Section II. Building Systems with SOPC Builder

Section II of this volume provides instructions on how to use SOPC Builder to achieve specific goals. Chapters in this section serve to answer the question, "How do I use SOPC Builder?" Many chapters in this handbook provide design examples which you can download free from **www.altera.com**. Design file hyperlinks are located with individual chapters linked from the Altera web site.

This section includes the following chapters:

- Chapter 6, Developing Components for SOPC Builder
- Chapter 7, Building Systems with Multiple Clock Domains

Revision History

The following table shows the revision history for Chapters 6– 7. These version numbers track the document revisions; they have no relationship to the version of SOPC Builder or the Quartus® II software.

Chapter(s)	Date / Version	Changes Made
6	February 2005 v1.0	Initial release.
7	February 2005 v1.0	Initial release.

6. Developing Components for SOPC Builder

QII54007-1.0

Introduction

This chapter describes the design flow to develop a custom SOPC Builder component. This chapter provides tutorial steps that guide you through the process of creating a custom component, integrating it into a system, and downloading it to hardware.

This chapter is divided into the following sections:

- *Component Development Flow* (see page 6–3) .
- *Design Example: Pulse-Width Modulator (PWM) Slave* (see page 6–8). This design example demonstrates developing a component with a single Avalon™ slave interface. In this section, you will start with a ready-made HDL design, package it into a SOPC Builder component, and then instantiate it in a system. If you have a development board, you can download the design to hardware and see the PWM work.
- *Sharing Components* (see page 6–28). This section shows you how to relocate component files to use them in other systems, or share them with other designers.

SOPC Builder Components and the Component Editor

SOPC Builder provides a component editor that lets you create and edit your own SOPC Builder components. By following the procedures described in this document, you will learn to use the component editor and turn any custom logic module into an SOPC Builder component.

Once your custom logic is packaged as component, you can instantiate it in an SOPC Builder system in the same manner as commercially available SOPC Builder Ready components. You can share your component with other designers to encourage design reuse.

Typically, a component is comprised of the following:

- Hardware files: HDL modules that describe the component hardware
- Software files: A C-language header file that defines the component register map, and driver software that allows programs to control the component
- Component description file (**class.ptf**): This file defines the structure of the component, and provides SOPC Builder the information it needs to integrate the component into a system. The component

editor generates this file automatically based on the hardware & software files you provide, and the parameters you specify in the component editor GUI.

After you create the hardware and software files that describe the component, you use the component editor to package those file into an SOPC Builder component. You can also use the component editor later to re-edit the component, if you ever update the hardware or software files.

Assumptions About the Reader

This chapter assumes that you are familiar with the following:

- Building systems with SOPC Builder. For details, see the *Introduction to SOPC Builder* and *Tour of the SOPC Builder User Interface* chapters in Volume 4 of the *Quartus II Handbook*.
- SOPC Builder components. For details, see the *SOPC Builder Components* chapter in Volume 4 of the *Quartus II Handbook*.
- Basic concepts of the Avalon interface. You do not need extensive knowledge of the Avalon interface, such as transfer types or signal timing, to use the design example(s) provided with this chapter. However, to create your own custom components, you need a fuller understanding of the Avalon interface. For details, see the *Avalon Interface Specification*.

Hardware and Software Requirements

To use the design example(s) in this chapter, you must have the following:

- Design files for the example design – A hyperlink to the design files appears next to this chapter on the SOPC Builder literature page. Visit **www.altera.com/sopcbuilder**.
- Quartus® II Software version 4.2 or higher – Both Quartus II Web Edition and the fully licensed version will work with the example design.
- Nios® II development kit version 1.1 or higher – Both the evaluation edition and the fully licensed version will work with the example design.
- Nios development board and an Altera® USB-Blaster™ download cable (Optional) – You can use any of the following Nios development boards:
  - Stratix® II Edition
  - Stratix Edition
  - Stratix Professional Edition
  - Cyclone™ Edition

If you do not have a development board, you can follow the hardware development steps, but you will not be able to download the complete system to a working board.

You can download the Quartus II Web Edition software and the Nios II Development Kit, Evaluation Edition for free from the Altera Download Center at **www.altera.com**.

Before you begin, you must install the Quartus® II software and Nios II development tools.

Component Development Flow

This section provides an overview of the development process for SOPC Builder components, covering both the hardware and software aspects. This section focuses on the design flow for components with a single Avalon slave interface. However, these steps are easily extrapolated to components with a master port, or multiple master and slave ports.

Typical Design Steps

A typical development sequence for a slave component includes the following steps, not necessarily in this order:

1. Specify the hardware functionality.
2. If a microprocessor will be used to control the component, specify the application program interface (API) to access and control the hardware.
3. Based on the hardware and software requirements, define an Avalon interface that provides:
    a. Appropriate control mechanisms
    b. Adequate throughput performance
4. Write HDL that describes the hardware in either Verilog or VHDL.
5. Test the component hardware alone to verify correct operation.
6. Write a C header file that defines the hardware-level register map for software.
7. Use the component editor to package the initial hardware and software files into a component.

8. Instantiate the component into a simple SOPC Builder system module.

9. Test register-level accesses to the component using a microprocessor, such as the Nios II processor. You can perform verification in hardware, or on an HDL simulator such as ModelSim.

10. If a microprocessor will be used to control the component, write driver software.

11. Iteratively improve the component design, based on in-system behavior of the component:

    a. Make hardware improvements and adjustments.

    b. Make software improvements and adjustments.

    c. Incorporate hardware and software changes into the component using the component editor.

12. Build a complete SOPC Builder system incorporating one or more instances of the component.

13. Perform system-level verification. Make further iterative improvements, if necessary.

14. Finalize the component and distribute it for design reuse.

The design process for a master component is similar, except for software development aspects.

Hardware Design

As with any logic design process, the development of SOPC Builder component hardware begins after the specification phase. Coding the HDL is an iterative process, as you write and verify the HDL logic against the specification.

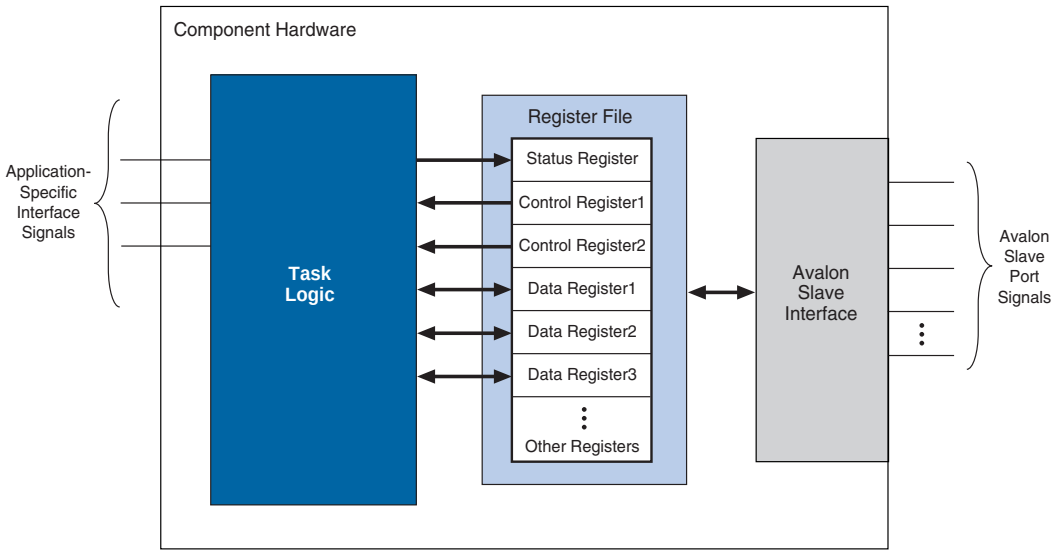
The architecture of a typical component consists of the following functional blocks:

- *Task Logic* - The task logic implements the component's fundamental function. The task logic is design dependent.

- *Register File* - The register file provides a path for communicating signals from inside the task logic to the outside world, and vice versa. The register file maps internal nodes to addressable offsets that can be read or written by the Avalon interface.
- *Avalon Interface* - The Avalon interface provides a standard Avalon front-end to the register file. The interface uses any Avalon signal types necessary to access the register file and support the transfer types required by the task logic. The following factors affect the Avalon interface:
  - How wide is the data to be transferred?
  - What is the throughput requirement for the data transfers?
  - Is this interface primarily for control or for data? That is, do transfers tend to be sporadic, or come in continuous bursts?
  - Is the hardware relatively fast or slow compared to other components that will be in a system?

Figure 6–1 shows a block diagram of a typical component with one Avalon slave port.

Figure 6–1. Typical Component with One Avalon Slave Port



Software Design

If your intent is for a microprocessor to control your component, then you must provide software files that define the software view of the component. At a minimum, you must define the register map for each

slave port that is accessible to a processor. The component editor lets you package a C header file with the component to define the software view of the hardware.

Typically, the header file declares macros to read and write each register in the component, relative to a symbolic base address assigned to the component. The following example shows an excerpt from the register map for an Altera-provided UART component for the Nios II processor.

