Altera Monitor Program

This tutorial presents an introduction to the Altera Monitor Program, which can be used to compile, assemble, download and debug programs for Altera's Nios II processor. The tutorial gives step-by-step instructions that illustrate the features of the Altera Monitor Program.

The process of downloading and debugging a Nios II application requires the presence of an actual FPGA device to implement the Nios II processor. For the purposes of this tutorial, it is assumed that the user has access to the Altera DE2 Development and Education board connected to a computer that has Quartus II (version 9.0 or higher) and Nios II EDS software installed.

The screen captures in the tutorial were obtained using version 4.1 of the Altera Monitor Program; if other versions of the software are used, some of the images may be slightly different.

Who should use the Altera Monitor Program

The Altera Monitor Program is intended to be used in an educational environment by professors and students. For commercial system and application development, Altera's *Nios II Integrated Development Environment* should be used.

Contents

Introduction

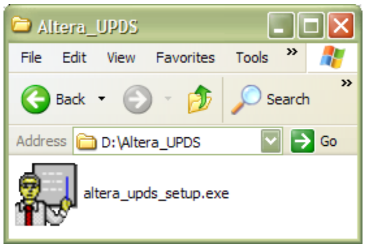
The Altera Monitor Program is a software application that runs on a host PC connected to a Nios II system. It allows the user to compile or assemble a Nios II application, download the application to the Nios II system, and then debug the running application. The Monitor Program provides functionality that allows the user to:

- Examine and modify register and memory contents.
- Disassemble the machine code present in any memory region.
- Single step through each assembly language instruction in the program.
- Set breakpoints that stop the execution of a program when certain instructions are reached or when certain data addresses are accessed.
- Set watch expressions and watch their values at different points in the execution of the program.
- Examine a graphical view of an *instruction trace* that records the set of recently executed instructions.
- Perform terminal input/output via the JTAG UART component.

1 Installing the Altera Monitor Program

The Altera Monitor Program is released as part of the Altera University Program Design Suite (UPDS). You must re-install it every time you re-install Quartus. To install the Altera UPDS, proceed as follows:

1. Download the Altera UPDS from the University Program section of Altera's website. It can be found by going to *www.altera.com* and clicking on *University Program* under *Education & Events*. Once in the University Program section, click on *Design Software* under *Educational Materials*, and select *Altera Monitor Program*. Download the EXE Install Package and extract it to a directory of your choice. In this tutorial, we have chosen D:\Altera_UPDS. As shown in Figure 1, the folder will contain a single executable file named *altera_upds_setup.exe*.



Address D:\Altera_UPDS

altera_upds_setup.exe

Figure 1. Altera UPDS installer file.

2. Double-click on the *altera_upds_setup.exe* executable file. This will bring up the first screen of the installer as illustrated in Figure 2. Click on the **Next** button and proceed to the next step.

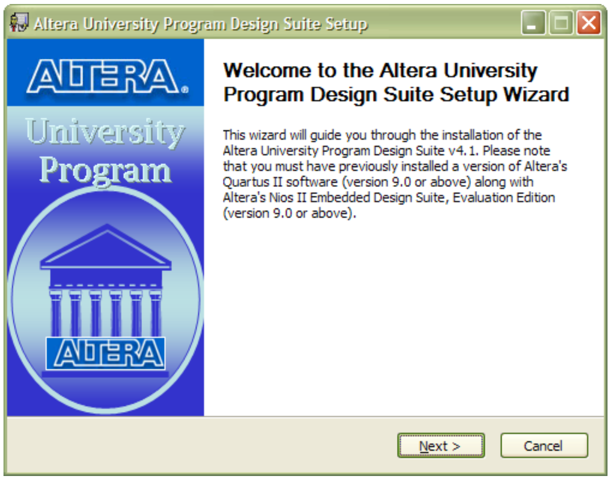
3. The installer will display the License Agreement; click **I Agree** to continue.

4. On the next screen, you will see the components that will be installed, as shown in Figure 3.

   The Altera Monitor Program component will be installed to the location *%ALTERA_HOME%\University_Program\Monitor_Program*, where *%ALTERA_HOME%* is the directory that is one level above the Quartus II installation directory.

   The tutorial files will be installed to the *%ALTERA_HOME%\University_Program\Monitor_Program\tutorial* directory, and we will refer to this path as <*TUTORIAL_FILES*> throughout the remainder of this tutorial.

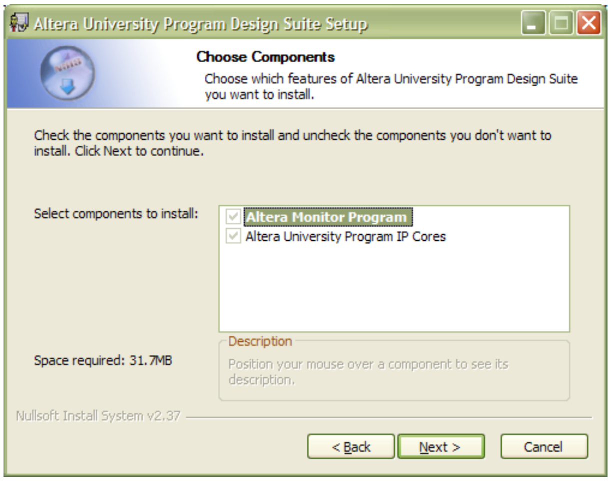
   Click **Next** to continue.



Welcome to the Altera University Program Design Suite Setup Wizard

This wizard will guide you through the installation of the Altera University Program Design Suite v4.1. Please note that you must have previously installed a version of Altera's Quartus II software (version 9.0 or above) along with Altera's Nios II Embedded Design Suite, Evaluation Edition (version 9.0 or above).

Figure 2. Altera UPDS Install Wizard.


Altera University Program Design Suite Setup
Nullsoft Install System v2.37
< Back
Next >
Cancel

Choose Components

Choose which features of Altera University Program Design Suite you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

☑ Altera Monitor Program

☑ Altera University Program IP Cores

Space required: 31.7MB

Description

Position your mouse over a component to see its description.

Figure 3. The components that will be installed.

5. The installer is now ready to begin copying files. Click **Install** to install the Altera UPDS, along with the Monitor Program. During the installation process, you will be asked if you would like a shortcut to the Altera Monitor Program to be placed on your **Windows Desktop**. Answering yes will place an icon similar to the one shown in Figure 4 on your desktop.

Figure 4. Altera Monitor Program desktop icon.

6. Assuming that the installation was successful, the screen shown in Figure 5 will be displayed. Click on the **Finish** button to complete the installation. Should an error occur, a window will suggest the appropriate action. Errors include:
   - Quartus II software is not installed or the Quartus II version is too old.
   - Nios II SDK software is not installed or the version is too old.

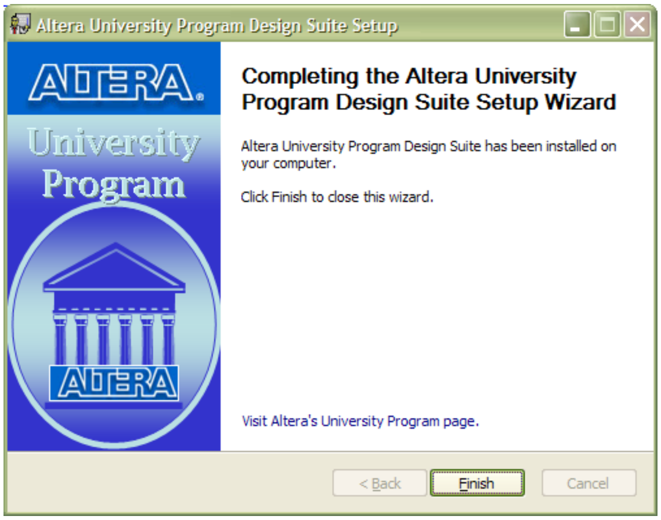

Altera University Program Design Suite Setup
ALTERA
University Program
Completing the Altera University Program Design Suite Setup Wizard
Altera University Program Design Suite has been installed on your computer.
Click Finish to close this wizard.
Visit Altera's University Program page.
< Back
Finish
Cancel

Figure 5. Altera UPDS installation finished.

2 Starting the Altera Monitor Program

Before downloading, running, and debugging applications on the Nios II processor, a Nios II system has to be downloaded onto the DE2 Development and Education board. The system can be downloaded by either using the Quartus II software, or using the Monitor Program user interface. This tutorial requires the Nios II system located at *<TUTORIAL_FILES>**example**hw**example.sof* to be downloaded.

The tutorial called *Introduction to the Quartus II Software* explains how to download a circuit onto the FPGA on the DE2 board using Quartus II, and the tutorial called *Introduction to the Altera SOPC Builder* tool shows you how to create Nios II systems. These tutorials are provided in the University Program section of Altera's web site.

Information on how to download the system via the Monitor Program user interface is provided later in this tutorial.

If you have chosen to place a shortcut on the Windows Desktop, you can start the Altera Monitor Program by double clicking on the desktop icon. The Monitor Program can also be started from the Windows Start Menu by following the menus to **Altera > University Program > Altera Monitor Program > Altera Monitor Program**.

After startup, the Monitor Program will appear as shown in Figure 6.

Working with Windows and Tabs

You can arrange your workspace by moving and resizing the internal windows inside the main Altera Monitor Program window.

To move a particular window to a different location, click on the window title or the tab associated with the window, and drag the mouse to the new location. As you move the mouse across the main window, the dragged window will snap to different locations. To detach the dragged window from the main window, drag it beyond the boundaries of the main window. To re-attach a window to the main window, drag the tab associated with the window onto the main window.

To resize a window, hover the mouse over one of its borders, and then drag the mouse. Resizing a window that is attached to the main window will cause any adjacent attached windows to also change in size accordingly.

To hide or display a particular window, use the **Windows** menu. To revert to the default window arrangement, simply restart the Altera Monitor Program. Figure 7 shows an example of a rearranged workspace.

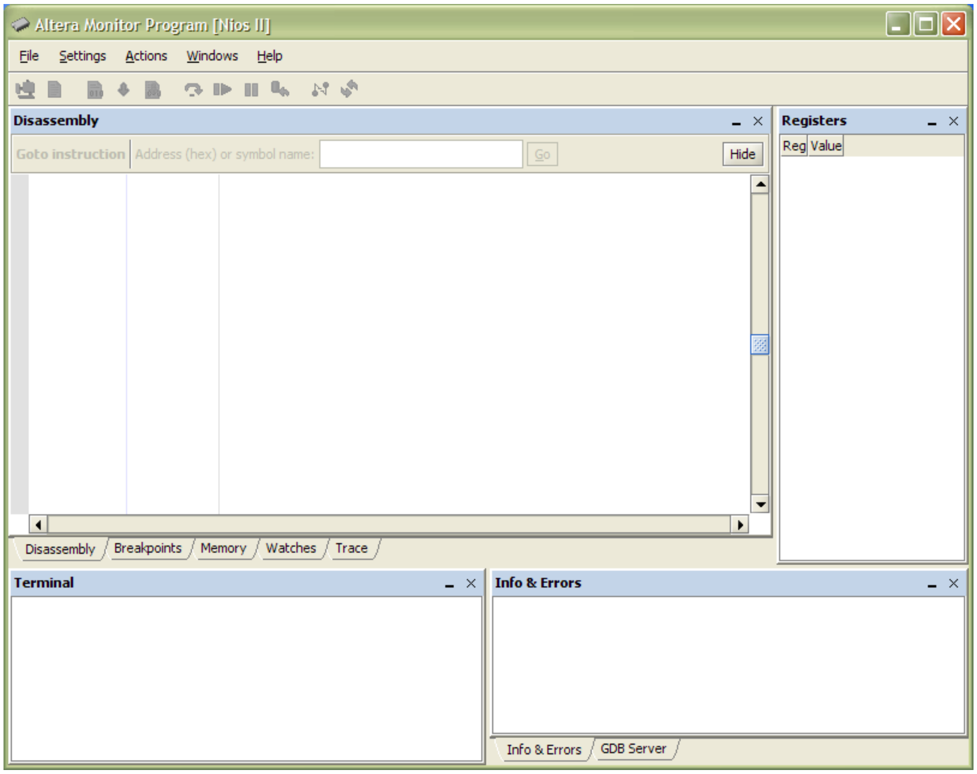

Altera Monitor Program [Nios II]
File Settings Actions Windows Help
Disassembly
Goto instruction Address (hex) or symbol name:
Go
Hide
Registers
Reg Value
Disassembly Breakpoints Memory Watches Trace
Terminal
Info & Errors
Info & Errors GDB Server

Figure 6. The Altera Monitor Program at startup.

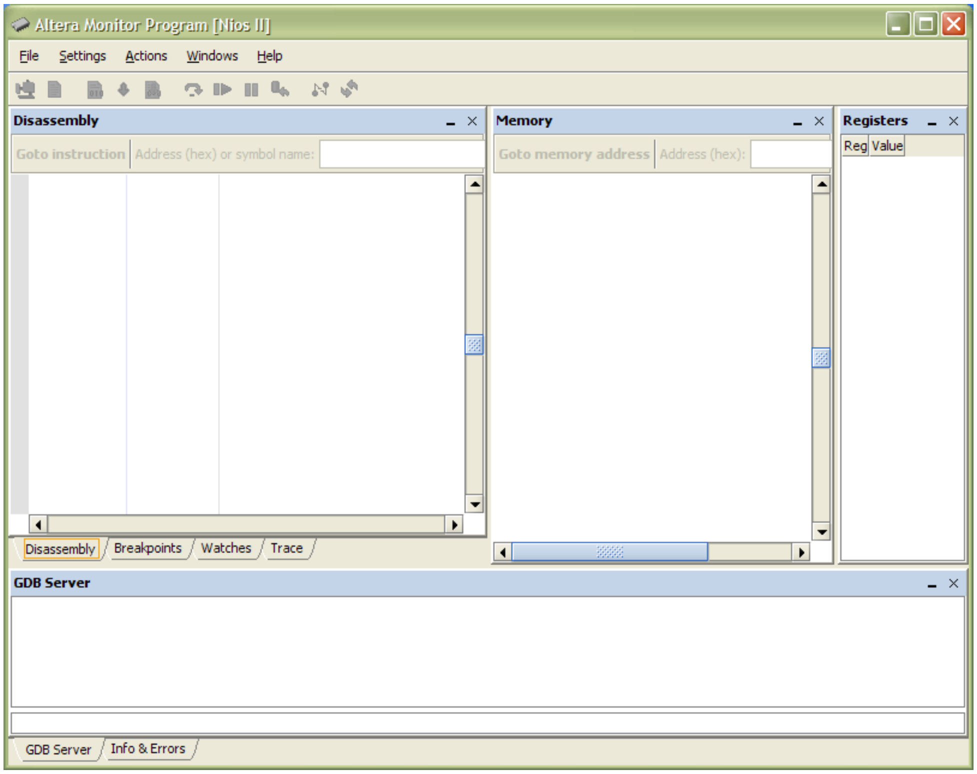

Altera Monitor Program [Nios II]
File Settings Actions Windows Help
Disassembly
Goto instruction Address (hex) or symbol name:
Memory
Goto memory address Address (hex):
Registers
Reg Value
Disassembly Breakpoints Watches Trace
GDB Server
GDB Server Info & Errors

Figure 7. The Altera Monitor Program with a Rearranged Workspace.

