XILINX®

The Low-Cost, Efficient Serial Configuration of Spartan FPGAs

XAPP098 November 13, 1998 (Version 1.0)

Application Note by Kim Goldblatt

Summary

This application note shows how to achieve low-cost, efficient serial configuration for Spartan FPGA designs. The approach recommended here takes advantage of unused resources in a design, thereby reducing the cost, part count, memory size, and board space associated with the serial configuration circuitry. As a result, neither processor nor PROM needs to be fully dedicated to performing Spartan configuration.

In particular, information is provided on how the idle processing time of an on-board controller can be used to load configuration data from an off-board source. As a result, it is possible to upgrade a Spartan design in the field by sending the bitstream over a network.

A brief summary of Spartan slave serial configuration, its protocol and signals, lays the groundwork for a discussion on ways to reduce bitstream storage and processing requirements. A detailed example illustrates how these techniques can be put into practice. Finally, different formats for configuration data are described along with instructions for their use.

Xilinx Family

Spartan and SpartanXL families

Introduction

In today's markets for electronic products, the designer strives for the continuous improvement of products. Cheaper components must be used to create more functionality with less board space. The low-cost, feature-rich Spartan series of FPGAs, consisting of the 5-Volt Spartan family and the 3.3-Volt SpartanXL family, plays a major role in helping the designer achieve these goals. This role is considerably enhanced when FPGAs are configured without the use of dedicated storage and processing resources.

This application note describes a number of techniques for achieving low-cost, efficient serial configuration. Extra room in on-board RAM or in a hard drive off the board can be used to store the configuration data. The idle time of a controller, whose primary purpose lies in executing other tasks, can be used to coordinate the loading of the bitstream into the Spartan device. Unused register bits accessible to the processor can be used for holding control, data and status bits. As a result, it is possible to configure Spartan FPGAs without the use of a PROM. Board space and component costs are saved. Besides these benefits, there are numerous others, including the following:

- The Spartan serial configuration *protocol* can be integrated with other initialization activities as part of the controller's general program. This makes board reset and initialization easier to coordinate.
- As part of the controller's program, the Spartan configuration protocol can be modified with relative ease. This can be useful not only for prototype development, but also product upgrades. For example, consider the case where, for a product under development, a new member is added to a daisy chain of existing Spartan devices.
- The Spartan *bitstream* can be embedded into the processor's program. Fewer separate code files are easier to manage.

This application note begins with some preliminary background on the serial configuration mode for Spartan FPGAs, including a description of the pertinent signals and their respective timing. Next, instructions show how an on-board controller can be used to program Spartan FPGAs without a dedicated PROM. Finally, information is provided on how to make the Spartan bitstream suitable for handling by the processor.

Spartan Serial Configuration

There are a number of different modes for configuring Spartan devices, including Slave Serial, Master Serial, JTAG, and Express (for SpartanXL only). Among these, the two serial modes, employing the least number of interface signals (a minimum of four), are the easiest to implement. In the present application, a controller, writing configuration data, plays the controlling role of "master", whereas the Spartan FPGA, receiving the data, serves as a "slave". For these reasons, this application note considers only the case of slave serial configuration.

As many as nine pins on the Spartan device may be used in Slave Serial Configuration: MODE, DIN, DOUT, CCLK, PROGRAM, INIT, DONE, HDC and LDC. A minimum of four signals, $\overline{\text{PROGRAM}}$, $\overline{\text{INIT}}$, DATA and CCLK are required. The principal functions of the nine signals are described in Table 1. Refer to the Spartan data sheet for more detailed information.

Table 1: Signals for Spartan Configuration

Signal	Type	Direction	Description
MODE (Spartan) **M0, M1 (SpartanXL)**	Mode select	Input	To select Slave Serial mode on Spartan devices, tie Mode High; on SpartanXL devices, tie both M0 and M1 High.
DIN	Data	Input	Write configuration data into the Spartan device
DOUT	Data	Output	Read configuration data from the Spartan device
CCLK	Clock	Input	Synchronizes data on the rising edge
$\overline{\text{PROGRAM}}$	Control	Input	Begin clearing the Spartan configuration memory
$\overline{\text{INIT}}$	Status	Open-drain output	A transition from Low to High indicates that the Spartan configuration memory is clear and ready to receive the bitstream
DONE	Status	Open-drain output	A High indicates that the configuration process is complete
HDC	Status	Output	High throughout configuration, until the I/Os go active
$\overline{\text{LDC}}$	Status	Output	Low throughout configuration, until the I/Os go active

Steps in the Configuration Process

The Slave Serial mode consists of four steps:

1. Clearing Configuration Memory
2. Initialization
3. Configuration
4. Start-Up

Let's have a look at each of these steps so that we may understand how the nine configuration signals work together to program a Spartan device. Refer to the Spartan datasheet for more details.

