XILINX®

XAPP1081 (v1.3) March 18, 2014

QuickBoot Method for FPGA Design Remote Update

Author: Randal Kuramoto

Summary

A primary advantage of an All Programmable FPGA is its remote update capability. This feature enables deployed systems to be updated with design patches or enhanced functionality.

This application note provides a solution that enables a reliable field update through a complementary combination of a fast, robust configuration method and an efficient HDL-based, in-system programming reference design. Together, the solution is referred to as the QuickBoot method.

Introduction

Figure 1 shows the architecture of a system with remote FPGA update capability.

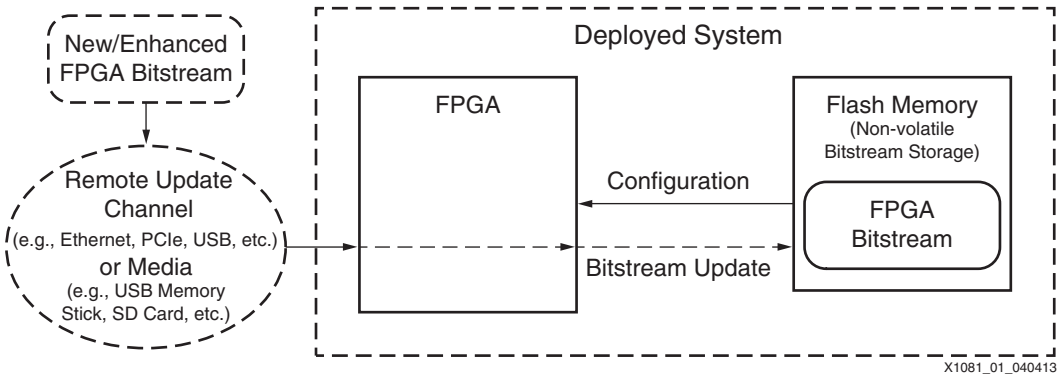

Figure 1: **System With Remote Update Capability**

This application note presents detailed descriptions of the QuickBoot method that are important for evaluating the QuickBoot solution and debugging implementation problems.

Demonstration implementations of the QuickBoot method are provided for the KC705 evaluation board using the serial peripheral interface (SPI) flash or byte-wide peripheral interface (BPI) flash. See KC705 Board Demonstrations, page 33 to run the QuickBoot demonstrations on the KC705 evaluation board.

See QuickBoot Reference Design Implementation Guide, page 14 to start implementing the QuickBoot reference design in an FPGA.

Features

Key features of the QuickBoot method include:

- Support for all 7 series FPGAs
- HDL-based flash programmer reference design
- Minimal remote update payload size approximately the size of a standard bitstream
- Simple programmer interface protocol for payload delivery
- Built-in remote update error recovery/fallback to a known good or “golden” bitstream
- Quick configuration time for either the update bitstream or golden bitstream

- Compatibility with several configuration options, including:
  - SPI mode
  - BPI mode
  - Encrypted bitstreams
  - Multiple FPGA configuration daisy-chains

Traditional Solution for Remote Update

The traditional solution for a remotely updatable FPGA involves a flash memory that has reserved areas within the main flash memory array to store these components:

- An update bitstream
- A known good or “golden” bitstream

A remote programming method is implemented and is used to program new or enhanced bitstreams into the update bitstream area. The FPGA preferably configures with the update bitstream.

If the remote update procedure fails or is interrupted, the FPGA must be able to reliably fallback and configure from the golden bitstream. Typically, the golden bitstream is never modified to ensure its known good condition for all cases.

Traditional remote update solutions use the FPGA’s built-in MultiBoot and Fallback features. The MultiBoot feature enables the FPGA to selectively load a bitstream from a specified address in flash memory. If the FPGA detects a configuration error, the Fallback feature resets the FPGA and retries configuration from address zero of the flash memory. See Figure 2 for the traditional Fallback MultiBoot flash memory components and configuration method. For details of the MultiBoot and Fallback features, see the “Fallback MultiBoot” section in *7 Series FPGAs Configuration User Guide* [Ref 1].