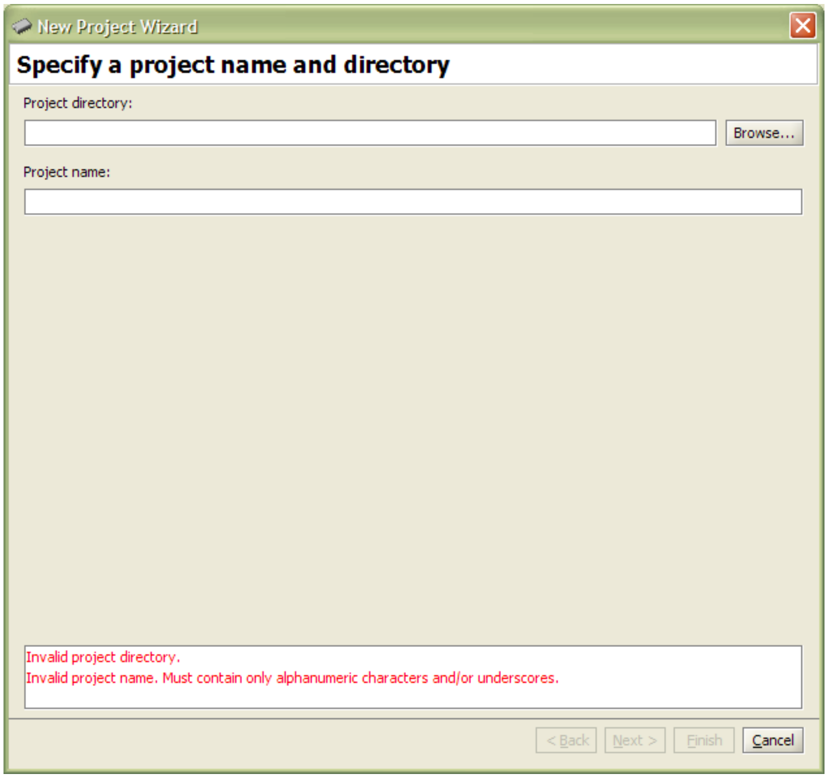
3 Creating a New Project

To use the Monitor Program, you first have to create a project. The project stores the information that the Monitor Program needs to compile, assemble, execute, and debug your application.

To create a new project, click the **File > New Project...** menu item. The **New Project Wizard** window will appear, as shown Figure 8. Using this wizard, we will specify the project's name and location, the Nios II system that will be used to run our program, and the program itself.

The window consists of a content area, a message area, and navigation buttons. The content area contains the fields that are used to input information about the project. The message area displays error and information messages. Double-clicking on an error message will select the field causing the error. The navigation buttons are used to move between the different steps of the wizard.

Proceed with the steps below to configure the new project.



Invalid project directory.
Invalid project name. Must contain only alphanumeric characters and/or underscores.

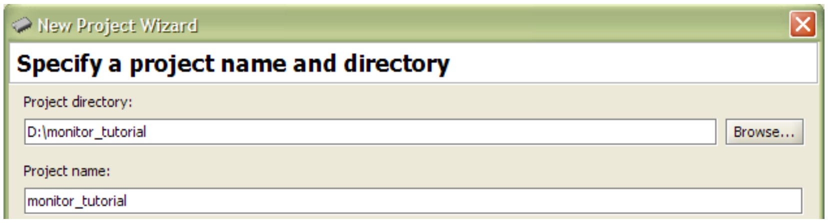
Figure 8. New Project Wizard.

Specify a project name and directory

The first step is to specify the project's name and the directory that will be used to store the files associated with the project. These files include the project file, which stores all the settings specified in this wizard, and any files that may be generated when your program is compiled.

In this tutorial, we will specify *monitor_tutorial* as the project's name, and *D:\monitor_tutorial* as the project's directory. If this directory does not yet exist, you will need to create it first. To select the directory without having to type its full path, click on the **Browse...** button. Note that a given directory may contain at most one project.

The window with the settings filled in is shown in Figure 9. Click **Next >** to continue.


New Project Wizard

Specify a project name and directory

Project directory: D:\monitor_tutorial

Project name: monitor_tutorial

Figure 9. New Project Wizard: Specify a project name and directory.

Specify a system

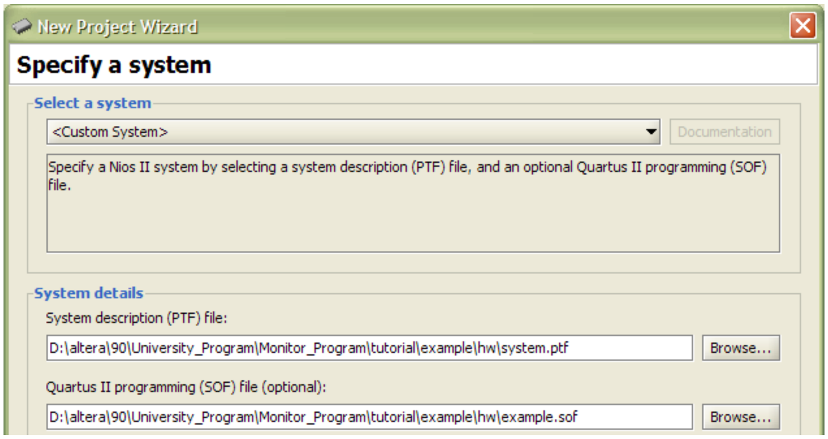
The next step is to specify a Nios II system for your program to run on. Nios II-based systems are described by a *.ptf* file, which is generated by the Altera SOPC Builder tool[1] when the system is created. An optional *.sof* file, if specified, allows the Monitor Program to download the Nios II-based system onto the board. If this file is not specified, you will have to use some other method (such as the Quartus II software) to do this.

The drop-down list on the **Select a system** pane allows you to select one of the sample systems provided with the Monitor Program, or **<Custom System>** that allows you to specify your own system. Both the *.ptf* and the *.sof* files are automatically filled in by the Monitor Program if a sample system is selected. However, if **<Custom System>** is selected, then the files need to be specified manually in the **System details** pane.

In this tutorial, select **<Custom System>** from the drop-down list at the top of the window, since we will use a system designed specifically for this tutorial. To specify the *.ptf* file for the system, click on the **Browse...** button next to the **System description (PTF) file** field, and navigate to *<TUTORIAL_FILES>\example\hw*. Select *system.ptf*, and click **Select**. To allow us to use the Monitor Program to download the system onto the board, we will also specify the *.sof* file for the system. Click on the **Browse...** button next to the **Quartus II programming (SOF) file (optional)** field, and select *system.sof*, which is located in the same directory as the *.ptf* file. This directory should be selected by default.

In addition to specifying a custom system, you can select one of the sample systems provided with the Monitor Program. You can later experiment with the sample systems, whose documentation can be accessed by selecting the system of interest and clicking on the **Documentation** button.

The window with the settings filled in is shown in Figure 10. Click **Next >** to continue.


New Project Wizard

Specify a system

Select a system: <Custom System>

Specify a Nios II system by selecting a system description (PTF) file, and an optional Quartus II programming (SOF) file.

System details

System description (PTF) file: D:\altera\90\University_Program\Monitor_Program\tutorial\example\hw\system.ptf

Quartus II programming (SOF) file (optional): D:\altera\90\University_Program\Monitor_Program\tutorial\example\hw\example.sof

Figure 10. New Project Wizard: Specify a system.

[1] More information about creating systems using SOPC Builder can be found in the tutorial called *Introduction to the Altera SOPC Builder*, available in the University Program section of Altera's website.

Specify a program type

The next step is to select the type of program that you would like to compile, assemble, debug, and run using the Monitor Program. The **Program Type** drop-down list allows you to choose from among the following program types:

1. **No Program**: Use this option if you want to connect to the Nios II system, without loading a program first. This is useful for running and debugging a program that was loaded into the system using some other method. It can also be used to read from and write to the system's address space, to access its memory-mapped devices.

2. **Assembly Program**: A program written using assembly language.

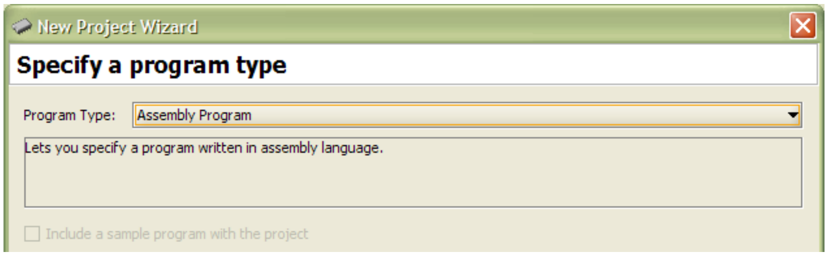
3. **C Program**: A program written using the C language.

4. **Program with Device Driver Support**: This is an advanced option, which can be used to build programs using the Nios II Software Build Tools. These tools generate a Board Support Package (BSP), which is a system library that contains device drivers for all the peripherals on the development board (e.g., the DE2 board) that are supported by the selected Nios II system. The programs that are used with this option can be written in either assembly, C, or C++ (or any combination). For more information about writing programs that can take advantage of the BSP, see Section 17.

5. **ELF or SREC File**: A pre-compiled program, in ELF or SREC format. Use this option to run a program that was already compiled, but to which you do not necessarily have the source code.

If you select a sample Nios II system for running your program, it may have sample programs bundled with it. Sample programs range from very simple examples that demonstrate basic concepts, to complete demonstrations that exercise many aspects of a system. They include the required source files and any settings that are needed for the program to run properly. You may extend or modify a sample program if you choose to. If you select a program type that has sample programs available for your sample system, you can check the **Include a sample program with the project** check box to display a list of all the available sample programs, and choose one for your project. This feature will not be used in this tutorial.

For this tutorial, select **Assembly Program** under **Program Type**. We will specify an assembly program written specifically for this tutorial in the next step. The window with the settings filled in is shown in Figure 11. Click **Next >** to continue.



Program Type: Assembly Program

Lets you specify a program written in assembly language.

□ Include a sample program with the project

Figure 11. New Project Wizard: Specify a program type.

Specify program details

The next step is to specify the source code files for your program, and any additional program options specific to the program type selected in the previous window.

You can add as many source files as your program requires, but in this tutorial, only a single source file is needed. When you have more than one source file in the list, the name of the very first file (at the top of the list) determines the name of the compiler's output binary files. To change the order of source files in the list, use the **Up** and **Down** buttons.

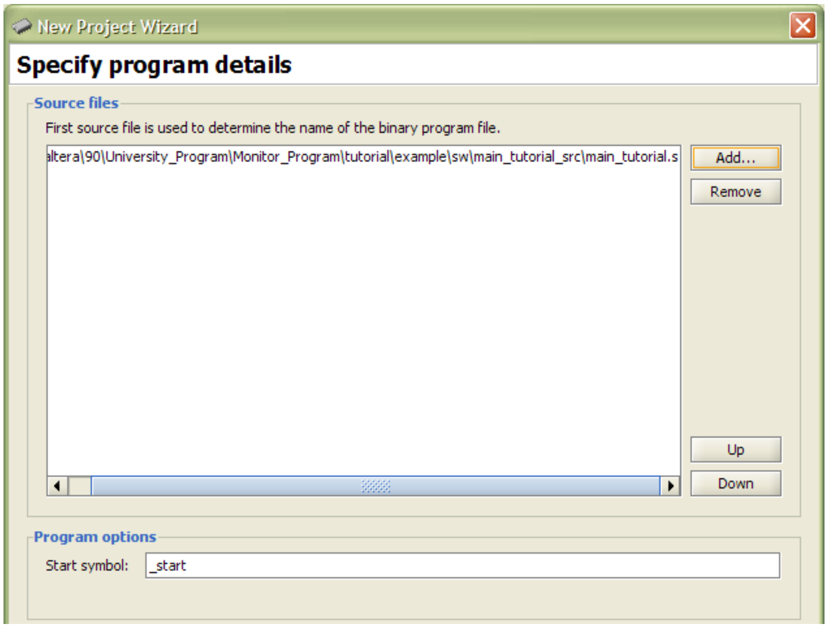
The only program option in this step that is available for assembly programs is the **Start symbol** option. The *start symbol* of an assembly-language program identifies the label that corresponds to the first instruction of the program.

To add the single assembly code file that is needed by this tutorial, click on the **Add...** button. In the file selection dialog that pops up, navigate to the *<TUTORIAL_FILES>\example\sw\main_tutorial_src* directory, select the *main_tutorial.s* file, and click **Select**.

Leave the **Start symbol** program option at its default value. The default start symbol is `_start` and this is the symbol that is used in *main_tutorial.s*.

Note that this wizard step is not displayed if you select the **No Program** program type.

The window with the settings filled in is shown in Figure 12. Click **Next >** to continue.



Specify program details

Source files

First source file is used to determine the name of the binary program file.

altera\90\University_Program\Monitor_Program\tutorial\example\sw\main_tutorial_src\main_tutorial.s

Program options

Start symbol: _start

Figure 12. New Project Wizard: Specify program details.

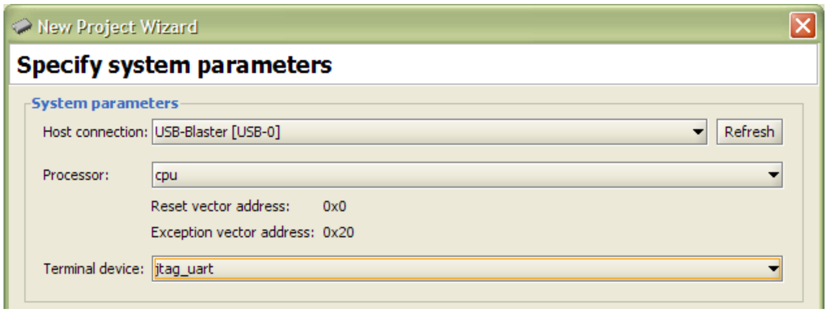
Specify system parameters

The next step is to specify the board, processor, and terminal device that we will use to run the program. The **Host connection** drop-down list contains the physical connection links (such as cables) that exist between the computer and any boards connected to it. The Nios II processors available in the system are found in the **Processor** drop-down list, and all the terminal devices connected to the selected processor are displayed in the **Terminal device** drop-down list.

For this tutorial, specify these options as follows:

1. Select the cable used to connect your host computer to the DE2 board from the **Host connection** drop-down list. The DE2 board is connected via a USB-Blaster cable.
2. The system used in this tutorial only has a single processor named `cpu`. It should already be selected in the **Processor** list. The addresses of the selected processor's reset and exception vectors are shown for reference.
3. The system used in this tutorial only has a single terminal device, named `jtag_uart`, which should already be selected in the **Terminal device** list. It will be connected to the terminal in the Monitor Program.

The window with the settings filled in is shown in Figure 13. Click **Next >** to continue.


New Project Wizard

Specify system parameters

System parameters

Host connection: USB-Blaster [USB-0]

Processor: cpu

Reset vector address: 0x0

Exception vector address: 0x20

Terminal device: jtag_uart

Figure 13. New Project Wizard: Specify system parameters.

Specify program memory settings

The next step is to specify the memory settings for compiling and linking your program.

Nios II programs are compiled into an *Executable and Linking Format* (ELF) file. This format supports *sections*, which can be used to divide a program into multiple parts: executable code section (called `.text`) and a data section (called `.data`). The partitioning of a program into different sections is performed by the linker.

The Altera Monitor Program allows the user to specify two sections of the program, as described below. Each section will be placed at the address value of *base address of memory device* + *start offset*. If a start offset is not specified, its default value will be zero.