Clearing Configuration Memory

On power-up, once V_{CC} reaches the Power-On-Reset threshold, the device automatically begins clearing the configuration memory. It is also possible to begin the clearing operation by applying a Low-level pulse to the $\overline{\text{PROGRAM}}$ input.

This line makes *reconfiguration* possible at any time during device operation. It is particularly useful when the controller needs to initiate Spartan configuration at a specific point in the power-up sequence.

As long as $\overline{\text{PROGRAM}}$ is Low, the device continues to cycle through the clearing step. After each pass through the configuration memory, $\overline{\text{PROGRAM}}$ is sampled. If $\overline{\text{PROGRAM}}$ is High, then one last clearing pass takes place, which concludes with a Low-to-High transition on $\overline{\text{INIT}}$.

Do not hold $\overline{\text{PROGRAM}}$ LOW for more than 500µs. Therefore, $\overline{\text{PROGRAM}}$ should not be used to delay the configuration process for periods of this magnitude. Hold $\overline{\text{INIT}}$ Low instead.

Initialization

Since $\overline{\text{INIT}}$ is an open-drain output, it requires a pull-up resistor to achieve a High level. Now that $\overline{\text{INIT}}$ has gone High, the internal memory is completely clear. At this point, the device identifies the selected configuration mode by sampling the level on the mode pins, after which it activates the appropriate configuration logic. The device is ready to begin the configuration step. Note that holding $\overline{\text{INIT}}$ Low can be used to delay the entry to the configuration step.

To select slave serial mode (which is required for the approach discussed in this application note) for the Spartan family, the MODE pin is tied High; for the SpartanXL family, the M0 and M1 pins are tied High.

Configuration

After $\overline{\text{INIT}}$ goes High, it is necessary for the controller to wait for a period between 55 µs and 275 µs before driving the CCLK input with the clock signal that transfers the bitstream.

DIN is the input for serial configuration data. DOUT is used when more than one device is connected in a daisy chain. See Figure 2 for an example. The bitstream, having filled the first device in the chain, will exit its DOUT line, which leads to the DIN line of the next device where configuration continues. The data exiting DOUT is clocked on the falling edge of CCLK. The data entering DIN is clocked on the rising edge of CCLK as shown in Figure 1. The clock oscillator internal to the device is not used in Slave Serial Mode to transfer data; it is only used during Initialization.

Start-Up

The Start-Up step provides a smooth transition from configuration to user operation. Three major events occur during Start-Up: The DONE output goes High, the I/Os go active and the GSR (Global Set/Reset) Net is released. Start-Up takes place over a period of four cycles labeled C1, C2, C3 and C4. Options in *BitGen*, the bitstream generation program in the Xilinx development software, determine which event takes place in which cycle. The menu for these options can be located as follows:

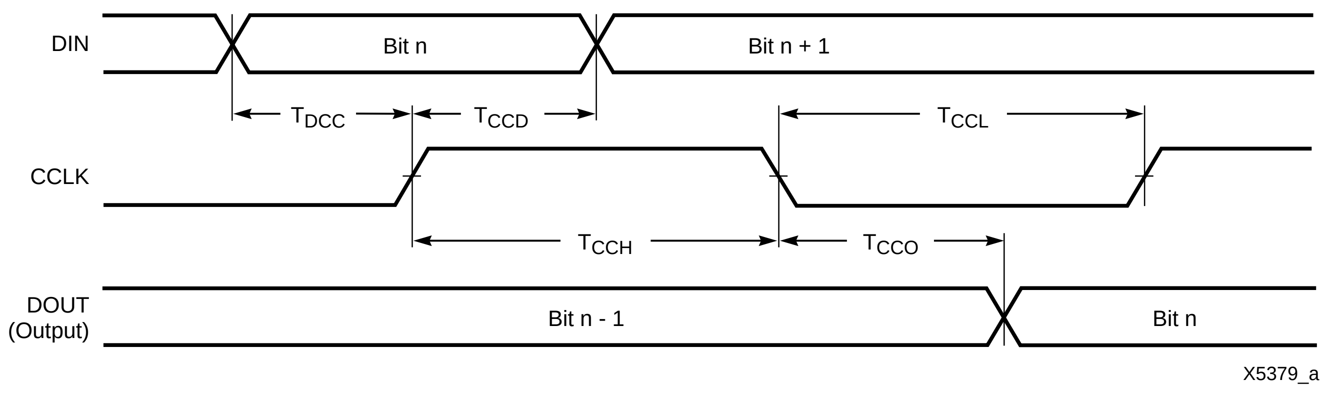

Figure 1: Loading and Readback of the Bitstream in Slave Serial Mode

1. Open the Design Manager.
2. Select Implement from under the **Design** menu.
3. Choose the **Options** button.
4. Click on **Edit Template** for Configuration.
5. Select the **Startup** tab. A menu will appear that permits the three events to be assigned to different cycles.

As an alternative, *BitGen* options can also be selected using **Template Manager**, which is found under the **Utilities** menu of the Design Manager.

The customary default option, known as CCLK_NOSYNC, is the most practical, since DONE goes High in C1, disconnecting the data source; I/Os go active in C2, avoiding contention; GSR released in C3, ensuring stable internal conditions. The CCLK is used to measure out the four start-up cycles. This application note only considers the default option.

There are two conditions that determine when the Start-Up step begins: (1) length count match and a (2) full configuration memory. The 24-bit length count is part of the bitstream header, that, once written to the Spartan device, is stored in a register. (See "The Anatomy of a Spartan Bitstream" on page 7.) From the time $\overline{\text{INIT}}$ goes High, a counter within the device begins counting, incrementing by one for each CCLK rising edge and comparing the result to the length count. Once the counter's contents match length count, the first condition is met. Meeting the second condition occurs when all the configuration *data* for a given Spartan device has been loaded into configuration memory.

A High on DONE is a result of the Start-Up process. While this transition on DONE indicates the completion of the configuration *step*, the configuration process, as a whole, ends with the last cycle of the Start-Up step, C4. It is important to provide CCLK rising edges for all four start-up cycles. This amounts to clocking the entire bitstream, from the first bits of the header to the last bit of the post-amble. Note that the total number of CCLK cycles required to write the bitstream equals the length count plus the four start-up cycles.

DONE is an optional signal, since the master knows how many bits of configuration bits need to be written. DONE's failure to go High generally indicates a problem with configuration (i.e. a bit error or incomplete loading of configuration data).

Like DONE, the HDC and $\overline{\text{LDC}}$ outputs provide status on the device's progress to user operation. HDC is High during configuration and takes on whatever I/O function is assigned to it at the time when all I/Os go active, in the Start-up step. Similarly, $\overline{\text{LDC}}$ is Low during configuration and takes on its respective I/O function when the I/Os go active as well.

The Controller Interface

By following three simple techniques, it is possible to implement serial configuration without adding any dedicated logic to a design.

1. Store the Spartan configuration data either in unused on-board memory or at an off-board location.
2. Use the idle time of an on-board controller for coordinating configuration.
3. A free register on-board (e.g., in a CPLD or an I/O port) can be used as a synchronous interface between the controller and the Spartan device.

These techniques can also be used independently. They are discussed in the sections that follow.

Storing the Spartan Bitstream

Storing the Spartan bitstream in unused *on-board* memory or downloading it from an *off-board* source means that additional memory components (i.e. PROMs) for storing configuration are unnecessary. This saves component costs and board space.

In the on-board case, a form of nonvolatile memory, such as ROM, is used to hold the configuration data. Generally, the data will be embedded in the processor's firmware. See "Embedding the Bitstream in Firmware" on page 8 for information on how to prepare configuration data for inclusion in C code.

As an alternative to the embedded approach, a free portion of on-board ROM can be set aside to store the bitstream in a table that is independent from the firmware. During board initialization, the firmware can then instruct the processor to access the table.

In the case of off-board storage, the Spartan bitstream can be downloaded from a number of different locations, including a hard drive (or floppy drive) within the product as well as outside of the product (i.e., belonging to a workstation). In just one of many possible scenarios, a company can permit their customers to perform their own updates to a Spartan-based product simply by downloading a new version of the bitstream via the internet. There is no need for the company to ship PROM replacements. The customer takes the bitstream and stores it on a hard drive that is accessible to a controller in the Spartan-based product.

