Guidelines for Developing a Nios II HAL Device Driver

This application note explains the process of creating and debugging a hardware abstraction layer (HAL) software device driver. An example of a HAL software device driver, called `my_uart_driver`, illustrates various software development stages. The example driver targets the **Altera_Avalon_UART** device, connected through a Vectored Interrupt Controller (VIC) to the Nios® II processor.

This application note helps you with custom device driver development for Nios II systems in the following ways:

- Shows the development process in steps, from sending bits out the transmit pin in `main()` through the construction of device access macros and automatic device initialization in `alt_sys_init()`

- Shows how to develop a driver with the command-line based Nios II Software Build Tools (SBT)

- Shows how to create applications and a board support package (BSP) based on your driver

- Shows how to import and debug the applications and BSP with the Nios II Software Build Tools for Eclipse

- Explains interrupt latency, interrupt nesting, and determinism

- Identifies system calls that cannot be included in a device driver interrupt service routine (ISR)

- Describes debugging tips and techniques, such as identifying UART transmission errors

This application note uses the Nios II Embedded Evaluation Kit (NEEK), Cyclone® III Edition, as an example hardware platform.

For more information about the HAL, refer to the *Overview of the Hardware Abstraction Layer* chapter in the *Nios II Software Developer's Handbook*. For more information about interrupt latency, refer to the *Exception Handling* chapter in the *Nios II Software Developer's Handbook*.

101 Innovation Drive
San Jose, CA 95134
www.altera.com

Subscribe

Prerequisites for HAL Device Driver Development

This document targets advanced systems developers with a basic understanding of the following concepts:

■ Nios II application development, including creating and building software applications and BSPs with the Nios II SBT, including the Nios II SBT for Eclipse.

■ The Quartus® II software, including opening Quartus II projects that match the target board, launching the Qsys system integration tool, and examining various peripheral settings.

■ Using the Quartus II Programmer tool to program an SRAM Object File (**.sof)** to an FPGA through an Altera® USB-Blaster™ download cable.

To gain the minimum prerequisite knowledge, refer to the following resources:

■ The *Nios II Hardware Development Tutorial*, available on the Nios II Hardware Development Design Example page of the Altera website

■ The *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*

■ The *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*

Using the HAL Architecture and Services

The HAL application programming interface (API) provides a standard POSIX-like interface to the hardware, abstracting the hardware details from upper-level clients of the HAL, such as operating systems, networking stacks, or Nios II applications. The HAL provides a variety of generic device classes, including character-mode, file subsystem, Ethernet, timestamp and system timers, direct memory access (DMA), and flash memory. The **Altera_Avalon_UART** is a member of the character-mode class of HAL devices. The HAL has an API for character-mode class devices, which you can use to manipulate the **Altera_Avalon_UART**. Mutual exclusion resources are available, provided either by MicroC/OS-II (if present) or by the HAL. These services include semaphores and event flags. When the HAL device driver makes calls to these resources, the calls are simply translated to non-operations when the multi-threading services are not available.

For additional information about HAL services, refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter in the *Nios II Software Developer's Handbook*.

For additional information about the HAL API, refer to the *HAL API Reference* chapter in the *Nios II Software Developer's Handbook*.

Software Requirements for the Driver Example

The following components are required:

■ Quartus® II software version 11.0 SP1 or higher.

■ Nios II Embedded Design Suite (EDS) version 11.0 SP1 or higher.

■ The **an459-design-files.zip** archive.

The **an459-design-files.zip** archive contains a hardware design example for the NEEK, along with software examples and a driver example named my_uart_driver. The following software example projects are included:

■ The **bit_bang_uart** application

■ The **hello_world_my_uart** application

■ The **hal_my_uart** BSP

> **an459-design-files.zip** is available on the HAL Device Drivers Design Example page of the Altera website.

HAL Device Drivers and Components

This application note discusses Nios II device drivers and components at several levels. To understand these levels, you need to be familiar with the following concepts:

■ *HAL-compatible component*—A design for a piece of hardware that can be incorporated into a Nios II system and supported by the HAL. A component is an abstract *IP core* that can be configured for a specific application. A component has abstract parameters, such as base address. These parameters have no specific value until a component instance is created. The Altera Avalon UART is an example of a component.

■ *Component instance*—A component that is instantiated in a system. Component instance parameters have specific values, assigned at the time of instantiation. For example, an instance of the Altera Avalon UART must have a specific base address. There can be multiple instances of a component in a system. Each instance has a unique name, such as uart1, assigned in Qsys.

■ *HAL device driver*—A piece of software written to interface a component to the HAL. A device driver supports a specific component. A Nios II BSP contains a single device driver for each component found in the system. If there are multiple instances of one component, they are all supported by a single driver. In this application note, my_uart_driver is an example of a device driver.

■ *HAL generic device model class*—A group of device drivers with similar characteristics and a common high-level API. Generic device models allow you to use a consistent set of API calls with a variety of hardware designs. The class of character-mode devices is an example of a HAL device class, and the printf() function is an example of a function call supported by this device class.

> For more information about HAL device classes, refer to "HAL Architecture" in the *Overview of the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook.*

Developing the HAL UART Device Driver

This section walks you through creation of the my_uart_driver device driver as an example of a HAL device driver.

Preparing the bit_bang_uart Application and hal_my_uart BSP

First, you need to set up a development and debugging environment for the UART. This example uses the NEEK with an accompanying design example in **an459-design-files.zip**.

Follow these steps to build the **bit_bang_uart** project:

1. Get the most up-to-date version of **an459-design-files.zip**. **an459-design-files.zip** is available on the HAL Device Drivers Design Example page of the Altera website.

2. Unzip **an459-design-files.zip** to a working directory, such as **C:/my_design**. This application note refers to this directory as *<my_design>*. Be sure to preserve the directory structure of the extracted software archive. Extraction creates a directory structure tree under *<my_design>* with the following four subdirectories:

   ■ **ip/my_uart**

   ■ **software_examples/bsp/hal_my_uart**

   ■ **software_examples/app/bit_bang_uart**

   ■ **software_examples/app/hello_world_my_uart**

   ☞ The working directory name you choose must not contain any spaces.

☞ After extracting **an459-design-files.zip**, refer to *<my_design>*/**readme.txt** for a list of any required software patches or other updated information. If a patch is required, install it according to the instructions in **readme.txt**.

Preparing the my_uart_driver Device Driver

This section provides some background on how the my_uart_driver device driver is associated with a component instance. You specify the name of the directory to store both the software device drivers and the custom components. The name should be descriptive enough to identify the custom component. The directory must be under the *<my_design>*/**ip** directory. The librarian searches for user component files named *<component_name>*_**sw.tcl** in directories below this **ip** directory.

Every HAL device driver has a software description file. The software description file name is of the form *<component_name>*_**sw.tcl**. This name must match the corresponding *<component_name>*_**hw.tcl** file generated by the Component Editor.

The my_uart_driver device driver's software description file is *<my_design>*/**ip/my_uart/my_uart_sw.tcl**.

☞ All components generated by the Component Editor have a *<component_name>*_**hw.tcl** file. However, certain components provided by Altera, such as the **Altera_Avalon_UART**, are generated outside the Component Editor, and therefore do not have a *<component_name>*_**hw.tcl** file.

For additional information about creating device driver Tcl scripts, refer to the "Driver and Software Package Tcl Script Creation" section of the *Developing Device Drivers for the Hardware Abstraction Layer* chapter in the *Nios II Software Developer's Handbook*.

Altera provides an additional tool with the Nios II processor, the System Console, that is useful for testing component instances and software device drivers, and for constructing BSPs. This application note does not describe the System Console.

For information about the System Console, refer to the *System Console User Guide*.

Configuring the Altera_Avalon_UART Component

In this section, you configure the **Altera_Avalon_UART** hardware component in Qsys. Perform the following steps:

1. Start the Quartus II software. On the File menu, click **Open Project**.

2. Browse to *<my_design>*.

3. Select the Quartus II project file **AN459.qpf**, and click **Open**.

4. On the Tools menu, click **Qsys** and open **system.qsys**.

5. In Qsys, in the **Module Name** column, double-click on uart1.

6. In the **UART (RS-232 Serial Port) - uart1** dialog box, verify the baud rate is set to 115200 bps, as shown in Figure 1.

Figure 1. Verify UART Baud Rate
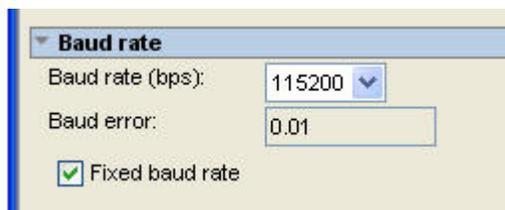
7. Click **Finish**.

8. In the **System Contents** tab of Qsys, verify the value for the UART base address.

   This design example uses a value of 0x80 for the UART's register base address. If you use a hardware design other than the design example accompanying this application note, the value of the UART's register base address might be different. Open Qsys and find the UART base address for your board. Figure 2 shows the base address for the UART used in this example.

9. Click **Generate** in Qsys to regenerate the system.

10. Recompile the Quartus II project.

11. In the Quartus II software, on the Tools menu, click **Programmer**.

12. To program the **AN459.sof** image to the development board, ensure that **Program/Configure** is turned on and click **Start**.

13. In Qsys, on the Tools menu, click **Nios II Command Shell**.

14. Change the directory to *<my_design>*/**software_examples/app/bit_bang_uart**.

Figure 2. uart1 Peripheral Register Base Address

15. Run the **create-this-app** script as follows:

```
./create-this-app↵
```

This step might take several minutes to complete.

The **create-this-app** script specifies the BSP named **hal_my_uart**. The **hal_my_uart** BSP associates the component instance uart1 with the software driver my_uart_driver, as shown in the **summary.html** excerpt in Figure 3. The **create-this-bsp** script for the **hal_my_uart** BSP selects component instance uart1 for the stdio device. Build messages are stored in **bsp/hal_my_uart/hal_my_uart_build_log.txt**.

☞   The compiler might report a small number of harmless warnings when you build **bit_bang_uart**. For a list of expected warnings, refer to the *<my_design>*/**readme.txt** file, extracted from **an459-design-files.zip**.

16. Change the directory to *<my_design>*/**software_examples/bsp/hal_my_uart**.

17. Edit **alt_sys_init.c** in your favorite editor. The **vi** editor is available from the Nios II Command Shell.

Figure 3. Driver Mapping in summary.html

Drivers

Module Name	Driver	Version
uart1	my_uart_driver	default

18. Disable the automatic invocation of the HAL UART device driver initialization function by commenting out invocations of the following macros in **alt_sys_init.c**:

- `ALTERA_AVALON_UART_INSTANCE()`

- `ALTERA_AVALON_UART_INIT()`

19. Save **alt_sys_init.c**.

20. Rebuild the **bit_bang_uart** project by changing the directory back to *<my_design>*/**software_examples/app/bit_bang_uart**, and executing `make`.

21. Connect a serial cable from the 9-pin console port on the Nios development board to an RS-232 serial port on your development host computer.

22. Ensure that your host serial port is configured with the following settings:

- 115200 baud

- 8 data bits

- 1 stop bit

- No parity

- No flow control

Importing Projects

Follow these steps to import the **bit_bang_uart** application project and the **hal_my_uart** BSP project:

1. In Qsys, in the Nios II menu click **Nios II Software Build Tools for Eclipse** to launch the Nios II SBT for Eclipse.

2. On the File menu, click **Import**. The **Import** dialog box appears.

3. Expand the **Nios II Software Build Tools Project** folder, and select **Import Nios II Software Build Tools Project**.

4. Click **Next**. The **Import Software Build Tools Project** wizard appears.

5. Next to **Project Location**, click **Browse**. Navigate to and select the *<my_design>*/**software_examples/app/bit_bang_uart** directory.

6. Click **OK**.

7. Name the project **bit_bang_uart**.

8. Turn on **Managed Project** to have Nios II SBT for Eclipse manage your makefile for you.

9. Click **Finish**. The wizard imports the **bit_bang_uart** application project.

10. Repeat steps 2 through 9, but at step 5 instead import the
    <my_design>/**software_examples/bsp/hal_my_uart** BSP**.**

   ☞   For additional information about importing Nios II SBT command-line
        projects, refer to "Importing a Command-Line Project" in the *Getting Started
        with the Graphical User Interface* chapter of the *Nios II Software Developer's
        Handbook.*

Understanding the Example Software

This example demonstrates how you can verify hardware functionality by specifying
an explicit, hard-coded memory address. In the Nios II SBT for Eclipse **Project
Explorer** tab, in the Nios II Project Explorer view, expand the **bit_bang_uart** project,
and open **bit_bang_uart.c**. In the main() procedure of **bit_bang_uart.c**, you can see
that the first call to IOWR() uses a hard-coded base address of 0x80 for uart1.