For this tutorial, use the following settings:

- **.text section**
  - Memory device: `memory/s1 (0h - 7fffh)`, corresponding to the on-chip memory of the Nios II system (this is the default and only choice in our example system)
  - Start offset in device: `1000`, corresponding to a 4096-byte offset into the on-chip memory to start the `.text` section
- **.data section**
  - Memory device: `memory/s1 (0h - 7fffh)`, corresponding to the on-chip memory of the Nios II system (this is the default and only choice in our example system)
  - Start offset in device: `1000`, corresponding to a 4096-byte offset into the on-chip memory to start the `.data` section.

The window with the settings filled in is shown in Figure 14.

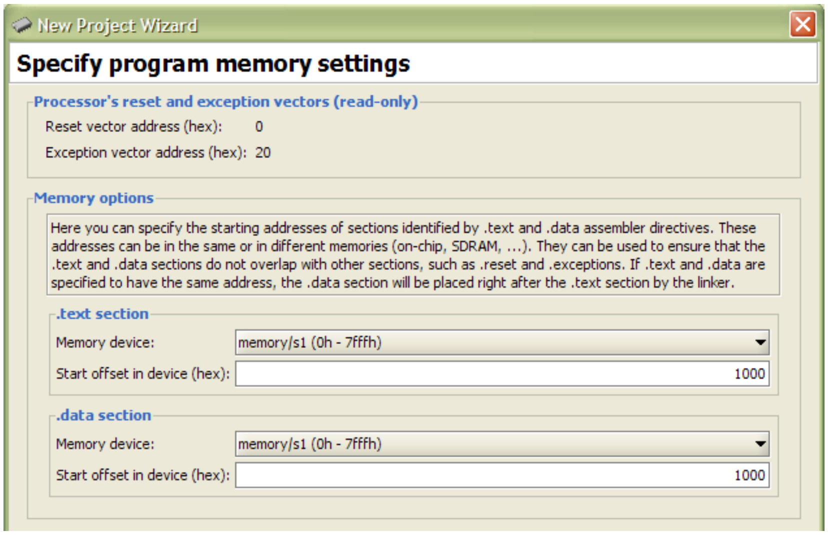
Note that the `.text` and `.data` offsets are the same, which would lead to an overlap of the two sections in memory. However, because the two offsets are exactly the same, the linker script used to produce the program executable will instead automatically place the `.data` section immediately after the `.text` section.

To make full use of these settings in your code, you will need to explicitly place the `.text` and `.data` assembler directives in your code. If you do not, the settings for the `.text` section will be used to locate the first instruction of your program. You should always use these directives if you want to separate your program text and data (for example, by placing each in a different memory device). Simply using `.org` directives for this purpose can cause the ELF and SREC files generated by assembling your program to be very large, especially if there is a wide gap between the text and data sections.

Note that the program memory settings are only available for the assembly and C program types.

Click **Finish** to create the new project and exit the wizard. At this point, you will be presented with a prompt, shown in Figure 15. This prompt appears whenever you create a new project (or open an existing project) that has the *.sof* file for the Nios II system specified. Clicking **Yes** will instruct the Monitor Program to attempt to download the Nios II system associated with the project onto the DE2 board. You can also choose to download the system at a later time, by following the instructions in Section 5.

The next section describes how to modify the settings of a project after it is created.



Specify program memory settings

Processor's reset and exception vectors (read-only)

Reset vector address (hex): 0

Exception vector address (hex): 20

Memory options

Here you can specify the starting addresses of sections identified by .text and .data assembler directives. These addresses can be in the same or in different memories (on-chip, SDRAM, ...). They can be used to ensure that the .text and .data sections do not overlap with other sections, such as .reset and .exceptions. If .text and .data are specified to have the same address, the .data section will be placed right after the .text section by the linker.

.text section

Memory device: memory/s1 (0h - 7fffh)

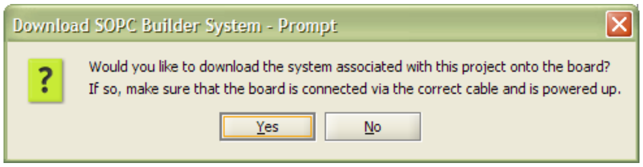
Start offset in device (hex): 1000

.data section

Memory device: memory/s1 (0h - 7fffh)

Start offset in device (hex): 1000

Figure 14. New Project Wizard: Specify program memory settings.



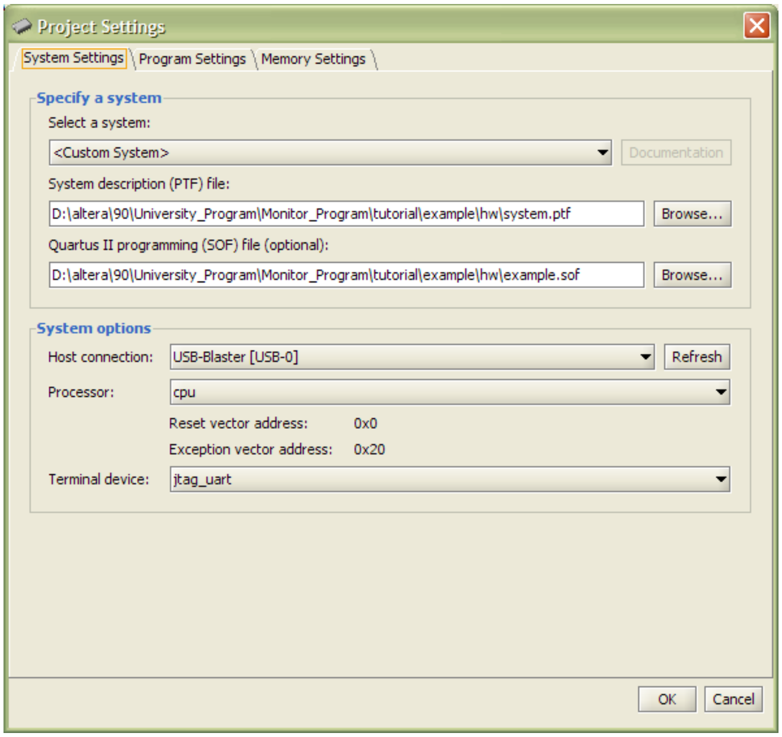
Would you like to download the system associated with this project onto the board?
If so, make sure that the board is connected via the correct cable and is powered up.

Figure 15. The Download SOPC Builder System prompt.

4 Modifying the Settings of an Existing Project

After you create your project (or load an existing project), you can modify many of the settings that were originally specified in the **New Project Wizard** by using the **Project Settings** window. To access the **Project Settings** window, click on the **Settings > System Settings...** menu item, or use the toolbar button. This will display the **Project Settings** window with the **System Settings** tab selected, as shown in Figure 16. To directly display the **Project Settings** window with the **Program Settings** tab selected, you can use the **Settings > Program Settings...** menu item, or the toolbar button.

This window contains most of the settings that were described in the previous section (about creating a new project).



Select a system: <Custom System>

System description (PTF) file: D:\altera\90\University_Program\Monitor_Program\tutorial\example\hw\system.ptf

Quartus II programming (SOF) file (optional): D:\altera\90\University_Program\Monitor_Program\tutorial\example\hw\example.sof

Host connection: USB-Blaster [USB-0]

Processor: cpu

Reset vector address: 0x0

Exception vector address: 0x20

Terminal device: jtag_uart

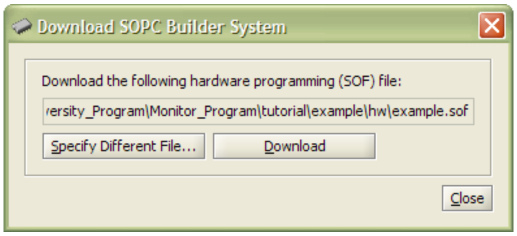
Figure 16. The Project Settings window.

5 Downloading the Nios II System onto the Board

Before running and debugging your program, the Nios II system needs to be downloaded onto the board. You have already seen two ways to do this, by either using the Quartus II software, or by answering **Yes** in the prompt that appears when creating a new project (or opening an existing project). It is also possible to download the Nios II system onto the board at any time using the Monitor Program. To do this, proceed with the steps below:

1. Click on **Actions > Download SOPC Builder System**. The window shown in Figure 17 will pop up, displaying the name of the *.sof* file that we specified earlier when creating the project. This file contains the data that will be downloaded onto the board, representing the Nios II system that we selected. You have the option of specifying a different file by clicking on **Specify Different File...**, but in this tutorial we will just use the file that we specified earlier (which will be the case most of the time).

2. Click on the **Download** button. The system will be downloaded onto the board, and after a short while you should get a success message. If the download failed, make sure that the board is powered up and properly connected via the cable that was specified in the **Host connection** list in the **New Project Wizard**.



Download the following hardware programming (SOF) file: versity_Program\Monitor_Program\tutorial\example\hw\example.sof

Figure 17. The Download SOPC Builder System window.

6 Compiling and Loading the Program

After successfully creating a project, the program can be compiled and downloaded onto the DE2 board. There are three different commands that can be used to compile and/or load a program:

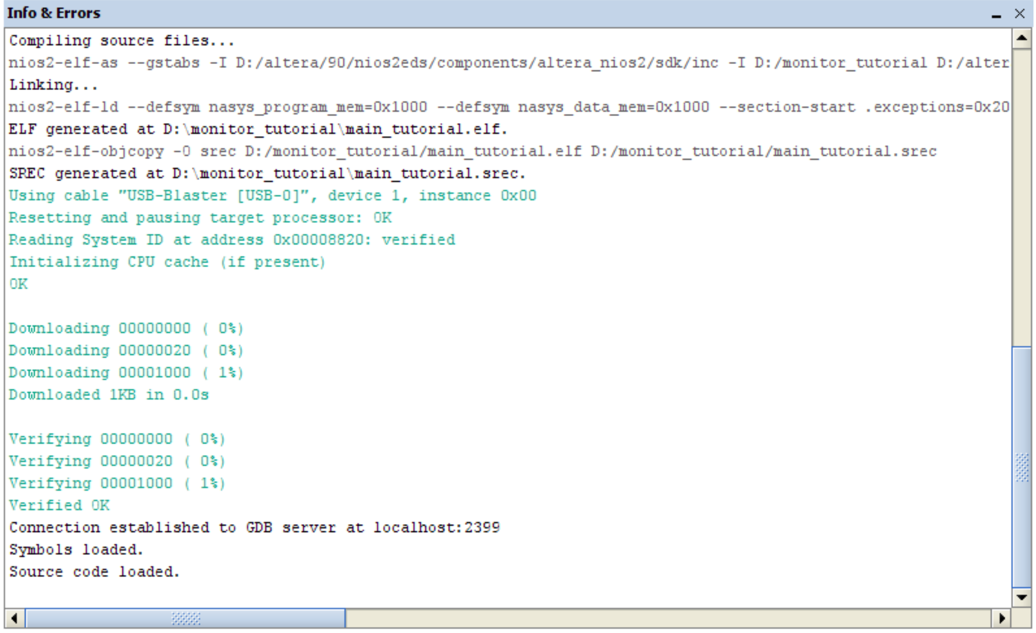
- **Actions > Compile** menu item or 010 toolbar button:
  Compiles the source files into an ELF and SREC file. Build warnings and errors will show up in the **Info & Errors** window. The generated ELF and SREC files are placed in the project's directory.

- **Actions > Load** menu item or ⬇ toolbar button:
  Loads the compiled SREC file onto the board and begins a debugging session in the Monitor Program. Loading progress messages are displayed in the **Info & Errors** window.

- **Actions > Compile & Load** menu item or 010 toolbar button:
  Performs the operations of both compilation and loading.

In this example, the program has not yet been compiled, so it cannot be loaded (the **Load** option is disabled).

Click the **Actions > Compile & Load** menu item or click the 010 toolbar button to begin the compilation and loading process. Throughout the process, messages are displayed in the **Info & Errors** window. The messages should resemble those shown in Figure 18.



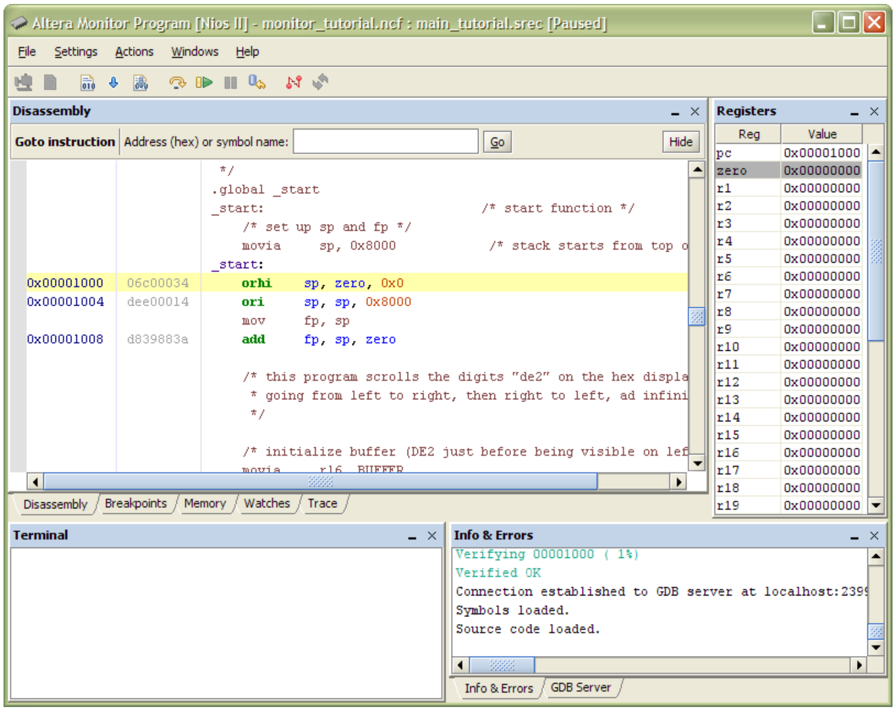
```
Compiling source files...
nios2-elf-as --gstabs -I D:/altera/90/nios2eds/components/altera_nios2/sdk/inc -I D:/monitor_tutorial D:/alter
Linking...
nios2-elf-ld --defsym nasys_program_mem=0x1000 --defsym nasys_data_mem=0x1000 --section-start .exceptions=0x20
ELF generated at D:\monitor_tutorial\main_tutorial.elf.
nios2-elf-objcopy -O srec D:/monitor_tutorial/main_tutorial.elf D:/monitor_tutorial/main_tutorial.srec
SREC generated at D:\monitor_tutorial\main_tutorial.srec.
Using cable "USB-Blaster [USB-0]", device 1, instance 0x00
Resetting and pausing target processor: OK
Reading System ID at address 0x00008820: verified
Initializing CPU cache (if present)
OK

Downloading 00000000 ( 0%)
Downloading 00000020 ( 0%)
Downloading 00001000 ( 1%)
Downloaded 1KB in 0.0s

Verifying 00000000 ( 0%)
Verifying 00000020 ( 0%)
Verifying 00001000 ( 1%)
Verified OK
Connection established to GDB server at localhost:2399
Symbols loaded.
Source code loaded.
```

Figure 18. Compilation and loading messages (the **Info & Errors** window has been maximized).

After successfully completing this step, your Monitor Program display should look similar to Figure 19. At this point, the program is paused at its first instruction.