Controlling Configuration

An on-board controller can use its idle processing time to supervise Spartan configuration. By having the controller do double duty, no dedicated processing components are required. The controller supervises serial configuration by monitoring status signals, issuing control signals, manipulating the bitstream, and providing for synchronization to a clock.

For most designs, sufficient processing time will be available during board initialization. As an example, the serial configuration of the largest Spartan devices presently available, the XCS40 and XCS40XL, would require 329,312 cycles and 330,696 cycles respectively. If CCLK is toggling at the maximum allowable frequency of 10 MHz, then serial configuration would take about 33 ms.

If insufficient continuous processing time is available for configuration, then the task of writing the bitstream may be interrupted so the controller can attend to other tasks, only to be resumed at a later point in time. In this case, the task of writing the bitstream exists as firmware subroutine, to which an interrupt priority can be assigned.

In brief, the $\overline{\text{INIT}}$ line on the Spartan device can be used to drive the interrupt line on the controller. A suitably low priority level can be assigned to this interrupt to ensure that the controller spends sufficient time servicing its primary tasks. As previously described, the processor initiates slave serial configuration by pulling $\overline{\text{PROGRAM}}$ Low. Once all the Spartan devices are clear, $\overline{\text{INIT}}$ goes High, requesting an interrupt of the controller. When the controller has no requests of higher priority than that of the Spartan device, it begins accessing configuration data from memory and writing them to the DIN input, bit-by-bit. While the controller will break away from configuration to attend to any higher priority requests, as soon as these are complete, it will continue with configuration until the DONE signal, monitored at the interface register, goes High.

When using interrupts, it is important to use a unique address for the Spartan device (or daisy chain). This avoids potential address conflicts when switching tasks. See "The Interface Register" on page 4 for how this is accomplished.

The Interface Register

A free register can be used to establish a synchronous interface between the controller and the Spartan device(s). The interface register is composed of two parts: the *output register* and the *input register,* which store Spartan configuration signals each time the processor does a read or write to the port. In order to support the set of signals commonly used for the slave serial mode ($\overline{\text{PROGRAM}}$, DATA and CCLK, $\overline{\text{INIT}}$ and DONE), the output register will be three bits wide for write operations and the input register will be two bits wide for read operation. More bits can be added for other control signals. For example, when using the readback feature, add two bits for the READ_DATA and READ_TRIGGER signals.

Ideally, this register will come from the unused flip-flops of a CPLD enabled by a unique address. Typically, the CPLD will have been placed on the board for the original purpose of decoding address lines. The unique address ensures that the configuration data on the processor's bus goes only to the register and nowhere else. It also ensures that data on the bus intended for other purposes cannot be written to the interface register. The nonvolatile nature of the CPLD is important, since, the interface register needs to be ready to support FPGA configuration on power-up.

A Practical Example

The techniques for low cost, efficient serial configuration of Spartan FPGAs, as presented in this application note, are commonly used in a wide variety of electronic products. Figure 2 shows a block diagram of a multi-board, multi-processor system with an ethernet connection to a host computer (external to the product). This kind of system is commonly used in automated test equipment. The ethernet connection makes it possible to download FPGA configuration data from a host computer, making changes to the Spartan design easy to implement. Since the configuration data does not require on-board, non-volatile storage, component costs and board area are saved. As an alternative to ethernet, other kinds of WAN or LAN (e.g., the internet) can be used.

Our system consists of several boards, each board with a number of processors. There is one *supervisor* processor on each board to manage general administrative tasks, including board initialization on reset as well as top-level

