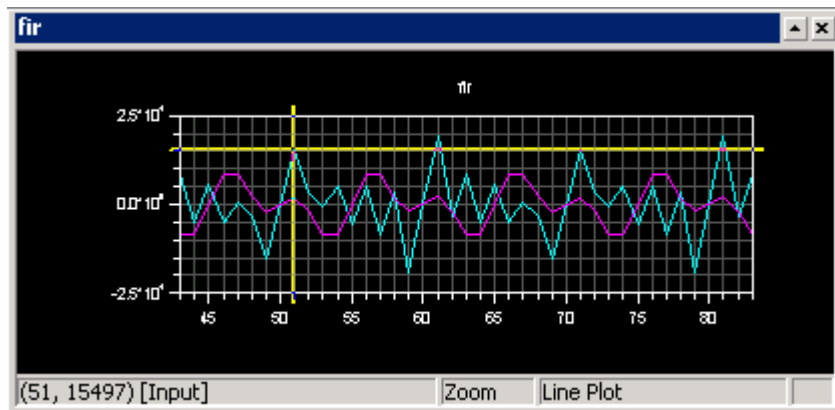


Figure 3-29. Plot Window: Viewing Individual Data Points by Using the Data Cursor Feature

7. Right-click in the Plot window and choose **Data Cursor** from the popup menu.

In the next step, you will look at data sets in the frequency domain.

8. Right-click in the Plot window and choose **Modify Settings** to open the Plot Settings dialog box. Then complete these steps:
  - a. Click the **Data Processing** tab.
  - b. In the **Data Sets** box, make sure that **Input** (the default) is selected. In the **Data Process** box, choose **FFT Magnitude**.
  - c. In the **Data Sets** box, select **Output**. In the **Data Process** box, choose **FFT Magnitude**.
  - d. Click **OK** to exit the Data Processing tab page.

VisualDSP++ performs a Fast Fourier Transform (FFT) on the selected data set before it is plotted. FFT enables you to view the signal in the frequency domain, as shown in [Figure 3-30](#).

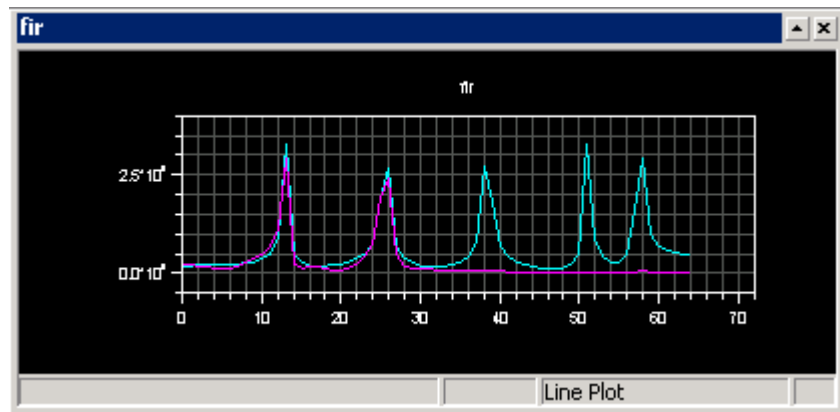


Figure 3-30. FFT Performed on a Selected Data Set

You are now ready to start Exercise Four.

# Exercise Four: Linear Profiling

In this exercise, you will load and debug the FIR program from the previous exercise. You will 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.

**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 FIR Program

To load the FIR program:

1. Close all open windows except for the Disassembly window and the Output window.

2. From the File menu, choose Load Program, or click  .

The Open a Processor Program dialog box appears.

3. Select the program to load as follows:
  - a. Open the Analog Devices folder and double-click the following sub-folders in succession:  
`VisualDSP\219x\Examples\Tutorial\fir\debug`
  - b. Double-click `fir.dxe` to load and run the FIR program.  
VisualDSP++ opens an Editor window.

You are now ready to enable linear profiling.