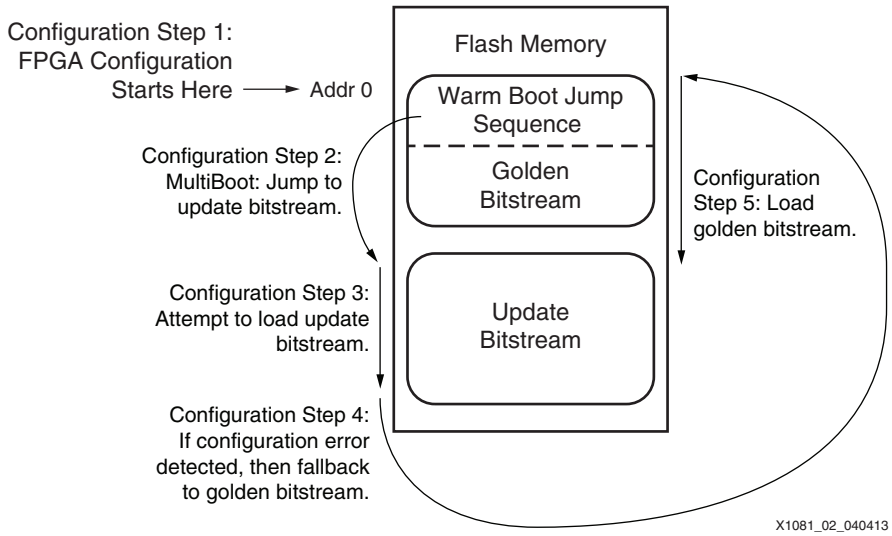

Figure 2: **Traditional Fallback MultiBoot Flash Memory Components and Configuration Method**

Note: During Fallback, the FPGA ignores the warm boot sequence.

For MultiBoot configuration, bitstream generation settings insert a “warm boot jump sequence” within the header of the golden bitstream that causes the FPGA to reboot to the next configuration address. Two FPGA configuration packets comprise the warm boot jump sequence:

- A packet for writing a new configuration start address into the FPGA’s warm boot starting address register (WBSTAR)
- A packet that issues the internal program (IPROG) command to restart the FPGA configuration from the address specified in the WBSTAR

Together, these packets comprise the warm boot jump sequence that directs the FPGA to reset configuration and jump to a specified configuration start address.

In summary, the Fallback MultiBoot solution uses a straightforward update bitstream overwrite method and a sequential try-and-recover configuration method. The FPGA configuration logic is solely responsible for recovery from any programming errors or interruptions. The FPGA tries to configure from the update bitstream, and if the FPGA detects a configuration error, the FPGA initiates a reconfiguration Fallback with the golden bitstream. Because of the additional, initial configuration attempt, the configuration time for the Fallback case can be twice as long as the standard configuration time.

QuickBoot Method for Remote Update

The QuickBoot method places the responsibility for programming error/interrupt recovery on the programming operation via a simple adjustment to the programming algorithm for the bitstream update process. The QuickBoot method integrates the programming method with a special configuration method that is based on a special configuration header to form the remote update solution. This solution is robust, compatible with many configuration setup variations, and quick to configure in all cases. Figure 3 shows an overview of the QuickBoot system.