Example: Register Map for a Component

```
#include <io.h>
#define IOADDR_ALTERA_AVALON_TIMER_STATUS(base)        __IO_CALC_ADDRESS_NATIVE(base, 0)
#define IORD_ALTERA_AVALON_TIMER_STATUS(base)          IORD(base, 0)
#define IOWR_ALTERA_AVALON_TIMER_STATUS(base, data)    IOWR(base, 0, data)

#define ALTERA_AVALON_TIMER_STATUS_TO_MSK              (0x1)
#define ALTERA_AVALON_TIMER_STATUS_TO_OFST             (0)
#define ALTERA_AVALON_TIMER_STATUS_RUN_MSK             (0x2)
#define ALTERA_AVALON_TIMER_STATUS_RUN_OFST            (1)

#define IOADDR_ALTERA_AVALON_TIMER_CONTROL(base)       __IO_CALC_ADDRESS_NATIVE(base, 1)
#define IORD_ALTERA_AVALON_TIMER_CONTROL(base)         IORD(base, 1)
#define IOWR_ALTERA_AVALON_TIMER_CONTROL(base, data)   IOWR(base, 1, data)

#define ALTERA_AVALON_TIMER_CONTROL_ITO_MSK            (0x1)
#define ALTERA_AVALON_TIMER_CONTROL_ITO_OFST           (0)
#define ALTERA_AVALON_TIMER_CONTROL_CONT_MSK           (0x2)
#define ALTERA_AVALON_TIMER_CONTROL_CONT_OFST          (1)
#define ALTERA_AVALON_TIMER_CONTROL_START_MSK          (0x4)
#define ALTERA_AVALON_TIMER_CONTROL_START_OFST         (2)
#define ALTERA_AVALON_TIMER_CONTROL_STOP_MSK           (0x8)
#define ALTERA_AVALON_TIMER_CONTROL_STOP_OFST          (3)
```

Software drivers abstract hardware details of the component so that software can access the component at a high level. The driver functions provide the software an API to access the hardware. The software requirements vary according to the needs of the component. The most common types of routines initialize the hardware, read data, and write data.

Driver software is dependent on the target processor. The component editor lets you easily package software drivers for the hardware abstraction layer (HAL) used by the Nios II processor development tools. To provide drivers for other processors, you must accommodate the needs of the development tools for the target processor.

For details on writing drivers for the Nios II HAL, see the *Nios II Software Developer's Handbook*. It is instructive to look at the software files provided for other ready-made components. The Nios II development kit provides many components you can use as reference. See <*Nios II kit path*>**/components/**.

Verifying the Component

You can verify the component in incremental stages, as you complete more and more of the design. Typically, you first verify the hardware logic as a unit (which might comprise multiple smaller stages of verification), and later you verify the component in a system.

Unit Verification

To test the task logic block alone, you use your preferred verification method(s), such as behavioral or register transfer level (RTL) simulation tools. Similarly, you can verify all component logic, including the register file and the Avalon interface(s), using your preferred verification tools.

After you package the HDL files into a component using the component editor, the Nios II development kit offers an easy-to-use method to simulate read and write transactions to the component. Using the Nios II processor's robust simulation environment, you can write C code for the Nios II processor that initiates read and write transfers to your component. The results can be verified either on the ModelSim simulator or on hardware, such as a Nios development board.

See *AN351: Simulating Nios II Embedded Processor Designs* for more information.

System-Level Verification

After you package the HDL files into a component using the component editor, you can instantiate the component in a system, and verify the functionality of the overall system module.

SOPC Builder provides support for system-level verification for RTL simulators such as ModelSim. While SOPC Builder produces a testbench for system-level verification, the capability of the simulation environment is largely dependent on the components included in the system.

During the verification phase, including a Nios II processor in the system can be useful to get the benefits of the Nios II simulation environment. Even if your component has no relationship to the Nios II processor, the auto-generated ModelSim simulation environment provides an easy-to-use base that you can build upon to verify other logic in the system.

Design Example: Pulse-Width Modulator Slave

This section uses a pulse-width modulator (PWM) design example to demonstrate the steps to create a component and instantiate it in a system. This component has a single Avalon slave port.

In this section, you will perform the following steps:

1. Install the design files.
2. Review the example design specifications.
3. Package the design files into an SOPC Builder component.
4. Instantiate the component in hardware.
5. Compile the hardware design in the Quartus II software, and download the design to a target board.
6. Exercise the hardware using Nios II software.

Install the Design Files

Before you proceed, you must install the Nios II development tools and download the PWM example design from the Altera web site. The hardware design used in this chapter is based on the **standard** hardware example design included with the Nios II development kit.

Do not use spaces in any directory path names when installing the design files. If the path contains spaces, SOPC Builder might not be able to access the files.

Perform the following steps to setup the design environment:

1. Unzip the contents of the PWM zip file to a directory on your computer. This document will refer to this directory as the <*PWM design files*> directory.
2. On your host computer file system, locate the following directory:

<Nios II kit path>/examples/*<verilog or vhdl>*/*<board version>*/standard

Each development board has a VHDL and Verilog version of the design. You can use either one. Table 6–1 shows the names of the directories for the available Nios development boards.

Table 6–1. Design File Directories

Nios Development Board	Tutorial Directory
Stratix II Edition	niosII_stratixII_2s60_es
Stratix Edition	niosII_stratix_1s10 or niosII_stratix_1s10_es
Stratix Professional Edition	niosII_stratix_1s40
Cyclone Edition	niosII_cyclone_1c20

For demonstration purposes, the figures in this chapter show the case of the Verilog design on the Nios Development Board, Cyclone Edition.

3. Copy the **standard** directory to a new location. By copying the design files, you avoid corrupting the original design. This document will refer to the newly-created directory as the **<*Quartus II project*>** directory.

Review the Example Design Specifications

This section discusses the design specifications for the provided PWM example design, giving details on each of the following topics:

- PWM Design Files
- Functional Specification
- PWM Task Logic
- Register File
- Avalon Interface
- Software API

In a typical design flow, it is the designer's responsibility to specify the behavior of the component.

PWM Design Files

Table 6–2 lists the contents provided in the <*PWM design files*> directory.

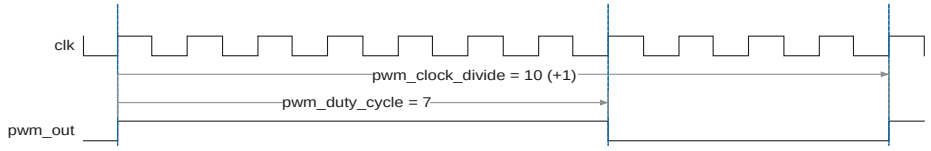
Table 6–2. PWM Design Files Directory

File Name	Description
/pwm_hw	Contains HDL files describing the component hardware.
pwm_task_logic.v	Contains the core of the PWM functionality.
pwm_register_file.v	Contains logic for reading and writing PWM registers.
pwm_avalon_interface.v	Instantiates task logic and register file, and provides an Avalon slave interface. This file contains the top-level module.
/pwm_sw	Contains C files describing the software interface to the component.
/inc	Contains header files defining low-level hardware interface.
avalon_slave_pwm_regs.h	Defines macros to access registers in the PWM component.
/HAL	Contains HAL driver files for the Nios II processor.
/inc	Contains HAL driver include files.
altera_avalon_pwm_routines.h	Declares function prototypes for accessing the PWM.
/src	Contains HAL driver source code files.
altera_avalon_pwm_routines.c	Defines functions for accessing the PWM.
/test_software	Contains an example program to test the component hardware & software.
hello_altera_avalon_pwm.c	`main()` initializes the PWM hardware, and uses the PWM to blink an LED.

Functional Specification

A PWM component outputs a square wave with modulated duty cycle. A basic pulse-width waveform is shown in Figure 6–2.

Figure 6–2. Basic Pulse-Width Modulation Waveform



The PWM component is specified and created as follows:

- The task logic operates synchronously to a single clock.
- The task logic uses 32-bit counters to provide a suitable range of PWM periods and duty cycles.
- A host processor is responsible for setting the PWM period value and duty-cycle value. This requirement implies the need for a read/write interface to control logic.
- Register elements are defined to hold the PWM period value and duty-cycle value.
- The host processor can halt the PWM output by using an enable control bit.

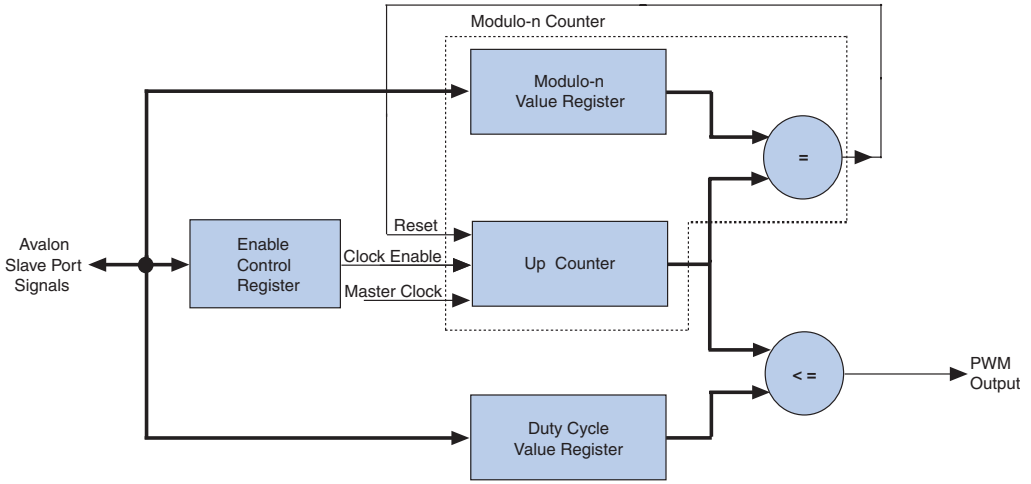
PWM Task Logic

The PWM task logic has the following characteristics:

- The PWM task logic consists of an input clock (`clk`), an output signal (`pwm_out`), an enable bit, a 32-bit modulo-n counter, and a 32-bit comparator circuit.
- `clk` drives the 32-bit modulo-n counter to establish the period of the `pwm_out` signal.
- The comparator compares the current value of the modulo-n counter to the duty-cycle value and determines the output of `pwm_out`.
- When the current count value is less than or equal to the duty-cycle value, `pwm_out` drives logic value 0; otherwise, it drives logic value 1.

The task-logic structure is shown in Figure 6–3.

Figure 6–3. PWM Task Logic Structure



Register File

The register file provides access to the enable bit, the modulo-n value and the duty cycle value, shown in Figure 6–3. The design maps each register to a unique offset in the Avalon slave port address space.

Each register has read and write access, which means that software can read back values previously written into the registers. This is an arbitrary design choice that provides software convenience at the expense of hardware resources. You could equally design the registers to be write-only, which would conserve on-chip logic resources, but make it impossible for software to read back the register values.

The register file and offset mapping is shown in Table 6–3. To support three registers, two bits of address encoding are necessary. This gives rise to the fourth register which is reserved.