Compilation errors

During the process of developing software, you will likely encounter compilation errors. Error messages from the Nios II assembler or from the C compiler are displayed in the **Info & Errors** window. An example of a compiler error message is shown in Figure 20. The file name and the line number corresponding to the source of the error are displayed, in addition to an indication of the cause of the error. You may be able to deduce the real cause of the error from the message or you may need to do some additional searching.

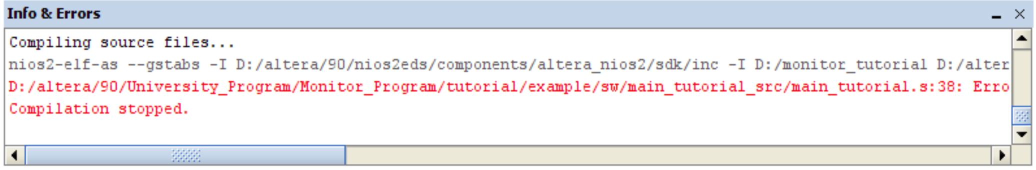


```
                                 */
                             .global _start
                             _start:                               /* start function */
                                 /* set up sp and fp */
                                 movia     sp, 0x8000               /* stack starts from top o
                             _start:
0x00001000      06c00034         orhi      sp, zero, 0x0
0x00001004      dee00014         ori       sp, sp, 0x8000
                                 mov       fp, sp
0x00001008      d839883a         add       fp, sp, zero

                                 /* this program scrolls the digits "de2" on the hex displa
                                  * going from left to right, then right to left, ad infini
                                  */

                                 /* initialize buffer (DE2 just before being visible on lef
                                 movia     r16, BUFFER
```

Reg	Value
pc	0x00001000
zero	0x00000000
r1	0x00000000
r2	0x00000000
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
r13	0x00000000
r14	0x00000000
r15	0x00000000
r16	0x00000000
r17	0x00000000
r18	0x00000000
r19	0x00000000

```
Verifying 00001000 ( 1%)
Verified OK
Connection established to GDB server at localhost:2399
Symbols loaded.
Source code loaded.
```

Figure 19. The Altera Monitor Program window after loading the example program.



```
Compiling source files...
nios2-elf-as --gstabs -I D:/altera/90/nios2eds/components/altera_nios2/sdk/inc -I D:/monitor_tutorial D:/alter
D:/altera/90/University_Program/Monitor_Program/tutorial/example/sw/main_tutorial_src/main_tutorial.s:38: Erro
Compilation stopped.
```

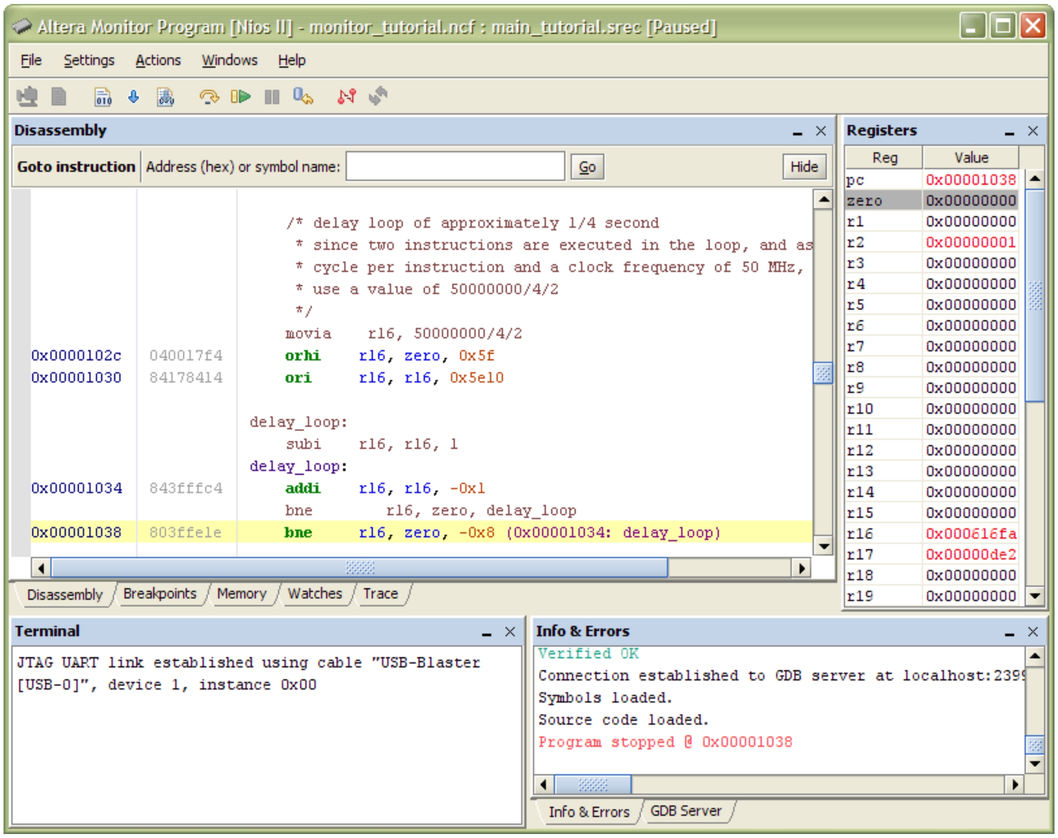
Figure 20. A compiler error message.

7 Running the Program

As mentioned at the end of the previous section, the program is paused at its first instruction after it has been loaded. To run the program, click the **Actions > Continue** menu item or click the ▶ toolbar button. The sample program will continuously scroll the digits dE2 across the 7-segment displays on the DE2 board.

The **Continue** command runs the program until something halts the processor's execution, such as a breakpoint or a forced user halt. To force the program to halt, click the **Actions > Stop** menu item or click the ⏸ toolbar button; the processor will stop at the instruction to be executed next.

When the program is stopped, all debugging windows are updated with new data. As seen in Figure 21, the **Disassembly** window highlights the next instruction to be executed in yellow and the **Registers** window highlights register values that have changed since the last program stoppage in red. The other windows in the Monitor Program are also updated, which will be shown in later parts of this tutorial.



```
                                    /* delay loop of approximately 1/4 second
                                     * since two instructions are executed in the loop, and as
                                     * cycle per instruction and a clock frequency of 50 MHz,
                                     * use a value of 50000000/4/2
                                     */
                                    movia    r16, 50000000/4/2
0x0000102c    040017f4              orhi     r16, zero, 0x5f
0x00001030    84178414              ori      r16, r16, 0x5e10

                                delay_loop:
                                    subi     r16, r16, 1
                                delay_loop:
0x00001034    843fffc4              addi     r16, r16, -0x1
                                    bne         r16, zero, delay_loop
0x00001038    803ffe1e              bne      r16, zero, -0x8 (0x00001034: delay_loop)
```

Reg	Value
pc	0x00001038
zero	0x00000000
r1	0x00000000
r2	0x00000001
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
r13	0x00000000
r14	0x00000000
r15	0x00000000
r16	0x00061 6fa
r17	0x00000de2
r18	0x00000000
r19	0x00000000

```
JTAG UART link established using cable "USB-Blaster
[USB-0]", device 1, instance 0x00
```

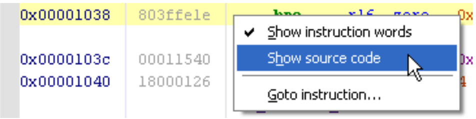
```
Verified OK
Connection established to GDB server at localhost:239
Symbols loaded.
Source code loaded.
Program stopped @ 0x00001038
```

Figure 21. **Disassembly** window after the program has been stopped.

8 Using the Disassembly Window

The **Disassembly** window displays human-readable machine code by interpreting the memory values as encoded instructions. As shown in Figure 21, there are three columns in the window. The left-most column gives the memory address corresponding to the instruction displayed on that line. The middle column displays the 32-bit instruction word corresponding to the machine encoding of the instruction. The right-most column displays the human-readable instruction along with the corresponding source code. For example, in Figure 21, the four instructions located at memory addresses 0x0000102c, 0x00001030, 0x00001034, and 0x00001038 have been disassembled.

The Disassembly window can be configured to display less information on the screen, such as not showing the source code from the *.s* assembly language file or not showing the machine encoding of the instructions. These settings can be changed by right-clicking on the window and selecting the appropriate menu item, as shown in Figure 22. The display in the window also uses a color-coded scheme, as detailed in Table 1.



```
0x00001038    803ffe1e      bne      r16, zero,  0x

0x0000103c    00011540
0x00001040    18000126
```

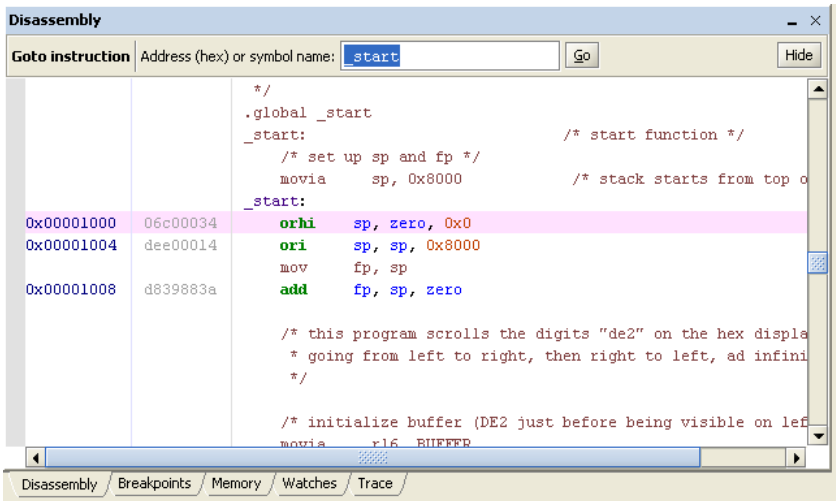
Figure 22. Pop-up menu to configure the display of the Disassembly window.

By scrolling using the vertical scrollbar on the right side of the **Disassembly** window or by using a mouse scroll wheel, different regions of memory can be disassembled and displayed. It is also possible to scroll to a memory address or an instruction symbol directly by using the **Goto instruction** panel in the Disassembly window. Access this panel through the **Actions > Goto instruction...** menu item and enter a symbol name or an instruction address in hexadecimal format. The instruction address must be a multiple of 4 because every instruction address is aligned

Color	Description
Brown	Source code
Green	Disassembled instruction name
Blue	Registers
Orange	Immediate & offset values
Dark blue	Address values & labels
Purple	Clickable link
Gray	Machine encoding of the instruction

Table 1. **Disassembly** window color-coded scheme.

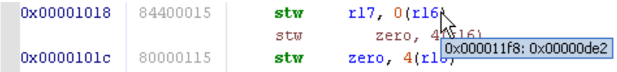
on a 32-bit word boundary. For example, enter `_start` or `1000` and press **Go**. The Disassembly window will show the `0x00001000` address as its first instruction, as shown in Figure 23, which also corresponds to the `_start` symbol. Also note that the instruction is highlighted with a pink background.



```
                          */
                         .global _start
                         _start:                              /* start function */
                             /* set up sp and fp */
                             movia     sp, 0x8000              /* stack starts from top o
                         _start:
0x00001000   06c00034        orhi      sp, zero, 0x0
0x00001004   dee00014        ori       sp, sp, 0x8000
                             mov       fp, sp
0x00001008   d839883a        add       fp, sp, zero

                             /* this program scrolls the digits "de2" on the hex displa
                              * going from left to right, then right to left, ad infini
                              */

                             /* initialize buffer (DE2 just before being visible on lef
                             movia     r16, BUFFER
```

Figure 23. **Goto instruction** panel in the **Disassembly** window.

Register and memory values can be examined in the Disassembly window while the program is in a *Paused* state. This is done by hovering your mouse over a *register* or a *register* + *offset* in the window, as shown in Figure 24.

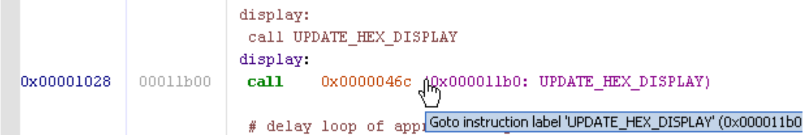


```
0x00001018   84400015        stw       r17, 0(r16)
                             stw          zero, 4(r16)
0x0000101c   80000115        stw       zero, 4(r16)
```

0x000011f8: 0x00000de2

Figure 24. Examining a memory value in the **Disassembly** window.

The Disassembly window also produces special clickable links in its display of branch instructions. Clicking one of these links will display the instruction that the processor would jump to if the branch was taken. Figure 25 shows one example of a link associated with a `call` instruction.



```
                            display:
                             call UPDATE_HEX_DISPLAY
                            display:
0x00001028   00011b00        call     0x0000046c (0x000011b0: UPDATE_HEX_DISPLAY)

                             # delay loop of appr
```

Goto instruction label 'UPDATE_HEX_DISPLAY' (0x000011b0

Figure 25. A clickable link in the **Disassembly** window.

Assembly Language and Machine Instructions

The Disassembly window is a good place to examine what machine instructions are produced by the compiler from your assembly-language instructions or C code. The translation from assembly-language instructions to machine instructions is handled by the Nios II assembler and it is a transparent process to the programmer. However, it is beneficial to examine the disassembled code and compare it with the source code. This is readily done because the Monitor Program displays both the source statements and the disassembled code in different colors. Observe that *pseudoinstructions* are implemented as different machine instructions. For example, the `movia` pseudoinstruction is implemented by the two instructions `orhi` and `ori`, as shown at the address values `0000102c` and `00001030`, respectively, in Figure 21.

9 Single step

Before discussing the single step action, it is convenient (for demonstration purposes) to restart execution of the program from the beginning. Click the **Actions > Restart** menu item or click the toolbar button to restart the program. Notice that the *pc* register value displayed in the **Registers** window is `0x00001000` and the Disassembly window is highlighting that instruction.

The Monitor Program has the ability to perform single step actions. Each single step consists of executing a single machine instruction and returning control to the Monitor Program. Note that if the program being debugged was written in C, each individual single step will still correspond to one assembly language instruction generated from the C code. The ability to step through statements in the high-level source code is not supported by the Monitor Program; however, Altera's *Nios II Integrated Development Environment* supports this advanced feature.

The single step action is invoked by clicking on the **Actions > Single step** menu item or by clicking on the toolbar button. The instruction that is executed by the processor is the one highlighted by the **Disassembly** window before the single step.

Since the first step in this section was to restart the program, the first single step will execute the instruction at `0x1000`, which will zero out the upper 16-bits of the *sp* register. Subsequent single steps will continue to execute one instruction at a time, in sequential order. Single stepping at a branch instruction may jump to a non-sequential instruction address if the branch is taken. You can observe this behavior by single stepping to `0x00001028`, which is a `call` instruction. Single stepping at this instruction will set the *pc* value to `0x000011b0`, which is the location of the `UPDATE_HEX_DISPLAY` label.

The Step Over Subroutine option in the Actions menu performs the Single Step operation unless the current instruction is a call instruction. In this case the program will run until the subroutine called is completed.