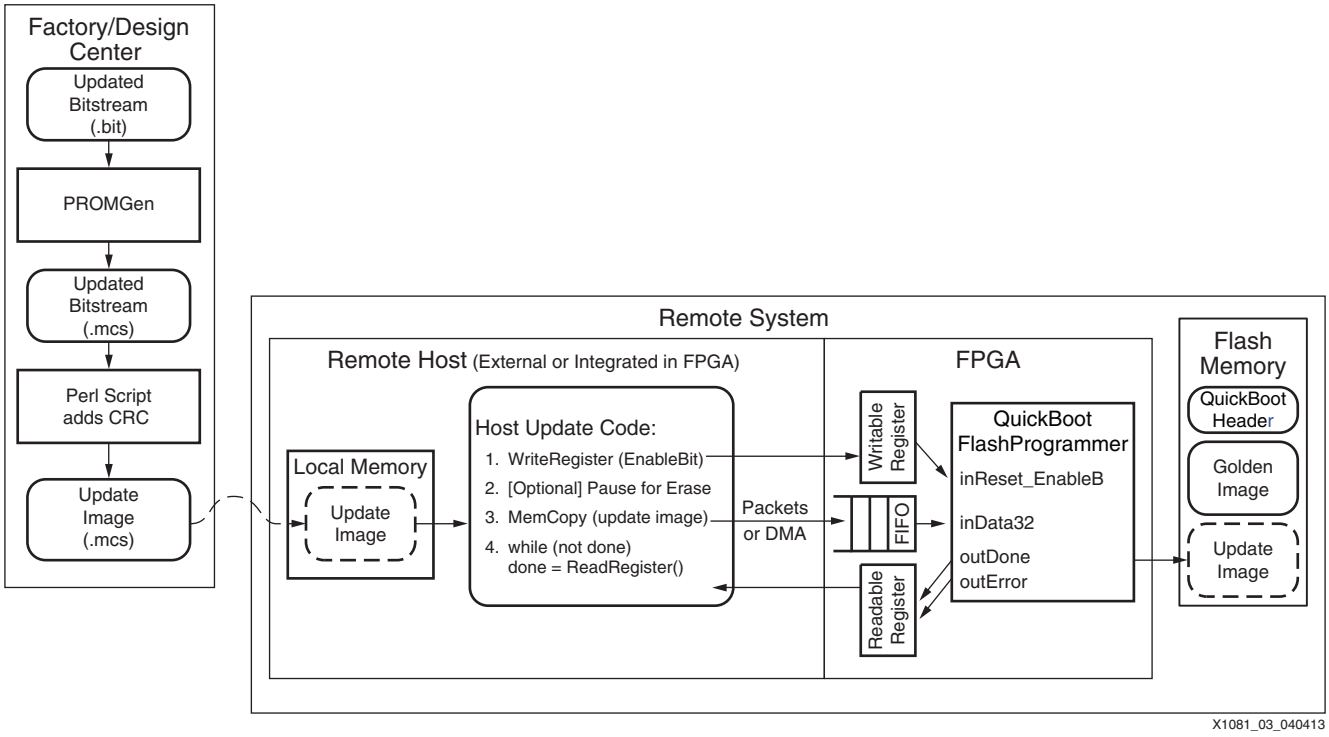

Figure 3: **Ecosystem for QuickBoot Remote Update Solution**

The reference design associated with this application note includes the QuickBoot flash programmer and scripts for generating the flash memory image with the QuickBoot header.

QuickBoot Configuration Method

Figure 4 illustrates the QuickBoot configuration method for remote update.

