


Introduction

This application note explains the process of developing and debugging a hardware abstraction layer (HAL) software device driver, to aid device driver development for the HAL of the Nios® II system. The various software development stages are illustrated using the **Altera_Avalon_UART** as an example hardware device, and an example of a HAL software device driver called **my_uart**.

The Nios II Development Board, Cyclone® II 2c35 Edition, is used as an example hardware reference platform. This document shows the development process in steps, progressing from sending bits out the transmit pin from `main()` up to the construction of device access macros and automatic device initialization via `alt_sys_init()`.

Debugging tips are included, such as identifying UART transmission errors. Development is shown via the Nios II Software Development Tools. The resulting applications and board support package created with the command-line based Nios II Software Build Tools are then imported and debugged with the Nios II IDE. Discussions on interrupt latency, interrupt nesting, determinism, and which type of system calls cannot be included in a device driver interrupt service routine are included.

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

Prerequisites

This document is targeted at advanced systems developers with a basic understanding of the following:


- Nios II application development, including creating and building software applications and board support packages with the Nios II Software Build Tools.
- The Quartus® II software, including opening Quartus II projects that match the target board, launching SOPC Builder, and examining various peripheral component settings.
- Using the Quartus II Programmer tool to program an SRAM object file (.sof) to an FPGA via an Altera® USB-Blaster™ download cable.

 Refer to the *Nios II Hardware Development Tutorial* and the *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook* to gain the minimum prerequisite knowledge.

Using the HAL Architecture and Services

The HAL 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, DMA, and flash memory. The **Altera_Avalon_UART** is a character-mode class of HAL device, and as such, can be manipulated by the HAL API for character-mode class devices. Mutual exclusion resources are provided by MicroC/OS-II or 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 very small applications that are severely resource limited, use the **software_examples/bsp/hal_reduced_footprint** board support package, which minimizes the HAL. Use the **software_examples/app/hello_alt_main** software example as a minimal starting point for your application.

 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

The following components are required:

- Quartus II software version 8.0 or higher.
- Nios II EDS version 8.0 or higher.
- The **an459_software_80.zip** software archive, located on the Altera Nios II Literature web page, in zip format, under the link to this document.

The **an459_software_80.zip** software archive contains path information for properly locating the various source files under **ip** and **software_examples** directories for any Nios Development Board full_featured hardware reference design. This software archive contains the **bit_bang_uart** application, **hello_world_my_uart** application, **my_uart** device driver, and **hal_my_uart** board support package (BSP).

Developing the HAL UART Device Driver

The `my_uart` device driver is used as an example of a HAL device driver.

Preparing the `bit_bang_uart` Application and `hal_my_uart` Board Support Package


The first step is to set up a test environment for the UART. This example uses the Cyclone II 2c35 Nios development board `full_featured` hardware example.


Follow these steps to build the `bit_bang_uart` project:

1. Make a copy of the entire hardware reference directory, so that you can make changes to the hardware peripherals and software source files, while preserving the hardware reference design that was originally installed. Copy the entire directory contents for the Cyclone II 2c35 `full_featured` design to a new working writable directory, identified in this document as `<my_design>`. The Cyclone II 2c35 `full_featured` design is located in the following path:

```
<Altera installation directory>/nios2eds/examples/verilog/  
niosII_cycloneII_2c35/full_featured/
```

The default `<Altera installation directory>` is `C:\altera\80`.

 The working directory name you choose may not contain any spaces.

 If you use a different Nios development board's `full_featured` design, or use VHDL instead of Verilog HDL, adjust the path and file names as appropriate in the instructions that follow.

2. In your working directory, delete the `software_examples` directory from the `full_featured` directory.
3. Extract the `an459_software_80.zip` file to the `<my_design>/full_featured` directory. Be sure to preserve the directory structure of the extracted software archive. This creates a directory structure tree under `<my_design>/full_featured` with the following four leaf nodes:
 - `ip/my_uart`
 - `software_examples/bsp/hal_my_uart`
 - `software_examples/app/bit_bang_uart`
 - `software_examples/app/hello_world_my_uart`

Preparing the `my_uart` Software Device Driver

This section provides some background on how the `my_uart` software device driver is associated with a hardware component.

The directory to use for storing both the software device drivers and the hardware components is named by you. The name should be descriptive enough to identify the hardware component. The directory is located in the `<my_design>/ip` directory. The librarian searches for user component files named `<component_name>_sw.tcl` in directories below this `ip` directory.

The hardware component's software description file used for the `my_uart` software device driver is named `<my_design>/ip/my_uart/my_uart_sw.tcl`. This name must match the corresponding `<component_name>_hw.tcl` file generated by the Component Editor.

Hardware components provided by Altera, such as the `Altera_Avalon_UART`, are actually generated by Java and do not have the `<component_name>_hw.tcl` file. All hardware components generated by the Component Editor will 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 *Using the Nios II Software Build Tools* chapter in the *Nios II Software Developer's Handbook*.

Altera provides an additional tool with the Nios II processor version 8.0, the System Console, that is useful for testing hardware components and software device drivers, and for constructing board support packages. The System Console is not described in this application note.

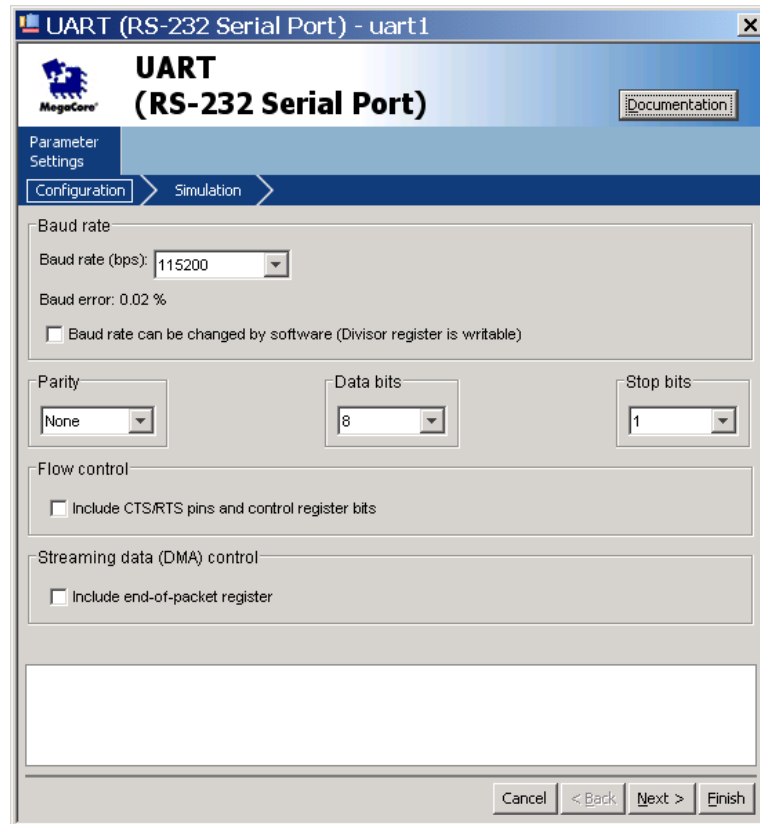
Configure the Altera_Avalon_UART Hardware Component Controlled by the my_uart Software Device Driver

Next, configure the `Altera_Avalon_UART` hardware component in SOPC Builder.

1. Open the Quartus II software, version 8.0 or later. On the File menu, click **Open Project**.
2. Browse to `<my_design>`.
3. Select the full_featured Quartus II project file, `NiosII_cycloneII_2c35_full_featured.qpf`, and click **Open**.
4. On the Tools menu, click **SOPC Builder**.
5. In SOPC Builder, in the **Module Name** column, double-click on `uart1`.

- In the **UART** dialog box, verify the baud rate is set to 115200 bps (**Figure 1**).
If you change the baud rate, click **Generate** in SOPC Builder to regenerate the system with the desired baud rate and then recompile the Quartus II project.

Figure 1. Verify UART Baud Rate

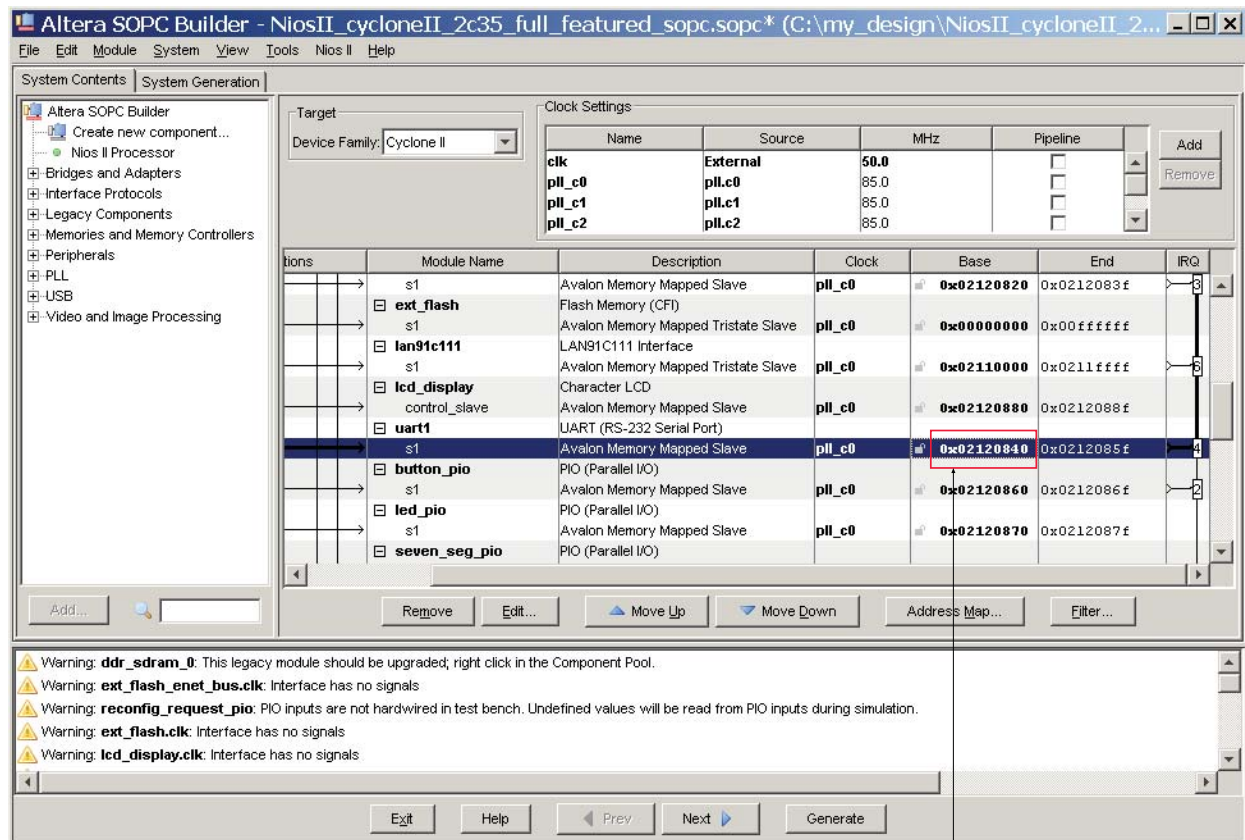


- Click **Finish**.