Table 6–3. Register File & Address Mapping

Register Name	Offset	Access	Description
clock_divide	00	Read / Write	The number of clock cycles counted during one cycle of the PWM output.
duty_cycle	01	Read / Write	The number of clock cycles in which the PWM output will be low.
enable	10	Read / Write	Enables/disables the PWM output. Setting bit 0 to 1 enables the PWM.
Reserved	11	-	

To read or write the registers requires only one clock cycle, which affects the wait-states for the Avalon interface.

Avalon Interface

The Avalon interface for the PWM component requires a single slave port using a small set of Avalon signals to handle simple read and write transfers to the registers. The component's Avalon slave port has the following characteristics:

- It is synchronous to the Avalon slave port clock.
- It is readable and writeable.
- It has zero wait states for reading and writing, because the registers are able to respond to transfers within one clock cycle.
- It has no setup or hold restrictions for reading and writing.
- Read latency is not required, because all transfers can complete in one clock cycle. Read latency would not improve performance.
- It uses native address alignment, because the slave port is connected to registers rather than a memory device.

Table 6–4 lists the Avalon signals types required to implement these transfer properties. The table also lists the names of each signal as defined in the HDL design file.

Table 6–4. PWM Signal Names & Avalon Signal Types

Signal Name in HDL	Avalon Signal Type	Bit-Width	Direction	Notes
clk	clk	1	input	Clock that synchronizes data transfers and task logic
resetn	reset_n	1	input	Reset signal; active low.
avalon_chip_select	chipselect	1	input	Chip-select signal
address	address	2	input	2-bit address; only three encodings are used.
write	write	1	input	Write enable signal
write_data	writedata	32	input	32-bit write-data value
read	read	1	input	Read enable signal
read_data	readdata	32	output	32-bit read-data value

For details on the behavior of Avalon signals and Avalon transfers, see the *Avalon Interface Specification*.

Software API

The PWM example design provides both a header file that defines the register map, and driver software for the Nios II processor. See Table 6–2 on page 6–10 for a description of the individual files. The driver functions are listed in Table 6–5.

Table 6–5. PWM Driver Functions (1)

Function	Prototype Description
altera_avalon_pwm_init();	Initializes the PWM hardware
altera_avalon_pwm_enable();	Activates the PWM output
altera_avalon_pwm_disable();	Deactivates the PWM output
altera_avalon_pwm_change_duty_cycle();	Deactivates the PWM output

Note for Table 6–5:
(1) Each function takes a parameter that specifies the base address of a specific instance of the PWM component.

Package the Design Files into an SOPC Builder Component

In this section, you will use the SOPC Builder component editor to package the design files into an SOPC Builder component. You will perform the following operations:

1. Open the Quartus II project and start the component editor.
2. Configure the settings on each tab of the component editor.
3. Save the Component.

Open the Quartus II Project & Start the Component Editor

To open SOPC Builder from the Quartus II software, you must have a Quartus II project open. Perform the following steps:

1. Start the Quartus II software.
2. Open the project **standard.qpf** in the **<*Quartus II project*>** directory.
3. Choose **SOPC Builder** (Tools menu). The SOPC Builder GUI appears, displaying a ready-made example design containing a Nios II processor and several components in the table of active components.
4. Choose **New Component** (File menu). The component editor GUI appears, displaying the **Introduction** tab.

HDL Files Tab

In this section you will associate the HDL files with the component using the **HDL Files** tab. Perform the following steps:

1. Click the **HDL Files** tab.

Each tab in the component editor GUI provides on-screen information that describes how to use each tab. Click the triangle at the top-left of each tab to view these instructions.

2. Click **Add HDL File**.
3. Browse to the <*PWM design files*>**/pwm_hw** directory. There are three Verilog HDL (**.v**) files in this directory.
4. Select all three HDL files in this directory and click **Open**. Use the control key to select multiple files.

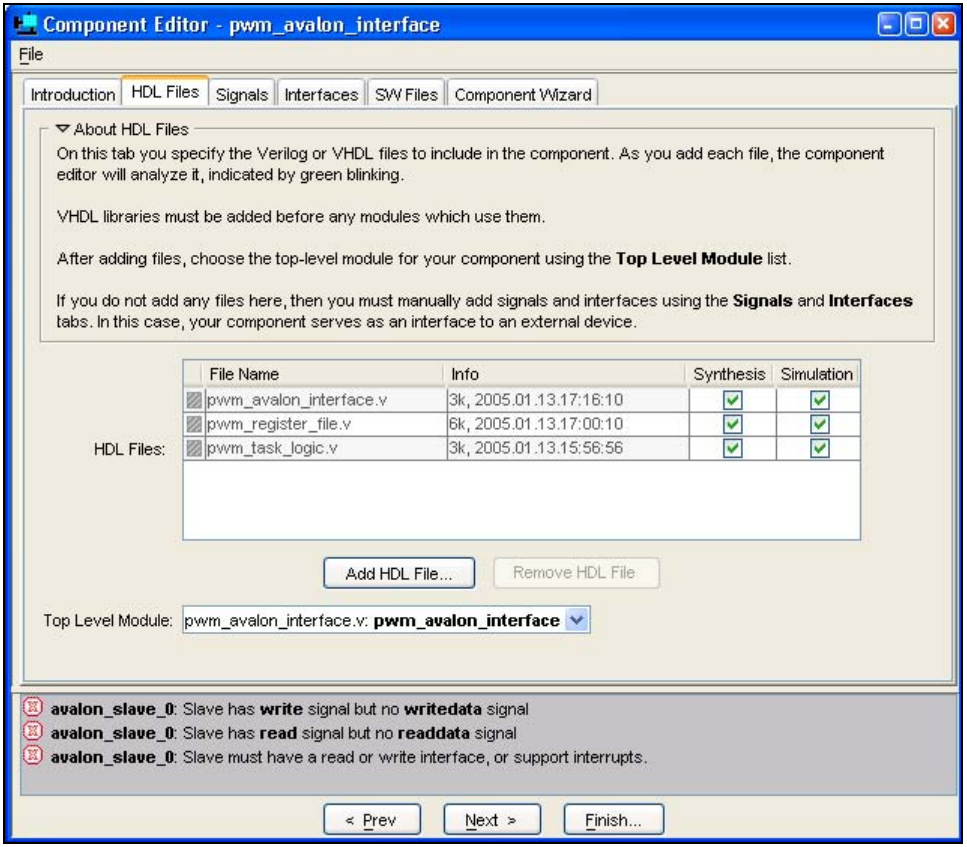
You will return to the **HDL Files** tab. The component editor immediately analyzes each file to read I/O signal and parameter information from the file.

5. Ensure that both the **Simulation** and **Synthesis** boxes are turned on for all files. This indicates that each file is appropriate for both simulation and synthesis design flows.

6. Select **pwm_avalon_interfave.v: pwm_avalon_interface** in the **Top Level Module** list to specify the top-level module.

At this point, the component editor GUI displays error messages. Ignore these messages for now, because you will fix them in later steps. Figure 6–4 shows the state of the **HDL Files** tab.

Figure 6–4. HDL Files Tab



Signals Tab

For every I/O signal present on the top-level HDL module, you must map the signal name to a valid Avalon signal type using the **Signals** tab. The component editor automatically fills in signal details that it finds in the top-level HDL source file. If a signal is named the same as a recognized Avalon signal type (such as `write` or `address`), then the component editor automatically assigns the signal's type. If the component editor cannot determine the signal type, it assigns it to type **export.**

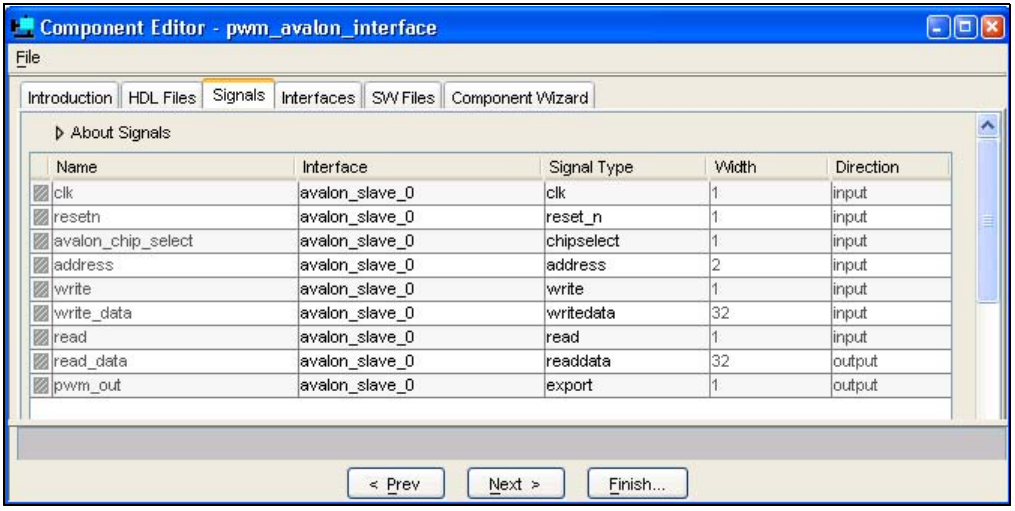
Perform the following steps to define the component I/O signals:

1. Click the **Signals** tab. All of the I/O signals in the top level HDL module **pwm_avalon_interface** appear automatically.

2. Assign the **Signal Type** settings for all signals, as show in Figure 6–5. To change a value, click the **Signal Type** cell to display a drop-down list, and select a new signal type from the list.

After you correctly assign each signal name to a signal type, the error messages should disappear.

Figure 6–5. Assigning Signal Names to Signal Types



You assign type **export** to the signal `pwm_out`, because it is not an Avalon signal. It is intended to be an output of the SOPC Builder system.

Interfaces Tab

The **Interfaces** tab lets you configure the properties of all Avalon interfaces on the component. In this case there is only one Avalon interface, as specified in the section "Avalon Interface" on page 6–13. Perform the following steps to configure the Avalon slave port:

1. Click the **Interfaces** tab. The component editor displays a default Avalon slave port that it created automatically, based on the top-level I/O signals in the component design.