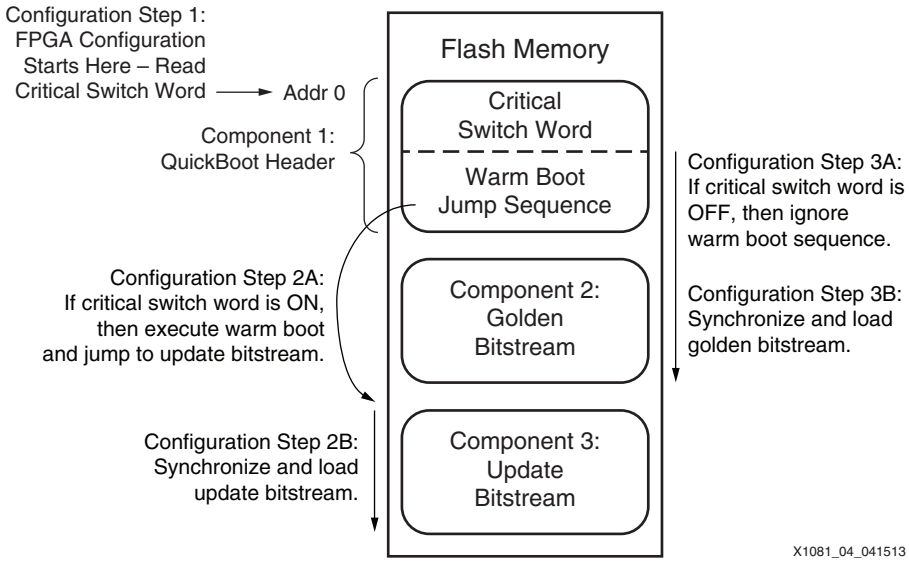

Figure 4: **QuickBoot Flash Memory Components and Configuration Method**

QuickBoot involves these components in flash memory:

- A special QuickBoot header
- A golden bitstream image area
- An update bitstream image area

The QuickBoot flash memory components are similar to the MultiBoot flash memory components, except that the MultiBoot warm boot jump sequence is a separate component from the golden bitstream component. The subcomponents of the QuickBoot header are further identified as:

- A “critical switch word”
- A warm boot jump sequence

The critical switch word is the key to the QuickBoot configuration method. The critical switch word is either ON or OFF. When the switch is ON, the FPGA loads the update bitstream. Otherwise, when the switch is OFF, the FPGA loads the golden bitstream.

The QuickBoot configuration sequence shown in Figure 4 occurs in this manner:

1. Start reading from flash address zero. Read the value from the critical switch word location and, depending on the switch value, go to either step 2 or step 3.
2. If the critical switch word is ON, configure with the update bitstream:
   a. Execute the warm boot jump sequence that follows the critical switch word and jump to the update bitstream area.
   b. Load the bitstream from the update bitstream area.
3. If the critical switch word value is OFF, configure with the golden bitstream:
   a. Ignore the warm boot jump sequence following the critical switch word and continue reading sequentially through the flash memory addresses toward the golden bitstream.
   b. Load the bitstream from the golden bitstream area.

In summary, the QuickBoot configuration method quickly determines which bitstream to load as the first step in the configuration process via the critical switch word. The FPGA then proceeds with a standard configuration process from the selected bitstream area.

QuickBoot Flash Programming Method

The update bitstream programming procedure determines the robustness of the QuickBoot solution. These conditions are required of the flash programming procedure for reliable QuickBoot configuration:

1. The critical switch word must be turned OFF before attempting any modification of the update bitstream. Changing the critical switch word to any value other than the exact ON value turns the switch OFF.
2. The critical switch word is turned ON only after the update bitstream has been verified to have been correctly programmed. The switch can only be turned ON by setting the critical switch word to the exact, unique ON value.
3. The golden image and part two of the QuickBoot header must never be modified to ensure their integrity for all cases.

Because the QuickBoot configuration method configures from the update bitstream image only when the critical switch word is ON, requirements 1 and 2 ensure that the FPGA never executes the warm boot jump sequence to the update bitstream area when there is any possibility that the update bitstream area does not contain a correctly programmed bitstream. Requirement 3 ensures that the golden bitstream can be safely loaded whenever the critical switch word is OFF or that the warm boot jump sequence correctly directs the FPGA to the update bitstream.

QuickBoot Flash Programming Algorithm

The QuickBoot flash programming algorithm is:

1. Erase the sub-sector or block containing the critical switch word to turn OFF the switch.
2. Erase the sectors or blocks of the update image area.
3. Program the update bitstream into the update image area.
4. Verify that the update image has been programmed correctly.
5. Program the critical switch word to its exact ON value to turn the switch ON, but only after the update image has been verified.