Using a hard-coded address can be helpful when you first bring up new hardware, to
rule out any software errors, such as C pointer dereference software coding errors, in
obtaining the peripheral's memory-mapped registers base address. This technique
provides confidence that your software is referencing the actual hardware peripheral
register.

Preview: Customizing the Design

This section describes typical modifications you might make to the software after you
have verified that it works with hard-coded addresses.

☞   Before carrying out these modification, work through the steps in "Debugging the
     bit_bang_uart Project" on page 9. Design modifications are described in detail starting
     in "The BitBangUartTransmit() Function" on page 15.

After you confirm successful communication from the software to the hardware, you
can change the hard-coded address to a symbolic name found in **system.h**. Replacing
the hard-coded register address with a symbolic definition enables the Nios II SBT to
update the software if the peripheral's register base address changes in the future.

UART1_BASE is a definition provided by **system.h**. When defining macros in **system.h**,
the Nios II SBT takes the peripheral name as defined in Qsys, and converts it to
uppercase. The Nios II SBT creates the peripheral's base address by appending _BASE
to the peripheral's name.

Making Software Modifications

UART1_BASE is defined in **system.h**, a generated header file, and used in the
bit_bang_uart.c source file. If you use a different hardware design, and the UART
peripheral name is not uart1, search and replace the occurrences of UART1_BASE in
bit_bang_uart.c with the name *<your_uart_peripheral_name>*_BASE. Find the
UART peripheral module name and register base on the **System Contents** tab in
Qsys. Refer to Figure 2 on page 6.

Making Hardware Modifications

If you make hardware design modifications, you must regenerate your Nios II system in Qsys and recompile the Quartus II project in the Quartus II software. In the regeneration step, Qsys updates the SOPC Information File (**.sopcinfo**) for your hardware design.

If you have a pre-existing BSP, such as the example design, an updated **.sopcinfo** file requires that you regenerate your BSP in the BSP Editor (or by executing **nios2-bsp** on the command line). Then you must do a clean build of the BSP as well as the application that depends on it. The clean build is required because the software needs to obtain the new value of the `_BASE` symbol from **system.h**.

For additional information about updating BSP files after a hardware change, refer to "Board Support Packages" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

In the Nios II Command Shell, to force the SBT to copy or regenerate all BSP and application files, simply perform the following steps:

1. Delete the application makefile, for example **app/bit_bang_uart/Makefile**.

2. Delete the BSP's **public.mk** file, for example **bsp/hal_my_uart/public.mk**.

3. Execute the **create-this-app** script in the application directory, for example **app/bit_bang_uart/create-this-app**. The **create-this-app** script runs the **create-this-bsp** script, which in turn runs **nios2-bsp**.

Debugging the bit_bang_uart Project

This section demonstrates debugging techniques with the **bit_bang_uart** project. To start debugging **bit_bang_uart**, perform the following steps:

1. In the Nios II Command Shell, open **nios2-terminal** in UART mode as follows:

   ```
   nios2-terminal --uart --port=<port name> --baud-rate=115200
   ```

   For example, on a Windows system, *<port name>* might be /dev/com1.

   In the Nios II SBT for Eclipse, the Nios II Console is connected to the JTAG UART, and receives the output to the JTAG UART from the Altera logging device.

2. In the Project Explorer view of the Nios II SBT for Eclipse, right-click the **bit_bang_uart** project, and click **Build Project**.

3. On the Run menu, click **Debug Configurations** to prepare a debug configuration for the **bit_bang_uart** project.

4. In the **Debug Configurations** dialog box, select **Nios II Hardware**.

5. Click the **New launch configuration** button, [icon], to create a new debug configuration. To name the debug configuration, in the **Name** box type neek_uart, and click **Apply**. Refer to Figure 4.

Figure 4. Debug Configuration Project Tab
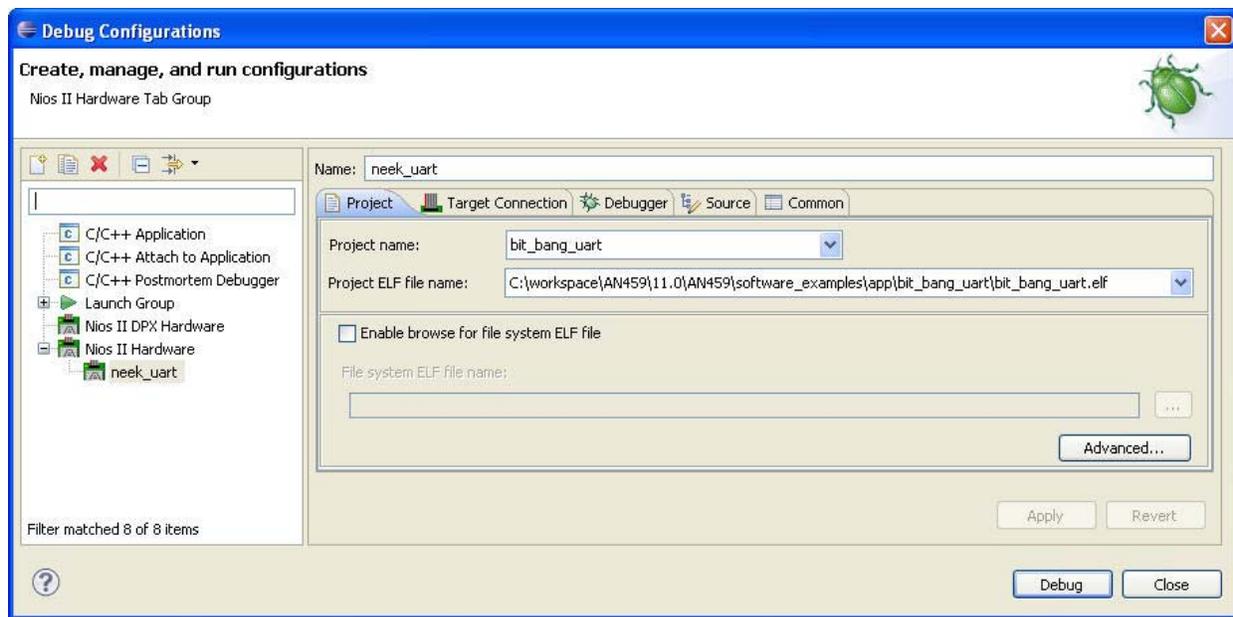
6. On the Project tab, set **Project name** to **bit_bang_uart**, and set **ELF file name** to the path name of the application project's Executable and Linking Format File (**.elf**).

7. Verify that none of the tabs contains a red "x", indicating an error. If any do, select that tab, and fill in the required data necessary to resolve the error as indicated by the tool's messages. For example, if more than one USB-Blaster cable is connected to your development host computer, the **Target Connection** tab has a red "x". In this case, you must select the appropriate cable under **Processors** to resolve the error.

☞ If the message at the top of the dialog box says **Actual system ID not found at target base address**, on the **Target Connection** tab, click **Refresh Connections**. You might need to click **Refresh Connections** several times to establish a connection.

8. Click the **Target Connection** tab (see Figure 5). The message at the top of the dialog box says **The expected Stdout device name does not match the selected target byte stream device name**. This message is expected, because in the **Connections** panel, under **Byte Stream Devices**, the listed device is **jtag_uart**, while the stdout device used by the **bit_bang_uart** application is uart1. You use **nios2-terminal** to send and receive serial I/O. **nios2-terminal** is required because the Nios II SBT for Eclipse does not support the use of a UART as a byte stream device.

Figure 5. Debug Configuration Target Connection Tab
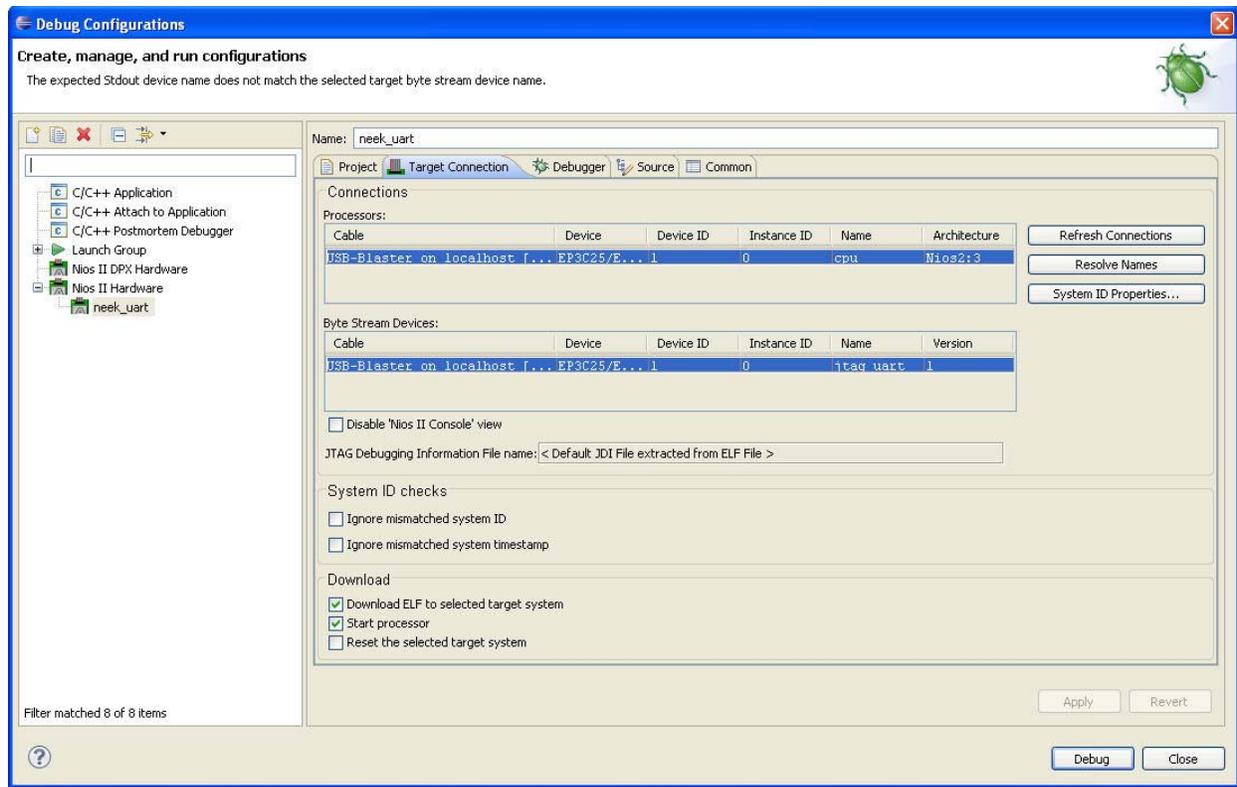
The **jtag_uart** byte stream device is used to receive Altera logging messages, as described in "Debugging with the Altera Logging Functions" on page 35.

For additional information about setting up a debug configuration for Nios II SBT projects, refer to "Run Configurations" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*.

9. Click **Debug**.

10. If Eclipse prompts you to switch to the Nios II Debug perspective, click **Yes**.

Depending on how the Eclipse preferences are configured, Eclipse might automatically switch to the Nios II Debug perspective.

11. Select the Nios II Console view.

12. On the Window menu, point to **Show View** and click **Memory** to open the Memory view.

13. If the Memory view appears in the lower left corner, sharing a tabbed area with the Console view, drag the memory tab to the upper right corner of the perspective. This arrangement allows you to view the Console and Memory views simultaneously.

14. Click the **Add Memory Monitor** button, ✚, in the Memory view, as shown in Figure 6. This action opens a **Monitor Memory** dialog box in which you can type the memory address that you want to monitor.

15. Enter the UART peripheral's register base address, as shown in Figure 6 (0x80 for the uart1 peripheral in the design example accompanying this application note).

16. Click **OK**.

17. In the Memory view, right-click any cell under the column labeled **0 – 3**, and click **Format**. Set **Column Size** to 1 unit per column.