10 Using Breakpoints

Breakpoints are special conditions that are checked by dedicated hardware in the Nios II processor as the application is running in real-time. Breakpoints can be triggered in four different ways:

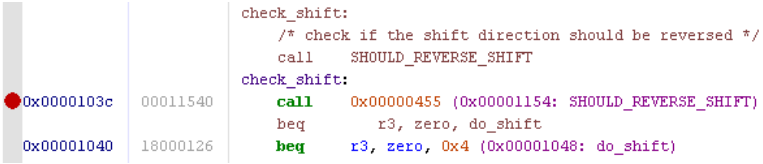
1. Program execution has reached a particular address
2. A read operation has been performed on a particular address
3. A write operation has been performed on a particular address
4. The processor has accessed the memory at a particular address

Note that trigger types 2-4 require the Nios II processor to be compiled with JTAG Debug Level 2 or higher. This section of the tutorial will cover the process of setting an instruction breakpoint (trigger type 1). There are two ways to set an instruction breakpoint. The first method can be used to set a simple instruction breakpoint as follows:

1. Switch to the **Disassembly** window.

2. Navigate to the instruction address that will have the breakpoint. For this example, display the `check_shift` instruction label.

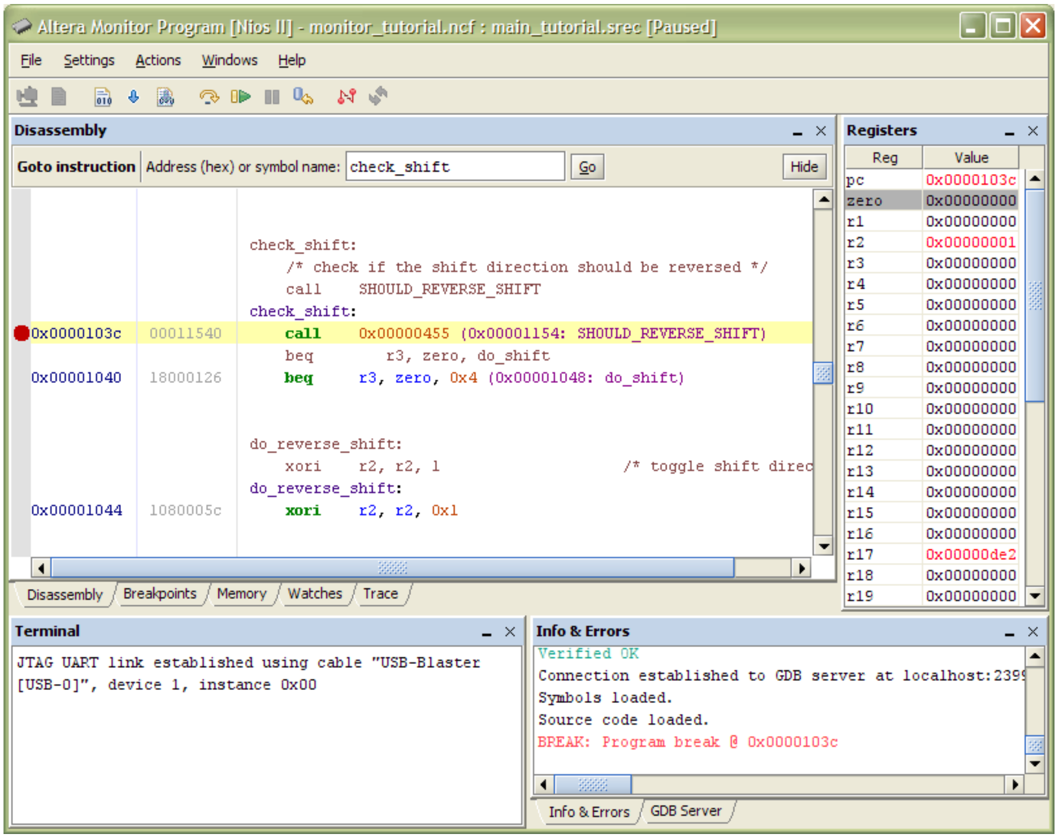
3. Click on the gray bar to the left of the address `0000103c` (address value of the `check_shift` label) to set an instruction breakpoint at this location. See Figure 26 for an illustration of what a breakpoint in the Disassembly window looks like. Clicking the same location again will remove the breakpoint.



```
                           check_shift:
                               /* check if the shift direction should be reversed */
                               call    SHOULD_REVERSE_SHIFT
                           check_shift:
●0x0000103c    00011540        call    0x00000455 (0x00001154: SHOULD_REVERSE_SHIFT)
                               beq        r3, zero, do_shift
 0x00001040    18000126        beq     r3, zero, 0x4 (0x00001048: do_shift)
```

Figure 26. Setting a simple instruction breakpoint in the **Disassembly** window.

Once the instruction breakpoint has been set, run the program and the breakpoint should trigger when the *pc* register value equals `0x0000103c`. The Monitor Program will look similar to Figure 27; notice the message in the **Info & Errors** window indicating that an instruction breakpoint has been triggered.



Address (hex) or symbol name: check_shift

```
                           check_shift:
                               /* check if the shift direction should be reversed */
                               call    SHOULD_REVERSE_SHIFT
                           check_shift:
●0x0000103c    00011540        call    0x00000455 (0x00001154: SHOULD_REVERSE_SHIFT)
                               beq        r3, zero, do_shift
 0x00001040    18000126        beq     r3, zero, 0x4 (0x00001048: do_shift)


                           do_reverse_shift:
                               xori    r2, r2, 1                  /* toggle shift direc
                           do_reverse_shift:
 0x00001044    1080005c        xori    r2, r2, 0x1
```

Reg	Value
pc	0x0000103c
zero	0x00000000
r1	0x00000000
r2	0x00000001
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
r13	0x00000000
r14	0x00000000
r15	0x00000000
r16	0x00000000
r17	0x00000de2
r18	0x00000000
r19	0x00000000

```
JTAG UART link established using cable "USB-Blaster
[USB-0]", device 1, instance 0x00
```

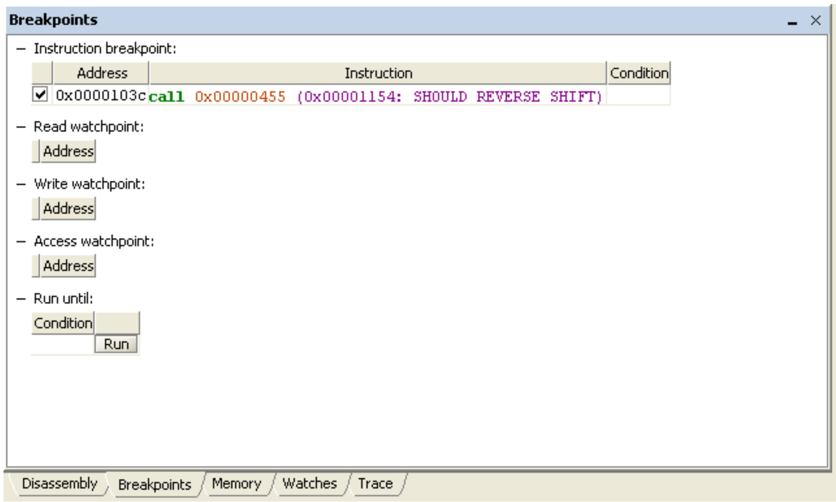
```
Verified OK
Connection established to GDB server at localhost:239
Symbols loaded.
Source code loaded.
BREAK: Program break @ 0x0000103c
```

Figure 27. The Monitor Program after the breakpoint has been triggered.

The second method of setting a breakpoint can be used for all four trigger types as follows:

1. Switch to the **Breakpoints** window, which is shown in Figure 28.

2. The breakpoint that was set earlier in the Disassembly window also appears in this window. The check mark beside the breakpoint can be used to enable or disable it. In this case, leave the check mark as it is.



	Address	Instruction	Condition
☑	0x0000103c	call 0x00000455 (0x00001154: SHOULD REVERSE SHIFT)	

Figure 28. **Breakpoints** window.

3. Right-click on the header corresponding to the breakpoint type you want to add. For example, to set a breakpoint that triggers when the processor writes to a particular address, right-click on the write watchpoints table, as shown in Figure 29.

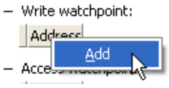


Figure 29. Adding a breakpoint, in this case a write watchpoint, in the **Breakpoints** window.

4. Click **Add**. A new entry will appear in the corresponding table. Enter the desired breakpoint address.

The Monitor Program also supports a more advanced form of a breakpoint, called a *conditional breakpoint*. A conditional breakpoint is an instruction breakpoint that only triggers when the usual instruction breakpoint condition is met and an additional user-specified condition is met. For this example, you will use the same breakpoint from before but with the condition `r2 == 0`, which in this program's context is when the scroll direction is to the left. The process to set this conditional breakpoint is as follows:

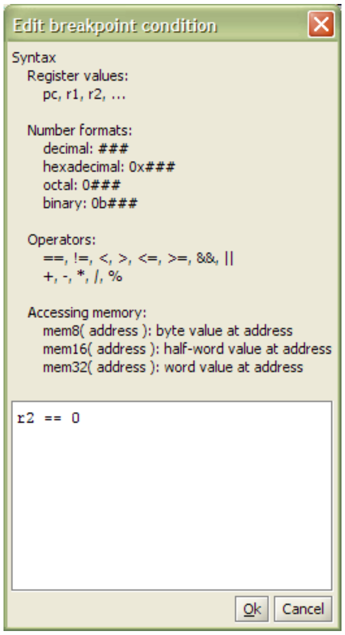
1. Switch to the **Breakpoints** window.

2. For the breakpoint at `0000103c`, double-click on the table cell under the **Condition** column.

3. The window in Figure 30 will appear. This window contains information about the syntax used to describe a condition. For this example type `r2 == 0`.

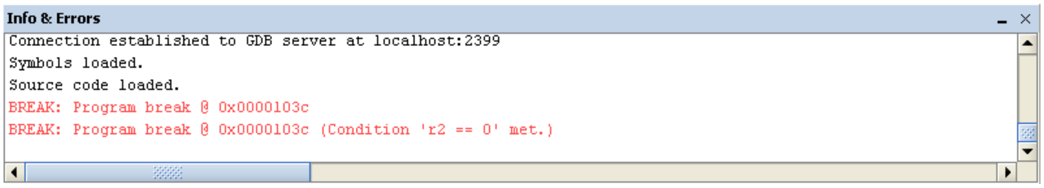
4. Press **Ok**. The **Condition** field for the breakpoint will now show the condition you entered.

The conditional breakpoint is now set. Run the program and as the `dE2` digits on the hexadecimal display disappear and the program begins to shift the digits to the left, the breakpoint will trigger. The **Info & Errors** window will again have a message about the cause of the breakpoint, including the trigger condition that was satisfied, as shown in Figure 31.



```
Syntax
   Register values:
      pc, r1, r2, ...

   Number formats:
      decimal: ###
      hexadecimal: 0x###
      octal: 0###
      binary: 0b###

   Operators:
      ==, !=, <, >, <=, >=, &&, ||
      +, -, *, /, %

   Accessing memory:
      mem8( address ): byte value at address
      mem16( address ): half-word value at address
      mem32( address ): word value at address

r2 == 0
```

Figure 30. **Edit breakpoint condition** window.



```
Connection established to GDB server at localhost:2399
Symbols loaded.
Source code loaded.
BREAK: Program break @ 0x0000103c
BREAK: Program break @ 0x0000103c (Condition 'r2 == 0' met.)
```

Figure 31. Message displayed in the **Info & Errors** window due to a triggered conditional breakpoint.

11 Examining and Changing Register Values

The **Registers** window displays the value of each register in the Nios II processor and allows the user to change most of the register values. The number format of the register values can be changed by right-clicking in the **Registers** window, as shown in Figure 32. You can choose among binary, octal, decimal, and hexadecimal representations in both signed and unsigned versions.

Every time program execution is stopped, the debugger updates all of the register values and highlights any changes in red. The user can also change the register values while the program is stopped.

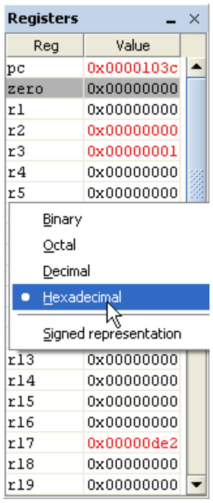
As a demostration of changing a register value, this section of the tutorial will set a breakpoint to halt the program when the hexadecimal display is showing `....dE2.` (`.` represents a blank) and the scroll direction is to the right. When the breakpoint is triggered, you will toggle the shift direction via the **Registers** window and then resume program execution. The detailed steps are as follows:

1. Switch to the **Breakpoints** window.

2. For the breakpoint at `0000103c`, change the condition string to `mem32(0x11fc) == 0x0000de20 && r2 == 1`. The `mem32` syntax is used to read a 32-bit value from memory at the specified address. In this program, the 32-bit value at the address `0x11fc` contains the value to be displayed that will be transferred to the hexadecimal display.

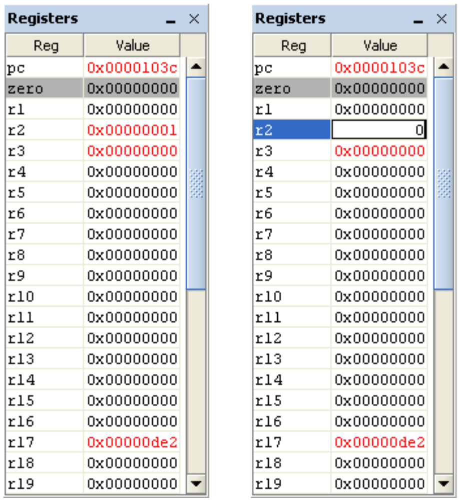
3. Resume program execution and wait for the breakpoint to trigger.

4. The **Registers** window should look similar to the left image of Figure 33. To edit the value of the *r2* register, which controls the scroll direction, double click on its value in the window. This will bring up a text box, as shown in the right image of Figure 33, and you can put in its new value of `0`.



Reg	Value
pc	0x0000103c
zero	0x00000000
r1	0x00000000
r2	0x00000000
r3	0x00000001
r4	0x00000000
r5	0x00000000
r13	0x00000000
r14	0x00000000
r15	0x00000000
r16	0x00000000
r17	0x00000de2
r18	0x00000000
r19	0x00000000

Figure 32. Changing the number format of the register values in the **Registers** window.



Reg	Value
pc	0x0000103c
zero	0x00000000
r1	0x00000000
r2	0x00000001
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
r13	0x00000000
r14	0x00000000
r15	0x00000000
r16	0x00000000
r17	0x00000de2
r18	0x00000000
r19	0x00000000

Reg	Value
pc	0x0000103c
zero	0x00000000
r1	0x00000000
r2	0
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
r13	0x00000000
r14	0x00000000
r15	0x00000000
r16	0x00000000
r17	0x00000de2
r18	0x00000000
r19	0x00000000

Figure 33. Register values after the breakpoint trigger; editing the value of the *r2* register.

5. Press **Enter** or click away from the text box to apply the change.

6. Resume the execution of the program and you should see that the value on the hexadecimal display is scrolling left now.

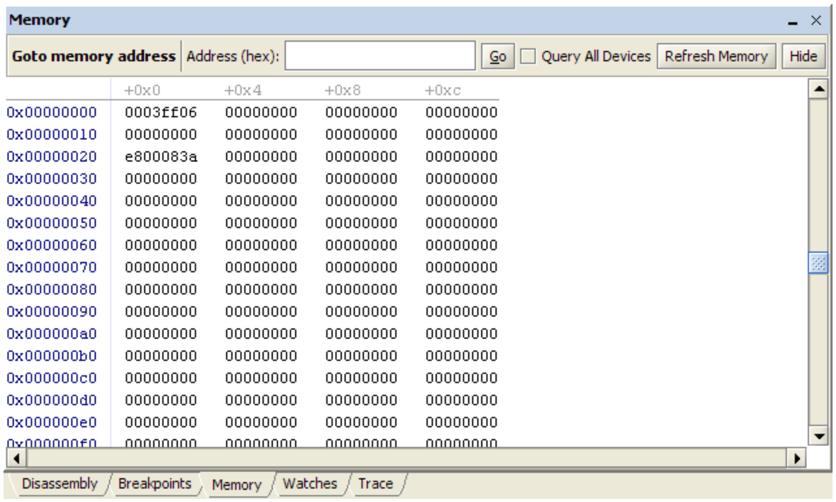
7. Eventually the conditional breakpoint that was set in step 2 will trigger again. Continue on to the next section of the tutorial when this occurs.