QuickBoot Implementation Details

The QuickBoot implementation relies on these elements:

- Critical Switch Word
- Flash Memory Overview and QuickBoot Flash Memory Component Mapping
- Bitstream Image Size and Flash Memory Size Selection
- QuickBoot Configuration Time
- QuickBoot Verification of the Update Image

Critical Switch Word

The critical switch word is a special word value. The switch is considered to be ON only when the critical switch word location contains an exact, predetermined value. The existence or omission of the exact critical switch word value dictates whether or not the FPGA configuration logic executes the warm boot jump sequence.

An understanding of the FPGA bitstream and of the FPGA's built-in configuration logic is required for understanding the operation of the critical switch word and its special ON value.

The Xilinx FPGA is configured via a bitstream (`.bit` file) comprising a sequence of 32-bit words. The 32-bit words are one of these types:

- Bus width auto detection word that the FPGA uses to automatically detect the parallel configuration data bus width at the beginning of a BPI and SelectMAP mode configuration.
- Sync word that marks the beginning of the bitstream and synchronizes the configuration logic to a 32-bit boundary of the packets that follow the sync word for all configuration modes.
- Packet header word that specifies a register and data word count for writing data to a register.
- Packet command or data words for each packet. For a packet that specifies a write to the configuration command register, the word is a command such as the IPROG command. For a packet that specifies a write to WBSTAR, the data word includes the warm boot jump address.

Note: For details about the bitstream bus width auto detect words, sync word, and packets see the "Configuration Details" chapter in *7 Series FPGAs Configuration User Guide* [Ref 1].

All bitstreams generated by Xilinx design tools are composed of the following in sequential order:

1. bus width auto detect words
2. sync word
3. data packets that define the FPGA configuration

When the FPGA reads data from a BPI flash memory, the FPGA configuration logic remains in its first stage until it detects a `0xBB` data value from the first word of the bus width auto detect pattern on its D[0:7] pins. During the initial search for the bus width auto detect pattern, the FPGA effectively ignores all incoming data until it recognizes a valid bus width auto detect pattern. Thus, for BPI mode configuration, the first word of the bus width auto detect pattern `0x000000BB` is the critical switch word for the QuickBoot method. In terms of the QuickBoot critical switch word, the remaining stages of the FPGA configuration logic are not turned ON until after the bus width auto detect pattern is found.

In SPI mode, the FPGA does not use bus width auto detect logic.

When the FPGA reads data from an SPI flash memory, the FPGA configuration logic remains in its first stage monitoring the incoming data for the sync word `0xAA995566` to synchronize itself to the boundary of the 32-bit words of the incoming bitstream. During the initial search for the sync word, the FPGA effectively ignores all incoming data until it recognizes the sync word. Thus, for SPI mode configuration, the sync word is the critical switch word for the QuickBoot method. In terms of the QuickBoot critical switch word, the remaining stages of the FPGA configuration logic are not turned ON until after the sync word is found.

Table 1 summarizes the exact value that enables the critical switch word to be turned ON for each configuration mode.

Table 1: **Critical Switch Word ON Value**

Configuration Mode	BPI Mode	SPI Mode
Critical switch word (32-bit value)	0x000000BB(1)	0xAA995566

Notes:

1. Due to bit and byte swapping, the critical switch word for a BPI flash hexadecimal data (MCS) file appears as `0x0000DD00` within the MCS file.

Flash Memory Overview and QuickBoot Flash Memory Component Mapping

Implementation of the QuickBoot flash programming algorithm requires knowledge of the NOR flash memory architecture and operations. In particular, the QuickBoot flash memory components must be mapped to specific flash memory regions, and the QuickBoot algorithm procedures must be mapped to the available flash operations for those specific regions.

Flash Memory Architecture and Operations

A NOR flash memory contains a linear array of data words or data bytes. The linear array is segmented into erasable blocks or sectors. For some NOR flash memories, the sectors are further segmented into subsectors. See the appropriate NOR flash memory data sheet for the segments and their sizes [Ref 2] [Ref 3].