- In the **System Contents** tab of SOPC Builder, verify the value for the UART base address.

This reference design uses a value of 0x02120840 for the UART's register base address. If you are using a hardware reference design other than the full_featured design for the Cyclone II 2c35 board, the value of the UART's register base address may be different. Open SOPC Builder and find the UART base address for your board. **Figure 2** shows the base address for the UART used in this example.

Figure 2. Uart1 Peripheral Register Base Address



Base address for the UART

- In the Quartus II window, on the Tools menu, click **Programmer**.
- To program the **full_featured.sof** image to the development board, turn on **Program/Configure** and click **Start**.
- In SOPC Builder, on the Nios II menu, click **Nios II Command Shell**.
- Change the directory to `<my_design>/software_examples/app/bit_bang_uart`

- Execute the **create-this-app** script:

```
./create-this-app
```

This step may take several minutes to complete.

The **create-this-app** script specifies the board support package is named **hal_my_uart**, which associates the hardware component **uart1** with the software driver **my_uart_driver** (Figure 3). Device **uart1** is selected for STDIO via the **create-this-bsp** script for **hal_my_uart** BSP.

Figure 3. uart1 Mapping to my_uart_driver

Module Name:	uart1
Version:	default
Driver:	my_uart_driver

- Change the directory to `<my_design>/software_examples/bsp/hal_my_uart`
- Edit **alt_sys_init.c**.

Use your favorite editor. The vi editor is available from the Nios II Command Shell.
- Disable the automatic invocation of the HAL UART device driver initialization function by commenting out the `ALTERA_AVALON_UART_INIT()` macro invocation in **alt_sys_init.c**.
- Save **alt_sys_init.c**.
- Rebuild the **bit_bang_uart** project by changing the directory back to `<my_design>/software_examples/app/bit_bang_uart`, and executing `make`.
- Connect a serial cable from the 9-pin console port on the Nios development board to the COM1 port on your development host computer.

Importing Projects

Follow these steps to import the **bit_bang_uart** application project (and later the **hello_world_my_uart** application project), in addition to the **hal_my_uart**:

- In SOPC Builder, on the **SystemGeneration** tab, click **Nios II IDE** to launch the Nios II IDE Debugger.
- On the File menu, click **Import**. The **Import** dialog box appears.
- Expand the **Altera Nios II** folder, and select **Existing Nios II software build tools project or folder into workspace**.
- Click **Next**. The **Import** wizard appears.
- Click **Browse**. Navigate to and select the `<my_design>/software_examples/app/bit_bang_uart` directory.
- Click **OK**.
- Click **Finish**. The wizard imports the `bit_bang_uart` application.

8. Repeat steps 2 through 7, but instead import the `<my_design>/software_examples/bsp/hal_my_uart` board support package.



For additional information about importing Nios II Software Build Tools created projects, refer to the “Debugging Hello_World” sub-section of the “Getting Started” section of the *Introduction to the Nios II Software Build Tools* chapter in the *Nios II Software Developer’s Handbook*.

9. In the Nios II IDE window, in the Nios II C/C++ Projects perspective, expand the **bit_bang_uart** project, and open **bit_bang_uart.c**.
10. The first call to `IOWR()` in the `main()` procedure of **bit_bang_uart.c** shows that you can write to a hard-coded base address for `uart1` of `0x02120840`. If you are not using a Cyclone II 2c35 Nios development board, change this address value to that of the UART’s register base address in your SOPC Builder design. This example of a hard-coded address value demonstrates that it may be convenient when first verifying hardware functionality to specify an explicit memory address. This avoids any C pointer dereference software coding errors, providing confidence that the actual hardware peripheral register is definitely getting referenced.

After the communication link from the software to the hardware is established, you can change the hard-coded address to `UART1_BASE` (where `UART1` is the name of your UART peripheral in SOPC Builder). Using a hard-coded address can be useful when first bringing up new hardware to rule out any software errors in obtaining the peripheral’s memory-mapped registers base address. However, replacing the hard-coded register address with a symbolic definition based on the component’s name, such as `UART1_BASE`, enables the Nios II Software Build tools to update the software if the peripheral’s register base address changes. When the project is regenerated in SOPC Builder and recompiled in the Quartus II software, you need to execute `nios2-bsp` to update and rebuild the BSP and application. The **bit_bang_uart** application needs the new value of `UART1_BASE`, which is passed via **system.h**, a generated header file, to the **bit_bang_uart.c** source file.



For additional information about updating BSP files after an SOPC Builder change, refer to the “Coordinating with Hardware Changes” sub-section of the “Board Support Packages” section of the *Using the Nios II Software Build Tools* chapter in the *Nios II Software Developer’s Handbook*.

A simple way to cause all BSP and application files to be copied or regenerated is to delete the application Makefile (**app/bit_bang_uart/Makefile**) and the BSP’s **public.mk** file (**bsp/hal_my_uart/public.mk**), followed by invoking the **create-this-app** script in the application directory (**app/bit_bang_uart/create-this-app**).


Additionally, if the UART peripheral name for the hardware design you are using does not match “`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 SOPC Builder. Refer to [Figure 2](#).



The peripheral name as defined in SOPC Builder is converted to uppercase in the macros defined in **system.h**. `UART1_BASE` is a definition provided by **system.h**. The peripheral’s base address is created by appending `_BASE` to the peripheral’s name.

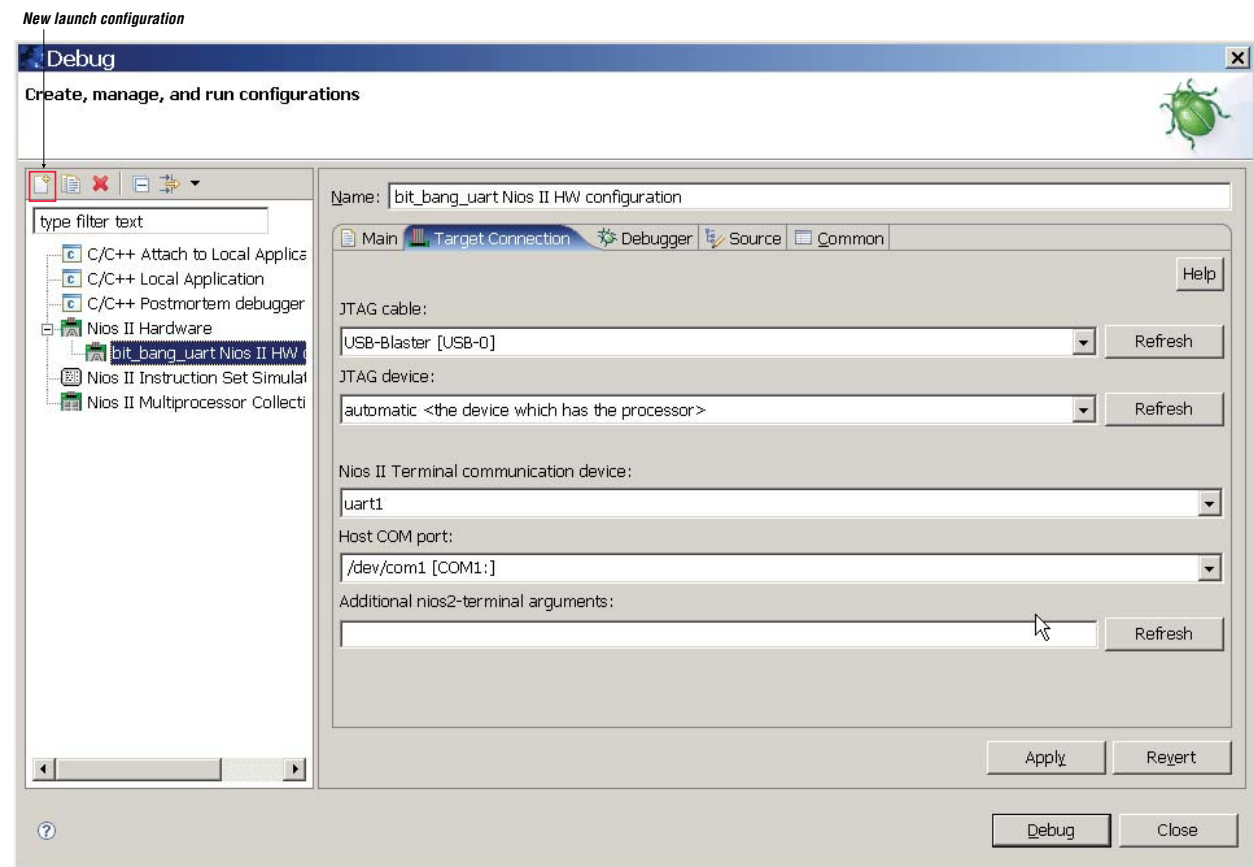
Debugging the bit_bang_uart Project

This section demonstrates debugging techniques with the **bit_bang_uart** project.

1. Open a Nios II Command Shell and run `nios2-terminal`. This shell receives the output to the `jtag_uart` from the `alt_log` device.
2. Click on the imported **bit_bang_uart** project in the C/C++ Projects window. On the Run menu, click **Debug** to prepare a debug configuration for the **bit_bang_uart** project.
3. In the Debug window, select **Nios II Hardware**.
4. Click the **New launch configuration** button, , to create a new debug configuration (refer to [Figure 4](#)).
5. On the **Main** tab, specify the SOPC Builder System PTF file, by browsing three directory levels up to the `<my_design>` root directory, and select the PTF file associated with this Quartus II project (for example, **NiosII_cycloneII_2c35_full_featured_sopc.ptf**).
6. Click **Open**.
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 fill in the “JTAG Cable” field with the matching USB-Blaster cable number to resolve the error.

8. Select the **Target Connection** tab, and set the **Nios II Terminal communication device** field to **uart1**.

Figure 4. bit_bang_uart Debug Configuration



For additional information about setting up a debug configuration for Nios II Software Build Tools created projects, refer to the “Setup a Debug Configuration” sub-section of the “Getting Started” section of the *Introduction to the Nios II Software Build Tools* chapter in the *Nios II Software Developer’s Handbook*.

9. Click **Debug**.
10. Depending on how the Nios II IDE preferences are configured, you might be automatically switched to the Debug perspective. If you are prompted to switch to the Debug perspective, click **Yes**.
11. On the Window menu, point to **Show View** and click **Memory** to open a Memory window.
12. If the Memory window is created in the lower left corner, sharing a tabbed area with the Console window, drag the memory tab to the upper right corner of the perspective. This arrangement allows you to view the Console and Memory windows simultaneously.