12 Examining and Changing Memory Contents

The **Memory** window displays the contents of the system's memory space and allows the user to edit its values. To update the memory displayed, stop the program and then single step once. The memory display will look similar to Figure 34, with hexadecimal addresses in the left-most column and consecutive values displayed horizontally. The numbers at the top of the window represent hexadecimal address offsets from the corresponding address in the left-most column. For example, referring to Figure 34, the address of the last word in the second row is `0x00000010 + 0xc = 0x0000001c`.

The colour of the contents displayed depends on whether that memory location corresponds to an actual memory device, a non-memory device, or is not mapped at all in the system. A memory location that corresponds to a memory device will be coloured black. A location that corresponds to a non-memory device will be coloured blue. A non-mapped memory location will be coloured grey. If in the last time the memory location was refreshed the value changed, the value in that memory location will be displayed in red.

Click the **Refresh Memory** button located in the **Goto memory address** panel of the memory window to refresh the current contents in memory. If a memory location being displayed corresponds to a non-memory device, it will only be updated if the **Query All Devices** checkbox is checked.

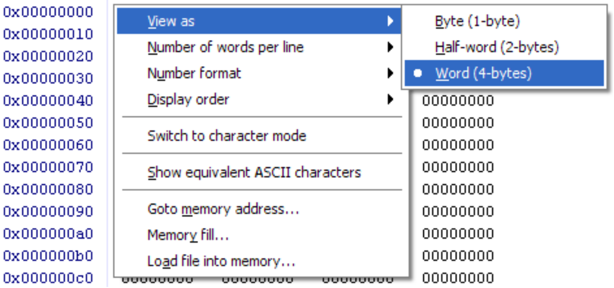


```
            +0x0      +0x4      +0x8      +0xc
0x00000000  0003ff06  00000000  00000000  00000000
0x00000010  00000000  00000000  00000000  00000000
0x00000020  e800083a  00000000  00000000  00000000
0x00000030  00000000  00000000  00000000  00000000
0x00000040  00000000  00000000  00000000  00000000
0x00000050  00000000  00000000  00000000  00000000
0x00000060  00000000  00000000  00000000  00000000
0x00000070  00000000  00000000  00000000  00000000
0x00000080  00000000  00000000  00000000  00000000
0x00000090  00000000  00000000  00000000  00000000
0x000000a0  00000000  00000000  00000000  00000000
0x000000b0  00000000  00000000  00000000  00000000
0x000000c0  00000000  00000000  00000000  00000000
0x000000d0  00000000  00000000  00000000  00000000
0x000000e0  00000000  00000000  00000000  00000000
0x000000f0  00000000  00000000  00000000  00000000
```

Figure 34. Example **Memory** window.

The display is configurable by a number of parameters:

- Memory element size: the display can format the memory contents as bytes, half-words (2-bytes), or words (4-bytes). This can be configured from the context menu accessible by right-clicking on the **Memory** window, as shown in Figure 35.



```
0x00000000
0x00000010
0x00000020
0x00000030
0x00000040                                00000000
0x00000050                                00000000
0x00000060                                00000000
0x00000070                                00000000
0x00000080                                00000000
0x00000090                                00000000
0x000000a0                                00000000
0x000000b0                                00000000
0x000000c0  00000000  00000000  00000000  00000000
```

Figure 35. **View as** menu used to select the memory element size in the **Memory** window.

- Number of words per line: the number of words per line can be configured to make it easier to find memory addresses. This can be configured from the context menu accessible by right-clicking on the **Memory** window, as shown in Figure 36.

- Number format: this is similar to the number format option in the **Register** window. This can also be configured from the context menu accessible by right-clicking on the **Memory** window.

- Display order: the display can display addresses increasing from left-to-right or right-to-left. Configure this option by right-clicking on the **Memory** window, as shown in Figure 37.

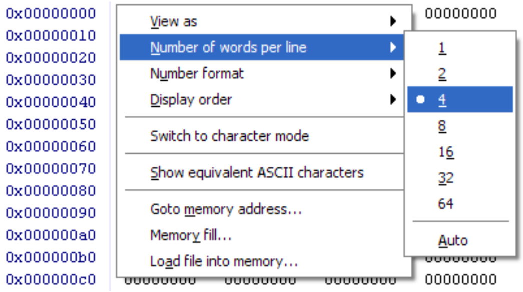


Figure 36. **Number of words per line** menu in the **Memory** window.

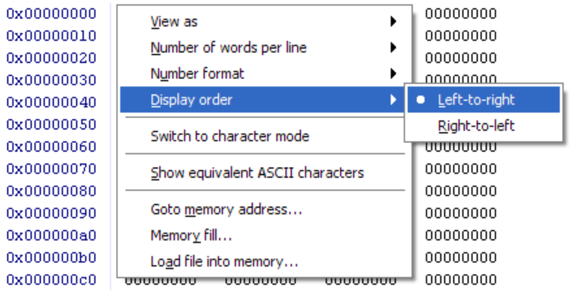
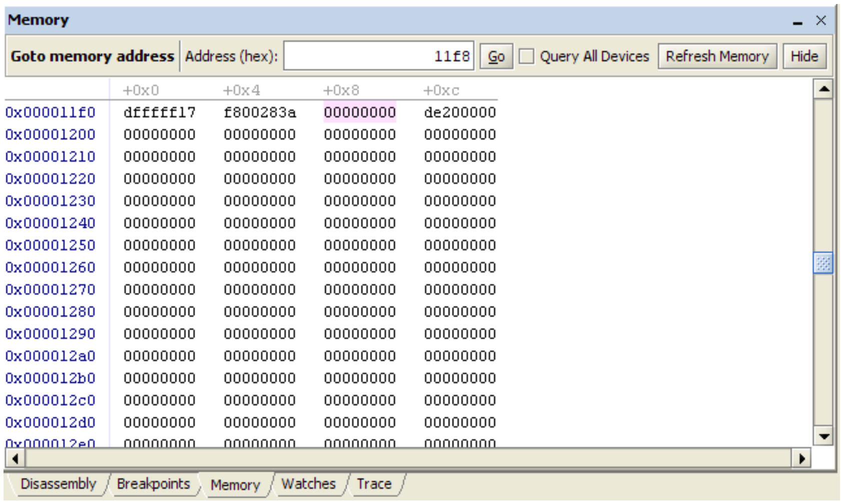


Figure 37. **Display order** menu in the **Memory** window.

Similar to the **Disassembly** window, you can view different memory regions by scrolling using the vertical scroll bar on the right or by using a mouse scroll wheel. There is also a **Goto memory address** panel in the **Memory** window analagous to the **Goto instruction** window. Click the **Actions > Goto memory address...** menu item to display the **Goto memory address** panel. As shown in Figure 38, you can enter any address in hexadecimal, press **Go**, and the Memory window will display that address. In this example, display the `11f8` address, which is where the buffer used by the program is stored.



Address (hex): 11f8

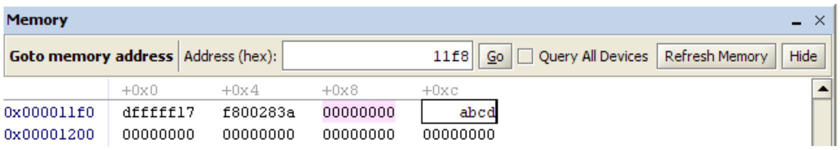
```
            +0x0      +0x4      +0x8      +0xc
0x000011f0  dffffff17 f800283a  00000000  de200000
0x00001200  00000000  00000000  00000000  00000000
0x00001210  00000000  00000000  00000000  00000000
0x00001220  00000000  00000000  00000000  00000000
0x00001230  00000000  00000000  00000000  00000000
0x00001240  00000000  00000000  00000000  00000000
0x00001250  00000000  00000000  00000000  00000000
0x00001260  00000000  00000000  00000000  00000000
0x00001270  00000000  00000000  00000000  00000000
0x00001280  00000000  00000000  00000000  00000000
0x00001290  00000000  00000000  00000000  00000000
0x000012a0  00000000  00000000  00000000  00000000
0x000012b0  00000000  00000000  00000000  00000000
0x000012c0  00000000  00000000  00000000  00000000
0x000012d0  00000000  00000000  00000000  00000000
0x000012e0  00000000  00000000  00000000  00000000
```

Figure 38. **Goto memory address** window.

Since the program reads from this buffer and passes the value to the hexadecimal display, the value shown on the hexadecimal display can be changed by changing the memory value. Proceed as follows:

1. In the row starting at the address `000011f0`, double-click the word under the `+0xc` column. This will bring up a text box to edit the word value at the address `000011fc`.

2. Type in `abcd`, as in Figure 39.



```
            +0x0      +0x4      +0x8      +0xc
0x000011f0  dfffff17  f800283a  00000000  abcd
0x00001200  00000000  00000000  00000000  00000000
```

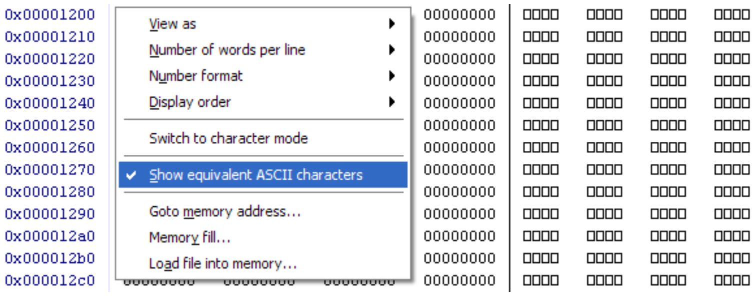
Figure 39. Editing the value at the `000011fc` memory address.

3. Press **Enter** or click away from the text box to apply the memory change.

Upon resuming program execution, you will see that the hexadecimal display is now scrolling `abCd`.

Character display

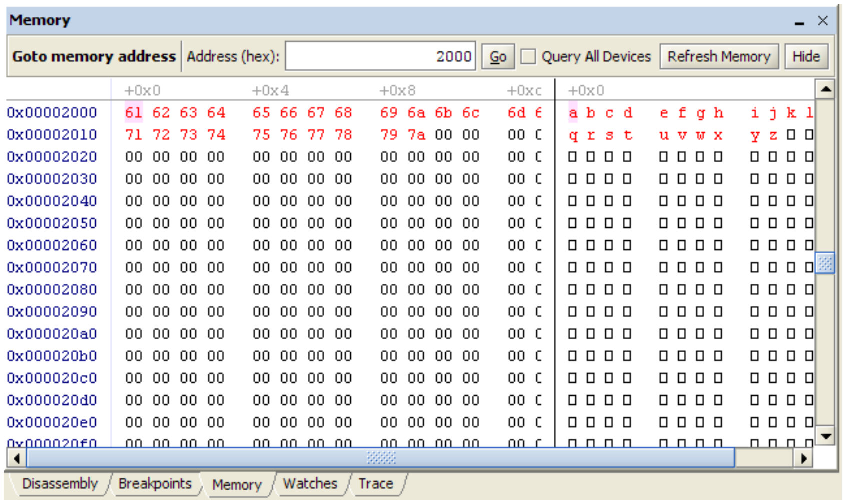
The **Memory** window can also be configured to interpret memory byte values as ASCII characters. This can be done by checking the **Show equivalent ASCII characters** menu item, accessible by right-clicking on the **Memory** window, as shown in Figure 40.



```
0x00001200  00000000  □□□□  □□□□  □□□□  □□□□
0x00001210  00000000  □□□□  □□□□  □□□□  □□□□
0x00001220  00000000  □□□□  □□□□  □□□□  □□□□
0x00001230  00000000  □□□□  □□□□  □□□□  □□□□
0x00001240  00000000  □□□□  □□□□  □□□□  □□□□
0x00001250  00000000  □□□□  □□□□  □□□□  □□□□
0x00001260  00000000  □□□□  □□□□  □□□□  □□□□
0x00001270  00000000  □□□□  □□□□  □□□□  □□□□
0x00001280  00000000  □□□□  □□□□  □□□□  □□□□
0x00001290  00000000  □□□□  □□□□  □□□□  □□□□
0x000012a0  00000000  □□□□  □□□□  □□□□  □□□□
0x000012b0  00000000  □□□□  □□□□  □□□□  □□□□
0x000012c0  00000000  00000000  00000000  00000000  □□□□  □□□□  □□□□  □□□□
```

Figure 40. Checking the **Show equivalent ASCII characters** menu item.

The right side of the figure shows a sample ASCII character display. Usually, it is more convenient to view the memory in bytes and characters simultaneously so that the characters appear in the correct sequence. This can be accomplished by clicking the **Switch to character mode** menu item, which can be seen in Figure 40. A sample character display in the character mode is shown in Figure 41.



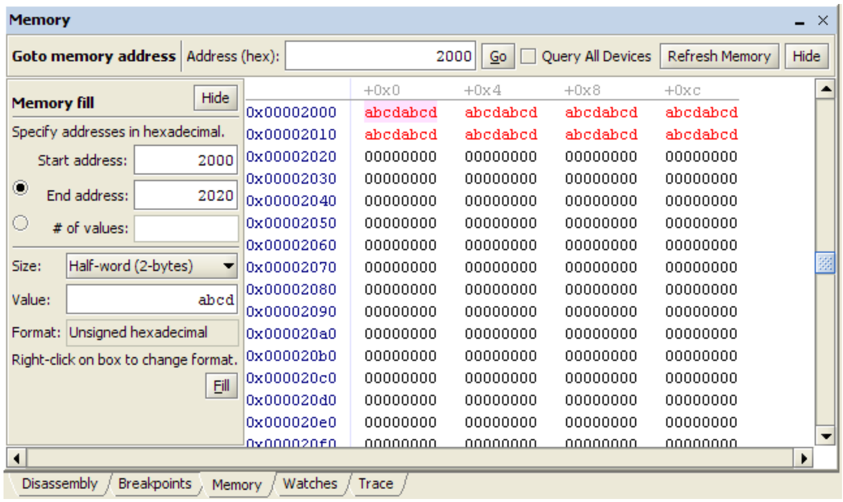
```
            +0x0         +0x4         +0x8         +0xc    +0x0
0x00002000  61 62 63 64  65 66 67 68  69 6a 6b 6c  6d 6   a b c d  e f g h  i j k l
0x00002010  71 72 73 74  75 76 77 78  79 7a 00 00  00 C   q r s t  u v w x  y z □ □
0x00002020  00 00 00 00  00 00 00 00  00 00 00 00  00 C   □ □ □ □  □ □ □ □  □ □ □ □
0x00002030  00 00 00 00  00 00 00 00  00 00 00 00  00 C   □ □ □ □  □ □ □ □  □ □ □ □
0x00002040  00 00 00 00  00 00 00 00  00 00 00 00  00 C   □ □ □ □  □ □ □ □  □ □ □ □
0x00002050  00 00 00 00  00 00 00 00  00 00 00 00  00 C   □ □ □ □  □ □ □ □  □ □ □ □
0x00002060  00 00 00 00  00 00 00 00  00 00 00 00  00 C   □ □ □ □  □ □ □ □  □ □ □ □
0x00002070  00 00 00 00  00 00 00 00  00 00 00 00  00 C   □ □ □ □  □ □ □ □  □ □ □ □
0x00002080  00 00 00 00  00 00 00 00  00 00 00 00  00 C   □ □ □ □  □ □ □ □  □ □ □ □
0x00002090  00 00 00 00  00 00 00 00  00 00 00 00  00 C   □ □ □ □  □ □ □ □  □ □ □ □
0x000020a0  00 00 00 00  00 00 00 00  00 00 00 00  00 C   □ □ □ □  □ □ □ □  □ □ □ □
0x000020b0  00 00 00 00  00 00 00 00  00 00 00 00  00 C   □ □ □ □  □ □ □ □  □ □ □ □
0x000020c0  00 00 00 00  00 00 00 00  00 00 00 00  00 C   □ □ □ □  □ □ □ □  □ □ □ □
0x000020d0  00 00 00 00  00 00 00 00  00 00 00 00  00 C   □ □ □ □  □ □ □ □  □ □ □ □
0x000020e0  00 00 00 00  00 00 00 00  00 00 00 00  00 C   □ □ □ □  □ □ □ □  □ □ □ □
0x000020f0  00 00 00 00  00 00 00 00  00 00 00 00  00 C   □ □ □ □  □ □ □ □  □ □ □ □
```

Figure 41. Character mode display.