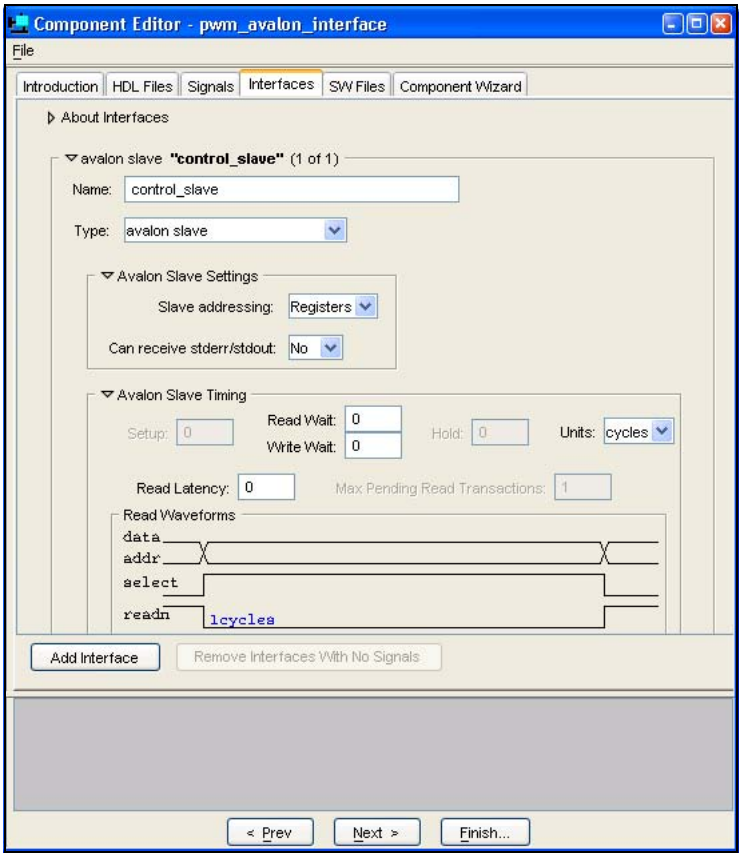
2. Type `control_slave` in the **Name** field to rename the slave port. This name appears in the SOPC Builder GUI when you instantiate the component in SOPC Builder.

3. Change the settings for the **control_slave** interface as listed in Table 6–6 below. Figure 6–6 on page 6–19 shows the **Interfaces** tab with the correct settings.

Table 6–6. Control Slave Interface Settings

Setting	Value	Description
Slave addressing	Registers	This setting is appropriate for slave ports used to access address-mapped registers
Read Wait	0	This setting means that the slave port responds to read requests in a single clock cycle (i.e, it does not need read waitstates.)
Write Wait	0	This setting means that the slave port captures write requests in a single clock cycle (i.e., it does not need write waitstates.)

Figure 6–6. Configuring the Interface Properties



Software Files (SW Files) Tab

The **SW Files** tab lets you associate software files with the component, and specify their usage. This component example design provides both a header file that defines the registers and driver software for the Nios II processor. For a description of each file, see Table 6–2 on page 6–10.

Perform the following steps to import the software files into the component:

1. Click the **SW Files** tab.
2. Click **Add SW File**. The Open dialog appears.

3. Browse to the directory <*PWM design files*>**/pwm_sw/inc**.
4. Select the file **altera_avalon_pwm_regs.h** and click **Open**.
5. Click the **Type** cell for **altera_avalon_pwm_regs.h** to change the file type. A drop-down list appears.
6. Select **type Registers (inc/)**.
7. Repeat steps 2 to 6 to add the file **<***PWM design files***>/pwm_sw/HAL/inc/altera_avalon_pwm_routines.h** and set its type to **HAL (HAL/inc/)**.
8. Repeat steps 2 to 6 to add the file **<***PWM design files***>/pwm_sw/HAL/src/altera_avalon_pwm_routines.c** and set its type to **HAL (HAL/src/)**.

Figure 6–7 shows the SW Files tab with the correct settings.

Figure 6–7. Software Files (SW Files) Tab

File Name	Info	Type
altera_avalon_pwm_regs.h	3k, 2004.12.17.16:21:10	Registers (inc/)
altera_avalon_pwm_routines.h	2k, 2004.12.17.16:04:28	HAL (HAL/inc/)
altera_avalon_pwm_routines.c	3k, 2004.12.17.16:27:04	HAL (HAL/src/)

Component Wizard Tab

This tab lets you control how SOPC Builder presents the component to a user. Perform the following steps to configure the user presentation of the component:

1. Click the **Component Wizard** tab.
2. For this example, do not change the default settings for **Component Name**, **Component Version**, and **Component Group**.

   These settings affect how SOPC Builder identifies the component and displays it in the list of available components. The component editor creates a default name for the component, based on the name of the top-level design module.
3. Under **Parameters**, in the **Tooltip** cell for the parameter **clock_divide_reg_init**, type the following:

```
Initial PWM Period After Reset
```

4. In the **Tooltip** cell for **clock_cycle_reg_init**, type:

```
Initial Duty Cycle After Reset
```

5. Click **Preview the Wizard** to preview how the component wizard will appear when instantiated from within SOPC Builder.

6. Close the preview window when you are done.

Save the Component

Perform the following steps to save the component and exit the component editor:

1. Click **Finish**. A dialog appears describing the files that will be created for the component.

2. Click **Yes** to save the files. The component editor saves the files to a subdirectory under **<*Quartus II project*>**. The component editor closes, and you return to the main SOPC Builder GUI.

3. Locate the new component **pwm_avalon_interface** in the list of available components under the **User Logic** group.

You are ready to instantiate the component into an SOPC Builder system.

Instantiate the Component in Hardware

At this point, the new component is ready to instantiate in an SOPC Builder system. The usage of a component is design dependent, based on the needs of the system. The remaining steps for this design example show one possible way to instantiate and test the component. However, there is an unlimited number of ways this component can be used in a system.

In this section you will add the new PWM component to a system, recompile the hardware design, and configure the FPGA. This section includes the following steps:

1. Add a PWM component to the SOPC Builder system and regenerate the system.

2. Modify the Quartus II design to connect the PWM output to an FPGA pin.

3. Compile the Quartus II design and configure the FPGA with the new hardware image.

Add a PWM Component to the SOPC Builder System

Perform the following steps to setup SOPC Builder's component search path:

1. In the SOPC Builder GUI, choose **SOPC Builder Setup** (File menu).
2. Under **Component/Kit Library Search Path**, enter the path to the ***<Quartus II project>*** directory. If there are pre-existing paths, use "+" to separate the path names.
3. Click **OK**.

The steps above make the component's software files visible to the Nios II IDE in later steps. These steps are necessary for the Quartus II software v4.2 and the Nios II IDE v1.1. Future releases will eliminate the need for these steps.

Perform the following steps to add a PWM component to the SOPC Builder system:

1. On the SOPC Builder **System Contents** tab, select the new component **pwm_avalon_interface** under the **User Logic** group in the list of available components, and click **Add**. The configuration wizard for the PWM component appears.

   If you want to, you can modify the parameters in the configuration GUI. The parameters affect the reset state of the PWM control registers, but have no affect on the outcome of the steps in this chapter.
2. Click **Finish**. You return to the SOPC Builder **System Contents** tab, and the component **pwm_avalon_interface_0** appears in the table of active components.
3. Right-click **pwm_avalon_interface_0** and choose **Rename**.
4. Type `z_pwm_0` for the component name and press **Enter**. (This name is unusual, but it minimizes effort later when you update the Quartus II design in section "Modify the Quartus II Design to Use the PWM Output" on page 6–23.

You must name the component exactly as directed, or else later steps in this chapter will fail.

5. Click **Generate** to start generating the system.

6. After system generation completes successfully, exit SOPC Builder and return to the Quartus II software.

Modify the Quartus II Design to Use the PWM Output

At this point, you have created an SOPC Builder system that uses the PWM component. Now you must update the Quartus II project to use the PWM output.

The file **standard.bdf** is the top-level Block Design File (BDF) for the Quartus II project. The BDF contains a symbol for the SOPC Builder system module, named **std_<*FPGA*>**, where **<*FPGA*>** refers to the FPGA on the target development board.

In the previous steps you added a PWM component which produces an additional output from the system module. Now you need to update the symbol for the system module, and connect the PWM output to an FPGA pin.

To complete this section, you must be familiar with the Quartus II Block Editor.

1. In the Quartus II software, open the file **standard.bdf**.
2. Right-click the symbol **std_<*FPGA*>** in the BDF and choose **Update Symbol or Block**. The Update Symbol or Block dialog appears.
3. Select **Selected Symbol(s) or Block(s)**.
4. Click **OK** to close the dialog. The symbol **std_<*FPGA*>** in the BDF is updated, and it now has an additional output port named **pwm_out_from_the_z_pwm_0**.

SOPC Builder creates unique names for all I/O ports on the system module, by combining the signal name in the component design file with the instance name of the component in the system module.

5. Delete the symbol for pins `LEDG[7..0]` which are connected to port `out_port_from_the_led_pio[7..0]` on the system module.

   These pins connect to LEDs on the development board. This example design uses one of the LEDs to display the output of the PWM.

6. Create a new output pin named `LEDG[0]`.

7. Connect the new pin `LEDG[0]` to `pwm_out_from_the_z_pwm_0` on **std_<*FPGA*>**.

The hardware design is now ready to compile.

Compile the Hardware Design and Download to the Target Board

Perform the following steps to compile the hardware design and download it to the target board.

1. Chose **Save** (File menu) to save changes to the BDF.

2. Choose **Start Compilation** (Processing menu) to start compiling the hardware design. The compilation begins.

If you performed all prior steps correctly, the Quartus II compilation will finish successfully after several minutes, and generate a new FPGA configuration file for the project.

You can only perform the remaining steps in this chapter if you have a development board.

Perform the following steps to download the hardware design to the board:

1. Connect your host computer to the development board using an Altera download cable, such as the USB Blaster, and apply power to the board.