## Step 2: Enable Linear Profiling

To enable linear profiling:

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

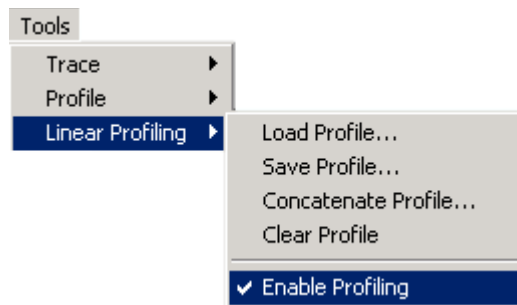


Figure 3-31. Enabling Linear Profiling for the FIR Program

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

Linear profiling will be performed when you next run the FIR program. By default, linear profiling is disabled.

2. From the View menu, choose Debug Windows. Then choose Linear Profiling Results to open the Linear Profiling Results window, shown in [Figure 3-32](#).

## Exercise Four: Linear Profiling

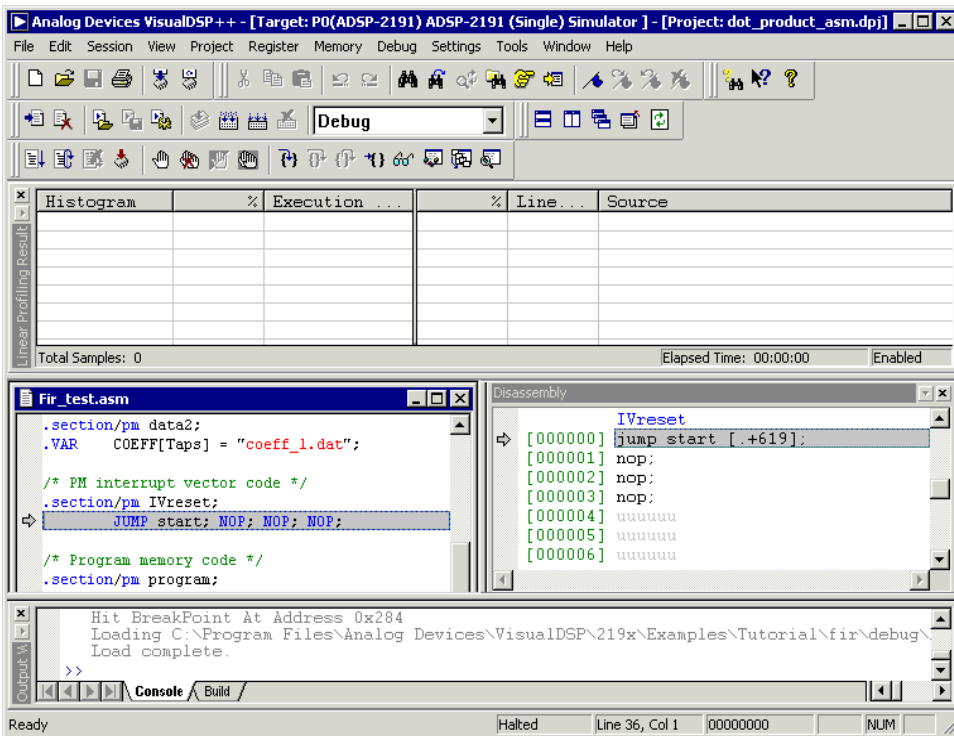


Figure 3-32. Linear Profiling Results Window (Empty)

The Linear Profiling Results window is initially 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 linear profile data.

## Step 3: Collect and Examine the Linear Profile Data

To collect and examine the linear profile data:

1. Set a breakpoint at the `looping` label if one is not set.
2. 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.

3. Examine the results of your linear profiling session.

The Linear Profiling Results window is divided into two, three-column panes. The left pane shows the results of the profile data. Double-clicking on a line in the left pane displays the corresponding source code for the profile data in the right pane, as shown in [Figure 3-33](#).