You can return to the previous memory view mode by right-clicking and clicking the **Revert to previous mode** menu item.

Memory fill

Memory fills can be performed in the **Memory** window. Click the **Actions > Memory fill...** menu item or right-click on the **Memory** window and click the **Memory fill...** menu item. The **Memory fill** panel will appear on the left-side of the **Memory** window. Simply fill in the desired values and click **Fill**. An example memory fill is shown in Figure 42, which starts at address `0x2000` and ends at `0x2020`. The fill value is 2-bytes in length and has a value of `0xabcd`.



Address (hex): 2000

□ Query All Devices

Memory fill

Specify addresses in hexadecimal.

Start address: 2000

● End address: 2020

○ # of values:

Size: Half-word (2-bytes)

Value: abcd

Format: Unsigned hexadecimal

Right-click on box to change format.

```
            +0x0      +0x4      +0x8      +0xc
0x00002000  abcdabcd  abcdabcd  abcdabcd  abcdabcd
0x00002010  abcdabcd  abcdabcd  abcdabcd  abcdabcd
0x00002020  00000000  00000000  00000000  00000000
0x00002030  00000000  00000000  00000000  00000000
0x00002040  00000000  00000000  00000000  00000000
0x00002050  00000000  00000000  00000000  00000000
0x00002060  00000000  00000000  00000000  00000000
0x00002070  00000000  00000000  00000000  00000000
0x00002080  00000000  00000000  00000000  00000000
0x00002090  00000000  00000000  00000000  00000000
0x000020a0  00000000  00000000  00000000  00000000
0x000020b0  00000000  00000000  00000000  00000000
0x000020c0  00000000  00000000  00000000  00000000
0x000020d0  00000000  00000000  00000000  00000000
0x000020e0  00000000  00000000  00000000  00000000
0x000020f0  00000000  00000000  00000000  00000000
```

Figure 42. **Memory fill** panel being used to perform a memory fill.

Load file data into memory

File data can also be loaded into the memory using the Monitor Program. This can be useful for providing different data sets to a program. To use this functionality, click the **Actions > Load file into memory...** menu item or right-click on the **Memory** window and click the **Load file into memory...** menu item. The **Load file** panel will appear on the left-side of the **Memory** window. Click **Browse...** to select the file to load. There are three types of files that are supported:

1. *Delimited hexadecimal value files*: These files are plain ASCII files aimed at loading numerical data into memory. The file format is defined as follows:

   - All memory values are specified as hexadecimal values, with an optional sign in front of the value.
   - Individual memory values are separated by a *delimiter character*.
   - There can be multiple lines of delimited values in the file.

   For the purpose of this tutorial, select the *<TUTORIAL_FILES>\sample.csv* file. This file is a *comma-*delimited hexadecimal value file.

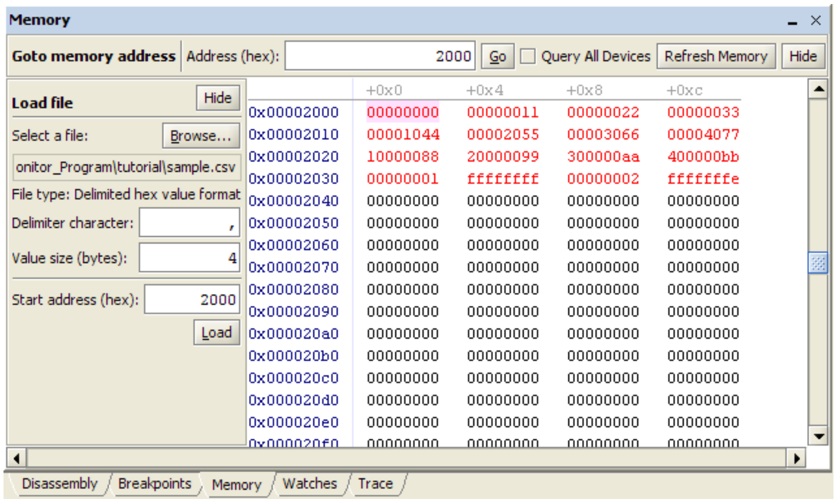
2. *Intel HEX-format files*: These files are in another special format aimed at loading numerical data into memory. You can use the Quartus II software to create Intel HEX-format files. When doing so, ensure that the word size is set to 8-bits.

3. *Binary files*: These files are loaded byte-by-byte without any interpretation. This is useful for loading binary data, such as audio or image files.

After selecting a file, a start address needs to be specified to indicate where to start loading the file data. In this example, specify `2000`. In the case of delimited hexadecimal value files, two additional parameters need to be specified:

1. *Delimiter character*: This is the character that separates consecutive values in the file. In this example, a comma (`,`) separates the values in the file.
2. *Value byte size*: The byte size of each value element. The Monitor Program will truncate values to be within the specified byte size. Specify a value of `4`.

After all the parameters have been specified, click **Load** and you should see the loaded values in the **Memory** window, as shown in Figure 43.



Memory

Goto memory address | Address (hex): 2000 | Go | ☐ Query All Devices | Refresh Memory | Hide

Load file | Hide

Select a file: Browse...

onitor_Program\tutorial\sample.csv

File type: Delimited hex value format

Delimiter character: ,

Value size (bytes): 4

Start address (hex): 2000

Load

```
            +0x0      +0x4      +0x8      +0xc
0x00002000  00000000  00000011  00000022  00000033
0x00002010  00001044  00002055  00003066  00004077
0x00002020  10000088  20000099  300000aa  400000bb
0x00002030  00000001  ffffffff  00000002  fffffffe
0x00002040  00000000  00000000  00000000  00000000
0x00002050  00000000  00000000  00000000  00000000
0x00002060  00000000  00000000  00000000  00000000
0x00002070  00000000  00000000  00000000  00000000
0x00002080  00000000  00000000  00000000  00000000
0x00002090  00000000  00000000  00000000  00000000
0x000020a0  00000000  00000000  00000000  00000000
0x000020b0  00000000  00000000  00000000  00000000
0x000020c0  00000000  00000000  00000000  00000000
0x000020d0  00000000  00000000  00000000  00000000
0x000020e0  00000000  00000000  00000000  00000000
0x000020f0  00000000  00000000  00000000  00000000
```

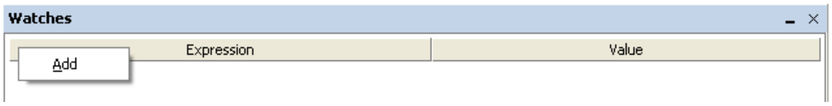
Disassembly | Breakpoints | Memory | Watches | Trace

Figure 43. **Load file** panel being used to load a delimited hexadecimal value file into memory.

13 Setting a Watch Expression

Watch expressions are simply expressions that are re-evaluated each time program execution is stopped. They provide a convenient means to keep track of the value of multiple expressions of interest. To add a watch expression:

1. Switch to the **Watches** window.
2. Right-click on the gray bar, like in Figure 44, and click **Add**.

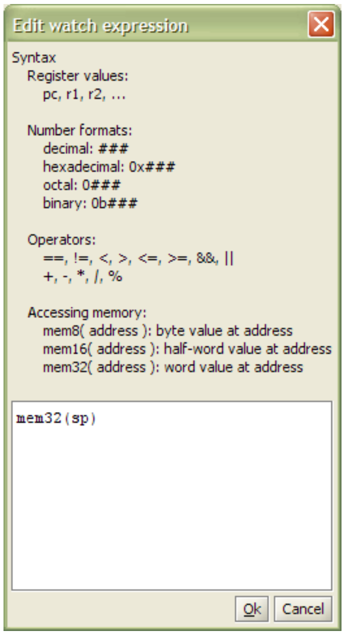


Watches

Expression	Value

Figure 44. Adding a watch expression via the **Add** menu item.

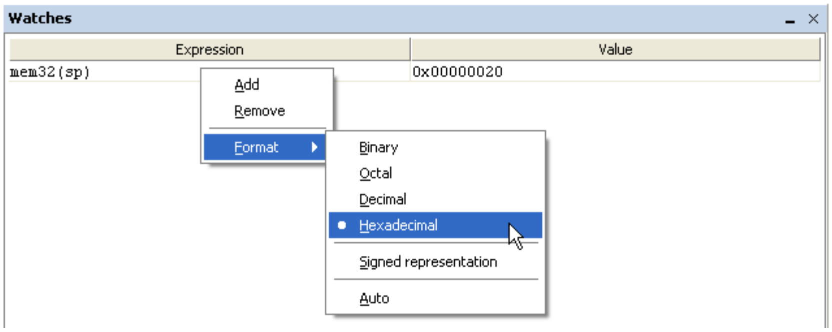
3. The **Edit Watch Expression** window will appear, like in Figure 45. Enter the desired watch expression, such as `mem32(sp)`, which will display the full-word value at the current stack pointer address.
4. Click **Ok**. The watch expression and its current value will show up in the table.
5. The number format of the displayed value can be changed by right-clicking on the row for that value, as shown in Figure 46.
6. As you repeatedly run the program and stop it at various points, the watch expression will be re-evaluated each time and its value shown in the table of watch values.



```
Syntax
   Register values:
      pc, r1, r2, ...

   Number formats:
      decimal: ###
      hexadecimal: 0x###
      octal: 0###
      binary: 0b###

   Operators:
      ==, !=, <, >, <=, >=, &&, ||
      +, -, *, /, %

   Accessing memory:
      mem8( address ): byte value at address
      mem16( address ): half-word value at address
      mem32( address ): word value at address
```

```
mem32(sp)
```

Figure 45. The **Edit Watch Expression** window.



Expression	Value
mem32(sp)	0x00000020

Figure 46. Changing the number format of a watch value.

14 Examining the Instruction Trace

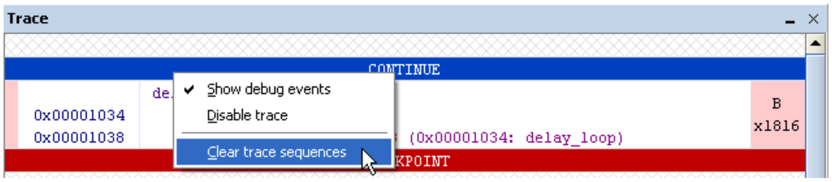
An instruction trace is a hardware-level mechanism to record a log of all recently executed instructions. The *Nios II JTAG Debug Module* has the instruction trace capability, but only if a Level 3 or higher debugging level is selected in the *SOPC Builder* configuration of the JTAG Debug Module[2]. If the required JTAG Debug Module is not present, a message will be shown in the **Info & Errors** window after loading the program. The message will say *INFO: Program Trace not enabled, because trace requires the Nios II processor to be configured with JTAG Debug Level 3.* This message can be safely ignored if there is no trace support in the Nios II system.

The example system used in this tutorial has a Level 3 JTAG Debug Module and so it supports the instruction trace. To demonstrate the **Trace** window in the Monitor Program, follow the steps below:

1. Switch to the **Trace** window. There is probably already something there because the instruction trace has been running since the program was loaded.

2. For this part of the tutorial, first clear the trace by right-clicking in the **Trace** window and clicking **Clear trace sequences**, as shown in Figure 47.

[2] See chapter 4 in the *Nios II Processor Reference Handbook* for more information about the configuration settings of the JTAG Debug Module. The Handbook can be found in the Nios II Literature section of Altera's web site.

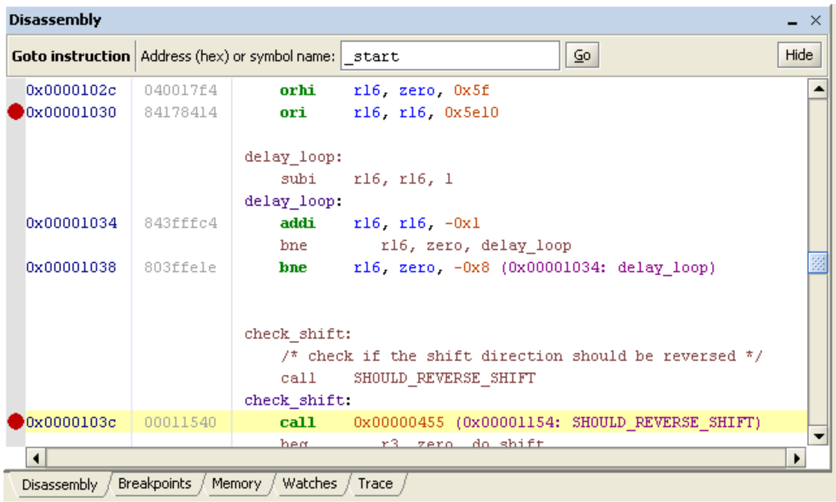


```
de
0x00001034
0x00001038                    (0x00001034: delay_loop)
```

Figure 47. Clear the trace sequences.

3. Remove all existing breakpoints and add two new instruction breakpoints at `1030` and `103c`. These two breakpoints will be used to skip the `delay_loop` section of the code, which does not produce very interesting results for the instruction trace.

4. Run the program until the breakpoint at `103c` is triggered. Your **Disassembly** window should look similar to Figure 48 with the breakpoints set and the instruction at `103c` highlighted.