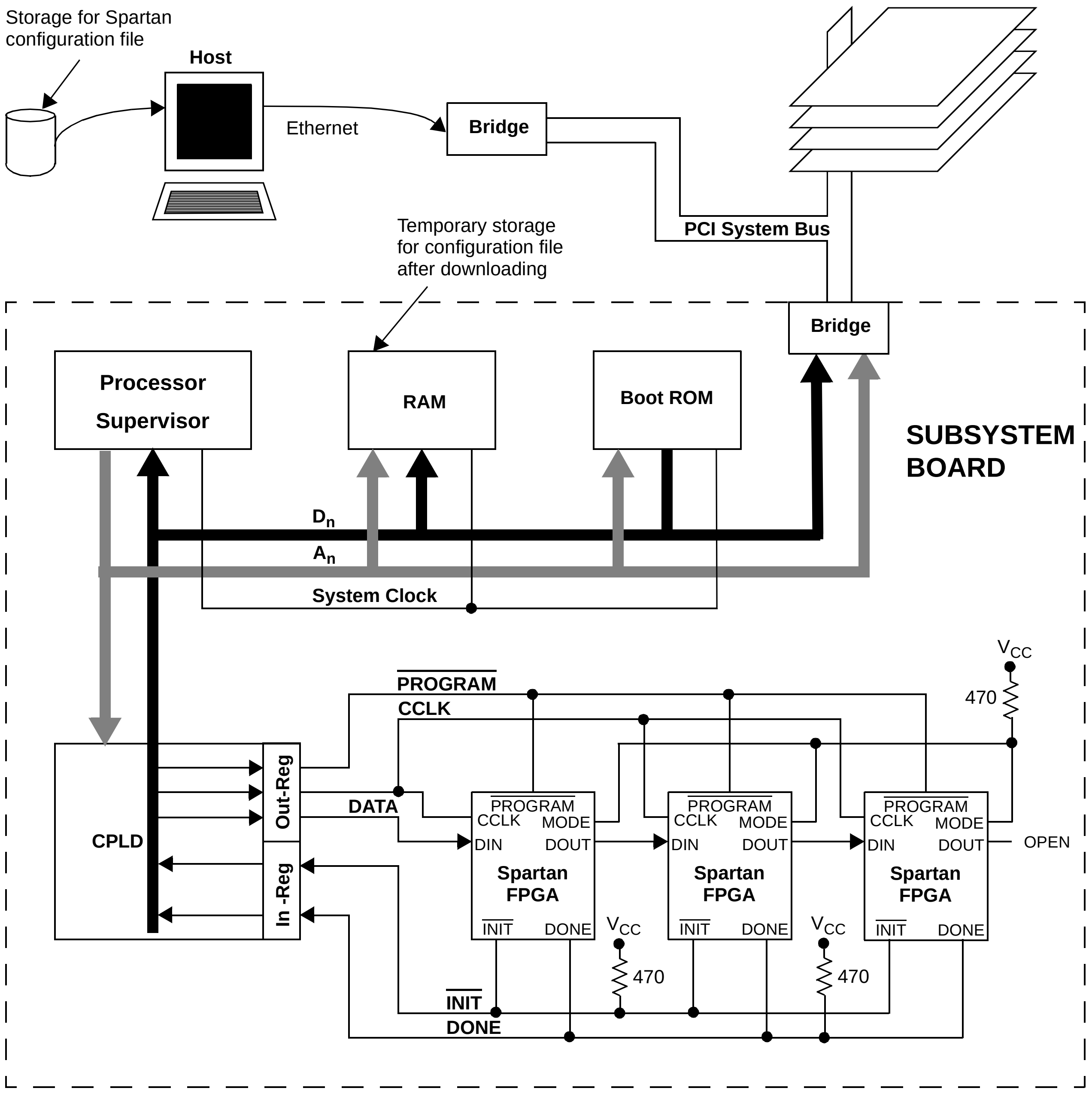

Figure 2: Configuring Spartan FPGAs from an Off-Board Location

Note: To select the Slave Serial mode on a Spartan device, MODE is tied High. On a SpartanXL device, both M0 and M1 are tied High.

functional coordination throughout operation. The supervisor serves as the *master* for serial configuration, receiving a download of the bitstream file, preparing it, and

coordinating the act of configuration. The three Spartan FPGAs serve as slaves, so MODE is tied High to select the slave serial configuration mode.

Figure 2 shows one of the boards within the system in detail (surrounded by the dashed line). For the sake of clarity, only the supervisor processor is shown, from here on referred to simply as “the processor”. Its data and address buses, D_n and A_n respectively, permit access to the Boot ROM, RAM, a network interface and a CPLD. The firmware for the processor resides in Boot ROM. On reset or power-up, the processor begins reading its instructions from here.

The Spartan configuration data is stored on the hard drive of the host computer. When configuring the Spartan design is desired, the bitstream is downloaded from the host, transferred from ethernet to the ATE system's internal PCI bus and, from there, passed onto the board's local data bus (D_n). Prompted by an interrupt request, the processor receives the configuration data file from the PCI bus and copies it into consecutive bytes of RAM for temporary holding.