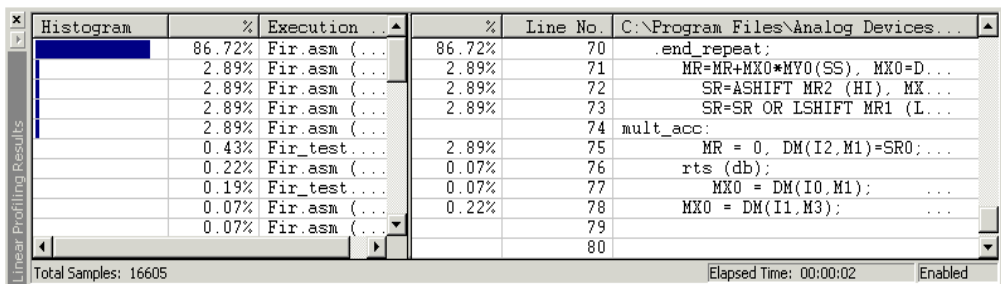


Figure 3-33. Linear Profiling Results of Analyzing the Performance of the FIR Program

## Exercise Four: Linear Profiling

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

### Histogram

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

### %

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

### Execution Unit

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

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

The left pane shows that one line of fir.asm has consumed over 86% of the total execution time. If you double-click this line, the right pane displays the line (70) in the source file, fir.asm. The source pane displays data for each line of executable code in the file for which linear profile data has been collected.

[Figure 3-34](#) shows the linear profile data for the `fir` function.

| %      | Line No. | C:\Program Files\Analog Devices...          |
|--------|----------|---------------------------------------------|
|        | 62       | <code>fir:</code>                           |
| 0.07%  | 63       | <code>MR = 0, MX0 = DM(I1,M1), MY...</code> |
| 0.07%  | 64       | <code>DM(I0,M3) = MX0; ...</code>           |
|        | 65       |                                             |
| 0.07%  | 66       | <code>CNTR=Samps_per_Bin; ...</code>        |
| 0.07%  | 67       | <code>DO mult_acc UNTIL CE;</code>          |
|        | 68       | <code>.repeat (Taps-1); ...</code>          |
|        | 69       | <code>MR=MR+MX0*MY0(SS), MX0=D...</code>    |
| 86.72% | 70       | <code>.end_repeat;</code>                   |
| 2.89%  | 71       | <code>MR=MR+MX0*MY0(SS), MX0=D...</code>    |
| 2.89%  | 72       | <code>SR=ASHIFT MR2 (HI), MX...</code>      |
| 2.89%  | 73       | <code>SR=SR OR LSHIFT MR1 (L...</code>      |

Figure 3-34. Linear Profile Data for the `fir` Function

The details of the `fir` function show that over 86% of the time spent running the entire FIR program is spent inside the `repeat` statement, which is part of the `DO` loop.

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 3-35](#).

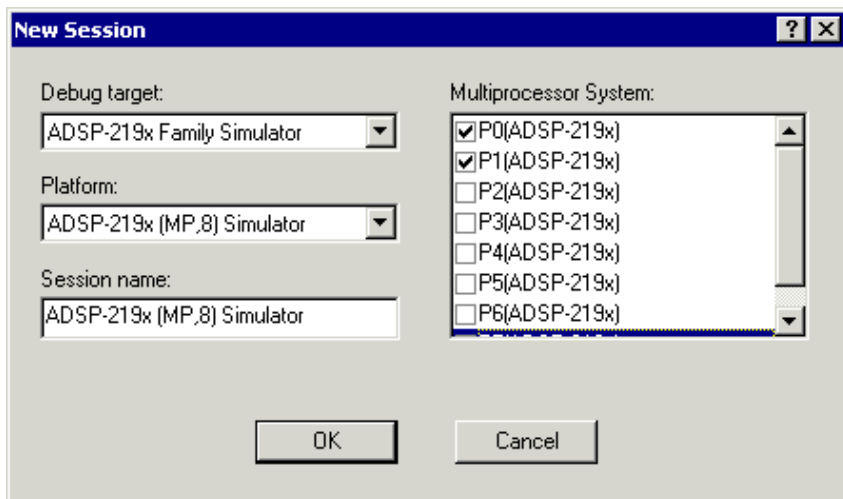


Figure 3-35. New Session Dialog Box: Setting up a Multiprocessor Simulator Session

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

| Box          | Value                                              |
|--------------|----------------------------------------------------|
| Debug Target | ADSP-219x Family Simulator                         |
| Platform     | ADSP-219x [MP,8] Simulator                         |
| Session Name | ADSP-219x [MP,8] Simulator Multiprocessor Tutorial |

4. Under **Multiprocessor System**, select the check boxes next to processors P0 and P1, as shown in [Figure 3-35](#).

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 3-36](#) shows the windows in the new multiprocessor session.