2. Choose **Programmer** (Tools menu) to open the Quartus II Programmer.

3. Use the Programmer window to download the following FPGA configuration file to the board: **<*Quartus II project*>/standard.sof**.

At this point, you have completed all the steps to create a hardware design and download it to hardware.

Exercise the Hardware Using Nios II Software

The PWM example design is based on the Nios II processor. You must execute software on the Nios II processor to exercise the PWM hardware. The example design files provide a C test program that pulses an LED by gradually modulating the PWM duty cycle. This test program accesses the hardware both by using the register map declarations directly, and by calling the driver functions.

In this section you will perform the following steps:

1. Start the Nios II IDE and create a new Nios II IDE project.
2. Build and run the C test program.
3. View the results.

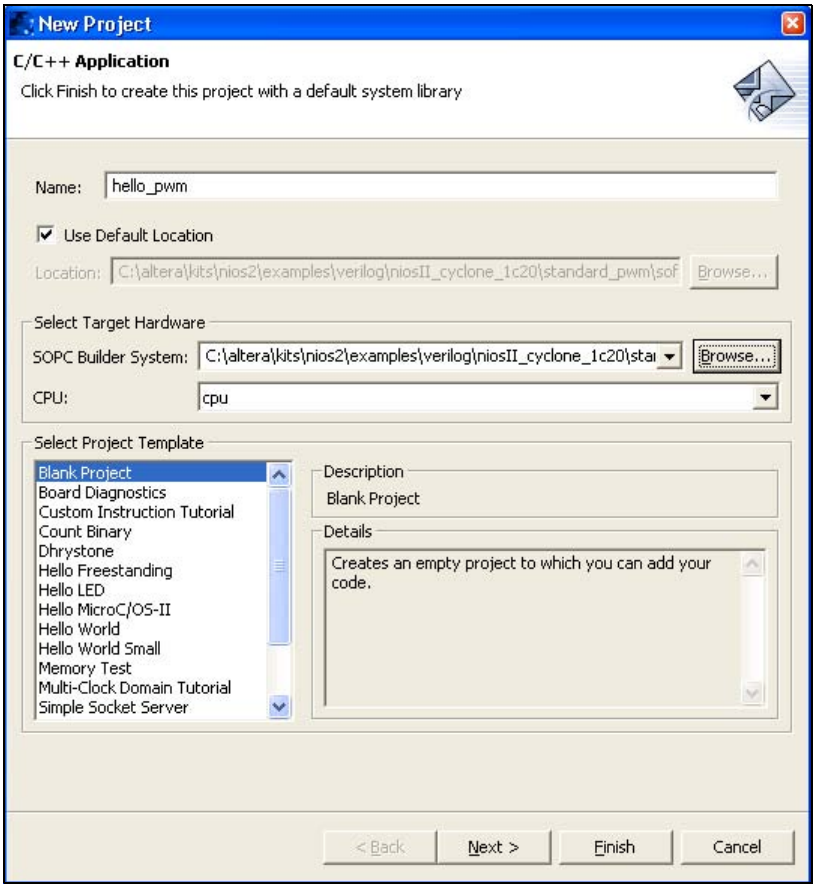
To complete this section, you must have performed all prior steps, and successfully configured the target board with the hardware design.

Start the Nios II IDE & Create a New IDE Project

Perform the following steps to start the Nios II IDE and create a new IDE project:

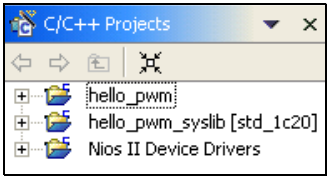
1. Start the Nios II IDE.
2. Choose **New > C/C++ Application** (File menu) to start a new project. The first page of the **New Project** wizard appears.
3. Under **Select Project Template**, select **Blank Project**.
4. In the **Name** field type `hello_pwm`.
5. Ensure that **Use Default Location** is turned on.
6. Click **Browse** under **Select Target Hardware**. The **Select Target Hardware** dialog box appears.
7. Browse to the **<*Quartus II project*>** directory.
8. Select the file **std_<*FPGA*>.ptf**.
9. Click **Open** to return to the New Project wizard. The **SOPC Builder System** and the **CPU** fields are now specified, as shown in Figure 6–8 on page 6–26.
10. Click **Finish**.

Figure 6–8. New Project Wizard



After the IDE successfully creates the new project, the C/C++ Projects view will contain two new projects, **hello_pwm** and **hello_pwm_syslib**, in addition to **Nios II Device Drivers**, as shown in Figure 6–9.

Figure 6–9. New Projects in the C/C++ Projects View



Compile the Software Project and Run on the Target Board

In this section you will compile the C test program provided with the PWM design files, and then download it to the target board.

First, perform the following steps to associate the C source file with the new C/C++ project.

1. In your computer's file system, copy the file <*PWM design files*>**/pwm_sw/test_software/hello_altera_avalon_pwm.c** to the directory **<*Quartus II project*>/software/hello_pwm/**.

2. In the Nios II IDE C/C++ Projects view, right-click **hello_pwm** and choose **Refresh**. This forces the IDE to recognize the new file in the project directory.

The project is now ready to compile and run. Perform the following steps:

1. Right-click **hello_pwm** and choose **Build Project** to compile the program. The first time you build the project, it can take a few minutes for the compilation to finish.

2. After compilation completes, select **hello_pwm** in the C/C++ Projects view.

3. Choose **Run** (Run menu). The Run dialog appears.

4. Under **Configurations** select **Nios II Hardware**, and click **New**. A new run/debug configuration named **hello_pwm Nios II HW configuration** appears.

5. If the **Run** button (in the bottom right of the Run dialog) is deactivated, perform the following steps:

    a. Click the **Target Connection** tab.

    b. Click **Refresh** next to the **JTAG cable** list.

    c. From the **JTAG cable** list, select the download cable you want to use.

    d. Click **Refresh** next to the **JTAG device** list.

6. Click **Run**.

7. View the results:

a. The **Console** view in the IDE displays messages similar to the following:

```
Hello from the PWM test program.
The starting values in the PWM registers are:
Period = 0
Duty cycle = 0
Notice the pulsing LED on the development board.
```

b. LED0 on the development board repeatedly pulses on and off.

Congratulations! You have finished all steps for the PWM design example.

Sharing Components

When you create a component using the component editor, SOPC Builder automatically saves the component in the current Quartus II project directory. To promote design reuse, you can use the component in different projects, and you can share your component with other designers.

Perform the following steps to share a component:

1. In your computer's file system, move the component directory to a central location, outside any particular Quartus II project's directory. For example, you could create a directory **c:\my_component_library** to store your custom components.

   The directory path name cannot contain spaces. If the path contains spaces, SOPC Builder might not be able to access the files.

2. In SOPC Builder, choose **SOPC Builder Setup** (File menu). The **SOPC Builder Setup** dialog appears, which lets you specify where SOPC Builder searches for component files.

3. Under **Component/Kit Library Search Path**, add the path to the enclosing directory of the component directory. For example, for a component directory **c:\my_component_library\pwm_avalon_interface**, add the path **c:\my_component_library**. If there are pre-existing paths, use "+" to separate the path names.

4. Click **OK**.

7. Building Systems with Multiple Clock Domains

QII54008-1.0

Introduction

This chapter guides you through the process of using SOPC Builder to create a system with multiple clock domains. You will start with a ready-made design that uses a single clock domain, and modify the design to use two clocks.

Example Design Overview

The design in this chapter mimics the common scenario of a system with separate control and data paths. Typically, the control path is slow, because the controller itself is relatively slow, and it is used only in short bursts to set up data transfers. On the other hand, the data path is fast so that, after the controller initiates a transfer, data moves as quickly as possible from source to destination.

Figure 7–1 on page 7–2 shows a simplified block diagram of the system structure. In this design, a Nios® II processor acts as the controller operating at 50 MHz. A DMA controller operating at 100 MHz manages the data path, and reads and writes data buffers that also operate at 100 MHz. The figure focuses on the multi-clock nature of the system, and shows the connections between master and slave ports.

Figure 7–1. Simplified Block Diagram of the Example Design

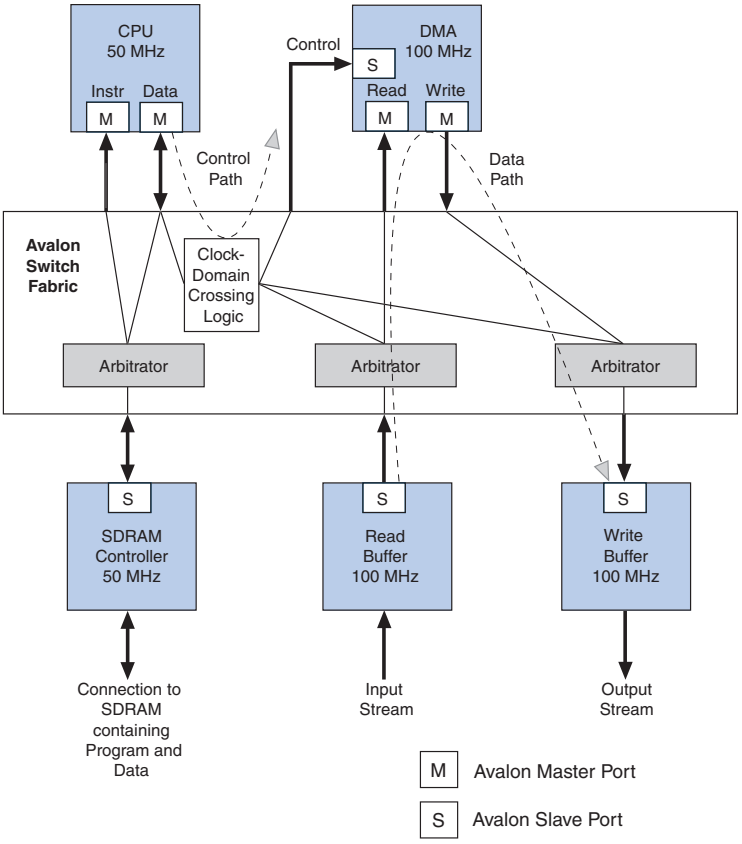


Figure 7–1 does not show example design features that are not directly related to the issue of multiple clocks, such as a JTAG UART communication peripheral and timer peripheral used by the Nios II processor.