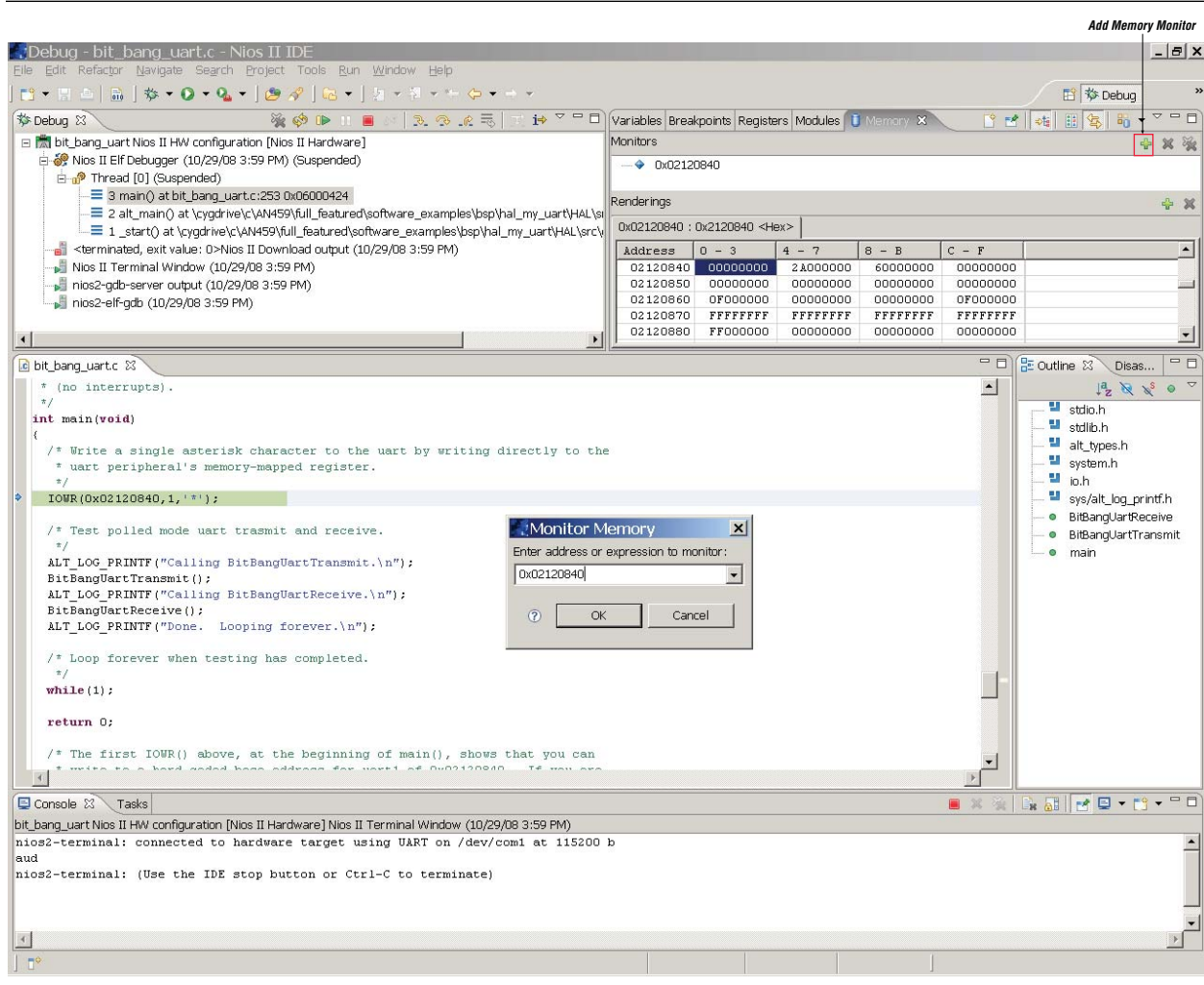


13. Click the  in the Memory window, as shown in [Figure 5](#). This action opens a **Monitor Memory** dialog box in which you can type the memory address that you want to monitor.
14. Enter the UART peripheral's register base address, as shown in [Figure 5](#) (0x02120840 for the Cyclone II full_featured design's uart1 peripheral).

Figure 5. Monitor Memory Address Specification Window

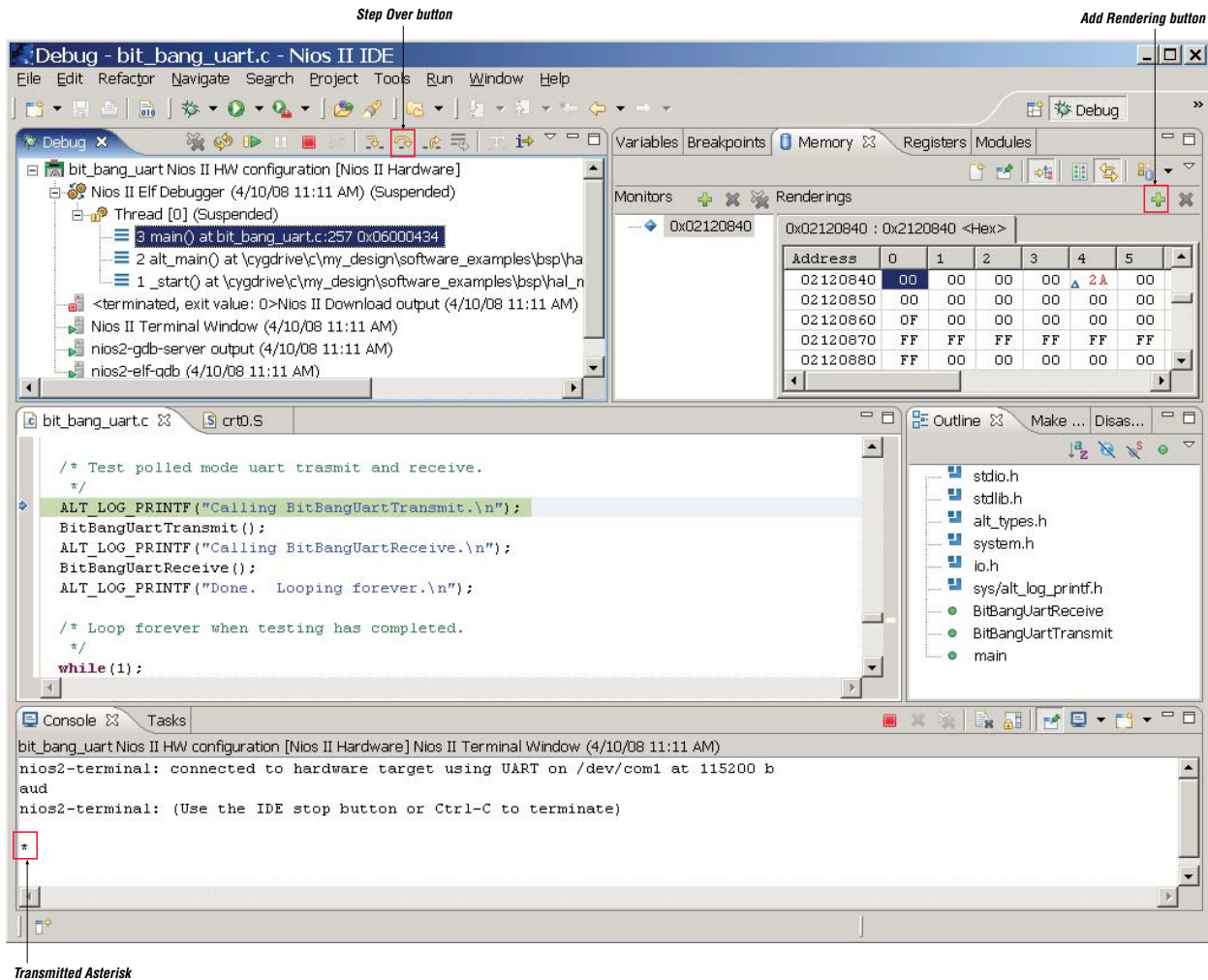


15. Click **OK**.
16. Adjust the size of the window so that you can see several memory address values in the Memory window, as shown in [Figure 6](#).
17. Click in the Memory window, under the column labeled 0-3.
18. Right-click and click **Format**. For **Column Size**, select 1 unit per column.
19. Click **OK**.
20. Use the **Step Over** button  to advance the program execution over the `IOWR()` macro. This macro transmits an asterisk to the debugger's Console window by writing directly to the UART's transmit register, as shown in [Figure 6](#).

 If you do not see an asterisk in the Console window, verify your hardware cable is properly connected and your UART peripheral base address matches the one in your SOPC Builder system.

The red numbers in the memory monitor window 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 window is the hexadecimal value for the asterisk character (*).

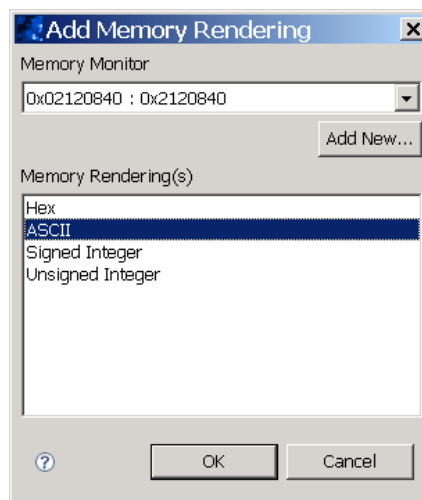
Figure 6. Transmit Asterisk



- You can view the Memory window in ASCII rather than hexadecimal. Click the **Add Rendering** button on the right of the Memory window (refer to [Figure 7](#)) to add a new ASCII rendering.

22. In the **Add Memory Rendering** dialog box (Figure 7), select **ASCII** and click **OK**.
The 2A in the Memory window changes to an asterisk.

Figure 7. Adding an ASCII Rendering to the Memory Window



23. You can transmit characters over the UART by directly changing memory values in the Memory window. Type an *h* into the cell currently occupied by the asterisk in the Memory window, followed by a return. This cell represents the transmit register, offset one long word from the UART's peripheral base address. Type an *i* into this same cell in the Memory window, followed by a return. The word *hi* appears in the Console window (Figure 8).