## Exercise Five: Multiprocessor Debugging

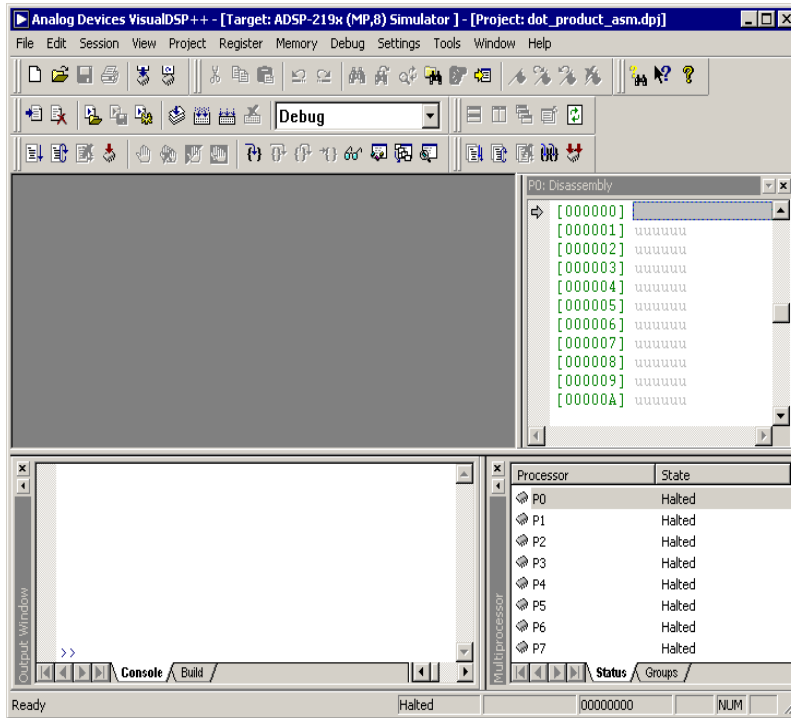


Figure 3-36. VisualDSP++ Windows: After Creating a Multiprocessor Session

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

The Multiprocessor window appears in the lower-right corner of the VisualDSP++ main window, as shown in [Figure 3-37](#).

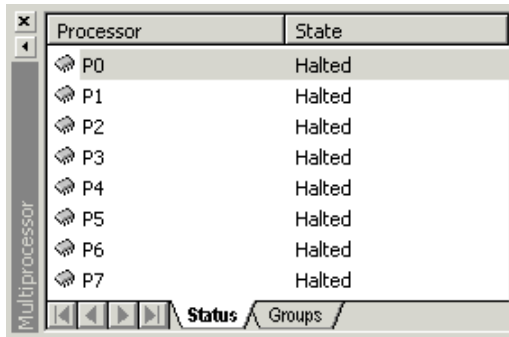


Figure 3-37. Multiprocessor Window: Status Tab Page

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 3-38](#) indicates that information was collected from processor P0.

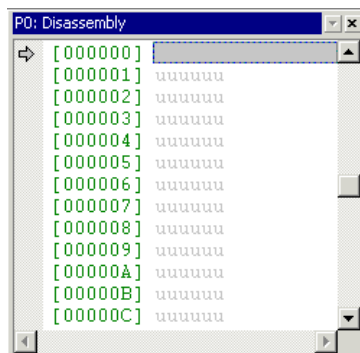


Figure 3-38. Disassembly Window: Disassembled Instructions in Processor P0

## Exercise Five: Multiprocessor Debugging

Figure 3-39 shows the Multiprocessor toolbar.



Figure 3-39. 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 3-37), 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 3-37), 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 000007 in the Disassembly window to highlight the field in gray.
3. Type the instruction `ax0=0x1234` and press Enter. The instruction is written into P1's memory, and the Disassembly window is updated as shown in Figure 3-40 to reflect this new instruction.