18. Click **OK**.

Figure 6. Specifying Memory Address to Monitor
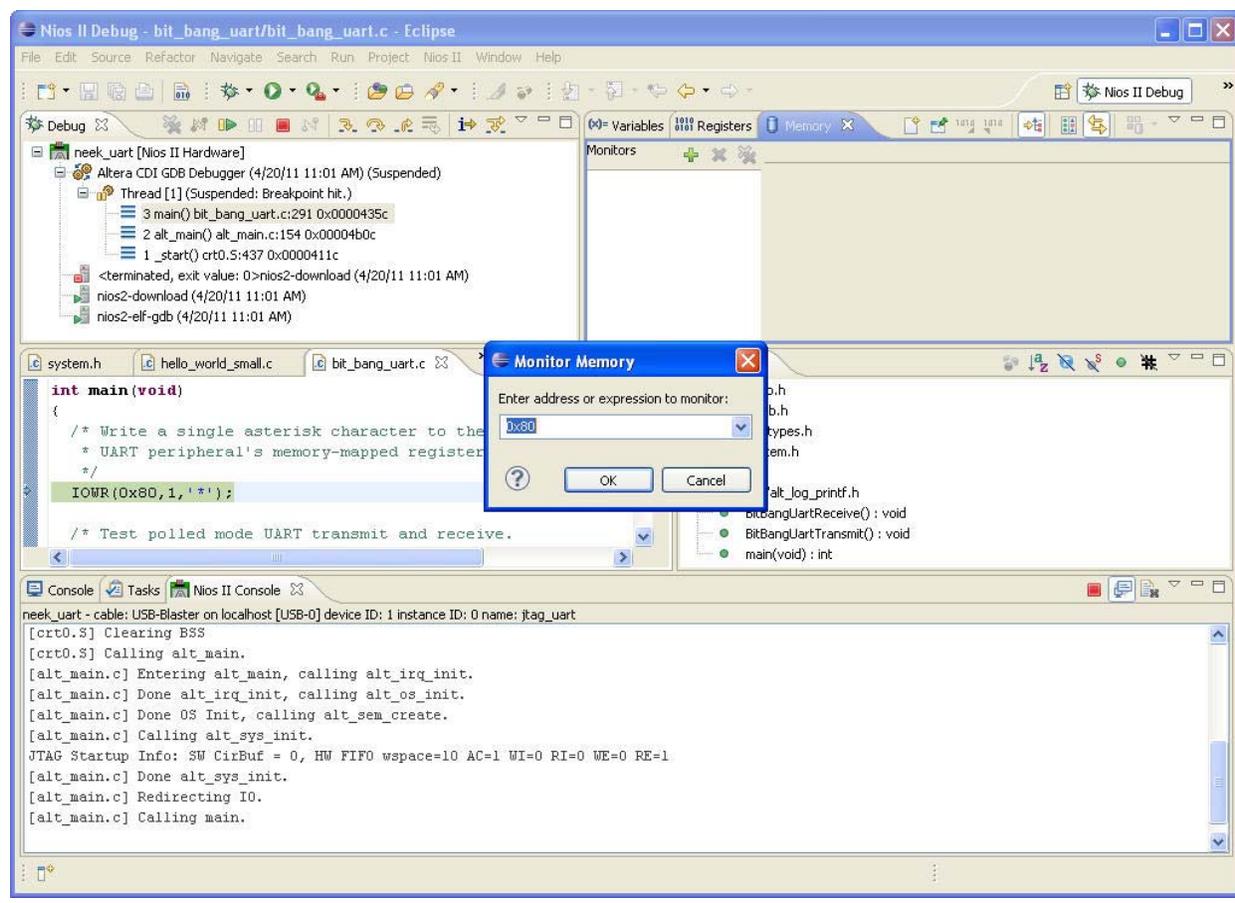
19. Use the **Step Over** button 🔄 to advance the program execution over the IOWR()
    macro. This macro transmits an asterisk to **nios2-terminal** by writing directly to
    the UART's transmit register, as shown in Figure 7.

    ☞ If you do not see an asterisk in **nios2-terminal**, verify your hardware cable
      is properly connected and your UART peripheral base address matches the
      one in your hardware design.

Figure 7. Asterisk Transmitted from Memory-Mapped Register
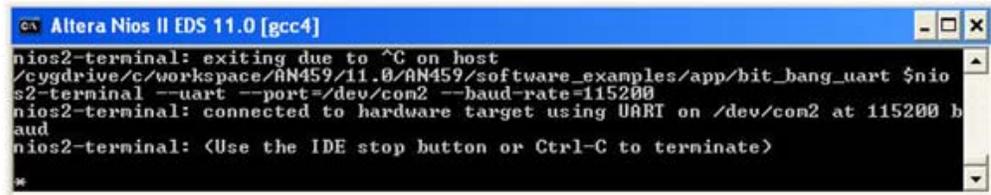
The red numbers in the Memory view indicate which memory values changed
during the last "step over" operation. This change helps you verify that a new
peripheral is functioning correctly. The 2A in the Memory view is the hexadecimal
value for the asterisk character (*), as shown in Figure 8.

Figure 8. Transmit Asterisk
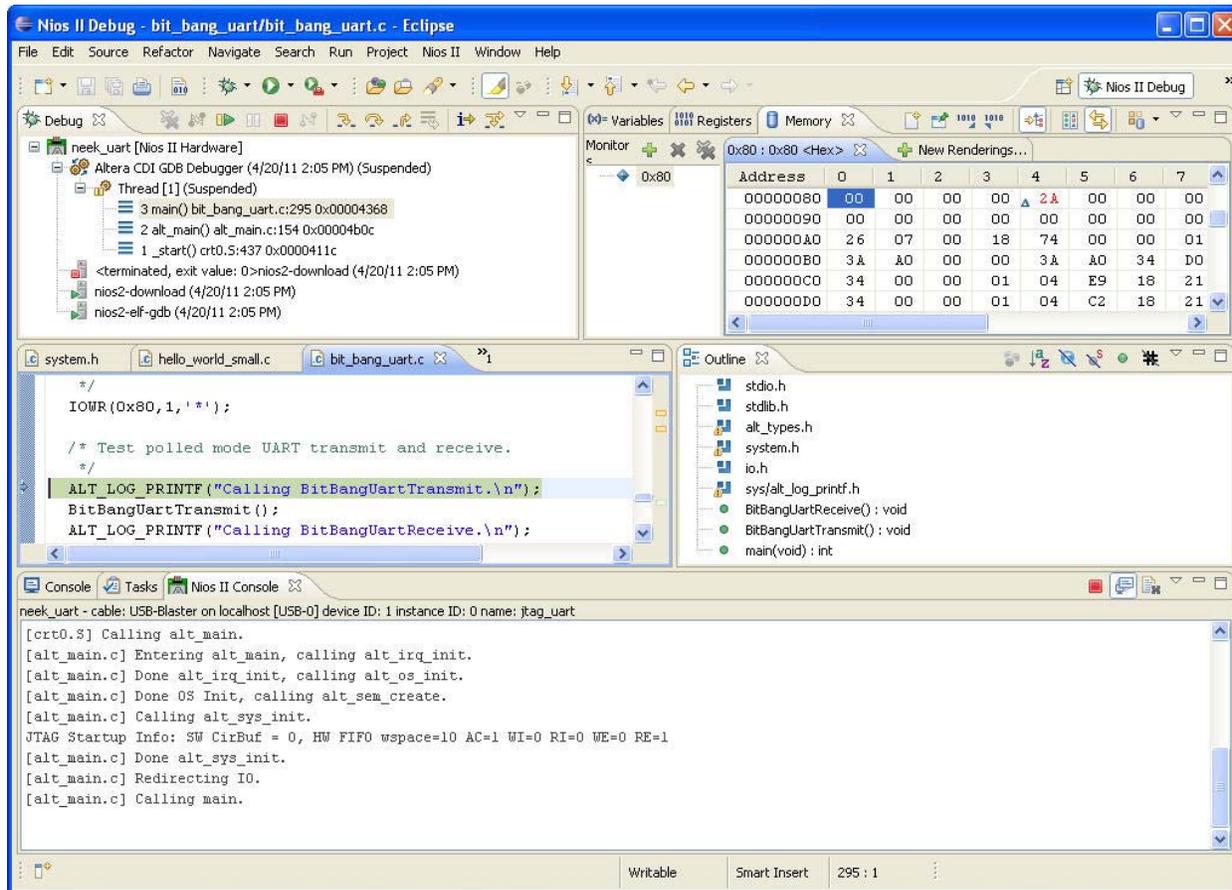
20. To show the Memory view in ASCII rather than hexadecimal, click the **New Renderings** tab (refer to Figure 9). In the **New Renderings** tab, select **ASCII** and click **Add Rendering(s)**.

The 2A in the Memory view changes to an asterisk.

Figure 9. Adding an ASCII Rendering to the Memory View
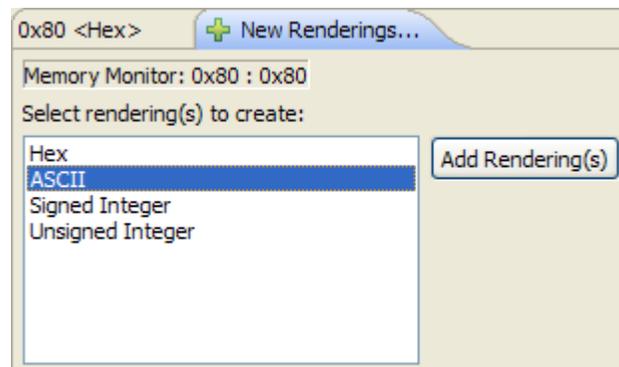
21. You can transmit characters over the UART by directly changing memory values in the Memory view as follows:

   a. In the ASCII rendering, type an h in the cell currently occupied by the asterisk in the Memory view. This cell represents the transmit register.

   b. Press **Enter**.

   c. Type an i in the same cell in the Memory view.

   d. Press **Enter**.

   The word hi appears in **nios2-terminal**, as shown in Figure 10.

☞ The peripheral memory-mapped registers bypass the cache. Therefore, the status register value displayed in the Memory view reflects any changes to the status register made by the peripheral. The IOWR() and IORD() macros always bypass the cache.

Figure 10. Characters Transmitted by Manipulating UART Register
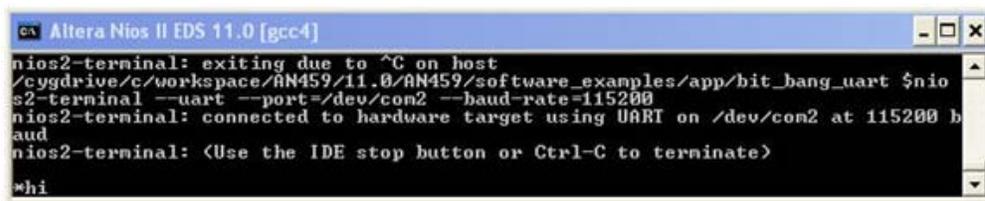
The BitBangUartTransmit() Function

This section examines the BitBangUartTransmit() function in **bit_bang_uart.c**. The BitBangUartTransmit() function demonstrates transmission of characters over the UART.

In the Nios II SBT for Eclipse, step over the BitBangUartTransmit() function. The characters BIT BANH appear in **nios2-terminal**, as shown in Figure 11. The following steps show why the string appears as it does.

Figure 11. Stepping Over the BitBangUartTransmit() Function
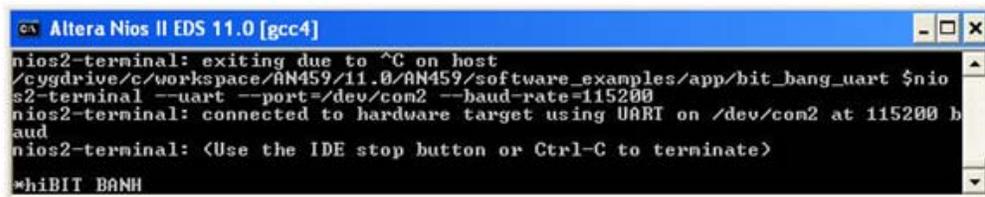
To begin analyzing `BitBangUartTransmit()`, perform the following steps:

1. Restart the debugging session as follows:

   a. Click **Terminate** ■ to stop the current debugging session.

   b. On the Run menu, click **Debug Configurations**.

   c. With the neek_uart debug configuration selected (the default), click **Debug**.