Hardware and Software Requirements

To use the design example(s) in this chapter, you must have the following:

- Design files for the example design – A hyperlink to the design files is located with this chapter on the SOPC Builder literature page. Visit **www.altera.com/sopcbuilder**.

- Quartus® II Software version 4.2 or higher – Both Quartus II Web Edition and the fully licensed version will work with the example design.
- Nios II development kit version 1.1 or higher – Both the evaluation edition and the fully licensed version will work with the example design.
- Nios development board and an Altera® USB-Blaster™ download cable (Optional) – You can use any of the following Nios development boards:

  - Stratix® II Edition
  - Stratix Edition
  - Stratix Professional Edition
  - Cyclone™ Edition

  If you do not have a development board, you can follow the hardware development steps, but you will not be able to download the complete system to a working board.

You can download the Quartus II Web Edition software and the Nios II Development Kit, Evaluation Edition for free from the Altera Download Center at **www.altera.com**.

Before you begin, you must install the Quartus II software and Nios II development tools.

Creating the Multi-Clock Hardware System

In this section, you will start with an example hardware design provided with the Nios II development kit and create a multi-clock system. You will perform the following steps:

1. Copy the hardware design files to a new directory
2. Modify the design in SOPC Builder to create a multi-clock hardware system
3. Update the Quartus II design to use the new clock domain
4. Compile the hardware design in the Quartus II software, and download the hardware design to a target board

Copy the Hardware Design Files to a New Directory

The hardware design used in this chapter is based on the **standard** hardware example design included with the Nios II development kit. Copy the design files by performing the following steps:

1. In your host computer file system, locate the following directory:

   *<Nios II kit path>***examples**\<verilog or vhdl>**<board version>***standard**

   Each development board has a VHDL and Verilog version of the design. You can use either one. Table 7–1 shows the names of the directories for the available Nios development boards.

Table 7–1. Design File Directories

Nios Development Board	Tutorial Directory
Stratix II Edition	niosII_stratixII_2s60_es
Stratix Edition	niosII_stratix_1s10 or niosII_stratix_1s10_es
Stratix Professional Edition	niosII_stratix_1s40
Cyclone Edition	niosII_cyclone_1c20

   For demonstration purposes, the figures in this chapter show the case of the Verilog design on the Nios Development Board, Cyclone Edition.

2. Copy the **standard** directory to a new location. This document will refer to the newly-created directory as *<hardware files directory>*.

Modify the Design in SOPC Builder

This section walks you through the process of implementing a multi-clock system in SOPC Builder. In this section you will do the following:

1. Open the system in SOPC Builder.
2. Add a DMA controller and two 4 Kbyte on-chip memory components.
3. Connect DMA master ports to memory slave ports.
4. Make clock domain assignments.
5. Regenerate the system.

Open the System in SOPC Builder

To open the system in SOPC Builder, perform the following steps:

1. Start the Quartus II software.
2. Choose **Open Project** (File menu).
3. Browse to *<hardware files directory>*.
4. Select **standard.qpf** and click **Open**.
5. Choose **SOPC Builder** (Tools menu) to start SOPC Builder.

The SOPC Builder window appears, displaying the contents of the system, which you will use as a starting point for your design.

Add DMA Controller and Memory Components

Perform the following steps to add the DMA controller and memory components to the system:

1. In the SOPC Builder list of available components, select **DMA** in the **Other** group, and click **Add**. The DMA configuration wizard displays.
2. In the DMA configuration wizard, click **Finish** to accept the default settings. You return to the SOPC Builder **System Contents** tab which displays the new DMA component, named **dma_0**.

   Errors are displayed in the SOPC Builder messages window. You can ignore these messages for now, because you will fix the errors in later steps.
3. In the list of available components, select **On-Chip Memory (RAM or ROM)** in the **Memory** group, and click **Add**. The On-Chip Memory configuration wizard appears.
4. In the On-Chip Memory configuration wizard, click **Finish** to accept the default settings. You return to the SOPC Builder **System Contents** tab, which displays the new memory component.
5. Right-click the new memory component and choose **Rename**.
6. Type `read_buffer`↵ to rename the component.
7. Repeat steps 3 and 4 to add another On-Chip Memory component to the system.
8. Right-click the new memory component and choose **Rename**.

9. Type `write_buffer`↵ to rename the component.

You must name the DMA and memory components exactly as specified above (**dma_0, read_buffer**, and **write_buffer**). If you name the components differently, later steps will fail.

Connect DMA Master Ports to Memory Slave Ports

Now that you have added the DMA controller and the memory components to the system, you must connect their master and slave ports appropriately. Perform the following steps:

1. Hover the mouse pointer over the connections panel in the SOPC Builder **System Contents** tab to display the potential master port connections for the DMA. See Figure 7–2.

2. Connect the DMA read master port (**dma_0/read_master**) to the **read_buffer** memory.

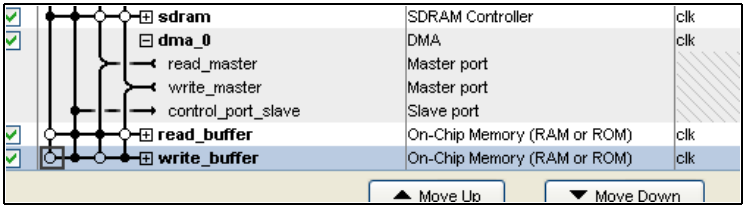
3. Connect the DMA write master port (**dma_0/write_master**) to the **write_buffer** memory.

4. Disconnect the Nios II processor instruction master (**cpu/instruction_master**) from both the on-chip memories' slave ports. For this example design, the processor does not use these memories to fetch instructions.

5. Verify that the Nios II processor data master (**cpu/data_master**) is connected to the DMA slave port (**dma/control_port_slave**).

Figure 7–2 shows the state of the connections panel after all components have been correctly connected.

Figure 7–2. Correct Connections Between Master & Slave Ports



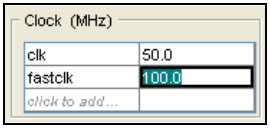
After the DMA controller is connected properly to the on-chip memories, the error messages disappear from the SOPC Builder messages window.

Make Clock Domain Assignments

In this section you will specify a 100 MHz clock input, and assign the DMA and memory components to the new clock domain. Then you will generate the system. Perform the following steps:

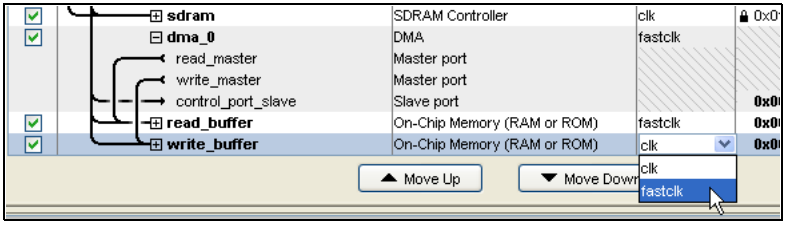
1. On the **System Contents** tab under **Clock MHz**, click the cell labeled **click to add** to enter a new clock entry.
2. Type **fastclk** for the name of the new clock, as shown in Figure 7–3.
3. Type **100** for the speed of `fastclk`, as shown in Figure 7–3.

Figure 7–3. Clock Settings



4. In the **Clock** list for the DMA component, select **fastclk** to assign the 100 MHz clock to the DMA component, as shown in Figure 7–4.
5. Repeat step 4 for the **read_buffer** and **write_buffer** components, as shown in Figure 7–4.

Figure 7–4. Assigning a Different Clock to the DMA & Memory Components



6. Click **Generate** to start generating the system.
7. After system generation completes successfully, exit SOPC Builder and return to the Quartus II software.

Update the Quartus II Design

At this point, you have created an SOPC Builder system that uses multiple clock domains. In this section you will do the following:

1. Update the system module symbol.
2. Update PLL settings to generate a 100 MHz clock.
3. Connect the 100 MHz clock to the system module
4. Compile the Design and download it to the board

To complete this section, you must be familiar with the Quartus II Block Editor.

Update the System Module Symbol

The file **standard.bdf** is the top-level Block Diagram File (BDF) for the Quartus II project. The BDF contains a symbol for the SOPC Builder system module, named **std_<*FPGA*>**, where **<*FPGA*>** refers to the FPGA on the target development board.

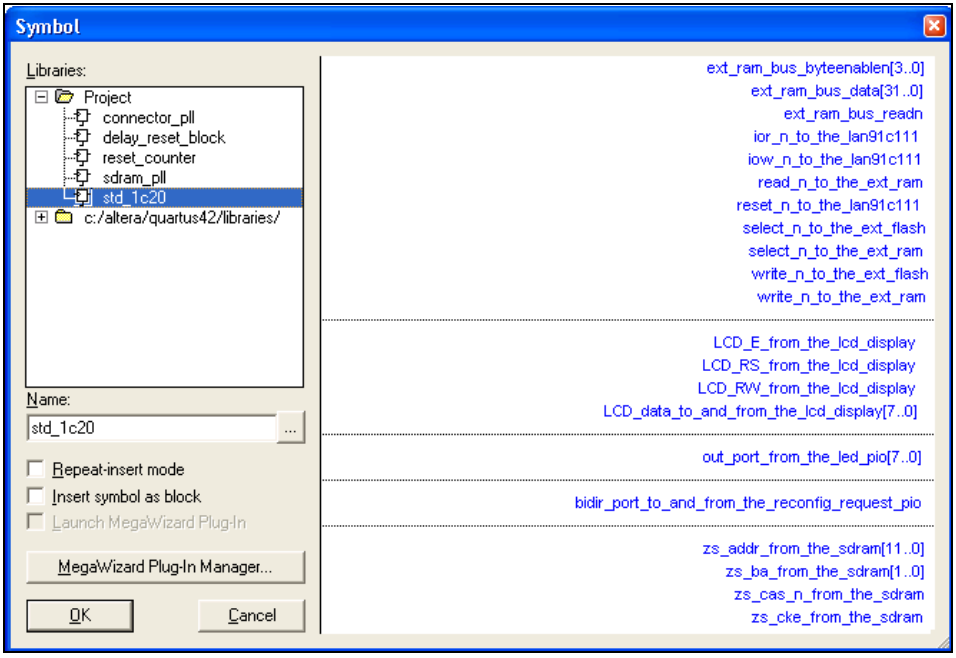
The SOPC Builder system module requires an additional clock input for the 100 MHz clock domain, and therefore you need to update the symbol for the system module. To remove the old symbol and insert an updated symbol, perform the steps below.