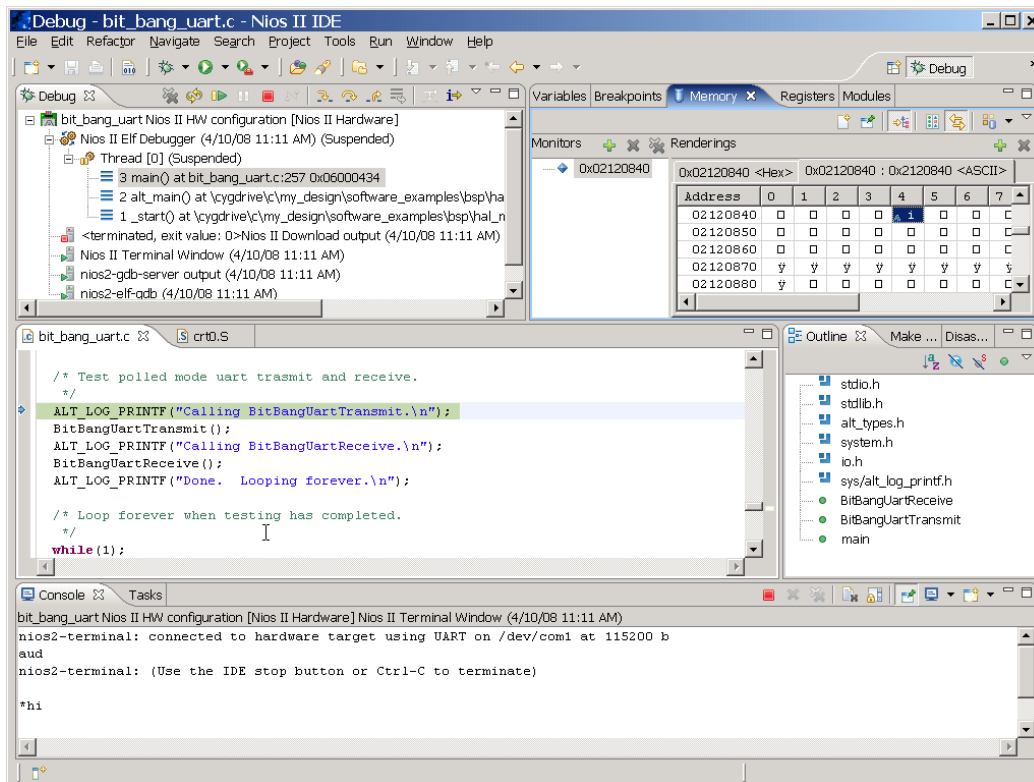
 The peripheral memory-mapped registers bypass the cache. Therefore, the status register value displayed in the Memory window reflects any changes to the status register made by the peripheral. The IOWR () and IORD () macros always bypass the cache.

Figure 8. Directly Manipulating the Peripheral Register Via the Memory Window

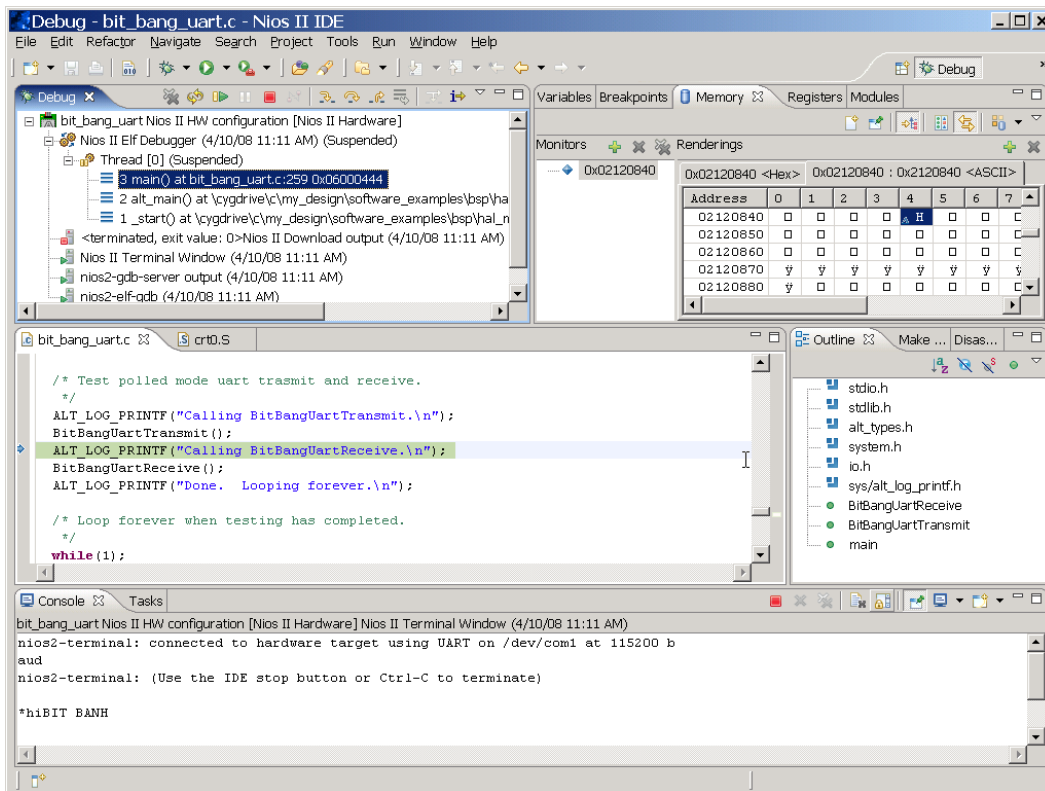


The BitBangUartTransmit Function


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

Step over the BitBangUartTransmit function. The characters displayed in the Console window are “BIT BANH”, as shown in Figure 9. The following steps explain why the string ends with an *H* instead of a *G*.

Figure 9. Stepping Over the BitBangUartTransmit Function Displays “BIT BANH”



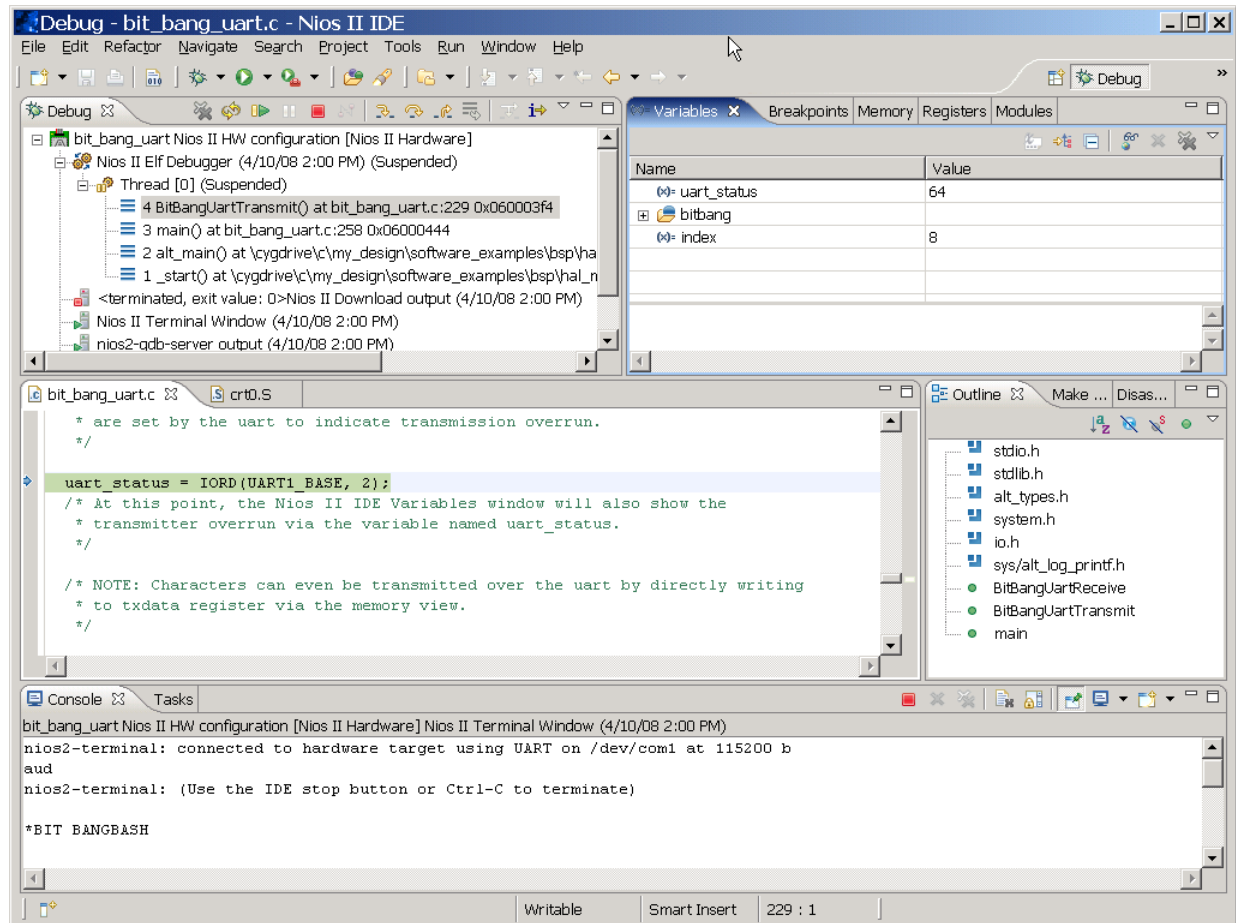
Perform the following steps:

1. Restart the debugging session. On the Run menu, click **Debug Last Launched**.
2. Step over until you reach the call to the BitBangUartTransmit function.
3. Use the **Step Into** button  to step into the BitBangUartTransmit function. Next, use the **Step Over** button to execute one line at a time. Continue stepping through the function until the string BIT BANGBASH is displayed.

To get this result, `bit_bang_uart.c` first writes a value of zero to the status register to clear any existing errors on the UART. This step is accomplished by the `IOWR()` macro, along with the `UART1_BASE`.

Next, a loop cycles through the `bitbang []` array, printing out the characters “BIT BANG” to the UART. The transmit ready bit is checked before each subsequent character transmission to prevent any overruns. Immediately after the loop, the characters “BASH” are transmitted one after the other. If you step each line to the end of the `BitBangUartTransmit` function, the characters “BIT BANGBASH” are transmitted over the UART, as shown in [Figure 10](#).