2. Click **Step Over** to step to the call to the `BitBangUartTransmit()` function.

3. Click **Step Into** 🔻 to step into the `BitBangUartTransmit()` function.

4. Click **Step Over** to execute one line at a time until the string BIT BANGBASH appears in **nios2-terminal**, as shown in Figure 12.

bit_bang_uart.c first writes a value of zero to the status register to clear any existing errors on the UART. The `IOWR()` macro accomplishes this step by writing to `UART1_BASE`.

Next, a loop cycles through the `bitbang[]` array, printing out the characters BIT BANG to the UART. To prevent overruns, the loop checks the transmit ready bit before each subsequent character transmission. Immediately after the loop, the software transmits characters BASH one after the other without checking the transmit ready bit.

If you step through each line to the end of the `BitBangUartTransmit()` function, the software transmits the characters BIT BANGBASH through the UART. These characters appear in **nios2-terminal**, as shown in Figure 12. There is no transmitter overrun, because the UART transmits each character much faster than you can single-step.

Figure 12. Transmitting BIT BANGBASH by Stepping Through the Function
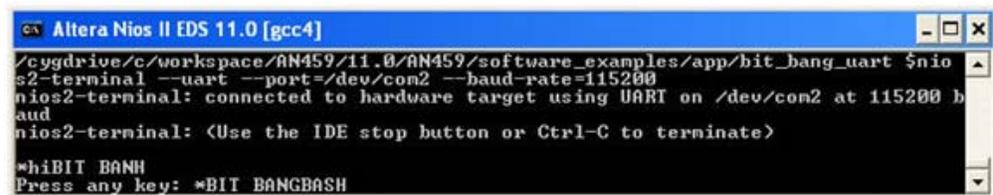
To observe `BitBangUartTransmit()`'s real-time behavior, perform the following steps:

1. Restart the debugging session, as in Step 1 on page 16.

2. Place a breakpoint in `BitBangUartTransmit()`, on the following statement:

   ```
   uart_status = IORD (UART1_BASE, 2);
   ```

   ☞ To set a breakpoint, double-click in the gray area left of the line, as shown in Figure 13.

3. Click the **Resume** button 🔘. The program runs until it reaches the breakpoint.

4. In the Variables view, right-click the uart_status variable name, point to **Format**, and click **Hexadecimal**.

Figure 13. Setting a Breakpoint in BitBangUartTransmit()
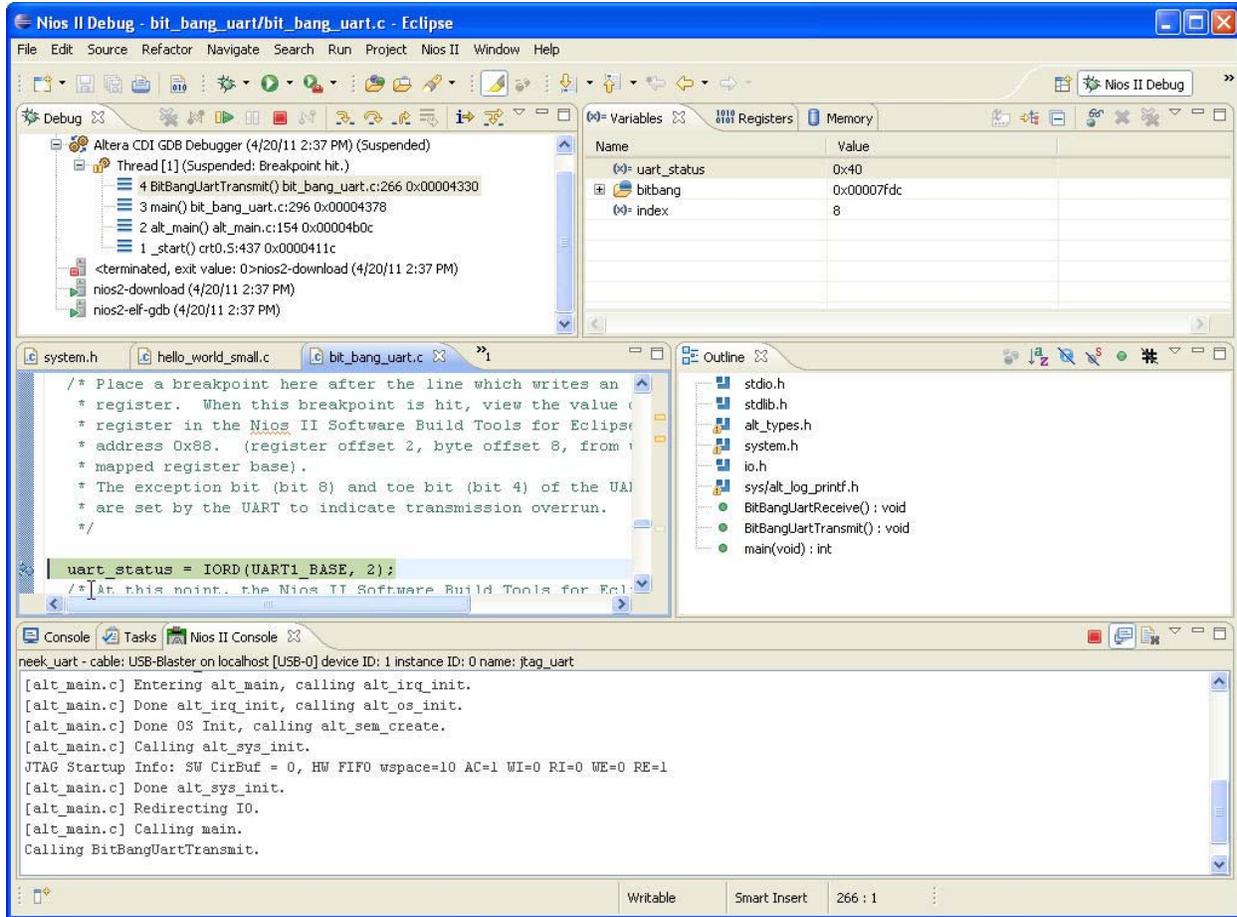
5. Step over the assignment of uart_status. The Variables view shows that the value of uart_status has changed to 0x170, as shown in Figure 14.

Figure 14. Value of uart_status Variable Is 0x170
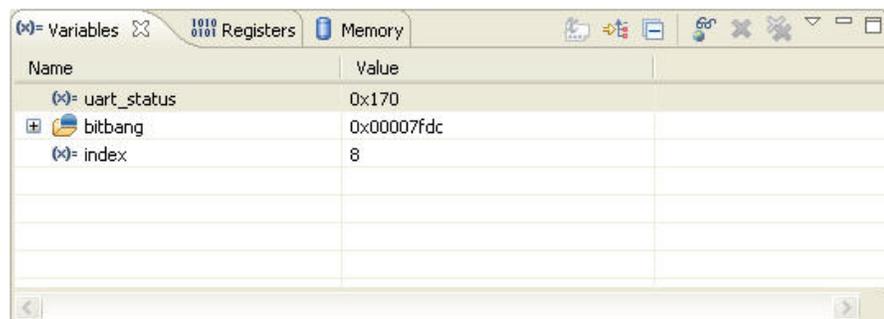
The register map for the Altera Avalon UART core shows that the status register's value of 0x170 means that the following bits are set:

■ Bit 8, the exception (E) bit

■ Bit 4, the transmitter overrun error (TOE) bit

The register map for the Altera Avalon UART core is described in the *UART Core* chapter in *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*.

Because the software does not wait for the transmitter to be ready before writing the final characters (GBASH), the transmitter is overrun and only the last character, H, is transmitted, as shown in Figure 15.

Figure 15. Transmitter Overrun
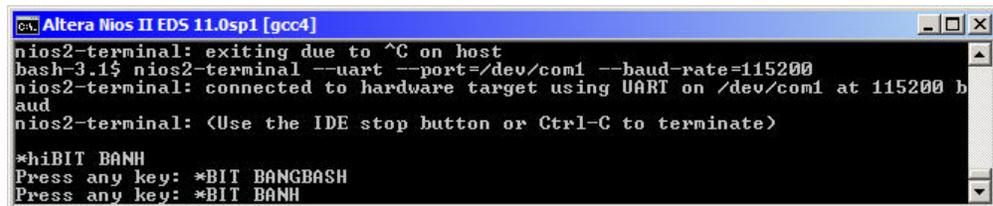
The BitBangUartReceive() Function

This section examines the BitBangUartReceive() function in **bit_bang_uart.c**. The BitBangUartReceive() function demonstrates receiving characters over the UART.

To analyze the BitBangUartReceive() function, perform the following steps:

1. Step into the BitBangUartReceive() function.

2. Set a breakpoint on the while loop immediately after the statement that reads a character into incoming_character, as shown in Figure 16.

3. Click the **Resume** button.

Figure 16. Setting a Breakpoint in BitBangUartReceive()
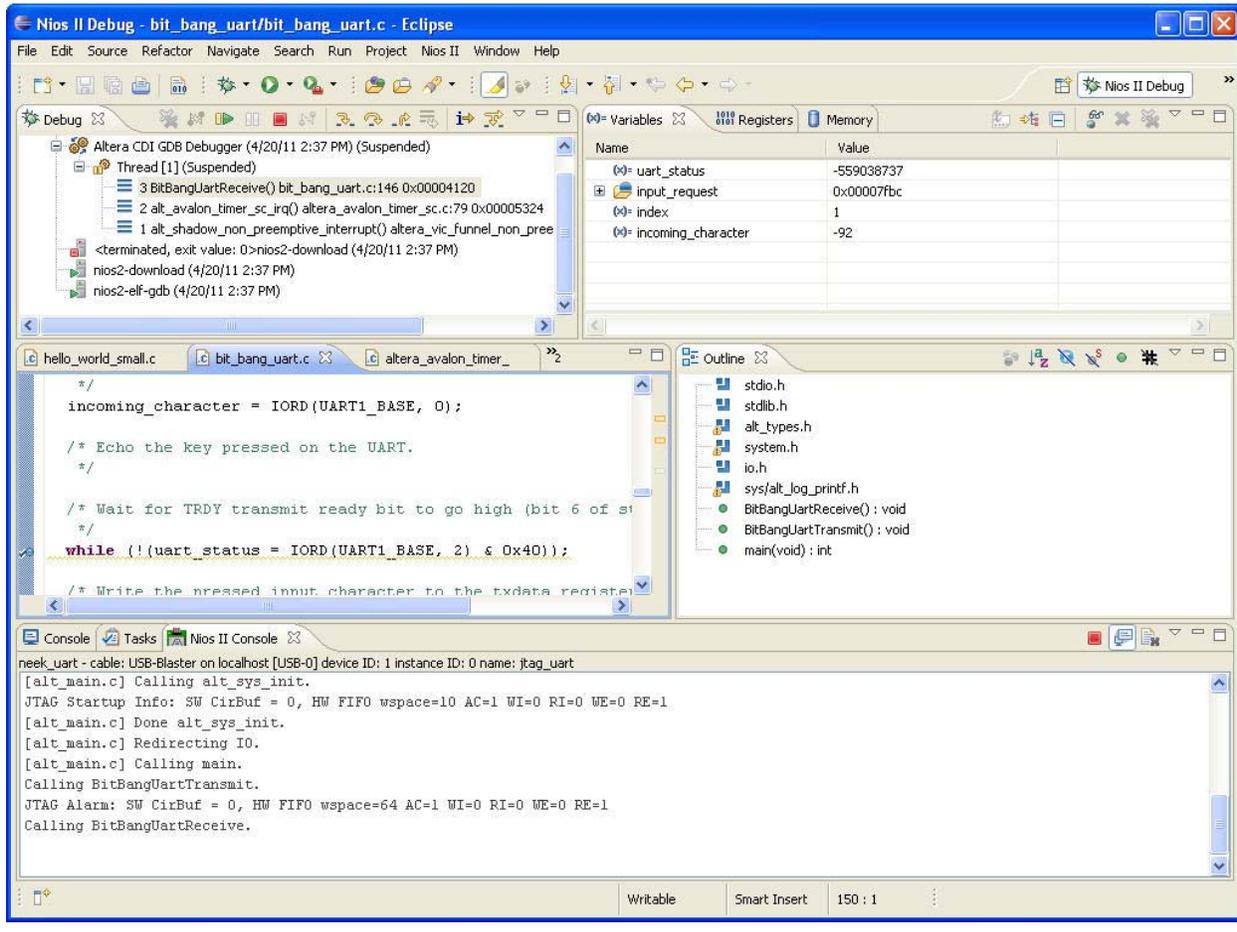
4. In the following loop statement, the Nios II processor is waiting for the receive ready (RRDY) bit to go high:

```
while (!(uart_status=IORD(UART1_BASE, 2) & 0x80));
```

nios2-terminal displays a prompt, as shown in Figure 17.