The CPLD contains the interface register that holds the bit values of the serial configuration signals. This example employs the minimum required number of signals: $\overline{\text{PROGRAM}}$, CCLK, DATA, $\overline{\text{INIT}}$ and DONE. The interface register consists of two parts, one called *Out-Reg* and the other called *In-Reg*. The processor writes bit values for $\overline{\text{PROGRAM}}$, DATA and CCLK into Out-Reg, which, in turn, applies those values to the corresponding inputs of the Spartan device(s). Also, on a regular basis, In-Reg samples $\overline{\text{INIT}}$ and DONE from the Spartan device(s) and makes those bit values available on D_n for monitoring by the processor. During serial configuration, the processor takes turns writing *control bits* to Out-Reg one instruction cycle and reading *status bits* from In-Reg the next. The three values contained in the control bits provide the logic levels that drive Out-Reg's $\overline{\text{PROGRAM}}$, DATA and CCLK signals. The values of the status bits communicate to the processor the levels of In-Reg's $\overline{\text{INIT}}$ and DONE signals.

A sample sequence of control bits is shown in Table 2. Each row in the table shows the bit values in the interface register at a given point in time. This is just one of a number of different possible sequences. The full sequence, from start to finish, passes through the four steps of Spartan serial configuration: Memory Clear, Initialization, Configuration and Start-up. The processor initiates memory clearing by issuing control bits with $\overline{\text{PROGRAM}}$ set Low. If $\overline{\text{INIT}}$ is not already Low, it will go Low at this time. During this step, CCLK can be High or Low, so long as there's no rising transition. Dummy bits occupy the DATA position. The processor monitors the In-Reg until it detects $\overline{\text{INIT}}$ at a High level. At this point, initialization takes place. With the beginning of the configuration step, the processor begins to write control bits with "real data" while, at the same time, continues to monitor In-Reg. Finally, according to the customary default order for events, as selected in *BitGen* (see "Start-Up" on page 2), the Start-up step readies the Spartan device for user operation over a series of four CCLK cycles: In C1, DONE goes High. In C2, the I/Os become active. In C3, the GSR net is released. With C4, user operation begins. It is important that a rising transition on CCLK be provided for C1, C2, C3 and C4. The bits clocked during those cycles, B_{n-3} through B_n, are dummy bits belonging to the post-amble.

We last left the configuration data stored in RAM. Before the processor can send this data to the Spartan devices, it is necessary to format them into control bits. The processor can accomplish this real-time by reading a byte of configuration data from RAM and distributing the eight bits of

Table 2: State Sequence for the Interface Register

Configuration Step	Contents of Interface Register				
	Control Bits in Out-Reg			Status Bits in In-Reg	
	$\overline{\text{PROGRAM}}$	CCLK	DATA	$\overline{\text{INIT}}$	DONE
Memory Clear	1	NRT[1]	X[2]	0[3]	0[4]
	0	NRT	X	0[3]	0[4]
	$\overline{\text{INIT}}$ goes Low (if not already Low).				
	1	NRT	X	0	0
	Wait for a period between 55 µs and 275 µs after $\overline{\text{INIT}}$ goes High				
Initialization	1	NRT	X	1	0
Configuration	1	0	b_0[5]	1	0
	1	1	b_0	1	0
	1	0	b_1	1	0
	1	1	b_1	1	0
	1	0	b_2	1	0
	1	1	b_2	1	0
	Continue writing bits. When the length count match occurs and configuration memory is full, then Start-Up begins.				
Start-Up[6]	1	0	b_{n-3}	1	0
	1	1 (C1)	b_{n-3}	1	0
	DONE goes High.				
	1	0	b_{n-2}	1	1
	1	1 (C2)	b_{n-2}	1	1
	I/Os become active.				
	1	0	b_{n-1}	1	1
	1	1 (C3)	b_{n-1}	1	1
	GSR is released.				
	1	0	b_n	1	1
	1	1 (C4)	b_n	1	1
	Begin User Operation				

Notes:
1. NRT means No Rising Transition.
2. X is a "don't care" input.
3. The logic level shown is for configuration after power up. For configuration during operation, prior to driving $\overline{\text{PROGRAM}}$ Low, DONE will be High.
4. The logic level shown is for configuration after power up. For configuration in mid-operation, prior to driving $\overline{\text{PROGRAM}}$ Low, $\overline{\text{INIT}}$ may be an active I/O, in which case, it will be driving either a High or a Low.
5. b_i represents the sequence of configuration bits i = 0 through n, starting with the first bit of the preamble, b_0, and ending with the last bit of the post-amble, b_n.
6. This example shows the Start-Up events ordered according to the customary default settings for CCLK_NOSYNC in *BitGen*.

which it is composed among control bits. Each bit needs to be placed in the DATA bit-position. Furthermore, the processor needs to provide the appropriate logic levels for the PROGRAM and CCLK positions. (As an alternative, the workstation can format the configuration data into control bits, which it then downloads to the board.)