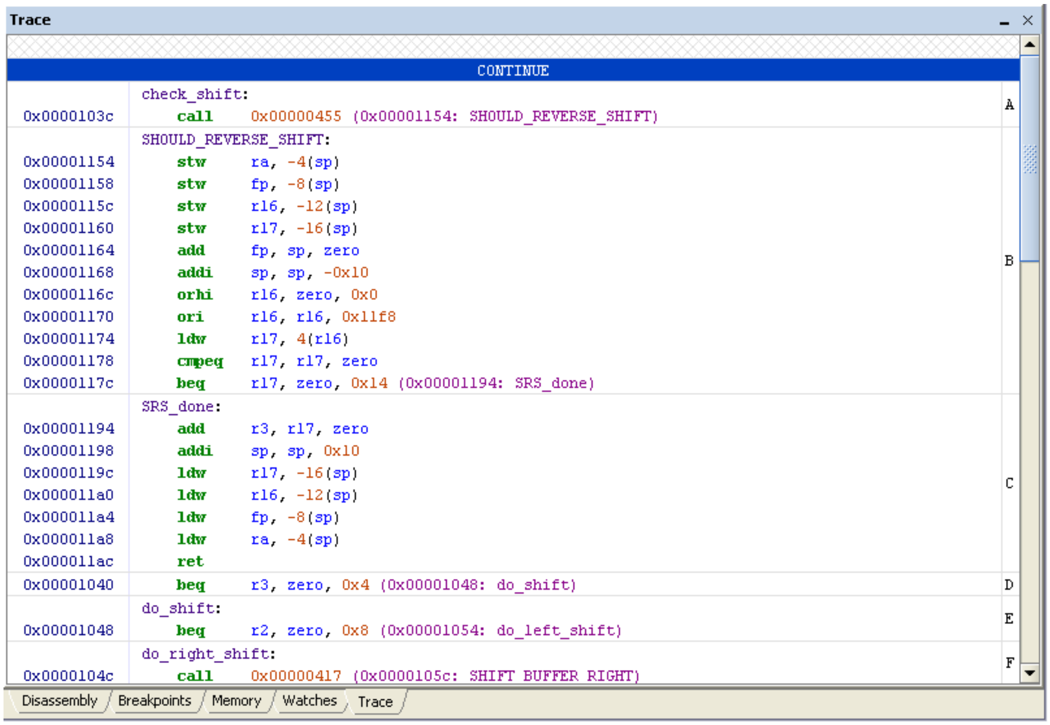


```
0x0000102c   040017f4      orhi    r16, zero, 0x5f
0x00001030   84178414      ori     r16, r16, 0x5e10

                        delay_loop:
                           subi    r16, r16, 1
                        delay_loop:
0x00001034   843fffc4      addi    r16, r16, -0x1
                           bne        r16, zero, delay_loop
0x00001038   803ffe1e      bne     r16, zero, -0x8 (0x00001034: delay_loop)


                        check_shift:
                           /* check if the shift direction should be reversed */
                           call    SHOULD_REVERSE_SHIFT
                        check_shift:
0x0000103c   00011540      call    0x00000455 (0x00001154: SHOULD_REVERSE_SHIFT)
                           beq        r3, zero, do_shift
```

Figure 48. **Disassembly** window display after setting breakpoints.

5. Continue the program's execution; it will execute the code to do some checks on the buffer as well as actually shift the contents of the buffer by one hexadecimal digit. The hexadecimal display will also be updated by a call to `UPDATE_HEX_DISPLAY` before the breakpoint at `1030` is triggered.

6. Switch back to the **Trace** window. The window will now be displaying all the instructions executed starting from the first breakpoint to the second breakpoint, as indicated in Figure 49. As seen in the figure, the trace is divided into *instruction blocks*. Although it is not evident in this trace, the Monitor Program will try to find repeated instruction blocks and common sequences to reduce the length of the trace on the screen.



```
              check_shift:
0x0000103c        call    0x00000455 (0x00001154: SHOULD_REVERSE_SHIFT)
              SHOULD_REVERSE_SHIFT:
0x00001154        stw     ra, -4(sp)
0x00001158        stw     fp, -8(sp)
0x0000115c        stw     r16, -12(sp)
0x00001160        stw     r17, -16(sp)
0x00001164        add     fp, sp, zero
0x00001168        addi    sp, sp, -0x10
0x0000116c        orhi    r16, zero, 0x0
0x00001170        ori     r16, r16, 0x11f8
0x00001174        ldw     r17, 4(r16)
0x00001178        cmpeq   r17, r17, zero
0x0000117c        beq     r17, zero, 0x14 (0x00001194: SRS_done)
              SRS_done:
0x00001194        add     r3, r17, zero
0x00001198        addi    sp, sp, 0x10
0x0000119c        ldw     r17, -16(sp)
0x000011a0        ldw     r16, -12(sp)
0x000011a4        ldw     fp, -8(sp)
0x000011a8        ldw     ra, -4(sp)
0x000011ac        ret
0x00001040        beq     r3, zero, 0x4 (0x00001048: do_shift)
              do_shift:
0x00001048        beq     r2, zero, 0x8 (0x00001054: do_left_shift)
              do_right_shift:
0x0000104c        call    0x00000417 (0x0000105c: SHIFT_BUFFER_RIGHT)
```

Figure 49. A partial view of the trace between the two breakpoints.

15 Using Project Files

Project files store the settings for a particular project, such as the system and program specifications (see Section 3). A project file is created for you in the project's directory whenever you create a new project. This allows you to easily reload a program that you were working on previously by opening the project file associated with it.

There are four commands for working with projects available under the **File** menu, as shown in Figure 50:

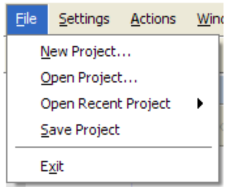


Figure 50. Menu options for working with projects.

1. **New Project**: Opens the **New Project Wizard**, which allows you to create a new project. Any previously opened project will be closed.

2. **Open Project**: Displays a window to select an existing project file and loads it.

3. **Open Recent Project**: This submenu will display the five most recently-used project files. Use this menu to open an existing project.

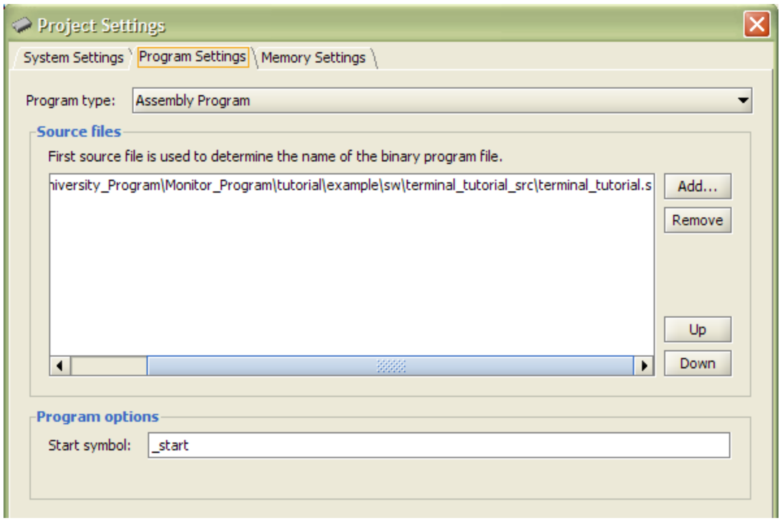
4. **Save Project**: Saves the current project. This is used to save the project's settings when they are modified via the **Project Settings** window.

16 Using the Terminal

This section of the tutorial demonstrates the functionality of the **Terminal** window. The **Terminal** window allows a program to interact with the user using character I/O. The window will interact with the program using a terminal device connected to the Nios II system. The terminal device can be selected in the **Terminal device** drop-down list, available in the **New Project Wizard** and the **Project Settings** windows.

This section uses a different source file, and thus the program needs to be reconfigured. A project file is provided to automatically load the correct settings for running the terminal example program. To load the project file, proceed as follows:

1. Click the **Actions > Disconnect** menu item or click the toolbar button to end the current GDB debugging session.
2. Click the **File > Open Project...** menu item.
3. Select the file located at *<TUTORIAL_FILES>\example\sw\terminal_tutorial.ncf* and click **Open**.
4. This step is not required but it is useful as a demonstration to verify that the correct program is now specified. Open the **Project Settings** window by clicking on the **Settings > Program Settings...** menu item, or by clicking on the toolbar button. You should see one source file listed, similar to Figure 51.



Program type: Assembly Program

First source file is used to determine the name of the binary program file.

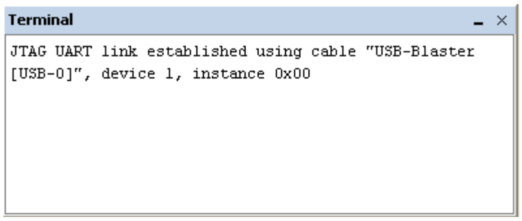
iversity_Program\Monitor_Program\tutorial\example\sw\terminal_tutorial_src\terminal_tutorial.s

Start symbol: _start

Figure 51. **Project Settings** window for the **Terminal** window tutorial.

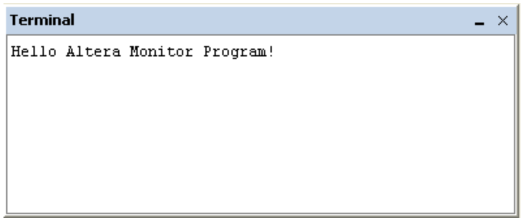
Now that the project for the terminal example program was opened, this section of the tutorial will demonstrate the functionality of the **Terminal** window in the Monitor Program.

1. **Compile & Load** the program. If the compilation fails, make sure that you have write permissions to the directory where the terminal example program resides. This may be the case if you are running the Monitor Program from a shared location on a public computer, such as in a university laboratory. In this case, try copying the tutorial example directory (*<TUTORIAL_FILES>\example*) to another location, and re-open the project from there. The project file stores the paths to all files (such as the system PTF file and the program source files) relative to itself, so as long as the location of the system and program files remains the same relative to the project, it can be opened from any location.
2. Once the program has been successfully loaded, the **Terminal** window will indicate that a connection has been established, similar to Figure 52.
3. Run the program. The program will clear the Terminal window and write the string `Hello Altera Monitor Program`, as shown Figure 53.



```
JTAG UART link established using cable "USB-Blaster
[USB-0]", device 1, instance 0x00
```

Figure 52. The **Terminal** window after a connection has been established to the JTAG UART on the board.



```
Hello Altera Monitor Program!
```

Figure 53. Output from the terminal tutorial program.

4. At this point of the program's execution, the program is ready to accept terminal input and echo it back. Click on the **Terminal** window and type something. You will see that what you type shows up in the window. This is done by the code in the program; the Terminal window, by default, does *not* automatically echo what is typed.

As mentioned in step 3, the program clears the Terminal window. This is accomplished through a special character sequence that is interpreted by the Monitor Program to be a *terminal command*. The Monitor Program's Terminal window supports a subset of the VT100 terminal commands. The supported commands are listed in Table 2. Note that `<ESC>` is actually one character, with the ASCII value `0x1B`.

Character Sequence	Description
`<ESC>[2J`	Erases everything in the Terminal window
`<ESC>[7h`	Enable line wrap mode
`<ESC>[7l`	Disable line wrap mode
`<ESC>[#A`	Move cursor up by # rows or by one row if # is not specified
`<ESC>[#B`	Move cursor down by # rows or by one row if # is not specified
`<ESC>[#C`	Move cursor right by # columns or by one column if # is not specified
`<ESC>[#D`	Move cursor left by # columns or by one column if # is not specified
`<ESC>[`$\#_1$`;`$\#_2$`f`	Move the cursor to row $\#_1$ and column $\#_2$
`<ESC>[H`	Move the cursor to the home position (row 0 and column 0)
`<ESC>[s`	Save the current cursor position
`<ESC>[u`	Restore the cursor to the previously saved position
`<ESC>[7`	Same as `<ESC>[s`
`<ESC>[8`	Same as `<ESC>[u`
`<ESC>[K`	Erase from current cursor position to the end of the line
`<ESC>[1K`	Erase from current cursor position to the start of the line
`<ESC>[2K`	Erase entire line
`<ESC>[J`	Erase from current line to the bottom of the screen
`<ESC>[2J`	Erase from current cursor position to the top of the screen
`<ESC>[6n`	Queries the cursor position. A reply is sent back in the format `<ESC>[`$\#_1$`;`$\#_2$`R`, corresponding to row $\#_1$ and column $\#_2$.

Table 2. VT100 commands supported by the **Terminal** window in the Monitor Program.

17 Using Device Drivers (Advanced)

Many of the IP cores that are used by the Altera SOPC Builder to create systems (including the *University Program IP Cores*) come bundled with device drivers that can be used by programmers to access their functionality. The device drivers are based on the hardware abstraction layer (HAL) system library. To use the HAL, the Nios II Software Build Tools generate a special Board Support Package (BSP) for the selected Nios II system. This allows easy access to the functionality of the HAL that is relevant to the specific system.

To make use of this feature, create a Monitor Program project, specifying **Program with Device Driver Support** as the program type. Usually, you will write C or C++ programs to take advantage of the HAL, but you can also specify assembly language source files if required.

When you compile your program, a BSP will be generated for your system in the \BSP subdirectory of the project directory. Your compiled program will then be linked with the BSP to produce the final binary program executable. Upon subsequent program compilations, the BSP will be re-generated and re-compiled only if the Nios II system has changed since the last compilation.

To re-generate the BSP manually, use the **Actions > Regenerate Device Drivers (BSP)** menu item. This is useful if the system has not changed, but the device drivers for some of the IP cores used by the system have changed. Re-generating the BSP in this case is the only way to ensure that your program uses the latest drivers.

To get access to the functionality of the HAL in your program, you will need to include the relevant header files. In addition to providing access to device drivers, the HAL supports a lot of other useful functionality. For detailed information about using the HAL, consult the *Nios II Software Developer's Handbook*, which can be found in the *Nios II Processor* subsection of the *Literature* section of Altera's website.

18 Running Multiple Instances of the Monitor Program (Advanced)

In some cases, it may be useful to run more than one instance of the Monitor Program on the same computer. For example, the selected system may contain more than one Nios II processor. An instance of the Monitor Program is required to run and debug programs on each available processor. As described in Section 3, it is possible to select a particular processor in a system via the **Processor** drop-down list in the **New Project Wizard** and **Project Settings** windows.

The Monitor Program uses GDB Server to interact with the Nios II system, and connects to the GDB Server using TCP ports. By default, the Monitor Program uses port 2399 as the base port, and to connect to each processor in a system, the Monitor Program will attempt to use a port located at a fixed offset from this base port. For example, a single system consisting of 4 processors corresponds to ports 2399-2402.

However, the Monitor Program does not detect any ports that may already be in use by other applications. If the Monitor Program fails to connect to the GDB Server due to a port conflict, you may try to change the base port number by creating an environment variable called `ALTERA_MONITOR_DEBUGGER_BASE_PORT` and specifying a different number.

It is also possible to have more than one board connected to the host computer. As described in Section 3, it is possible to select a particular board via the **Host connection** drop-down list in the **New Project Wizard** and **Project Settings** windows. In this case, a separate instance of the Monitor Program is needed to interact with each processor on each physical board. By default, the Monitor Program assumes a maximum of 10 Nios II processors per board. This means that ports 2399-2408 are used by the Monitor Program for the first board connected to the computer, and the first processor on the second board will use port 2409.

You may specify a different value for the maximum number of processors per Nios II system by creating an environment variable called `ALTERA_MONITOR_DEBUGGER_MAX_PORTS_PER_CABLE` and specifying a different number. This is useful if any of the systems that you want to use contain more than 10 Nios II processors. It is also useful if you experience a port conflict and none of your systems contain 10 or more processors. In this case, decreasing this number (in conjunction with changing the base port number) may provide a solution.

19 The GDB Server Panel (Advanced)

To see this panel, select the GDB Server panel of the Monitor Program. This window will display the low level commands being sent to the GDB Server, used to interact with the Nios II system being used. It will also show the responses that GDB sends back. The Monitor Program provides the option of sending your own GDB commands to the debugger. Consult online resources to learn what commands are available.

This document is being provided on an "as-is" basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.