Figure 17. Waiting to Receive Character on UART
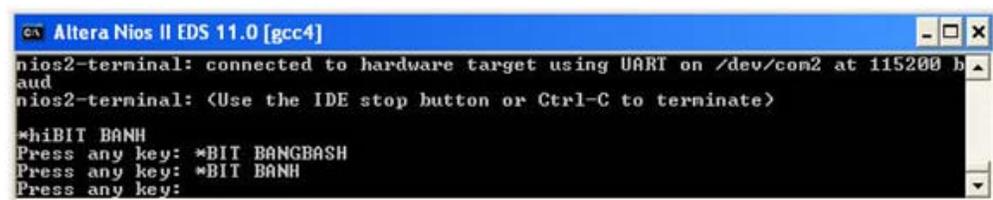
5. In **nios2-terminal**, type * (asterisk). The debugger hits the breakpoint you set, as shown in Figure 18.

6. Examine the Variables view (expand it if necessary to see the `incoming_character` variable). The `incoming_character` variable holds the asterisk you sent through **nios2-terminal**, as shown in Figure 18.

Completing these steps verifies that both the transmit and receive functions of the UART work in polled mode.

Figure 18. incoming_character Variable Is Set to the Character Entered on the Console
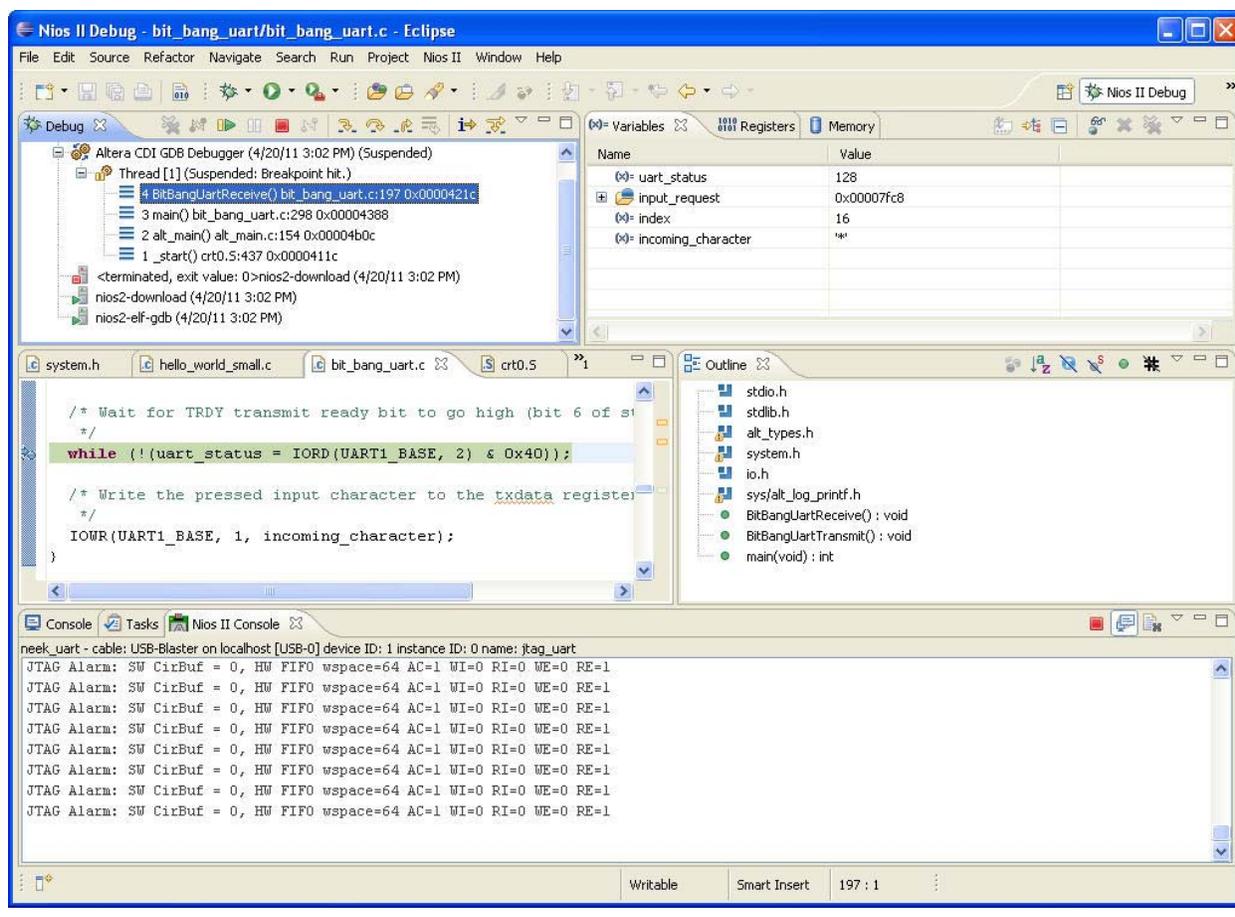
Creating Device Hardware Access Macros

After you verify the functionality of the peripheral registers with the **bit_bang_uart** test software, you can replace the `IORD()` and `IOWR()` macros and their hard-coded address parameters with register access macros. You define the register access macros for the component, under the
<my_design>/**ip**/*<componentfolder>*/**inc**/*<component>*_**regs.h** source code header file.

The base address, component instance name, and interrupt request (IRQ) priority are all available to HAL device drivers from **system.h**. You can write macros that access specific peripheral registers by name, constructed from the information provided in **system.h**. The macros remove the hard-coded nature of the register accesses and instead pull the register base address information out of **system.h**. This procedure

allows automatic incorporation of any changes made to the component instance base address in the hardware design. For example, to access the UART's transmit register, the code in **bit_bang_uart.c** uses an `IOWR()` macro with a hard-coded offset (value 1). Convert this method to a device access macro that can adapt to changes in **system.h** automatically.

Example 1 (from **my_uart_regs.h**) defines a set of device access macros and related access masks for the UART status register.

Example 1. Device Access Macros in my_uart_regs.h

```
#define MY_UART_STATUS_REG               2
#define IOADDR_MY_UART_STATUS(base)      IO_CALC_ADDRESS_NATIVE(base, MY_UART_STATUS_REG)
#define IORD_MY_UART_STATUS(base)        IORD(base, MY_UART_STATUS_REG)
#define IOWR_MY_UART_STATUS(base, data)  IOWR(base, MY_UART_STATUS_REG, data)

#define MY_UART_STATUS_PE_MSK            (0x1)
#define MY_UART_STATUS_PE_OFST           (0)
#define MY_UART_STATUS_FE_MSK            (0x2)
#define MY_UART_STATUS_FE_OFST           (1)
#define MY_UART_STATUS_BRK_MSK           (0x4)
#define MY_UART_STATUS_BRK_OFST          (2)
#define MY_UART_STATUS_ROE_MSK           (0x8)
#define MY_UART_STATUS_ROE_OFST          (3)
#define MY_UART_STATUS_TOE_MSK           (0x10)
#define MY_UART_STATUS_TOE_OFST          (4)
#define MY_UART_STATUS_TMT_MSK           (0x20)
#define MY_UART_STATUS_TMT_OFST          (5)
#define MY_UART_STATUS_TRDY_MSK          (0x40)
#define MY_UART_STATUS_TRDY_OFST         (6)
#define MY_UART_STATUS_RRDY_MSK          (0x80)
#define MY_UART_STATUS_RRDY_OFST         (7)
#define MY_UART_STATUS_E_MSK             (0x100)
#define MY_UART_STATUS_E_OFST            (8)
#define MY_UART_STATUS_DCTS_MSK          (0x400)
#define MY_UART_STATUS_DCTS_OFST         (10)
#define MY_UART_STATUS_CTS_MSK           (0x800)
#define MY_UART_STATUS_CTS_OFST          (11)
#define MY_UART_STATUS_EOP_MSK           (0x1000)
#define MY_UART_STATUS_EOP_OFST          (12)
```

The Altera Nios II component also provides the address construction macro `IO_CALC_ADDRESS_NATIVE()`. The UART device access macros in **nios2eds/components/altera_nios2/HAL/inc/io.h** use this macro. `IO_CALC_ADDRESS_NATIVE()` computes the native address of a specified peripheral register. To compute this address, it adds the second parameter (offset) to the first parameter (peripheral base address). The offset is represented in system bus width units, for example, 32 bits. The `IORD()` and `IOWR()` macros translate to the Nios II assembler instructions, `ldwio` and `stwio`, respectively.

Native addressing mode is deprecated, because Altera is moving to a direct addressing model. New components should be written to use byte-enable signals. Write new device drivers for these components with direct addressing macros, such as `IORD_32DIRECT()`, which utilize the byte-enable signals. Offsets for direct address macros are always represented in bytes. The **bit_bang_uart** example application uses native addressing. The **my_uart** device driver also uses native addressing.

For example, the following addressing macro:

```
IOWR(UART1_BASE, 2, 0);
```

translates to the following direct addressing macro:

```
IOWR_32DIRECT(UART1_BASE, 8, 0);
```

Notice that the offset specified is now eight bytes, instead of two long words.

For more details on direct addressing macros, refer to "Writing Device Drivers" in the *Cache and Tightly-Coupled Memory* chapter in the *Nios II Software Developer's Handbook*.

In the `BitBangUartTransmit()` function in **bit_bang_uart.c**, you used an `IORD()` macro with hard-coded values to read the UART status register:

```
uart_status = IORD(UART1_BASE, 2);
```

You can achieve the same functionality by using the UART's device access macro:

```
uart_status = IORD_MY_UART_STATUS(UART1_BASE)
```

Using this macro makes the device driver code easier to write and easier to understand after it is written.

Altera recommends that you create device access macros for all of your custom component's registers, and that you create masks for each of the bits represented in those macros. These steps result in a driver that is much easier to understand; therefore, it is easier to verify the correctness of the device driver.

Staging the HAL Device Driver Development

The following sections describe the existing `my_uart_driver` source code, particularly the device access descriptors used to manipulate the peripheral. `my_uart_driver` is based on the Altera Avalon UART device driver, with all of the names changed to represent the "my" flavored device, as an illustration of how you can incorporate your own device driver. All of the function and macro names (except for the `*_INIT()` and `*_INSTANCE()` macros) in the Altera Avalon UART device driver have had the "altera_avalon" portion of the name replaced with "my". For example, `ALTERA_AVALON_UART_STATUS_REG()` has become `MY_UART_STATUS_REG()`.

The two macros for `_INSTANCE()` and `_INIT()` are exceptions, because their names must match the hardware component name. As a result, the `my_uart_driver` device driver has definitions for `ALTERA_AVALON_UART_INIT()` and `ALTERA_AVALON_UART_INSTANCE()`. These `_INIT()` and `_INSTANCE()` macros must be defined in a header file that also matches the device name, which in this case is **altera_avalon_uart.h**. This restriction is necessary for the automatic construction of **alt_sys_init.c**, a generated C source file that takes care of component instance initialization.

This example shows you how to write a software device driver that fits the HAL structure, either for manipulation of your own new device, or to override the functionality of the provided software device driver for an Altera component or other third party device.

The file **bit_bang_uart.c** demonstrates how to write source code. The source code development progresses toward a complete device driver. Source code development starts from direct access of the peripheral's registers and goes on to validating the proper functioning of the **Altera_Avalon_UART** hardware. **bit_bang_uart.c** is the first piece of software to communicate with the Altera Avalon UART hardware.

To develop the source code that accesses a new device, perform the following steps:

1. Use `IOWR()` macros with hard-coded address values in `main()` to write values directly to the memory-mapped UART registers. `IOWR()` macros are the most direct way to access the UART hardware. Direct hardware access is useful for validating proper functioning of the component instance, while minimizing the potential for any software coding errors to interfere with hardware validation.

   For more information about HAL device driver access macros, refer to the "Accessing Hardware" section of the *Developing Device Drivers for the Hardware Abstraction Layer* chapter in the *Nios II Software Developer's Handbook*.

2. After developing some direct peripheral manipulation code for your custom component, modeled after **bit_bang_uart.c**, write custom device access macros.

3. Using the custom device access macros from the previous step, develop and test polled routines for the `init()`, `read()`, and `write()` functions.