Reprogramming a NOR flash memory with new data values requires a two-step process:

1. An erase operation to reset all data bits of a selected segment to a `1` state.
2. A program operation to change data bits only from a `1` state to a `0` state.

The erase operation can be performed on a block, sector, or sub-sector. That is, all data words or bytes within the specified segment are erased via each erase operation. The program operation can be performed on a single word or byte.

Note: Usually for speed, each program operation programs a buffer of words or page of bytes.

General Mapping of QuickBoot Components Within Flash Memory

Because modification of the flash memory requires an erase operation and because an erase operation affects an entire flash memory segment, the flash memory segment architecture dictates the placement of the QuickBoot components within a flash memory:

1. The QuickBoot header is placed in this sequence:
   a. QuickBoot header part 1, the critical switch word, is placed in its own erasable segment.
   b. QuickBoot header part 2 follows part 1 but in a different flash segment than part 1.
2. The golden bitstream image follows the QuickBoot header in the flash memory array.
3. The update bitstream image is placed in its own erasable segments. The update image does not share any erasable segments with the golden bitstream image.

The critical switch word is placed within its own erasable segment such that the destruct process, which is an erase operation, affects only the critical switch word. Similarly, the update image is placed in its own set of segments such that the reprogramming process does not affect the golden bitstream image.

Detailed Mapping of QuickBoot Components to BPI Flash Memory

A detailed map of the BPI flash memory contents for the QuickBoot solution in a Micron P30 flash memory [Ref 2] is shown in Figure 5. The critical switch word is located at the end of the first erasable block, which is 65,536 (`0x00010000`) 16-bit words in size. The figure shows examples of a good and bad update bitstream. In a good update bitstream, the area reserved for the update bitstream contains a valid bitstream. In a bad bitstream, the area reserved for the update bitstream contains data that is in a corrupted, incomplete, or unknown condition.