In the following steps you will disconnect and then reconnect all connections to the system module, including connections to FPGA pins. You must make these connections exactly, or else the design will not work in hardware, potentially damaging the development board. If possible, print the BDF as a reference to help you reconnect pins to the system module later.

1. In the BDF, select the symbol **std_<*FPGA*>** and delete it.
2. Click-and-drag to select all of the pins that previously connected to the right-hand-side of the symbol.
3. Press the right-arrow key ten times to move the pins to the right, creating space for the new symbol.
4. Double click in the space where the symbol used to be to insert a new symbol. The **Symbol** window appears.
5. Expand the **Project** folder under **Libraries**.

6. Select the **std_<*FPGA*>** symbol, and click OK. See Figure 7–5.

Figure 7–5. Selecting and Inserting the New Symbol



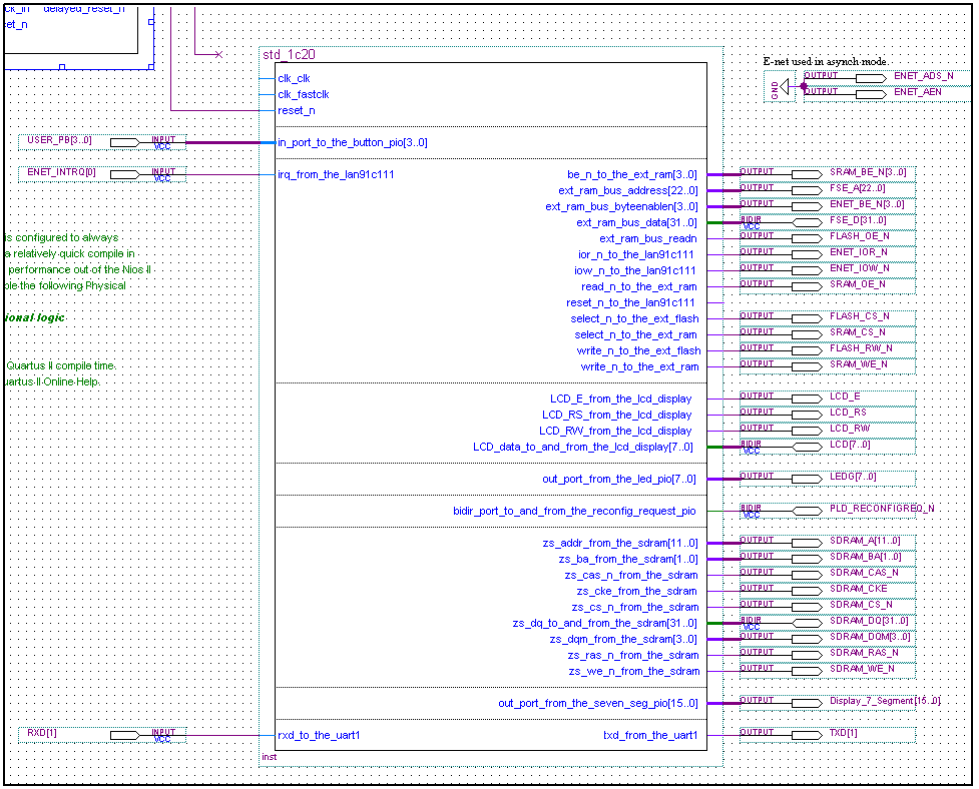
7. Move the mouse to position the symbol between the pin symbols, then click the left mouse button to place the symbol.

8. Look at the symbol, and notice the new clock input, **clk_fastclk**, and the old clock input, which is now named **clk_clk**.

9. Reconnect all previous connections to the **std_<*FPGA*>** symbol, except for the clock inputs **clk_clk** and **clk_fastclk**. You will connect the clock inputs in later steps.

Figure 7–6 shows the new symbol reconnected to all signals except for the clocks.

Figure 7–6. Updated Symbol without Clock Connections



Update PLL Settings to Generate a 100 MHz Clock

Perform the following steps to modify the PLL instance in the BDF to generate a 100 MHz clock:

1. Locate the symbol **sdram_pll** in the BDF.

2. Right-click the symbol and select **MegaWizard Plug-in Manager** to configure the PLL settings. The MegaWizard for the ALTCLKLOCK function displays.

At this point, the Quartus II software might display a benign warning: "Delay shifts (time delay elements) are no longer supported in the Stratix PLLs." If you see this message, click **OK**.

3. Click **Next** until you reach page 6 of 15 of the wizard flow, shown in Figure 7–7.

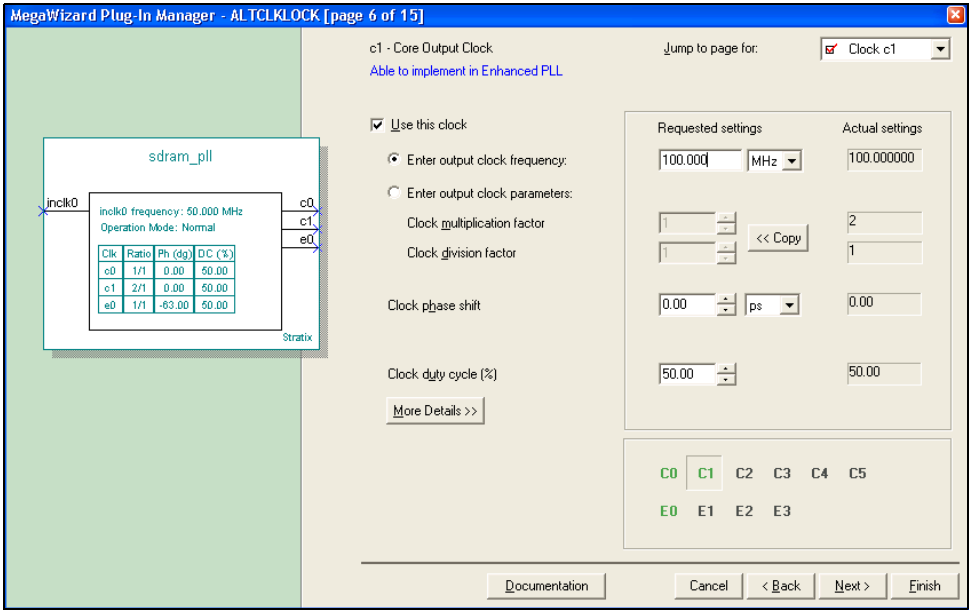
4. Click **Use this clock**.

5. Select **Enter output clock frequency**.

6. Under **Requested settings,** type `100` and select **MHz**.

Figure 7–7 shows the **ALTCLKLOCK** MegaWizard with the correct settings.

Figure 7–7. ALTCLKLOCK Settings to Generate a 100MHz Clock



7. Click **Finish** to jump to the final stage (page 15 of 15) of the wizard flow.

8. Click **Finish** again to return to the Quartus II software.

9. Right-click the **sdram_pll** symbol and choose **Update Symbol or Block**. The Update Symbol or Block dialog appears.

10. Click **OK** to update the selected instance of the symbol.

The PLL is now configured correctly to generate a 100 MHz output clock.

For further information on using the PLLs in Altera devices, see the handbook for the target device family.

Connect the 100 MHz clock to the system module

You now must connect the clock signals to the SOPC Builder system module. The easiest way to accomplish this is to symbolically link the PLL outputs to the system module inputs using conduit aliases. This section will guide you through the process of making one connection, and leave the remaining connections to you as an exercise.

To connect the 100 MHz clock signal from the PLL to the system module, perform the following steps:

1. Move the mouse pointer over node **c1** on **sdram_pll** until the pointer changes to a cross-hairs.

2. Click and drag right to add a conduit (i.e. a connection line) to node **c1**.

3. Click on the conduit line to select it.

4. Type `dmaclk` and press Enter.

5. Move the mouse pointer over node **clk_fastclk** on **std_<FPGA>** until the pointer changes to a cross-hairs.

6. Click and drag left to add a conduit to node **clk_fastclk**.

7. Click on the conduit line to select it.

8. Type `dmaclk` and press **Enter**.

The two nodes are now linked symbolically (by the name `dmaclk`) via a conduit alias.

Follow the instructions below to complete the remaining clock connections. Depending on which Nios development board you are targeting, the remaining PLL connections to the PLL(s) are slightly different. This section gives instructions to accommodate all Nios development boards.

Be sure to use the instructions that apply to your board, or else the design will fail in hardware.

For the Nios Development Board, Stratix II Edition, Stratix Professional Edition, and Stratix Edition, connect the PLL according to Table 7–2.