4. Write the ISRs for interrupt driven mode. An ISR is an interrupt-driven software routine, responding to a hardware interrupt that the peripheral generates when it requires servicing. An interrupt-driven device driver is much more efficient than a polled device driver, which wastes processor clock cycles by repeatedly querying the peripheral to determine if there is work to be performed. An ISR allows the Nios II processor to do other work while the peripheral is idle, or while it is operating autonomously and does not require servicing by the Nios II processor. Call `alt_ic_isr_install()` from `main()` to install the ISRs.

5. After you have tested the ISR and polled routines from `main()`, create and test the `INIT` and `INSTANCE` macros. **alt_sys_init.c** invokes these initialization macros to initialize both the software device driver and the hardware driver. The `INIT` macro needs to initialize an **alt_dev** structure for the software device driver with the tested functions for reading and writing to the UART hardware device. The `INSTANCE` macro declares a structure for each component instance to hold component instance-specific information, such as the baud rate and the transmit and receive memory buffers. At this point, you move the `alt_ic_isr_install()` calls from function `main()` to the device's initialization code, as described in the next section.

For more information about the **alt_dev** structure, refer to "Character-Mode Device Drivers" in the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Understanding the Hardware-Specific INSTANCE and INIT Macros

The `INSTANCE` macro creates the **alt_dev** structure, which represents the component instance. This macro creates unique component instance-specific data structures.

The INIT macro must perform the following tasks:

■ Create mutual exclusion resources

■ Install the component's ISR with `alt_ic_isr_register()`

☞ The `my_uart_driver` example driver supports both the enhanced and the legacy HAL interrupt APIs. Supporting both APIs ensures that the driver can be used in combination with legacy drivers supporting only the legacy API. For information about supporting both APIs, refer to "Interrupt Service Routines" in the *Exception Handling* chapter in the *Nios II Software Developer's Handbook.*

■ Register the **alt_dev** structure with `alt_dev_reg()`

■ Enable interrupts

Integrating a New HAL Device Driver in the BSP

When you integrate a HAL device driver into a BSP, the following services are enabled:

■ Automatic initialization with the `alt_sys_init()` function for the HAL device drivers.

`alt_sys_init()` is an automatically generated function. `alt_sys_init()` calls the INIT and INSTANCE macros for each component instance found in the hardware design that has a specific source code directory structure and set of file names. The directory structure for hardware components provided by Altera conforms to:

*<Altera installation>***/ip/altera/sopc_builder_ip/***<component_folder>*

The easiest option for a directory structure for your custom components conforms to:

*<my_design>***/ip/***<component_folder>*

Place the device driver source code files in a folder structure under *<component_folder>*. The file names conform to the following:

■ **/inc/***<component>***_regs.h**

■ **/HAL/inc/***<component>***.h**

■ **/HAL/src/***<component>***.c**

☞ This document uses the variable *<Altera installation>* to represent the location where the Altera Complete Design Suite is installed. On a Windows system, by default, that location is **c:/altera/***<version number>*.

■ HAL device-class services can access any specific HAL device in that class. For example, **Altera_Avalon_UART** is a character-mode device, and so has access to higher level services such as buffer management. HAL software device drivers become available to the UNIX-style POSIX API for device functions such as `open()` and `read()`.

For more information about adding device drivers using the Nios II SBT, refer to the "Integrating a Device Driver in the HAL" section of the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

For more information about how to integrate your own VHDL or Verilog HDL source code as a new HAL-compatible Qsys component, and for details about the Component Editor tool, refer to the *Creating Qsys Components* chapter of *Volume 1: Design and Synthesis* in the *Quartus II Handbook*.

Understanding HAL Mutual Exclusion Resources

Software device drivers can use mutual exclusion resources to control access to any data structure or peripheral register. Event flags and semaphores provide synchronization and mutual exclusion services. These resources allow only one task to access a shared piece of data at a time in a multi-threaded environment.

If the MicroC/OS-II operating system is present, its resources are used. Otherwise, the HAL provides its own set of event flags and semaphores. The HAL event flags and semaphores support device driver source code portability. The event flags and semaphores do nothing in this example.

The `my_uart_driver` device driver creates two semaphores and one event flag. The two semaphores are called **read_lock** and **write_lock**. `my_uart_driver` uses them to control access to the transmit and receive circular buffers. The event flag, called **events**, indicates to the software device driver when data is ready to be transmitted or received.

Overview of Debugging Tools for HAL Device Drivers

The Nios II EDS and Quartus II software tools provide a variety of mechanisms for debugging device drivers:

■ You can monitor individual component instance signals for activity with the SignalTap® II logic analyzer. For example, you can hook up the SignalTap II logic analyzer to the UART hardware transmit line to watch for any activity while you write characters to the **Altera_Avalon_UART** component instance through the `my_uart_driver` device driver.

For information about using SignalTap II with Nios II systems, refer to AN446: *Debugging Nios II Systems with the SignalTap II Embedded Logic Analyzer*.

■ You can step into the `fprintf()` function, stepping through the various layers of abstraction until you reach the HAL's invocation of `my_uart_write()` function in the `my_uart_driver` device driver.

■ You can set breakpoints in the driver's ISRs, or set watchpoints on UART memory-mapped registers to pause the processor when a character is received.

☞ Use caution when setting a breakpoint in an ISR. When you resume, there might be problems with other devices, if they generated interrupts that were not handled. However, sometimes the best way to debug a specific device driver is to set a breakpoint in it. When this technique is required, you can reset or download the software containing the device driver again when you are done with a particular debugging session.

These mechanisms can help you diagnose an incorrectly configured system. For example, if the interrupt controller receives a spurious interrupt signal, the interrupt might not be properly handled. When interrupts are enabled after low-level system initialization, there is no way to clear the interrupt source. As a result, the application does not work correctly. The Nios II SBT for Eclipse debugger might even stop communicating with the processor.

Debugging the HAL UART Device Driver

For the next set of debugging examples, you must create a new application. For these examples, create the **hello_world_my_uart** application and import it to the Nios II SBT for Eclipse. Next, regenerate the files which make up the **hal_my_uart** BSP. This time, instead of commenting out the invocation of the macros `ALTERA_AVALON_UART_INSTANCE()` and `ALTERA_AVALON_UART_INIT()`, let the `alt_sys_init()` function install the Altera Avalon UART HAL device driver, after which you can inspect its operation.

The following sections show examples of placing breakpoints and watchpoints in HAL device driver source code to analyze component instance behavior.

To create and import the **hello_world_my_uart** project, perform the following steps:

1. Delete the generated file **public.mk** from the **hal_my_uart** BSP. Enter the following command in the Nios II Command Shell:

   ```
   rm <my_design>/software_examples/bsp/hal_my_uart/public.mk ↵
   ```

   Deleting these files causes the next build to regenerate the **hal_my_uart** BSP files, including **alt_sys_init.c**.

2. Create the **hello_world_my_uart** application by invoking its **create-this-app** script. Enter the following commands:

   ```
   cd <my_design>/software_examples/app/hello_world_my_uart ↵
   ./create-this-app ↵
   ```

   This action accomplishes several tasks:

   ■ Invokes the **create-this-bsp** script for the **hal_my_uart** BSP.

   ■ In the BSP, deletes and rebuilds generated files and object files from the previous build.

- Builds the **libhal_bsp.a** BSP library in the
  <*my_design*>**/software_examples/bsp/hal_my_uart** directory.

- Builds the **hello_world_my_uart.elf** file in the
  <*my_design*>**/software_examples/app/hello_world_my_uart** directory.

- Generates a new **public.mk** file.

- Sets the software device descriptors stdout, stderr, and stdin to uart1. The
  <*my_design*>**/software_examples/bsp/hal_my_uart/create-this-bsp** script sets
  up these software device descriptors by calling **nios2-bsp**.

- Creates the software device driver, my_uart_driver, in the
  <*my_design*>**/ip/my_uart** directory, and associates it with the
  Altera_Avalon_UART device. The <*my_design*>**/ip/my_uart/my_uart_sw.tcl**
  script creates the driver and associates it with its device.

- Sets the software device driver called my_uart_driver to the component
  instance named uart1. The
  <*my_design*>**/software_examples/bsp/hal_my_uart/ hal_my_uart.tcl** script,
  passed to **nios2-bsp**, sets the driver to uart1.

- Regenerates **alt_sys_init.c** by invoking of ALTERA_AVALON_UART_INIT().

☞ You can invoke **nios2-bsp** with the --debug parameter, which displays
  verbose information about the construction steps in this section. The
  --debug parameter can be very useful for finding errors in the construction
  of the relevant Tcl scripts and command shell scripts.

3. Import the **hello_world_my_uart** application to the Nios II SBT for Eclipse as
   described in "Importing Projects" on page 7 in Steps 2 through 8, substituting the
   hello_world_my_uart application for the **bit_bang_uart** application. It is not
   necessary to re-import the BSP.

Setting Breakpoints in the my_uart_driver Device Driver

This section demonstrates the use of breakpoints to examine HAL device driver
activity. Perform the following steps:

1. After importing the **hello_world_my_uart** project, open the **my_uart_init.c** device
   driver source file, located in the **hal_my_uart** project, at the following directory:

   <*my_design*>**/software_examples/bsp/hal_my_uart/drivers/src/my_uart_init.c**

2. Place a breakpoint at the top of the function named my_uart_irq(), as shown in
   Figure 19.

3. Restart **nios2-terminal** if it is not already running.

4. Create a debug configuration for **hello_world_my_uart** by following the steps in
   "Debugging the bit_bang_uart Project" on page 9, Step 3 through 8, and
   substituting the **hello_world_my_uart** application for the **bit_bang_uart**
   application.

5. Start debugging the **hello_world_my_uart** application. The processor pauses at
   the top of function main().

6. Click the **Resume** button. The Nios II processor pauses at the `my_uart_irq()` invocation.

Figure 19. Setting a Breakpoint on my_uart_irq()
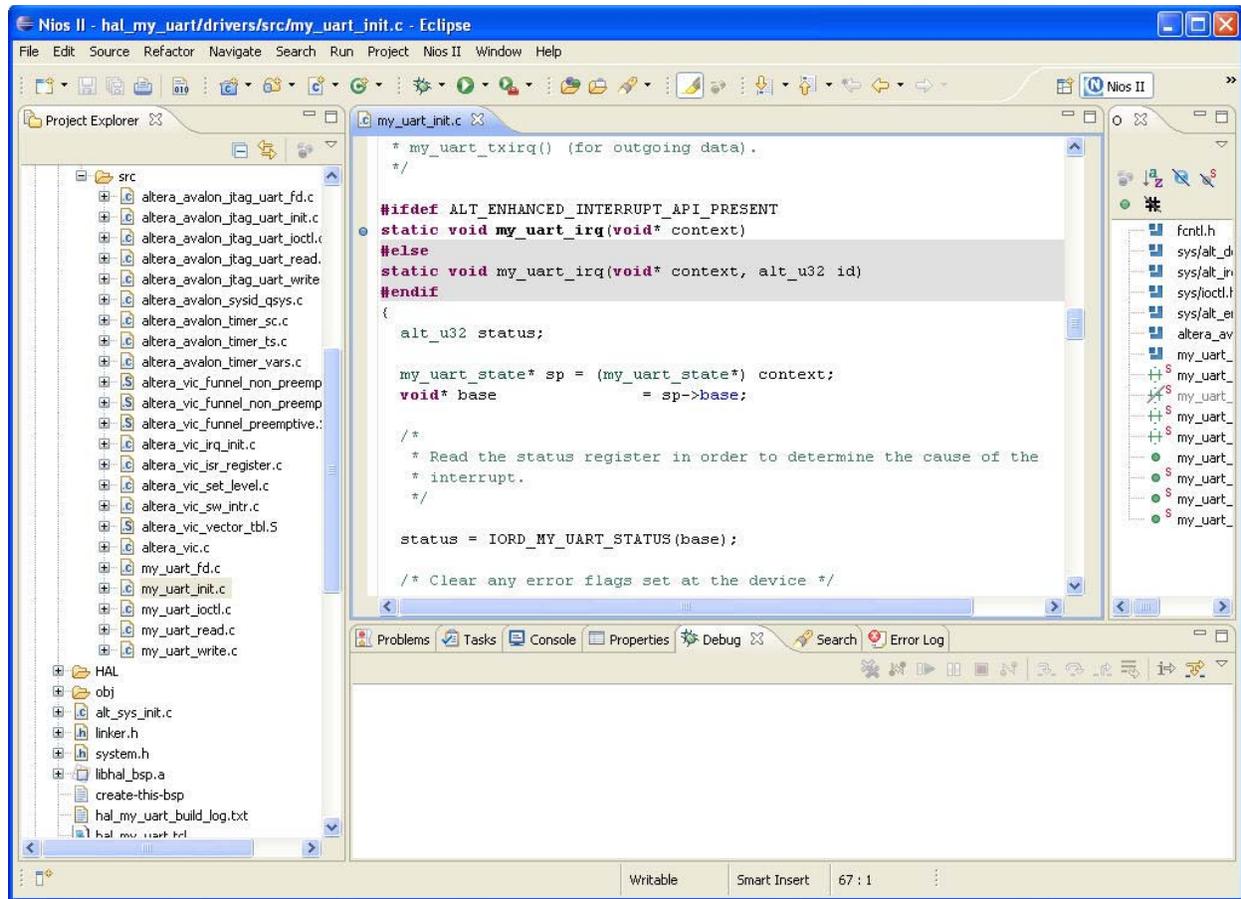
7. Step up to and over the following assignment of the status register:

   `status = IORD_MY_UART_STATUS(base);`

8. In the Variables view, set the format of the `status` variable to `hex`. You can see that the status register now holds the value 0x60. This value indicates bits 5 and 6 are set. According to the `my_uart_driver` register description in **drivers/inc/my_uart_regs.h**, these two bits indicate transmit ready and transmit. The UART driver is in an interrupt context, ready to transmit the first character of the string `"Hello from Nios II!"`.