Figure 10. BIT BANGBASH Transmitted if Function Is Stepped One at a Time



Perform the following steps:

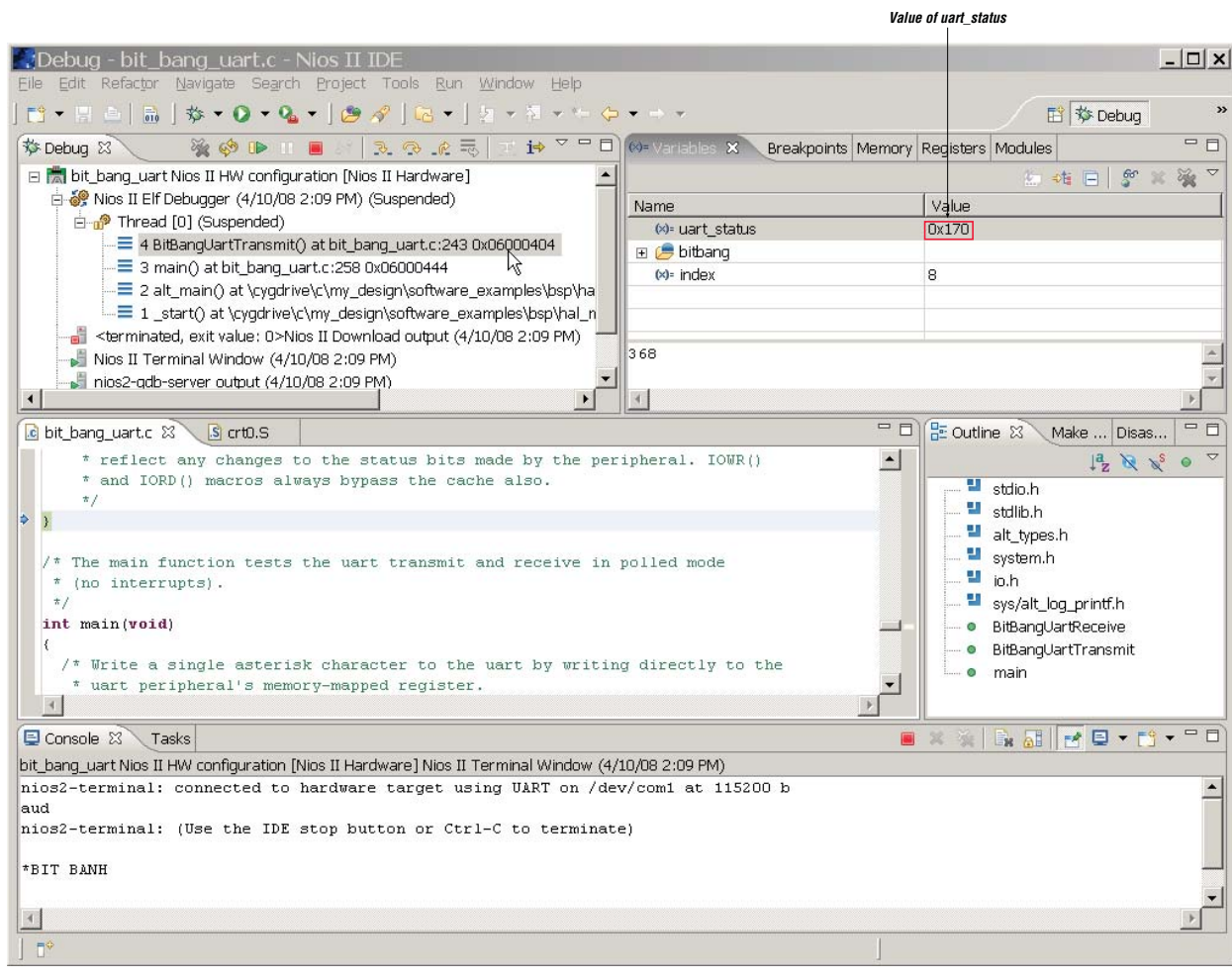
1. Restart the debugging session. On the Run menu, click **Debug Last Launched**.
2. Instead of stepping into `BitBangUartTransmit()`, place a breakpoint on the `uart_status` variable assignment that follows the calls to `IOWR` with the letters “BASH”. (To set a breakpoint, double-click in the gray area left of the line.)

If the Variables window is not set to display hexadecimal as the default, select the `uart_status` variable name. Right-click, point to **Format**, and click **Hexadecimal**.

3. Click the **Resume** button . The program runs until the breakpoint is hit.

- Step over the assignment of `uart_status`. The Variables window shows that the value of `uart_status` has changed to `0x170`, as shown in [Figure 11](#).

Figure 11. Value of `uart_status` Variable Is `0x170`



The register map for the Altera Avalon UART core, described in the [UART Core](#) chapter in volume 5 of the *Quartus II Handbook*, shows that the status register's value of `0x170` indicates that the exception bit (bit 8) and the toe bit (bit 4) are set. The toe bit is the transmitter overrun bit. By not waiting for the transmitter to be ready before writing the additional characters ("GBASH"), the transmitter has been overrun and only the last character, *H*, is transmitted.

The BitBangUartReceive Function

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

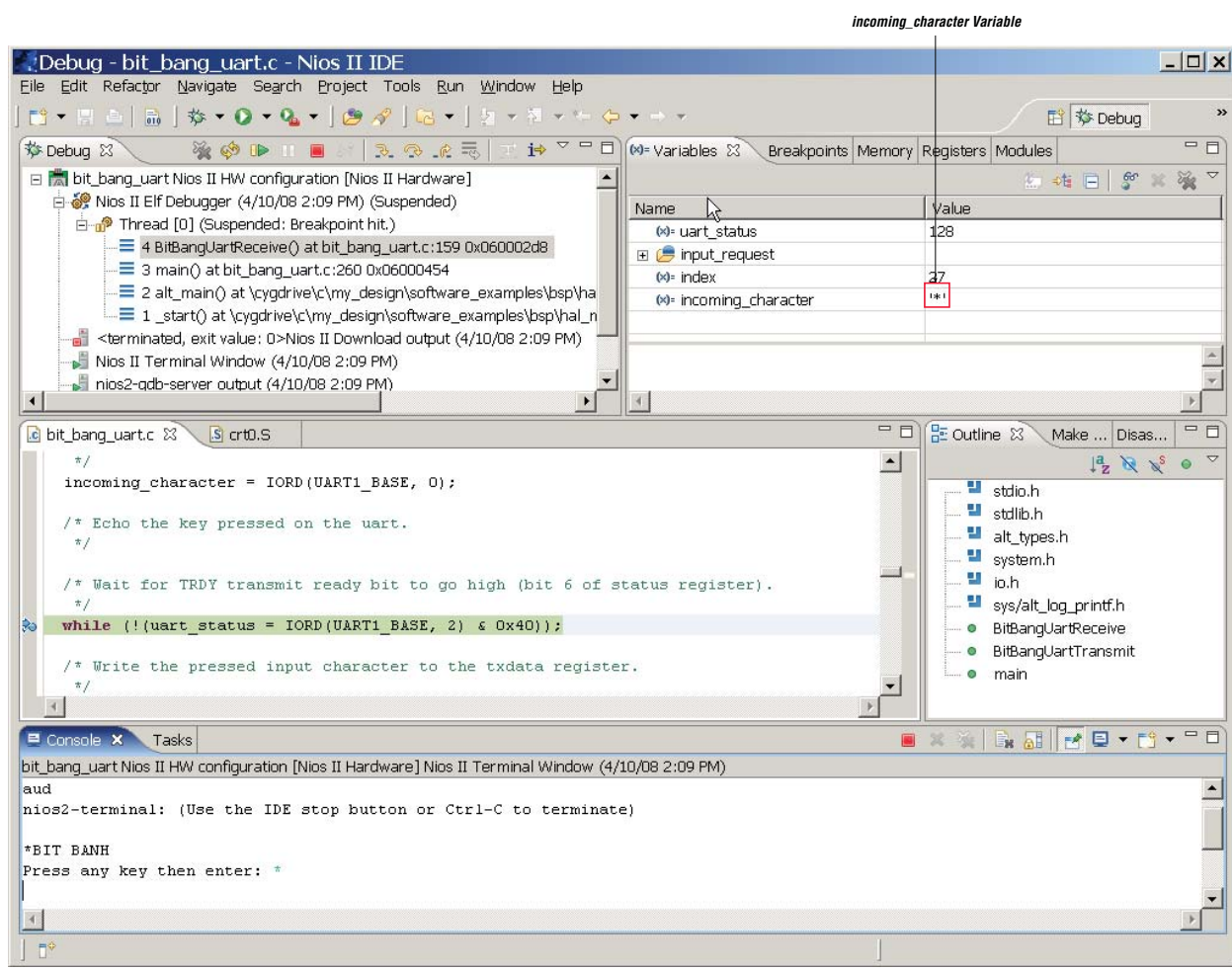
Perform the following steps:

- Step into the `BitBangUartReceive` function.

2. Set a breakpoint on the while loop immediately after the assignment of the incoming_character variable.
3. Click the **Resume** button.
4. The Nios II processor is waiting in the while loop just above the incoming_character assignment for the RRDY (receive ready) bit to go high. Click at the bottom of console and enter an "*" (asterisk).
5. Press **Enter**. The debugger hits the breakpoint you set.
6. Examine the Variables window (expand it if necessary to see the incoming_character variable). The incoming_character variable holds the asterisk you sent via the Console window, as shown in [Figure 12](#).

Both the transmit and receive functions of the UART in polled mode have been verified to work.

Figure 12. incoming_character Variable Is Set to the Character Entered on the Console



Creating Device Access Macros

When the functionality of the various peripheral registers has been validated by 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 name, and 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**. The benefit of this procedure is automatic incorporation of any changes made to the component base address in SOPC Builder. For example, to access the UART's transmit register in **bit_bang_uart.c**, an `IOWR()` macro is used, along with a hard-coded offset (with a value of 1) for indexing this register. 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)
```

Also, the Altera Nios II component provides the address construction macro, `IO_CALC_ADDRESS_NATIVE()`, that is used in the UART device access macros (from `nios2eds/components/altera_nios2/HAL/inc/io.h`). `IO_CALC_ADDRESS_NATIVE()` is a macro that adds the second parameter (the offset in system bus width units [for example, 32 bits]) to the first parameter (the peripheral's register base address), to derive the direct address of the specified peripheral register. The `IORD()` and `IOWR()` macros translate to the Nios II assembler instructions, `ldwio` and `stwio`, respectively.

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);
```

The same functionality can be achieved 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 has been written.

Altera recommends that you create device access macros for all of your custom peripheral'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. The 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 device name. As a result, the MY UART software 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 hardware device name, which in this case is `altera_avalon_uart.h`. This restriction is necessary for the automatic construction of the `alt_sys_init.c` device initialization generated C source file.

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

The file `bit_bang_uart.c` demonstrates how to write source code. The source code development progresses toward a complete device driver. It starts from direct access of the peripheral's registers and goes on to validating the proper functioning of the `Altera_Avalon_UART` hardware. This `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 hardware device, perform the following steps:

1. Use `IOWR()` macros with hard address values in `main()` to write values directly to the memory-mapped UART registers. This method is the most direct way to interface with the UART, and is useful for validating proper functioning of the hardware component, 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 peripheral, modeled after `bit_bang_uart.c`, write the device access macros.
3. Use these device access macros to develop and test polled routines for the `init`, `read`, and `write` functions.
4. Write the interrupt service routines (ISRs) for interrupt driven mode. An interrupt-driven software device driver routine responds to hardware interrupts generated by the hardware device to indicate that there is useful work to be performed. This method is much more efficient than a polled mode device driver routine, which consumes and wastes Nios II microprocessor clock cycles by constantly querying the hardware device to ask if there is useful work to be performed. An interrupt service routine for a hardware device allows the Nios II microprocessor to do other useful work while the hardware device is either idle or operating autonomously, and therefore does not require the involvement of the Nios II microprocessor. Use `alt_irq_register()` to install the ISRs in `main()`.
5. After the ISR and polled routines are tested from `main()`, create and test the `INIT` and `INSTANCE` macros. These initialization macros are invoked by `alt_sys_init.c` to automatically 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 instance of the hardware device to hold device instance specific information, such as the baud rate and the transmit and receive memory buffers. At this point, the `alt_irq_register` calls are moved from `main` to the device `init` function.