## Exercise Five: Multiprocessor Debugging

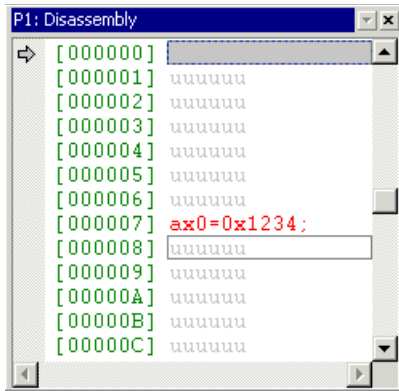


Figure 3-40. Disassembly Window: Patching a New Instruction into Processor P1's Memory

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 3-41](#).

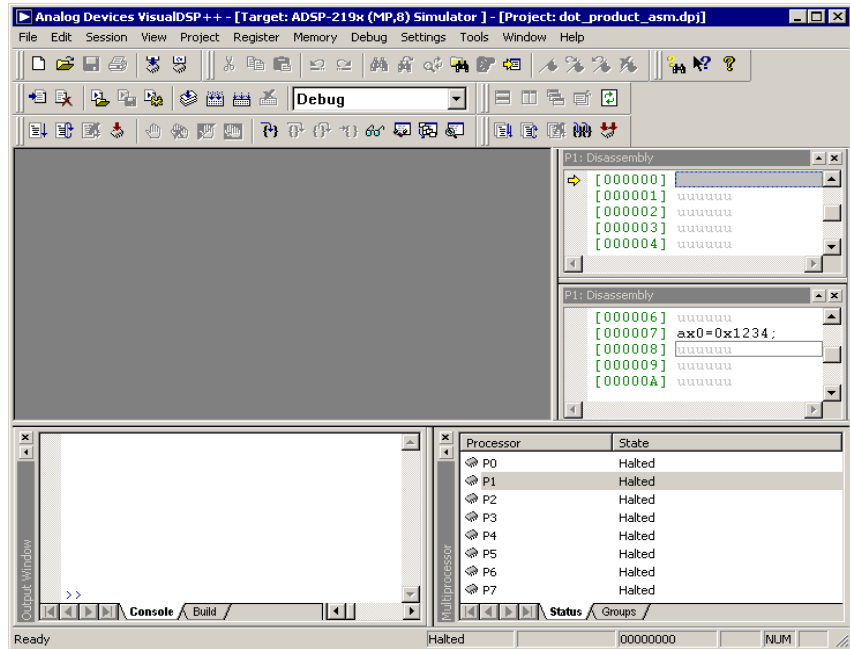


Figure 3-41. Two Open Disassembly Windows Focused on Processor P1

5. Select different processors from the Multiprocessor window's Status tab page ([Figure 3-37](#)) and notice that both Disassembly windows change focus to the currently selected processor. Also note that the `ax0=0x1234;` instruction is present only when the Disassembly windows are focused on P1.
6. Select processor P0.
7. Right-click the mouse in the top Disassembly window and choose Pin to Processor from the popup menu, shown in [Figure 3-42](#).

## Exercise Five: Multiprocessor Debugging

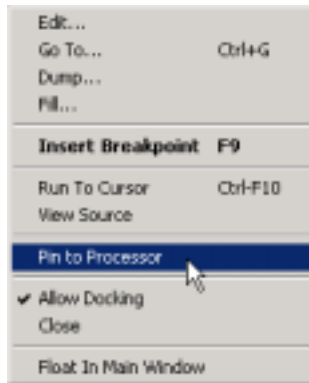


Figure 3-42. Selecting Pin to Processor

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 3-37](#)), 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 3-43](#) shows the top (P0) and bottom (P1) Disassembly windows.