9. Continue stepping through the procedure. The `my_uart_irq()` function invokes `my_uart_txirq()` in response to a transmit interrupt.

   Press **Resume** after each character is transmitted. Stop after the entire string "Hello from Nios II!" is transmitted.

10. Remove the breakpoint.

☞ After you pause the debugger in an ISR, the rest of the system is in an unknown state, because it could not respond to other interrupt requests while paused in the driver. Therefore, you need to start a new debugging session to perform further debugging.

Setting Watchpoints in the HAL UART Device Driver

In this section, you intercept the Nios II processor by placing a watchpoint on a UART peripheral register. A watchpoint is a special breakpoint that pauses the execution of an application whenever the value of a given expression changes. To watch for any writes to the transmit register on the UART, you can set up a write-access watchpoint on the register.

☞ To use watchpoints, you must configure the Nios II processor's JTAG debug module to level 2 or higher. This setting enables two data triggers, which are required for watchpoints. The hardware design example accompanying this application note is configured with debug level 2. You can specify a higher debug level to get four data triggers, enabling four simultaneously-available watchpoints. Configure the JTAG debug level in the Nios II processor parameter editor in Qsys, on the **JTAG Debug Module** page.

To set a watchpoint, perform the following steps:

1. Start the debugging session for the **hello_world_my_uart** project.

2. Open the Breakpoints view. If the Breakpoints view is not visible, open it through the Window menu, by pointing to **Show View** and clicking **Breakpoints**.

3. Open the menu by clicking the drop-down arrow in the upper right corner of the view, and click **Add Watchpoint (C/C++)**.

4. In the **Add Watchpoint** dialog box, type a value in the **Expression to watch** field that equals the uart1 base value plus an offset of one long word. This value accesses the transmit register. In the case of the design example accompanying this application note, this value is 0x84.

5. In the Access section, turn on **Write** and turn off **Read**.

6. Click **OK**. The **Add Watchpoint** dialog box closes.

7. Click the **Resume** button.

The Nios II processor executes until it writes the first character, H. The processor writes this character when the my_uart_txirq() function invokes the macro IOWR_MY_UART_TXDATA(), as shown in Figure 20.

View the transmit register value in the Eclipse Memory view. Notice that the value changes when the debugging stops at the watchpoint.

Look at the call stack in the upper left corner of the Nios II Debug perspective. The call stack records each call leading up to this point, including each function invoked to process the transmit interrupt. The alt_shadow_non_preemptive_interrupt() function calls my_uart_irq().

Figure 20. Setting Watchpoints in the UART's Transmit Register
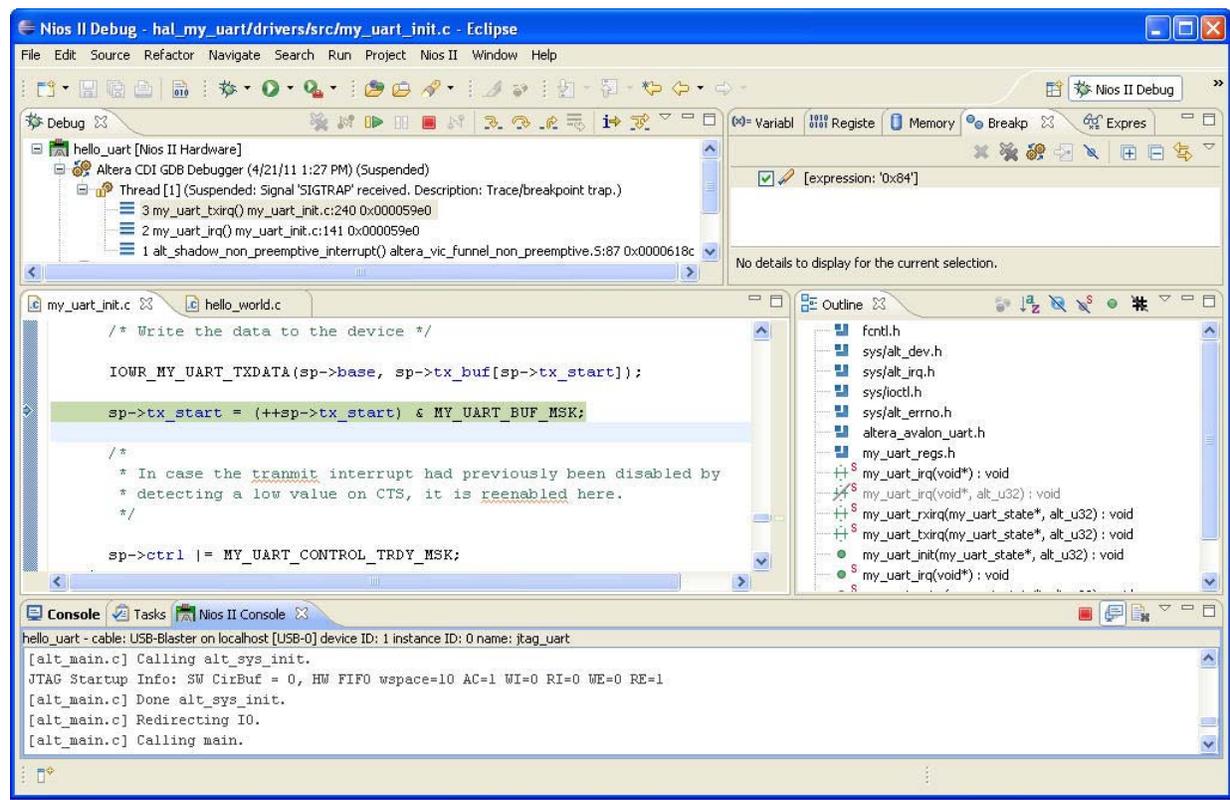
Reducing Driver Code and Memory Footprint

The Nios II SBT provides BSP settings to configure the HAL. You manipulate these settings through the BSP Editor, or with the --set parameter to **nios2-bsp**.

The Nios II SBT settings are described in "Settings" in the *Nios II Software Build Tools Reference* chapter in the *Nios II Software Developer's Handbook*.

The **Reduced device drivers** and **Lightweight device driver API** options are of particular interest, because they reduce the code and data footprint at the expense of device driver functionality. Additionally, they set #define parameters that the my_uart_driver device driver needs to examine and handle. The setting values are documented in the **summary.html** file generated by **nios2-bsp**, in <*my_design*>**/software_examples/bsp/hal_my_uart**.

Using the Reduced Device Drivers API Option

The **Reduced device drivers** option generates a #define statement for ALT_USE_SMALL_DRIVERS. To turn on this option, set **hal.enable_reduced_device_drivers** to true. Setting this option has the following effects on correctly-written UART device drivers:

- Sets #define ALT_USE_SMALL_DRIVERS

- Activates polled mode only for the UART device

- Disables floating-point support in printf() and sprintf()

- Configures drivers to ignore flow control.

Figure 21 shows excerpts from the **summary.html** file generated by the SBT for a BSP with reduced device drivers enabled.

Figure 21. hal.enable_reduced_device_drivers in summary.html

Setting Name:	hal.enable_reduced_device_drivers
Identifier:	ALT_USE_SMALL_DRIVERS
Default Value:	0
Value:	0
Type:	Boolean
Destination:	public_mk_define
Description:	Certain drivers are compiled with reduced functionality to reduce code footprint. Not all drivers observe this setting. The altera_avalon_uart and altera_avalon_jtag_uart drivers switch from interrupt-driven to polled operation. CAUTION: Several device drivers are disabled entirely. These include the altera_avalon_cfi_flash, altera_avalon_epcs_flash_controller, and altera_avalon_lcd_16207 drivers. This can result in certain API (HAL flash access routines) to fail. You can define a symbol provided by each driver to prevent it from being removed. If true, adds -DALT_USE_SMALL_DRIVERS to ALT_CPPFLAGS in public.mk.
Restrictions:	none

For more information about the **Reduced device drivers** option, refer to "Reducing Code Footprint" in the *Developing Programs Using the Hardware Abstraction Layer* chapter and to "Reducing Code Footprint" in the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Using the Lightweight Device Drivers API Option

The **Lightweight device driver API** option generates a #define statement for
ALT_USE_DIRECT_DRIVERS. To turn on this option, set
hal.enable_lightweight_device_driver_api to true. Setting this option has the
following effects on correctly-written UART device drivers:

■ Sets #define ALT_USE_DIRECT_DRIVERS.

■ Eliminates the option of using a file system. File descriptors cannot be created.

■ Disables stdio device descriptor redirection. alt_main() cannot call
  alt_io_redirect().

■ Disabling the open() and close() functions. Attempting to call these functions
  generates a link time error.

■ Causes direct calls to your UART device driver using macros, bypassing the device
  manipulation function invocations normally accessed through the file descriptor
  structure. The macros are defined in **alt_driver.h**, which is in the BSP folder, in
  HAL/inc/sys.

For example, a call to alt_putstr() is normally treated as a call to the run-time library
function fputs(). With lightweight device drivers enabled, alt_putstr() is
translated to ALT_DRIVER_WRITE() (defined in **alt_driver.h**) and state-obtaining
macros. The ALT_DRIVER_WRITE() macro in turn calls the ALT_DRIVER_FUNC_NAME()
macro (also defined in **alt_driver.h**), and eventually ALTERA_AVALON_UART_WRITE(),
which is defined in the **altera_avalon_uart_write.c** driver file for the UART, where the
UART is defined for stdout. Calling ALT_DRIVER_FUNC_NAME(uart1, write) returns
ALTERA_AVALON_UART_WRITE.

ALT_USE_DIRECT_DRIVERS is dual-purposed in the my_uart_driver device driver. It
provides a convenient way to map the names of the ALTERA_AVALON_UART_INIT() and
ALTERA_AVALON_UART_INSTANCE() macros, which are tied to the component name, to
names that are specific to the my_uart_driver device driver. This setting of
ALT_USE_DIRECT_DRIVERS already maps ALTERA_AVALON_UART_INIT() and
ALTERA_AVALON_UART_INSTANCE() to macros that change based on the setting of
ALT_USE_DIRECT_DRIVERS in **altera_avalon_uart.h**. At the same time, the
ALTERA_AVALON_UART_INIT() and ALTERA_AVALON_UART_INSTANCE() macros have the
ALTERA_AVALON portion of their names change to MY_UART. The resulting four macro
name mappings are MY_UART_DEV_INIT(), MY_UART_STATE_INIT(),
MY_UART_DEV_INSTANCE(), and MY_UART_STATE_INSTANCE().

Figure 22 shows excerpts from the **summary.html** file generated by the SBT for a BSP with lightweight device drivers enabled.

Figure 22. hal.enable_lightweight_device_driver_api in summary.html

Setting Name:	hal.enable_lightweight_device_driver_api
Identifier:	ALT_USE_DIRECT_DRIVERS
Default Value:	0
Value:	0
Type:	Boolean
Destination:	public_mk_define
Description:	Enables lightweight device driver API. This reduces code and data footprint by removing the HAL layer that maps device names (e.g. /dev/uart0) to file descriptors. Instead, driver routines are called directly. The open(), close(), and lseek() routines will always fail if called. The read(), write(), fstat(), ioctl(), and isatty() routines only work for the stdio devices. If true, adds -DALT_USE_DIRECT_DRIVERS to ALT_CPPFLAGS in public.mk.
Restrictions:	The Altera Host and read-only ZIP file systems can't be used if hal.enable_lightweight_device_driver_api is true.