 For more information about this `alt_dev` structure, refer to the “Character-Mode Device Drivers” section of the *Developing Device Drivers for the Hardware Abstraction Layer* chapter in the *Nios II Software Developer’s Handbook*.

Understanding the Device-Specific INSTANCE and INIT Macros

The `INSTANCE` macro creates the `alt_dev` structure, which represents an instance of the HAL device (the hardware component). This macro creates unique device instance specific data structures.

The `INIT` macro must perform the following tasks:

- Create mutual exclusion resources
- Install the component's interrupt service routine with `alt_irq_register()`
- Register the `alt_dev` structure with `alt_dev_reg()`
- Enable interrupts

Integrating a New HAL Device Driver into the Board Support Package

Integration enables the following services:

- 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 found in the SOPC Builder 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:

`DeviceDrivers[SopcBuilder]\<component_folder>`

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

`<my_design>\ip\<component_folder>`

The device driver source code files are placed in folders in `<component_folder>`. The file names conform to the following:

- `\inc\<component>_regs.h`
- `\HAL\inc\<component>.h`
- `\HAL\src\<component>.c`
- HAL devices can be accessed by services that are provided for a particular HAL device class. For example, `Altera_Avalon_UART` is a character mode hardware 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 software build tools, refer to the “Integrating a Device Driver into 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 integrating a device driver with the HAL, refer to 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 SOPC Builder component, refer to the *Component Editor* and *SOPC Builder Components* chapters in volume 4 of the *Quartus II Handbook*.
- For details about the Component Editor tool, refer to the *Component Editor* chapter in volume 4 of 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, which do nothing in this example. This is a device driver source code portability feature.

The MY UART software device driver creates two semaphores and one event flag. The two semaphores are called **read_lock** and **write_lock**. They are used by the MY UART software device driver 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.

Mechanisms for Debugging the HAL UART Device Driver

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

- You can monitor individual hardware component 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** hardware device via the MY UART software device driver.
- 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()` MY UART software device driver function.
- You can set breakpoints in the driver's interrupt service routines, or even set watchpoints on UART memory-mapped registers to halt the processor when a character is received. There may be consequences to setting a breakpoint in an interrupt service routine (ISR). When you resume, there may be problems with other devices that did not get their interrupts handled. However, this is sometimes the best way to debug the driver for a particular device. You can always just 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 wrong interrupt number is passed to `alt_irq_register()`, the result is a device interrupt that is not properly handled. When interrupts are enabled after low-level system initialization, there is no way to clear the interrupt source. The application will not work correctly. The Nios II IDE debugger may even stop communicating with the processor.

Techniques for 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 as a Nios II IDE project. Next, regenerate the files which make up the `hal_my_uart` board support package. This time, instead of commenting out the invocation of the `ALTERA_AVALON_UART_INIT` macro, 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 device behavior.

Perform the following steps:

1. Delete the `public.mk` generated file. Enter the following command in the Nios II Command Shell:

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


This causes the `hal_my_uart` BSP files, including `alt_sys-init.c`, to be regenerated at the next build.

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:

- The `create-this-bsp` script for the `hal_my_uart` board support package is invoked.
- A new `public.mk` file is generated.
- The software device descriptors `stdout`, `stderr`, and `stdin` are set to `uart1`. This is done during the `nios2-bsp` invocation in the `<my_design>/software_examples/bsp/hal_my_uart/create-this-bsp` script.
- The software device driver called `my_uart_driver` is created in the `<my_design>/ip/my_uart` directory and is associated with the `Altera_Avalon_UART` hardware device. This is done in `<my_design>/ip/my_uart/my_uart_sw.tcl`.
- The software device driver called `my_uart_driver` is set to the hardware component named `uart1`. This is done via the Tcl script passed to the `nios2-bsp` invocation called `<my_design>/software_examples/bsp/hal_my_uart/hal_my_uart.tcl`.
- `alt_sys_init.c` is regenerated with the invocation of `ALTERA_AVALON_UART_INIT`.
- The `libhal_bsp.a` board support package library is built in the `<my_design>/software_examples/bsp/hal_my_uart` directory.
- The `hello_world_my_uart.elf` file is built in the `<my_design>/software_examples/app/hello_world_my_uart` directory.

 `nios2-bsp` can be invoked with the `--debug` parameter, which causes verbose generation of information about the above construction steps, and 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 into the Nios II IDE as described in “Importing Projects” on page 7, substituting the `hello_world_my_uart` application for the `bit_bang_uart` application.

Setting Breakpoints in the HAL MY UART Software Device Driver

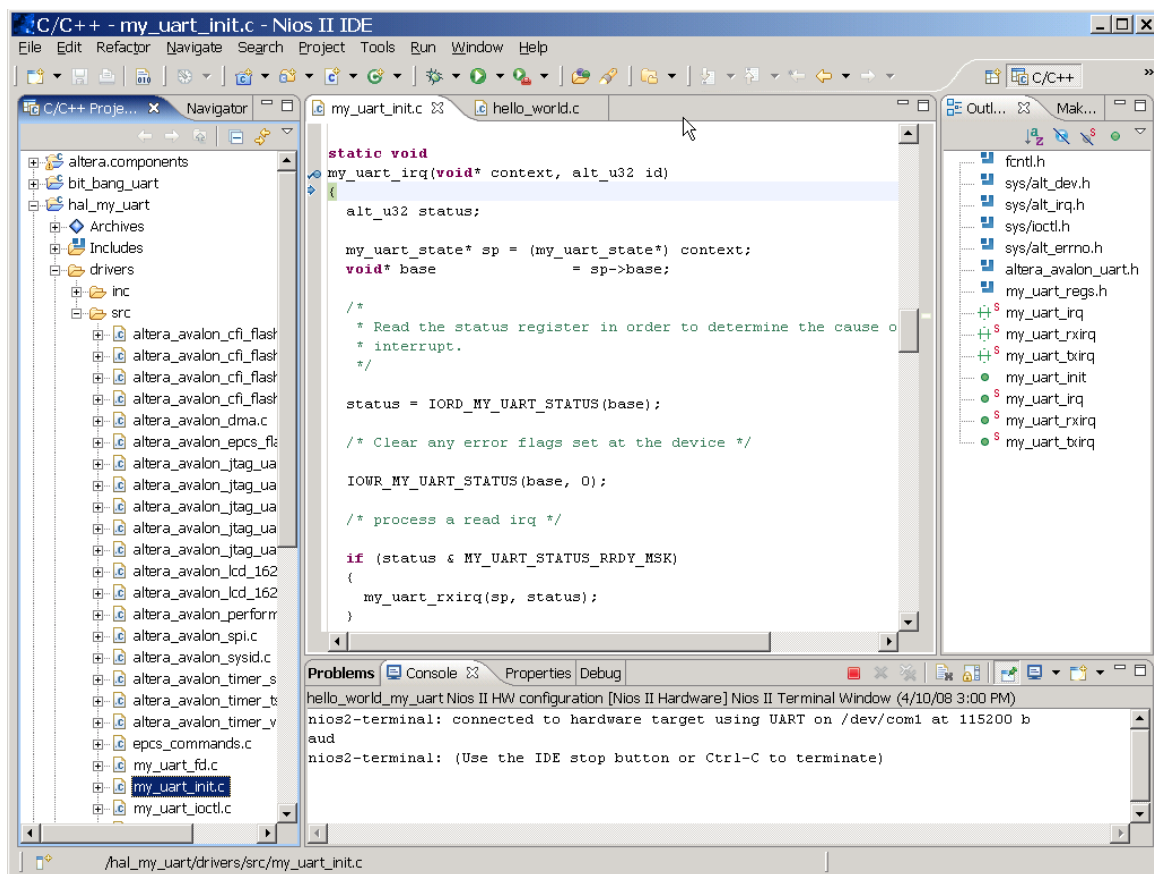
This section demonstrates the use of breakpoints to examine HAL device driver activity. Perform the following steps:

1. After the `hello_world_my_uart` project is imported, 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 on the function named `my_uart_irq()`, as shown in Figure 13.

Figure 13. Setting a Breakpoint on `my_uart_irq`



3. Create a Debug configuration for **hello_world_my_uart** by following the same steps in “[Debugging the bit_bang_uart Project](#)” on page 9, starting at step 2, and substituting the **hello_world_my_uart** application for the **bit_bang_uart** application. Download, and execute the **hello_world_my_uart** application. The Nios II processor stops at the `my_uart_irq()` invocation.
4. Step up to and over the following assignment of the status register:


```
status = IORD_MY_UART_STATUS(base);
```

The status register now holds the value 0x60. This value indicates bits 5 and 6 are set. According to the MY UART register description, these two bits indicate transmit ready and transmit. The UART driver is now in an interrupt context, ready to transmit the first character of the string “Hello from Nios II!”.
5. 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 when the entire string “Hello from Nios II!” has been transmitted.
6. Remove the breakpoint.



After exploring the actions of an interrupt service routine, the rest of the system is in an unknown state, because it could not respond to other interrupt requests while stopped in the driver. Therefore, you need to start a new debugging session. Download the elf image again to restart the program with Debug.

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 stops 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.

On the Run menu, click **Debug**. Click the **Debugger** tab. Turn off the **Use FS2 console window for trace and watchpoint support** option.

To set a watchpoint, perform the following steps:

1. Start the debugging session for the **hello_world_my_uart** project.
2. Click on the **Breakpoints** tab. Right-click in the Breakpoints window, and click **Add Watchpoint**.
3. 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 Cyclone II full_featured hardware design, this value is 0x02120844.
4. In the Access section, turn on **Write** and turn off **Read**.
5. Click **OK**. The **Add Watchpoint** dialog box closes.