The bit values for all three signals need to be chosen in compliance with the protocol summarized in "Steps in the Configuration Process" on page 2 as well as the timing requirements described in the Spartan datasheet. For example, note in Table 2 that during the configuration step, each data bit is repeated for two consecutive writes to Out-Reg. CCLK is Low for the first occurrence and High for the second. This ensures that the setup time for DATA with respect to CCLK is met.

It is important that the order of the control bits, as written to the Out-Reg, preserve the bit order of the original configuration data file. The first bit of the header (just after the title declaration) needs to be the first bit written to the Spartan FPGA.

Figure 2 shows three Spartan devices connected in a daisy chain, though any number of Xilinx FPGAs can easily be accommodated in such a loop. The DATA line applies the bitstream to the DIN pin of the left-most FPGA first. After this device is configured, its DOUT pin drives the middle FPGA's DIN pin with the bitstream. Only after this second device is configured, does its DOUT pin pass the bitstream onto the right-most device. For a daisy chain, the configuration files for the individual Spartan devices need to be combined into a single bitstream. For details, see "Bitstream Considerations" on page 7 for more information.

If the same bitstream is to be loaded into more than one Spartan device, then those devices can be connected with their configuration signals in parallel. See the Spartan datasheet for more information on daisy chain and parallel configuration.

Bitstream Considerations

The *BitGen* utility within Xilinx development software produces configuration data files in a number of different formats. The ones that are most useful for configuring Spartan FPGAs with a controller are: the rawbits file (.RBT), the hex file (.HEX), and the bit (.BIT) file. Table 3 summarizes the distinguishing characteristics of these files.

The Anatomy of a Spartan Bitstream

A distinct benefit of the rawbits file is that the bitstream can be easily viewed using a common text editor. Figure 3 shows the internal organization of a Spartan bitstream. Note that for the same design, the data frames for Spartan and SpartanXL devices will contain a different number of bits. Nevertheless, the internal organization of the bitstream, as described below, applies to both Spartan families. At the top of the file is a title declaration, which provides information about the configuration data such as:

- Configuration data file format
- Version of Xilinx development system in use
- The name of the design
- The target device
- The date the file was created
- The number of bits of actual configuration data

Following the title declaration, the actual bitstream begins tion with a 40-bit header, the beginning and end of which are shown in bold. It consists the following parts:

- A minimum of eight dummy bits, all High
- A preamble code of '0010'
- A 24-bit length count
- At least four more dummy bits, all High

Following the header is the first data frame, which, like all data frames, begins with a 0 and ends with a four-bit CRC code (the default option). The file ends with the post-amble code 0111111 (shown in bold at the bottom of the figure), followed by eight start-up bits (all High). Note that the title declaration is never loaded into the Spartan FPGA, only the header and the data frames that follow go into the device during configuration.

The Rawbits File

When applying the techniques for efficient serial configuration discussed in this application note, a rawbits file will typically be stored in free on-board RAM or in an off-board location (i.e., the floppy disk or hard drive of a workstation). Before the configuration data can be written to the Spartan device, it is necessary to first strip off the title declaration, then convert the header and data frames from ASCII to binary. This can be accomplished by the on-board controller if the file is stored in RAM. Alternatively, if stored on a hard drive, the workstation can perform the translation prior to downloading. This latter strategy is especially useful if

Table 3: Configuration Data Files

File Format	File Extension	Title Declaration	Description
Rawbits	.RBT	Yes	Bitstream is coded in ASCII, one byte for each configuration data bit
Hex	.HEX	No	Each group of four consecutive configuration data bits is represented as one Hex digit (i.e., 0 through F) which, in turn, is coded as one ASCII byte
Binary	.BIT	Yes	Bitstream is coded in binary, one configuration bit after the next

```
Xilinx ASCII Bitstream
Created by Bitstream M1.5
Design name: cntlogix.ncd
Architecture:spartan
Part:       s30pq240
Date:        Tue Sep 29 16:40:13 1998
Bits:       247968
11111111001000000001111001000100110011111
0101011111111111111011110101111111010111111 1
0101111111010111111101011111110101111111010
1111111010111111101011111110101111111010111
1111011011111110101111111010111111101011111
1101011111110101111111010111111101011111110
1011111110101111111010111111101011111110111
11110010
                                  .
                                  .
                                  .
0111111111111110111111101011111110101111111
0101111111010111111101011111110101111111010
1111111010111111101011111110101111111010111
1111011011111110101111111010111111101011111
1101011111110101111111010111111101011111110
1011111110101111111010111111101011111110011
0111001001111111111111111
```

Figure 3: Spartan Rawbits Configuration File

on-board RAM is scarce, since the rawbits file takes up eight times the space of the binary version.