Table 7–2. PLL Connections

Node on sdram_pll	Connects to
c0	`clk_clk` on **std_<*FPGA*>**
c1	`clk_fastclk` on **std_<*FPGA*>**
e0	`PLD_CLKOUT` pin

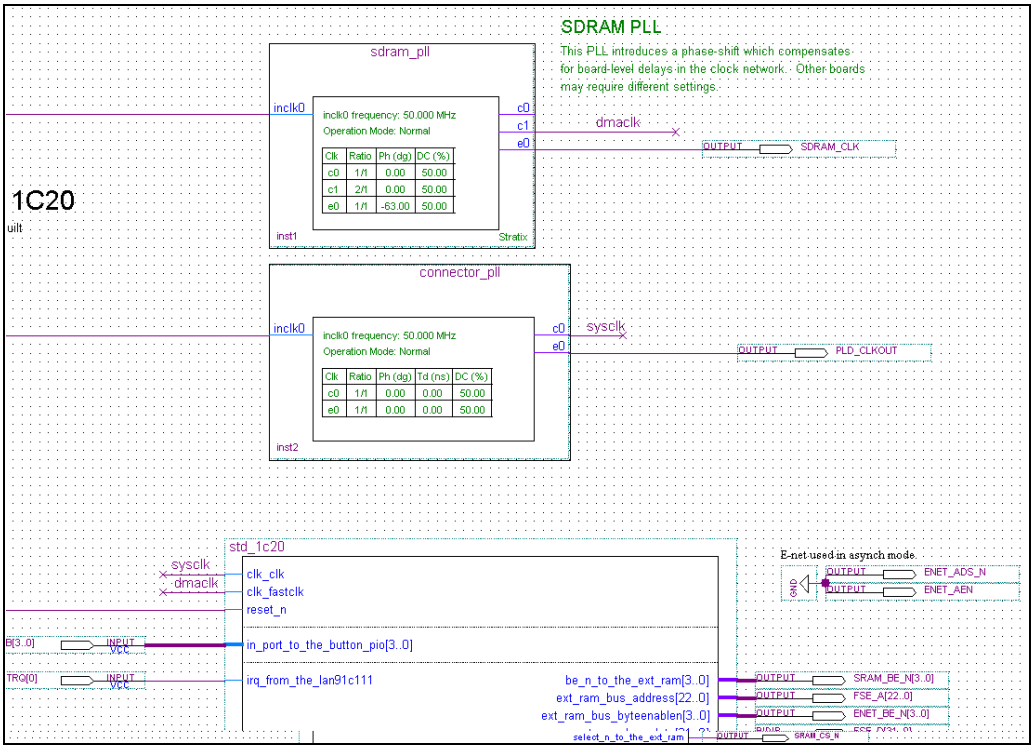
The BDF for the Nios Development Board, Cyclone Edition contains a second PLL symbol named **connector_pll** in addition to **sdram_pll**. Connect both PLLs according to Table 7–3.

Table 7–3. PLL Connections for the Nios Development Board, Cyclone Edition

Node	Connects to
c0 on **sdram_pll**	Nothing
c1 on **sdram_pll**	`clk_fastclk` on **std_<*FPGA*>**
e0 on **sdram_pll**	`SDRAM_CLK` pin
c0 on **connector_pll**	`clk_clk` on **std_<*FPGA*>**
e0 on **connector_pll**	`PLD_CLKOUT` pin

Figure 7–8 shows an example of the BDF for the Nios Development Board, Cyclone Edition with all connections completed using conduit aliases.

Figure 7–8. Symbolic Conduit Aliases Connecting PLL(s) to System Module



Compile the Design and Download to the Board

To compile the design and download it to the target board, perform the following steps:

1. Chose **Save** (File menu) to save changes to the BDF.

2. Choose **Start Compilation** (Processing menu) to start compiling the hardware design. The compilation begins.

If you performed all prior steps correctly, the Quartus II compilation will finish successfully after several minutes, and generate a new FPGA configuration file for the project.

You can only perform the remaining steps in this chapter if you have a development board.

To download the hardware design to the board, perform the following steps:

1. Connect your host computer to the development board via an Altera download cable, such as the USB Blaster.
2. Choose **Programmer** (Tools menu) to open the Quartus II Programmer.
3. Use the Programmer window to download the following FPGA configuration file to the board: ***<Hardware files directory>*\standard.sof**.

You have completed all the steps to create a multi-clock hardware design and download it to hardware. This design is based on the Nios II processor, and therefore you will have to run a software program on the processor to exercise the hardware.

Running Software to Exercise the Multi-Clock Hardware

In this section, you will run a program on the Nios II processor to exercise the multi-clock domain hardware. This program sets up and initiates a DMA transfer using the Nios II processor, and measures the time for the DMA transfer to complete.

There is nothing special about this program that makes it specific to multi-clock domain systems. Because the Avalon switch fabric abstracts the details of clock domain crossing, the Nios II processor benefits from the fast performance of the DMA controller without needing to be aware of the system clock domain properties.

You will perform the following steps:

1. Install the example software design files.
2. Create a new Nios II IDE project using the software files.
3. Build and run the program.
4. Analyze the results.

To complete this section, you must have performed all prior steps, and successfully configured the target board with your multi-clock hardware design.

Install the Example Software Design Files

In this section, you will install the example software design files on your computer. You can download the example design files from the Altera web site. Before you proceed installing the files, download the file **multi_clock.zip** associated with the URL for this chapter.

The file **multi_clock.zip** contains the following C-language source files:

- **dma_xfer.c** — Contains `main()`.
- **init.c**, **init.h** — Contain initialization routines to set up memory buffers, and initialize the timer.
- **settings_check.h** — Provides basic error checking to verify that the hardware contains the necessary DMA and memory components.

The file **multi_clock.zip** contains example software files packaged as a Nios II IDE software template. Perform the following steps to install the files:

1. Extract the contents of **multi_clock.zip** into a new directory called **multi_clock**.

2. Move the **multi_clock** directory under the following directory: **<***Nios II kit path***>\examples\software**

The files are packaged as a Nios II IDE template only to minimize the number of steps required for you to run the software. Using a template is not a necessary part of writing software for multi-clock domain systems. Using the template automatically provides the following functionality in the Nios II IDE, which you otherwise would have to perform manually:

1. Imports source files into a new project directory.

2. Sets up the system library settings.

3. Sets up the project settings.

Create a New Nios II IDE Project

To create a new Nios II IDE project using the provided example files, perform the following steps:

1. Start the Nios II IDE.

2. Choose **New > C/C++ Application** (File menu) to start a new project. The first page of the **New Project** wizard appears.

3. Under **Select Project Template**, select **Multi-Clock Domain Tutorial**.

4. Ensure that **Use Default Location** is turned on.

5. Click **Browse** under **Select Target Hardware**. The **Select Target Hardware** dialog box appears.

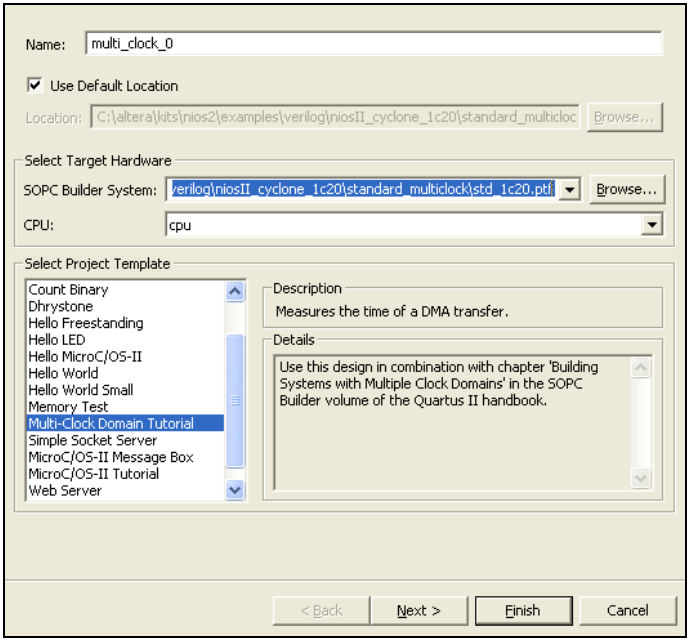
6. Browse to the *<hardware files directory>* directory where you created the hardware design earlier.

7. Select the file **std_<*FPGA*> .ptf** .

8. Click **Open** to return to the **New Project** wizard. The **SOPC Builder System** field is now specified and the **CPU** field now contains the name of the CPU in the system, as shown in Figure 7–9.

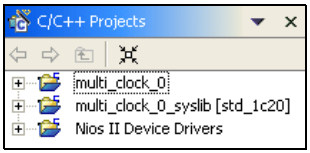
9. Click **Finish**.

Figure 7–9. New Project Wizard Filled in with Correct Settings



After the IDE successfully creates the new project, the **C/C++ Projects** view contains two new projects, **multi_clock_0** and **multi_clock_0_syslib**, in addition to **Nios II Device Drivers**, as shown in Figure 7–10.

Figure 7–10. New Projects Displayed in the C/C++ Projects View



Build and Run the Program

In this section, you will build the software in the Nios II IDE and run it on a development board. In short, the program does the following:

- Initializes the read and write memory buffers, filling the read memory buffer with random values.
- Performs timer initialization to accurately measure how long a DMA transaction will take.
- Sets up a DMA transaction to copy the contents of the read buffer to the write buffer.
- Starts the timer, initiates the DMA transaction, and waits for the DMA to generate an interrupt.
- Stops the timer when the DMA transaction finishes.
- Verifies that the write buffer contents are correct.
- Reports the duration of the DMA transaction.

To build and run the program, perform the following steps:

1. In the Nios II IDE C/C++ Projects view, select the **multi_clock_0** project.

2. Choose **Run As** > **Nios II Hardware** (Run menu) to build the program, download it to the board, and run it. The IDE automatically builds the program before attempting to run it. The build process can take several minutes. After the build completes, the IDE will download the program to the target board and run it.

3. View the results in the **Console** view.

The **Console** view will display results similar to the following:

```
nios2-terminal: starting in terminal mode (Control-C
exits)

Hello from Nios II!
Starting DMA transfer...

Starting a DMA transfer of 4096 bytes of data.
DMA transfer completed.
It took 34.3600006104 useconds to complete the
transfer.
Comparing send and receive buffer data...
Data Matches.

Program completed successfully.
```

See the Nios II IDE online help for more information on building and downloading projects.

In this example, the DMA achieved approximately 120 MBytes per second (4096 bytes / 34.36 usec). The source (read) and destination (write) buffer memories have zero wait states, and therefore can perform a maximum of one transfer per clock cycle. For successive 32-bit transfers at 100 MHz, the theoretical maximum DMA transfer performance is 400 MBytes per second. The 34.36 usec time includes the following processor overhead, which accounts for the difference between 400 and 120:

- Time spent in the DMA driver setting up the DMA transfer.
- Time spent in the interrupt handler after the DMA flags that it has completed the transfer.
- Time spent entering the DMA callback function to capture the finish time.

Conclusion

Congratulations! You have completed all tutorial steps in this chapter to create a multi-clock domain system with SOPC Builder and exercise the system in hardware.