For more information about the **Lightweight device driver API** option, refer to "Reducing Code Footprint" in the *Developing Programs Using the Hardware Abstraction Layer* chapter and to "Reducing Code Footprint" in the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Interrupt Latency and Determinism

This section discusses the crucial topics of interrupt latency and determinism. For the purposes of this discussion, you need to be familiar with the following concepts:

- Interrupt latency—The difference between the time that a component instance asserts an interrupt and the execution of the first instruction at the interrupt vector address. This instruction is typically part of the interrupt funnel, rather than the interrupt handler itself.

- Interrupt response time—The time elapsed between the event that causes the interrupt and the execution of the handler.

- Determinism—An attribute of a piece of source code that is guaranteed to execute within a fixed amount of time. Overall interrupt latency impacts the deterministic behavior for all source code in the system for which interrupts are not disabled.

For more information, refer to "Nios II Exception Handling Overview" in the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

To minimize interrupt latency, thus directly improving system determinism, follow these guidelines:

■ In the software ISR, perform the minimum processing necessary to clear the interrupt.

■ Complete noncritical-section interrupt processing outside of the interrupt context. If your software is based on an operating system, a high priority task can be pending on an event flag. The ISR posts to the event flag, notifying the task to complete interrupt processing.

■ Use an External Interrupt Controller (EIC), such as the VIC used in the hardware design for this application note.

   If it is not possible to use an EIC, you can improve the performance of the internal interrupt controller (IIC) by using the interrupt vector custom instruction.

   ☞ The interrupt vector custom instruction is not compatible with the EIC interface. The performance of the IIC with the interrupt vector custom instruction is generally inferior to the performance of the VIC.

■ Use shadow register sets

■ To achieve the lowest possible interrupt latency, consider using tightly-coupled memories so that interrupt handlers can run without cache misses.

👣 For information about using the EIC and shadow register sets, or the interrupt vector custom instruction, refer to "Improving ISR Performance" in the *Exception Handling* chapter in the *Nios II Software Developer's Handbook* and "Exception and Interrupt Controllers" in the *Processor Architecture* chapter in the *Nios II Processor Reference Handbook.* For information about tightly coupled memory, refer to the *Using Tightly Coupled Memory with the Nios II Processor Tutorial*.

👣 For details of the interrupt vector custom instruction implementation, refer to the "Exception and Interrupt Controllers" section in the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

👣 For more information about tightly-coupled memories, refer to the "Tightly-Coupled Memory" section in the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

☞ Restrict the use of synchronization resources to post-function calls. Do not call the following types of functions from within an ISR:

   ■ Functions and macros, such as `ALT_SEM_PEND()`, that explicitly wait for a resource

   ■ Library functions, such as `printf()`, that might wait for a resource

   ■ Other functions that wait for resources

   Calling these types of functions from an ISR can have serious consequences, from the destruction of overall system latency to complete system deadlock.

Avoid using `alt_irq_interruptible()`, which can enable ISR nesting, but is likely to worsen interrupt latency (unless the ISR is abnormally long) because of the interrupt context switch overhead. If the ISR is lengthy, instead of making it interruptible, consider moving much of the less time-critical processing of the interrupt outside of the ISR to a task. Write the ISR to do only as much as is required to clear the interrupt and capture state so that the hardware can proceed, and then signal a task to complete processing of the interrupt request.

Debugging with the Altera Logging Functions

The Altera logging functions are a very useful mechanism for debugging device drivers. Altera logging uses macros to bypass the HAL driver and access the peripheral directly. As a result, software can print debugging messages during the boot process before the devices are initialized.

You do not need to regenerate the **.sopcinfo** file in Qsys or recompile the **.sof** image in the Quartus II software.

Another advantage of Altera logging is that you can disable it without modifying your source code. You simply change a BSP setting and recompile, leaving zero residual impact in the compiled and linked application **.elf** file. All the Altera logging mechanisms are macros, and so the compiler eliminates them when they are not enabled. As a result, you can leave these calls to obtain debugging information in the source code for your released final product, with no loss of speed or code memory space. Compiling with Altera logging disabled creates a **.elf** file identical to a **.elf** compiled from source code without the Altera logging macros.

☞ When Altera logging is enabled, the behavior of the application might be less deterministic, due to the collection and output of Altera logging messages.

Altera Logging Usage

You enable Altera logging with the `hal.log_port` BSP setting. This setting causes the SBT to define `ALT_LOG_ENABLE` in **public.mk**. You set the logging level with `ALT_LOG_FLAGS`.

You can add Altera logging diagnostic messages to your code by invoking `ALT_LOG_PRINTF()`, a macro that handles most `printf()` formatting options.

Writes to the Altera logging device are blocking. Therefore, when `hal.log_port` is set to a component instance of type **altera_avalon_jtag_uart**, you must run an application to accept the Altera logging output in order for the Nios II application to complete initialization. Otherwise, the application pends on an `ALT_LOG_PRINTF()` statement until the Altera logging device's output buffer can be drained.

You can handle the JTAG UART logging output in either of the following ways:

■ Run your application in the Nios II SBT for Eclipse. JTAG UART logging output appears in the Nios II Console view.

■ Run your application from the Nios II Command Shell, with the nios2-download command. From another Nios II Command Shell, run nios2-terminal to accept the JTAG UART logging output.

You can disable the Altera logging feature by setting the `hal.log_port` setting in the BSP to `none`. This is the default setting. In the example BSP, **hal_my_uart**, the **create-this-bsp** script initially sets `hal.log_port` to `jtag_uart`. You can enable and disable it in the BSP Editor. After regenerating a BSP, you can check the value of `hal.log_port` in **summary.html**, located in the **bsp/hal_my_uart** folder.

Disabling Altera logging has the effect of leaving `ALT_LOG_ENABLE` undefined in **bsp/hal_my_uart/public.mk**. When this feature is disabled, the application does not pend on `ALT_LOG_PRINTF()` statements, even when no terminal capable of receiving Altera logging output is connected.

You can leave your `ALT_LOG_PRINTF()` debugging statements in the final source code version intended for production release, provided you set `hal.log_port` to `none`. With this setting, the definition of `ALT_LOG_PRINTF()` is empty, and so the compiler effectively removes these macro invocations. They have no impact on code footprint or performance unless you re-enable Altera logging.

For complete information about using the Altera logging functions, refer to "Using Character-Mode Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter in the *Nios II Software Developer's Handbook.*

Altera Logging Example

The example applications, **bit_bang_uart** and **hello_world_my_uart**, and the example BSP, **hal_my_uart**, use the Altera logging functions.

To see an example of Altera logging, run or debug **hello_world_my_uart**. You can see the logging messages in the Nios II Console view, as shown in Figure 23.

You enable Altera logging by setting the `hal.alt_log_flags` BSP setting to 3. Refer to the **create-this-bsp** script in the **bsp/hal_my_uart** directory for an example.

Figure 23. Altera Logging Output for hello_world_my_uart.c with ALT_LOG_FLAGS=3
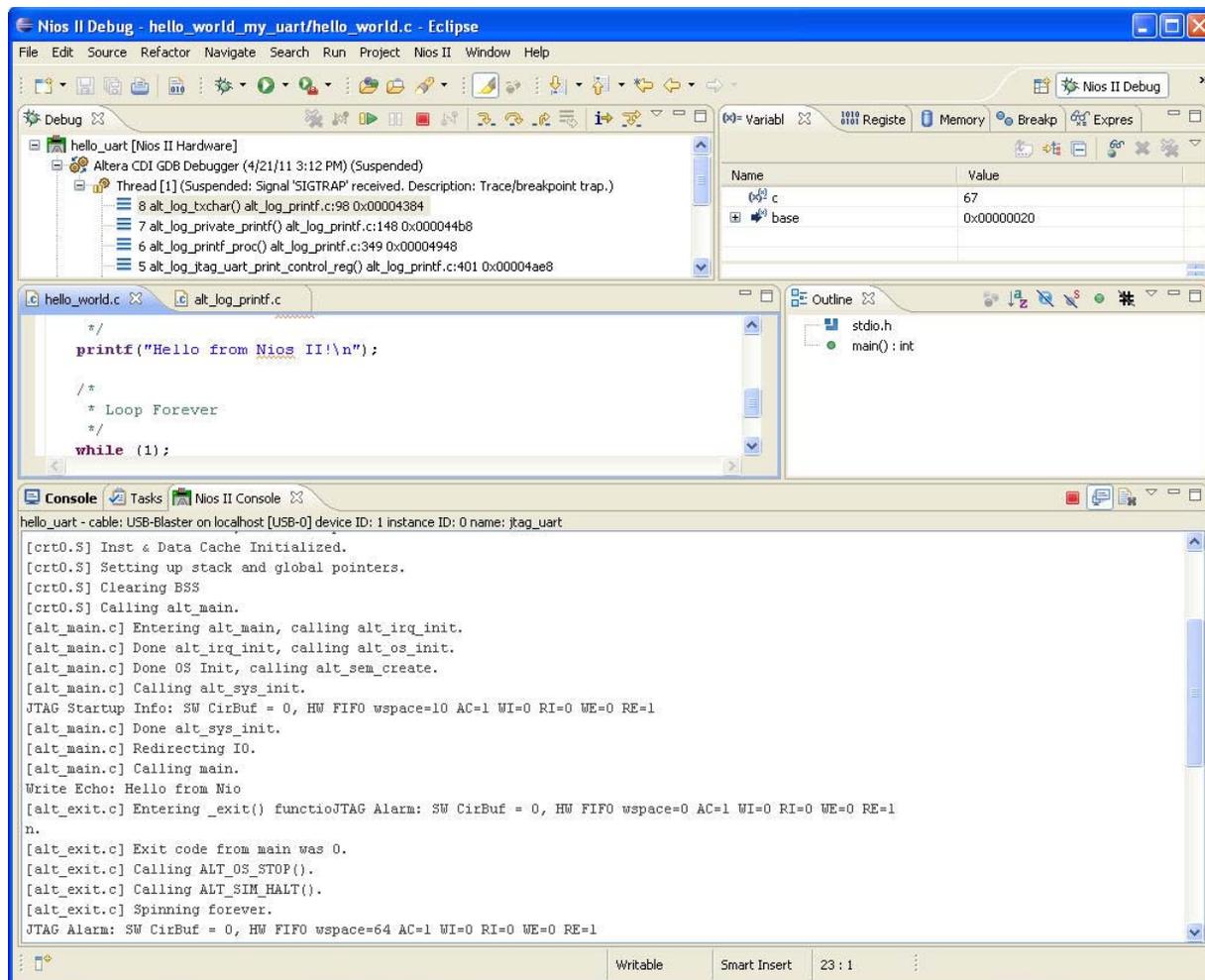
Conclusion

By dissecting the **Altera_Avalon_UART** component and the `my_uart_driver` HAL software device driver, and examining the UART status register bit manipulation at a fine-grained level of detail, you gained insight into the HAL device driver development process. You now have the tools necessary to develop and debug at this low level of the system, close to the hardware. Your set of tools includes analysis and debugging techniques for tackling even the most elusive and deterministic embedded software specification deviations.

With your new knowledge about the HAL's facilities, and with the array of techniques for debugging and development described in this document, you are now better prepared to write HAL software device drivers for your own embedded system's components. You can also apply these tools at higher levels in the software hierarchy.

Document Revision History

Table 1 shows the revision history for this document.

Table 1. Document Revision History

Date	Version	Changes
July 2011	4.0	Updated instructions and design example for Qsys
January 2010	3.0	■ Update for the Nios II Software Build Tools for Eclipse ■ Update the examples to run on the NEEK ■ Update the design example to use the EIC and VIC ■ Update the software examples to use the HAL enhanced interrupt API
November 2008	2.0	■ Nios II version 8.0 upgrade, adaptation of the **Altera_Avalon_UART** device driver to become the `my_uart_driver` device driver ■ Nios II Software Build Tools conversion for **my_uart** IP, **hal_my_uart** BSP, and **bit_bang_uart** and **hello_world_my_uart** applications ■ Changed size of document to 8.5 x 11 inches
August 2007	1.0	Initial release.