The Hex File

The hex file is handled in a similar fashion. It has an advantage over the rawbits file in that it is more compact, therefore taking up less space in memory. On the other hand, the hex file requires more processing to convert it into the binary that is used to configure the Spartan device. The Hex file does not have a text title declaration. Before configuration, it is necessary to convert ASCII to hex (i.e., each byte becomes a single hex digit), and then from hex to binary (i.e., each hex digit becomes a nibble).

The Binary File

A binary format has two benefits over the other two file types. First, it is the most compact of all, taking up half the storage space of a hex file and one eighth the space of a rawbits file. As a result, the format is ideal for storage on board. Second, once in binary form, the header and data frames require no further translation and can be written directly to the device.

Because of the difficulty in identifying and removing the title declaration, the binary (.bit) file created by *BitGen* is not recommended for use. Instead, one should use *BitGen* to create a hex file, which, in turn, is converted to binary as described in the preceding section.

Embedding the Bitstream in Firmware

As an alternative to storing the configuration file in on-board RAM or on the hard drive of a workstation, it is also possible to embed it in the controller's firmware. To perform this task, use a utility called *makesrc* which is available for free downloading from the Xilinx web site, WebLINX. The file can be found as follows:

1. Visit WebLINX at *www.xilinx.com*.
2. Perform a Xilinx site search by selecting **Search**.
3. In the blank denoted by the words "**Search for:**", type the name of the utility you are looking for followed by an asterisk. (In this instance, type *makesrc**.) The asterisk is a wild card character that will allow for any file extension the utility may have. Press enter to begin the search.
4. The browser will report the results of the search. Click on any of the hypertext links found. This takes you to a page from where the utility can be downloaded.
5. Click the file name (i.e., *makesrc.zip* for PCs*)* to download the file.
6. De-compress using an unzip program (e.g., PKZip) and it is ready to use.

There are actually a host of useful utilities for managing configuration data files, more of which will be described in the next section. They can all be downloaded from WebLINX using the same procedure.

Perform a search for the file name (followed by an asterisk for any extension). It is available in both a tar file for a UNIX-based workstation and a PKZip file for PCs. This utility accepts only the MCS PROM format as an input. The first step for embedding configuration data in firmware is to generate a configuration data file in MCS format using *PromGen* in the Xilinx development software. The *pconfig* utility available on WebLINX converts .bit, .rbt, and .hex files to MCS format. The MCS file is used as an input to *makesrc*, which produces a HEX file with formatting customized to suit the needs of different assemblers and compilers. Consult the read-me file that accompany the utility for more details.

Combining Files for a Spartan Daisy Chain

The integrated bitstream used for configuring a Spartan daisy chain is not a simple concatenation of the configuration files for the individual devices. *PromGen*, a utility in the Xilinx development software, must be used to join the files. This utility only combines binary (.bit) files, which the *BitGen* utility readily supplies. *PromGen* takes the binary files for the different devices, strips off the title declarations and the headers, merges the data frames, and, finally, adds a new header at the top. The output file is a hex file (no title declaration). If a rawbits file is desired, then use the utility *hex2bits* available on WebLINX to convert the hex file. *Hex2bits* is available in both zip and tar versions. Consult the accompanying read-me file.

Verifying Configuration

Successful loading of the bitstream into the Spartan device can be verified by reading back the configuration data in serial. This is accomplished by instantiating readback symbols into the Spartan design. Refer to the Spartan datasheet and XAPP015 for directions on how to use this feature.

Bibliography

The Xilinx Spartan Series Datasheet
(www.xilinx.com/partinfo/spartan.pdf)

The Xilinx Programmable Logic Data Book 1998
(www.xilinx.con/partinfo/databook.htm)

XAPP015: Using the XC4000 Readback Capability
(www.xilinx.com/xapp/xapp015.pdf)