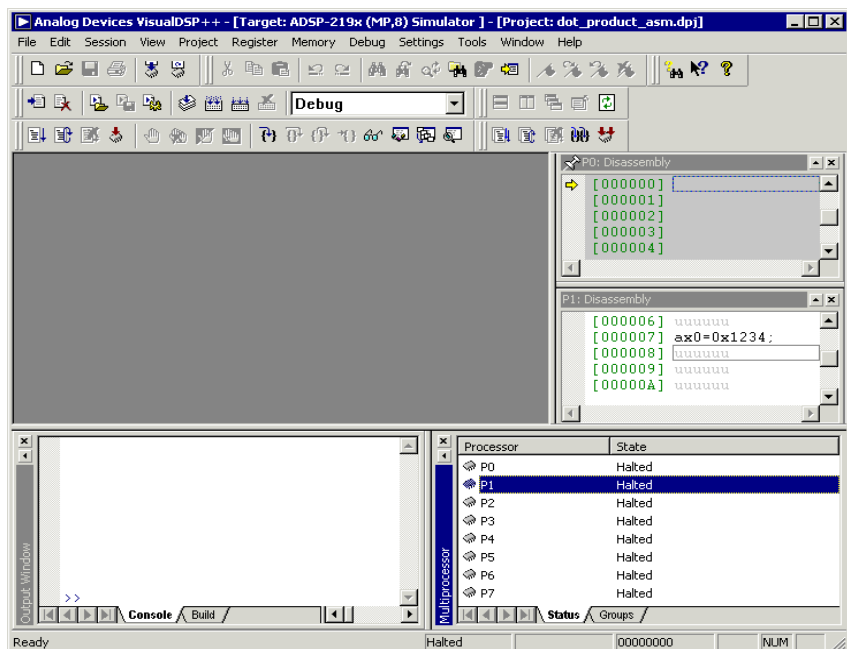


Figure 3-43. Pinning a Disassembly Window

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.

## Exercise Five: Multiprocessor Debugging

### 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 3-37](#)), select P0 to set focus to processor P0.
2. Click the Load Program button , or from the File menu, choose Load Program. The Open a Processor Program dialog box appears.
3. Open your Analog Devices folder and double-click the following sub-folders in succession:  
`VisualDSP\219x\examples\tutorial\dot_product_c\debug`
4. Double-click `dotprodc.dxe` to open the Load Multiprocessor Confirmation dialog box, shown in [Figure 3-44](#).

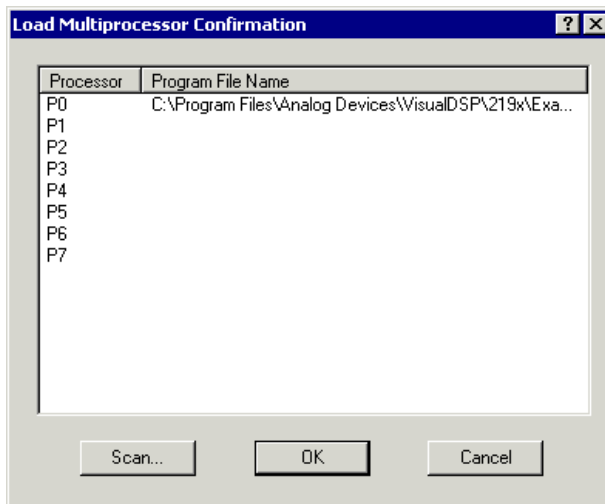


Figure 3-44. Load Multiprocessor Confirmation Dialog Box: Specifying Load `dotprodc.dxe` into Processor P0

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 3-45](#)).