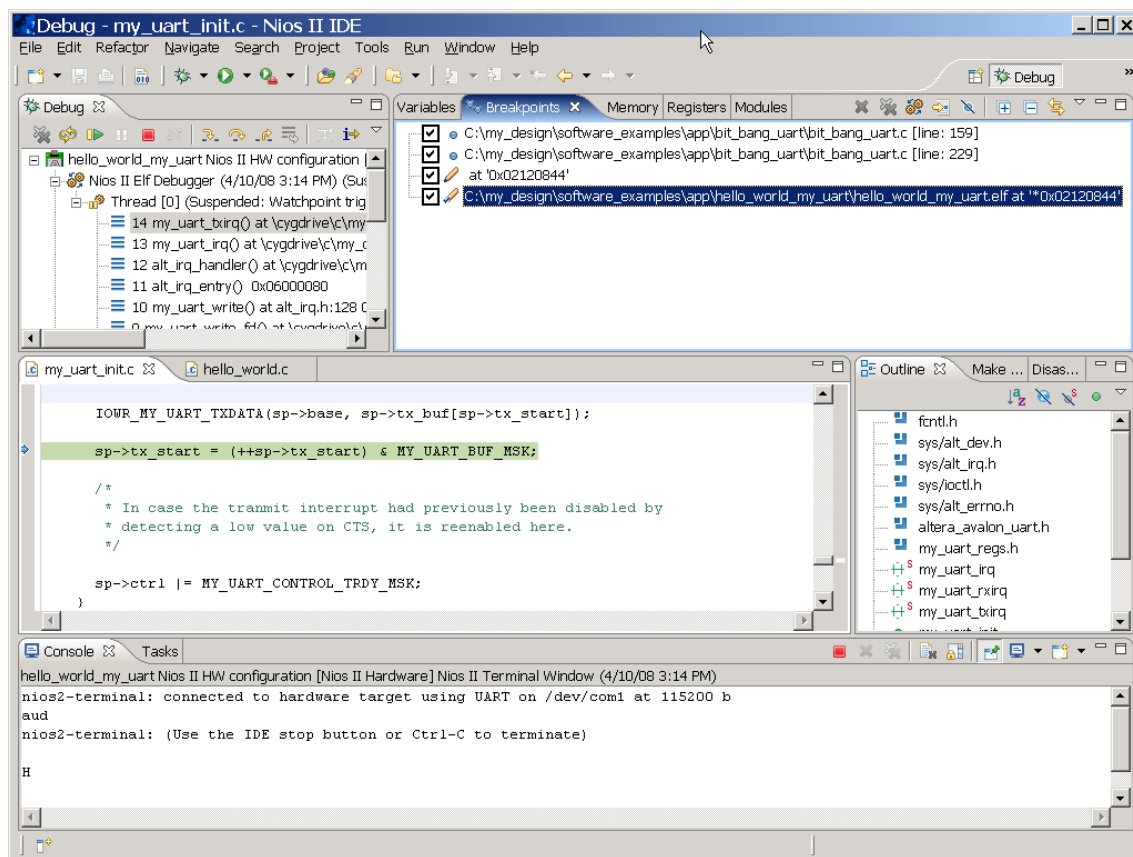
6. Click the **Resume** button.

The Nios II processor executes until it writes the first character to be transmitted via the UART, an “H”. This step occurs in the `my_uart_txirq()` function when the macro `IOWR_MY_UART_TXDATA()` gets invoked, as shown in [Figure 14](#).

View the transmit register value in the Nios II IDE Memory window. Note that the value changes when the watchpoint is hit.

Look at the call stack in the upper left corner of the Debug Perspective. Each call leading up to this point is recorded, including each function invoked to process the transmit interrupt. The `alt_irq_entry()` function calls `alt_irq_handler()`, which calls `my_uart_irq()`, which calls `my_uart_txirq()`.

Figure 14. Setting Watchpoints in the UART's Transmit Register



Setting the Reduced Device Drivers and Lightweight Device Drivers API Options

There are HAL settings that can be defined for the board support package to allow the HAL to be configured. These settings are configured with the `--set` parameter to the `nios2-bsp` invocation in the `create-this-bsp` script. The settings are described 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 functionality. Additionally, they set `#define` parameters that need to be examined and handled in the MY UART software device driver. The settings are stored in the `summary.html` file generated by `nios2-bsp`, in `<my_design>/software_examples/bsp/hal_my_uart`.

- **Reduced device drivers**—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 the UART device:
 - Sets `#define ALT_USE_SMALL_DRIVERS`
 - Activates polled-mode only for the UART device
 - No floating-point `printf()` or `sprintf()`
 - Flow control is ignored
- **Lightweight device driver API**—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 the UART device:
 - Sets `#define ALT_USE_DIRECT_DRIVERS`
 - File descriptors cannot be created, eliminating the option of using a file system.
 - STDIO device descriptors cannot be redirected to actual device descriptors `alt_main()` by the `alt_io_redirect()` call.
 - Calling `open()` or `close()` generates a link time error.
 - Causes direct calls to your UART device driver via macros, bypassing the device manipulation function invocations normally accessed through the file descriptor structure. The macros used are defined in `alt_driver.h` (in the **Device Drivers [NiosII]\altera_hal\HAL\inc\sys** directory).

Figure 15 shows an excerpt of the `summary.html` file generated by `nios2-bsp`.


Figure 15. Excerpt of `summary.html` Generated by `nios2-bsp`

Setting Name:	<code>hal_enable_lightweight_device_driver_api</code>
Identifier:	<code>ALT_USE_DIRECT_DRIVERS</code>
Default Value:	0
Value:	0
Type:	Boolean
Destination:	<code>public_mk_define</code>
Description:	Enables lightweight device driver API. This reduces code and data footprint by removing the HAL layer that maps device names (e.g. <code>/dev/uart0</code>) to file descriptors. Instead, driver routines are called directly. The <code>open()</code> , <code>close()</code> , and <code>lseek()</code> routines will always fail if called. The <code>read()</code> , <code>write()</code> , <code>fstat()</code> , <code>ioctl()</code> , and <code>isatty()</code> routines only work for the stdio devices. If true, adds <code>-DALT_USE_DIRECT_DRIVERS</code> to <code>ALT_CPPFLAGS</code> in <code>public.mk</code> .
Restrictions:	The Altera Host and read-only ZIP file systems can't be used if <code>hal_enable_lightweight_device_driver_api</code> is true.
Setting Name:	<code>hal_enable_mul_div_emulation</code>
Identifier:	<code>ALT_NO_INSTRUCTION_EMULATION</code>
Default Value:	0
Value:	0
Type:	Boolean
Destination:	<code>public_mk_define</code>
Description:	Adds code to emulate multiply and divide instructions in case they are executed but aren't present in the CPU. Normally this isn't required because the compiler won't use multiply and divide instructions that aren't present in the CPU. If false, adds <code>-DALT_NO_INSTRUCTION_EMULATION</code> to <code>ALT_CPPFLAGS</code> in <code>public.mk</code> .
Restrictions:	none
Setting Name:	<code>hal_enable_reduced_device_drivers</code>
Identifier:	<code>ALT_USE_SMALL_DRIVERS</code>
Default Value:	0
Value:	0
Type:	Boolean
Destination:	<code>public_mk_define</code>
Description:	The drivers are compiled with reduced functionality to reduce code footprint. Not all drivers observe this setting. The <code>altera_avalon_uart</code> and <code>altera_avalon_jtag_uart</code> drivers switch to a polled-mode of operation. The <code>altera_avalon_cfi_flash</code> , <code>altera_avalon_epcs_flash_controller</code> , and <code>altera_avalon_lcd_16207</code> drivers are removed. You can define a symbol provided by each driver to prevent it from being removed. If true, adds <code>-DALT_USE_SMALL_DRIVERS</code> to <code>ALT_CPPFLAGS</code> in <code>public.mk</code> .
Restrictions:	none

For example, a call to `alt_putstr()`, which normally is treated as a call to the run-time library function `fputs()`, instead gets 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` software device driver. It provides a convenient way to map the names of the `ALTERA_AVALON_UART_INIT` and `ALTERA_AVALON_UART_INSTANCE` macros that are tied to the hardware component class name to names that are specific to the `my_uart` software 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`.

 For more information about the **Reduced device drivers** and **Lightweight device driver API** options, refer to the “Reducing Code Footprint” section in the *Developing Programs Using the Hardware Abstraction Layer* chapter and the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer’s Handbook*.

Interrupt Latency and Determinism

Interrupt latency is defined as the difference between the time that a hardware component asserts an interrupt and the time that the first instruction of the interrupt service routine (ISR) executes.

Determinism is defined as 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 the discussion on latency in the *Exception Handling* chapter of the *Nios II Software Developer’s Handbook*.

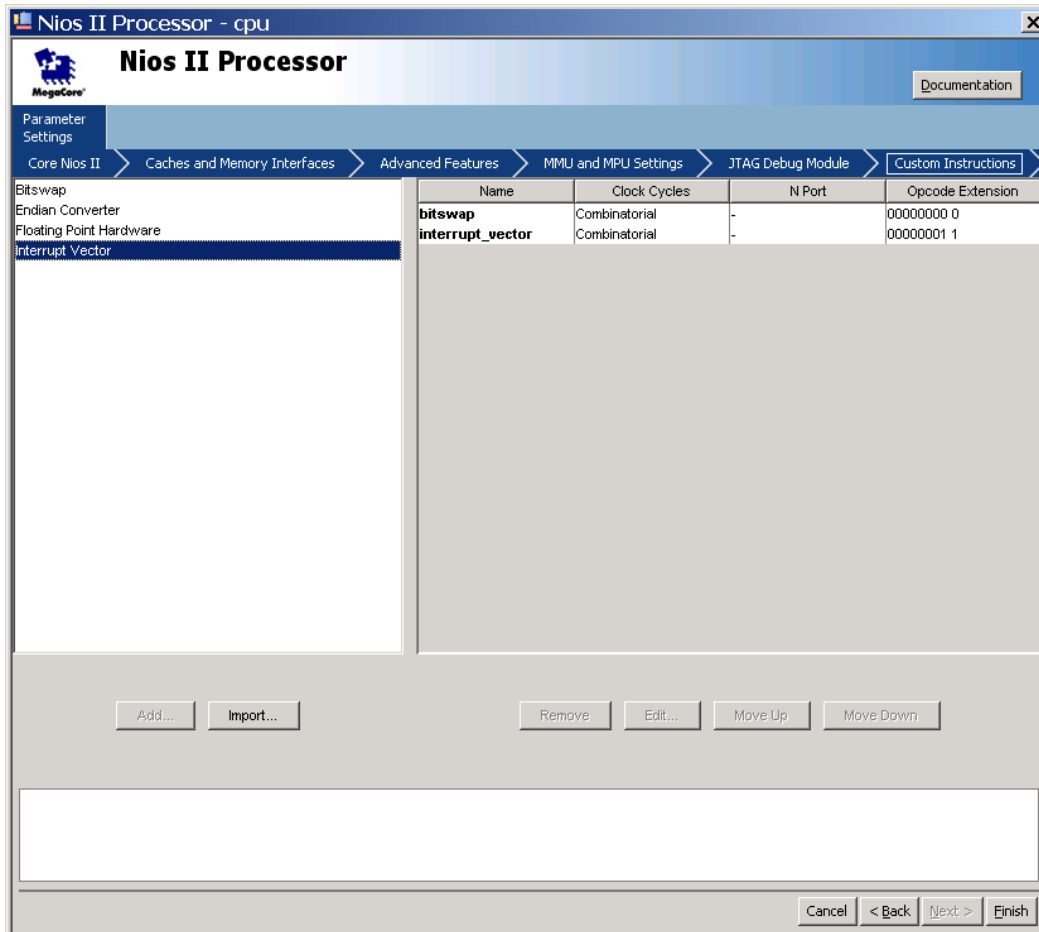
To minimize interrupt latency, thus directly improving system determinism, follow these guidelines:

- In the software interrupt service routine, do as little as possible to clear the interrupt.
- Complete non-critical-section interrupt processing outside of the interrupt context. If an operating system is used, 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.