## Exercise Five: Multiprocessor Debugging

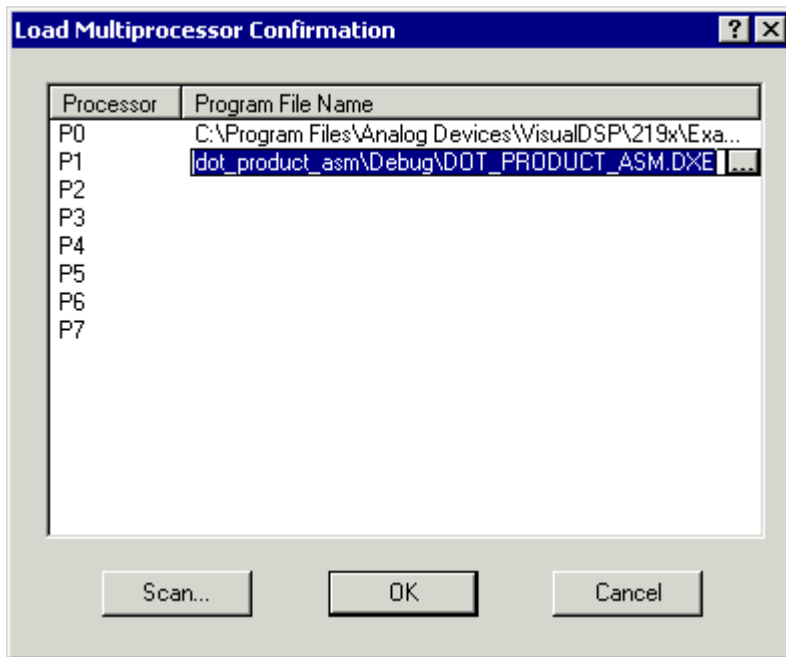


Figure 3-45. Load Multiprocessor Confirmation Dialog Box:  
Specifying Load dot\_product\_asm.dxe into Processor P1

d. Click OK to load the programs.

After you load each program into its respective processor, both processors run to the first executable line in `main()` and open the source files for each program.

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 3-46 shows the Editor windows tiled vertically.

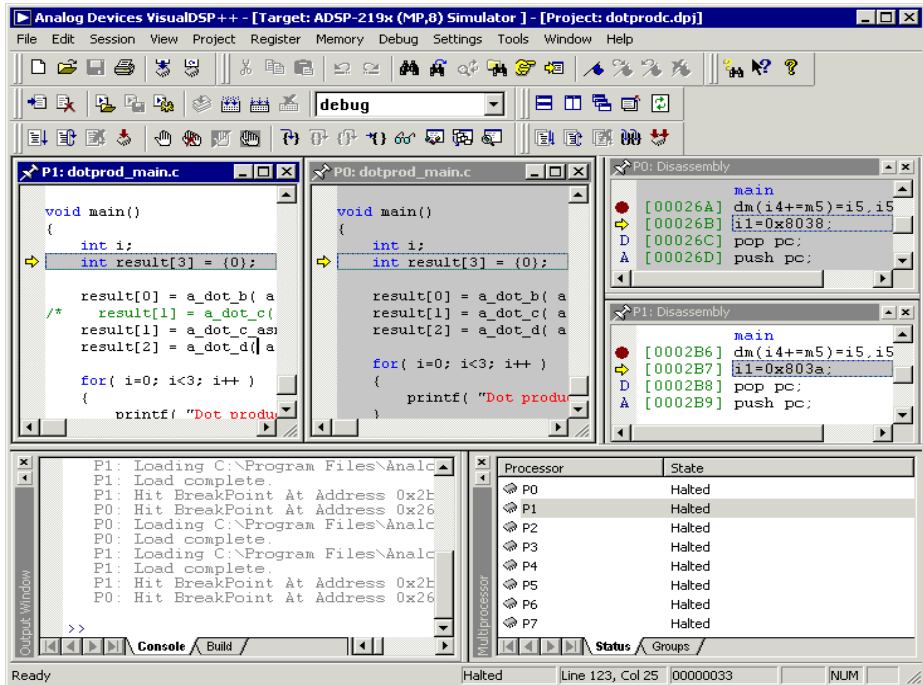


Figure 3-46. Two Programs Loaded onto Two Separate Processors

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

- 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. This feature facilitates navigation between processors.

## Exercise Five: Multiprocessor Debugging

### 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 00026D, and the P1 Disassembly window is still at address 0002B7.

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 00026F, and P1 is at address 0002B9.

## Step 5: Configuring and Using Multiprocessor Groups

Often, in large multiprocessor systems, individual processors are grouped into a “cluster” 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 3-50.](#)) Up to this point, you have used only two of the eight processors in the multiprocessor session. The remaining processors, P2–P7, have not been used.