On the **Custom Instructions** tab, use the **Interrupt Vector** setting (Figure 16) for the Nios II component in SOPC Builder to process the interrupt funnel in hardware as a custom instruction, which is faster than processing the interrupt funnel in software. This setting essentially replaces the `alt_irq_handler()` source code with equivalent hardware. There can only be one interrupt vector custom instruction component in a Nios II processor. If the interrupt vector custom instruction is present in the Nios II processor, the HAL source detects it at compile time and generates code using the custom instruction. The interrupt vector custom instruction improves both average and worst-case interrupt latency by up to 20%. To achieve the lowest possible interrupt latency, consider using tightly-coupled memories so that interrupt handlers can run without cache misses.

For more information about tightly coupled memory, refer to *Using Tightly Coupled Memory with the Nios II Processor Tutorial*.

Figure 16. Interrupt Vector Custom Instruction



For details of the interrupt vector custom instruction implementation, refer to the “Exception and Interrupt Controller” 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. Functions that pend on resources must *not* be called from within an ISR. Doing so can have fatal consequences, from the destruction of overall system latency to complete system deadlock. This includes any functions that can end up waiting for any resource, such as `printf`, as well as any direct resource pend calls, such as `Alt_Sem_Pend`.

Avoid using `alt_irq_interruptible()`, which can enable ISR nesting, but is likely to worsen interrupt latency (unless ISR is abnormally long) because of the interrupt context switch overhead. If the ISR is long enough that you are considering making it interruptible, consider moving much of the less time-critical processing of the interrupt outside of the ISR into a task. The ISR should 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.

ALT_LOG Message Logging Mechanism

`alt_log` is a logging mechanism that can be very useful for debugging device drivers. To enable this function for your application and BSP, perform the following steps:



The example `bit_bang_uart` and `hello_world_my_uart` applications and `hal_my_uart` BSP already incorporate these steps.

1. Include the following header file in source files that call `ALT_LOG_PRINTF()`:

```
#include "sys/alt_log_printf.h"
```

2. In the `nios2-bsp` command, set the `hal.log_port` parameter to the desired logging device, such as `jtag_uart` (refer to the **create-this-bsp** script in the `bsp/hal_my_uart` directory). The logging device must be of type `altera_avalon_jtag_uart` or `altera_avalon_uart`. This causes `system.h` to be generated with definitions for `ALT_LOG_PORT_TYPE` and `ALT_LOG_PORT_BASE`, as well as injecting into `public.mk` the define statement for `ALT_LOG_ENABLE`.

3. Set the desired `ALT_LOG_FLAGS` level with the following Tcl command:

```
add_sw_property alt_cppflags_addition -DALT_LOG_FLAGS=3
```

This command adds `-DALT_LOG_FLAGS=3` to the `ALT_CPP_FLAGS` make variable in `public.mk`. (See the `my_uart_sw.tcl` custom software component TCL script in the `ip/my_uart` directory.)

Macros are used to bypass the HAL driver and access the peripheral directly, so that messages can be printed during the boot process before the devices get initialized. The `.sof` image does not need to be regenerated in SOPC Builder or recompiled in the Quartus II software.

The `alt_log` device writes are blocking, so the `nios2-terminal` command must be executed from a Nios II Command Shell prompt (refer to [Figure 17 on page 35](#)) to accept the `alt_log` output to the `altera_avalon_jtag_uart` device (when `hal.log_port` is set to `jtag_uart`, a device of type `altera_avalon_jtag_uart`) in order for the Nios II application to complete initialization. Otherwise, the application pends on an `ALT_LOG_PRINTF()` statement until the `alt_log` device's output buffer can be drained. You can add `alt_log` diagnostic messages to your code by invoking `ALT_LOG_PRINTF()`, a macro that handles most `printf` options except for floating point (`%f` or `%g`).

The `alt_log` feature can be disabled by not defining `hal.log_port` as a parameter to the `nios2-bsp` command invoked by the `bsp/hal_my_uart/create-this-bsp` script. Disabling `alt_log` has the effect of not setting `ALT_LOG_ENABLE` in the Makefile for board support package in `bsp/hal_my_uart/public.mk`. Disabling this feature prevents the application from pending on the completion of `ALT_LOG_PRINTF` statements, even when no terminal capable of receiving `ALT_LOG` output is connected, such as the `nios2-terminal` in this example. Additionally, disabling the `alt_log` feature has the advantage of leaving zero residual impact in the compiled and linked application elf file. This creates an elf image which is identical when compared with an elf image compiled with the same `bit_bang_uart.c` source code written without the additional `ALT_LOG_PRINTF()` macro invocations.

You can leave your `ALT_LOG_PRINTF()` debugging statements in the final source code version intended for production release, ready to be turned on by a simple recompile with `-DALT_LOG_ENABLE`. Of course, the determinism of the application is impacted when `ALT_LOG_ENABLE` is defined, due to the collection and output of `alt_log` messages. All the `alt_log` mechanisms are macros, and so get eliminated by the compiler when not enabled. The result is that 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.

Extra Logging Options and `ALT_LOG_FLAGS`

Aside from boot messages, these extra logging options are built into `alt_log`. Each option has its own on-flag define, the details of each option is outlined in the table below.

A second preprocessor define, `ALT_LOG_FLAGS`, can be set to provide some grouping for turning on these extra logging options. The flag levels are based on the intrusiveness of performance—the higher the level, the more processor time the `alt_log` options take, slowing execution. The `ALT_LOG_FLAGS` levels are defined as follows:

- `ALT_LOG_FLAGS = 0` or not defined (Default)—Only `alt_log_boot` is turned on.
- `ALT_LOG_FLAGS = 1`—Above, plus `alt_log_sys_clk` and `alt_log_jtag_uart_startup_info`.
- `ALT_LOG_FLAGS = 2`—All of the above, plus `alt_log_jtag_uart_alarm` and `alt_log_write`.
- `ALT_LOG_FLAGS = 3`—All of the above, plus `alt_log_jtag_uart_isr`.
- `ALT_LOG_FLAGS = -1`—Silent mode – no `alt_log` outputs.

Each logging option has their own on-flag define, so the default flag groupings can be overridden. Setting the on-flag to 1 will turn that option on; anything else will turn off the option.

Table 1 shows the extra logging options.

Table 1. Extra Logging Options (Part 1 of 2)

Option	Description	
alt_log_sys_clk	Description	Prints out a message from the system clock interrupt handler every interval. This tells the user if the system is still alive. Every message is appended with a count that increments with each print out. The default interval is 1 second.
	On-flag name	ALT_LOG_SYS_CLK_ON_FLAG_SETTING
	Extra switches	ALT_LOG_SYS_CLK_INTERVAL – the interval in number of ticks. The default is (number of ticks for one second) * MULTIPLIER.ALT_LOG_SYS_CLK_INTERVAL_MULTIPLIER – can increase the interval in multiples of one second. Default is 1.
	Sample Output	System Clock On 0 System Clock On 1
alt_log_write	Description	Every time alt_write() is called, (basically any print statements that go to STDOUT), the first N characters will be echoed to a logging message. The message starts with "Write Echo." Default – N is 15 characters.
	On-flag name	ALT_LOG_WRITE_ON_FLAG_SETTING
	Extra switches	ALT_LOG_WRITE_ECHO_LEN. Default is 15.
	Sample Output	Write Echo: Hello from Nio
alt_log_jtag_uart_startup_info	Description	At JTAG UART driver initialization, print out a line with the number of characters in the software transmit buffer (SW CirBuf), and JTAG UART control register contents (HW FIFO). The SW CirBuf number might be negative as it is the (tail pointer – head pointer) value for a circular buffer; the JTAG UART control register fields can be found in the Altera Embedded Peripherals Handbook.
	On-flag name	ALT_LOG_JTAG_UART_STARTUP_INFO_ON_FLAG_SETTING
	Extra switches	None
	Sample Output	JTAG Startup Info: SW CirBuf = 0, HW FIFO wspace=64 AC=0 WI=0 RI=0 WE=0 RE=1
alt_log_jtag_uart_alarm	Description	Creates an alarm object to print out the same JTAG UART information as alt_log_jtag_uart_startup_info, but at a repeated interval. Default interval is 0.1 second, or 10 messages a second.
	On-flag name	ALT_LOG_JTAG_UART_ALARM_ON_FLAG_SETTING
	Extra switches	ALT_LOG_JTAG_UART_TICKS = number of ticks between alarms. Default is ticks_per_second / DIVISOR. ALT_LOG_JTAG_UART_TICKS_DIVISOR = number of times a second to print out. Default is 10.
	Sample Output	JTAG Alarm: SW CirBuf = 0, HW FIFO wspace=45 AC=0 WI=0 RI=0 WE=0 RE=1

Conclusion

By dissecting the `Altera_Avalon_UART` peripheral hardware and the `my_uart` 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, close to the hardware, level of the system. The `printf()` function is not available this deep in the software hierarchy, but your set of tools now 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 peripheral devices. Many more device drivers are provided with the Nios II EDS, ranging from the more simple PIO to the relatively complex, such as Ethernet. Furthermore, these tools can also be applied at higher levels in the software hierarchy.

Referenced Documents

This application note references the following documents:

- *Component Editor* chapter in volume 4 of the *Quartus II Handbook*
- *Developing Device Drivers for the Hardware Abstraction Layer* chapter in the *Nios II Software Developer's Handbook*
- *Developing Programs Using the Hardware Abstraction Layer* chapter in the *Nios II Software Developer's Handbook*
- *Exception Handling* chapter in the *Nios II Software Developer's Handbook*
- *HAL API Reference* chapter in the *Nios II Software Developer's Handbook*.
- *Introduction to the Nios II Software Build Tools* chapter in the *Nios II Software Developer's Handbook*
- *Nios II Hardware Development Tutorial*
- *Nios II Software Build Tools Reference* chapter in the *Nios II Software Developer's Handbook*
- *Overview of the Hardware Abstraction Layer* chapter in the *Nios II Software Developer's Handbook*
- *Processor Architecture* chapter in the *Nios II Processor Reference Handbook*
- *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*
- *UART Core* chapter in volume 5 of the *Quartus II Handbook*
- *Using the Nios II Software Build Tools* chapter in the *Nios II Software Developer's Handbook*
- *Using Tightly Coupled Memory with the Nios II Processor Tutorial*

Document Revision History

Table 2 shows the revision history for this application note.

Table 2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2008 v2.0	<ul style="list-style-type: none"> ■ Nios II version 8.0 upgrade, adaptation of the Altera_Avalon_UART device driver to become the my_uart 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 	Updated for version 8.0.
August 2007 v1.0	Initial release.	—



101 Innovation Drive
San Jose, CA 95134
www.altera.com
Technical Support
www.altera.com/support

Copyright © November 2008. Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.



I.S. EN ISO 9001