In this step you will complete the following tasks:

- Add processors P2–P7 to the Default group and issue an MP Reset command to all eight processors
- Create new groups and add processors to them
- Make different groups active

## Exercise Five: Multiprocessor Debugging

### 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 3-47](#).

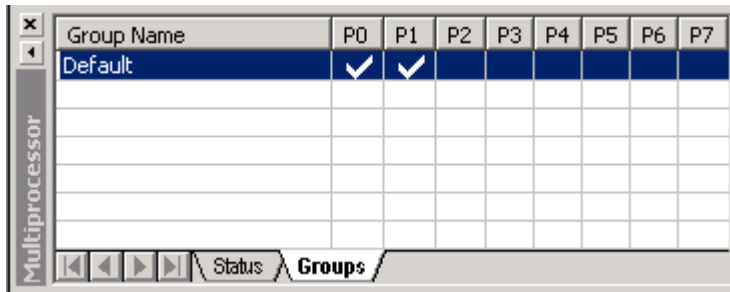


Figure 3-47. Multiprocessor Window: Groups Tab Page

Notice that processors P0 and P1 are checked to show that they are in the Default group, and processors P2–P7 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 3-48](#).

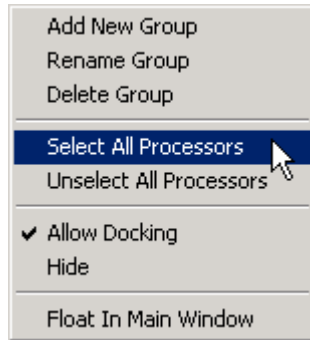


Figure 3-48. Moving All Processors into the Default Group

In the Multiprocessor Group window, a check mark appears for all eight processors in the Default group (see [Figure 3-49](#)).

4. Issue an **MP Reset** command to all eight processors by performing one of these actions:
  - Click the Multiprocessor Reset button .
  - From the **Debug** menu, choose **Multiprocessor** and then choose **Reset**.

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

## Exercise Five: Multiprocessor Debugging

### 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 3-49](#). You can change the group's name. For this exercise, however, you will use the default.

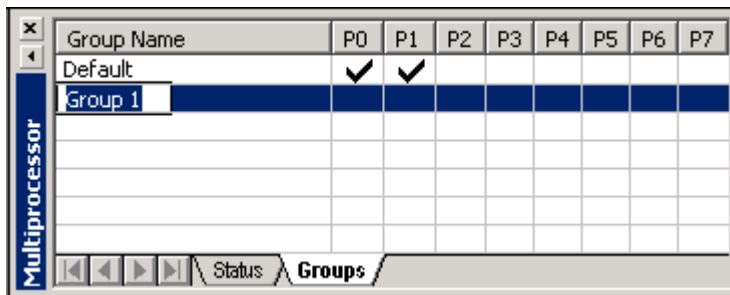


Figure 3-49. Multiprocessor Window, Groups Tab Page: Creating a New Group

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 3-50](#) shows the two new multiprocessor groups.

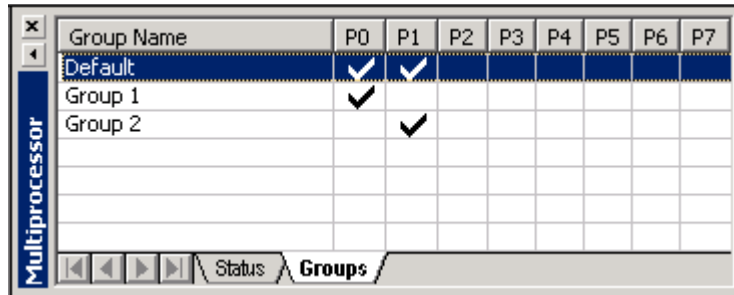


Figure 3-50. Multiprocessor Window, Groups Tab Page: Creating Two New Groups

### Activating New Multiprocessor Groups

To make the new groups active:

1. Make **Group 1** active by clicking the left mouse button on the **Group 1** label.
2. 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.

3. Make **Group 2** active by clicking the left mouse button on the **Group 2** label.
4. 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 other examples included with your VisualDSP++ software.

These sample programs are in the following folder:

VisualDSP\219x\Examples\Simulator Peripheral Examples

The sample programs are documented in the appendix “2191 Peripherals Used with the Simulator” in the *VisualDSP++ 2.0 User's Guide for ADSP-21xx DSPs*.