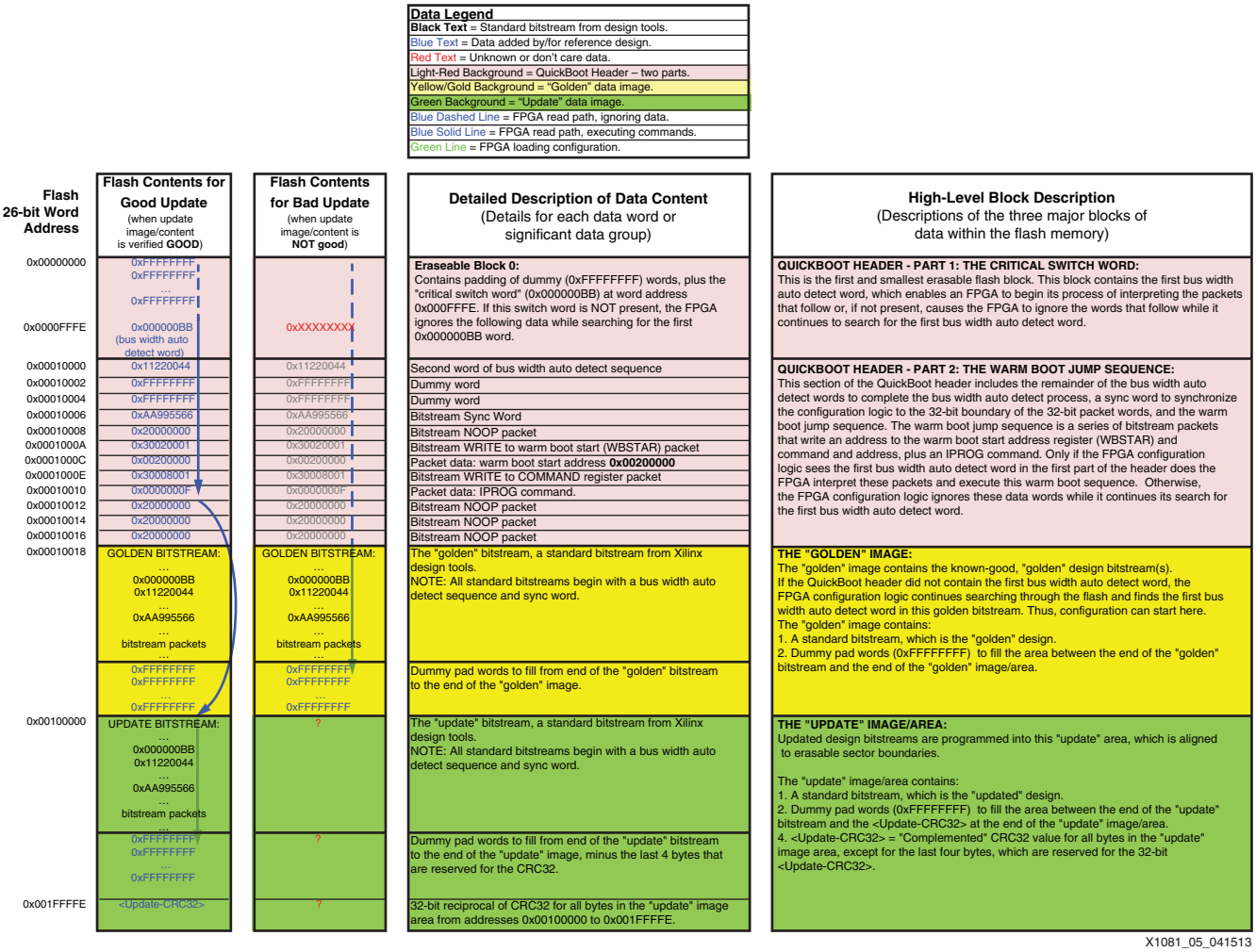

Figure 5: **QuickBoot BPI Flash Memory Map for Good and Bad Update Images**

Note: The addresses shown in Figure 5 for the update image are for a bitstream that can fit in a 16 Mb space. The update bitstream image addresses can be different for different sized FPGAs. See the reference design files for an Excel version of this information.

For the good update example, the critical switch word location contains the first bus width auto detect word `0x000000BB`. The FPGA configuration logic goes through this sequence for the good update example:

1. Begins the bus width auto detect sequence.
2. Executes the warm boot jump sequence that follows in part 2 of the QuickBoot header.
3. Concludes by loading the update bitstream.

For the bad update example, the critical switch word location does not contain the first bus width auto detect word. Therefore, the FPGA configuration logic goes through this sequence for the bad update example:

1. Continues to search for the bus width auto detect words, ignoring part 2 of the QuickBoot header.
2. Finds the standard bus width auto detect words that begin the golden bitstream.
3. Concludes by loading the golden bitstream.

The update bitstream is located within its own set of erasable blocks. The start of the update bitstream is aligned to the beginning of its first erasable block at word address `0x00100000`.

Note: The addresses of the update bitstream image for a specific implementation can be different from that shown in Figure 5 because the update image depends on the bitstream size.

Detailed Mapping of QuickBoot Components to SPI Flash Memory

The mapping of the QuickBoot components to SPI flash memory is similar to the mapping to BPI flash memory, except for a difference in the critical switch word. In SPI mode, the FPGA configuration logic skips its bus width auto detect stage and instead first begins searching for the sync word. Therefore, the bus width auto detect words are omitted from the QuickBoot header, and the QuickBoot header for SPI flash memory begins with the sync word as the critical switch word.

Figure 6 shows a detailed map of the SPI flash memory contents for the QuickBoot solution in a Micron N25Q flash memory [Ref 2]. The critical switch word is located at the end of the first erasable sub-sector, which is 4,096 (0x00001000) 8-bit bytes in size.

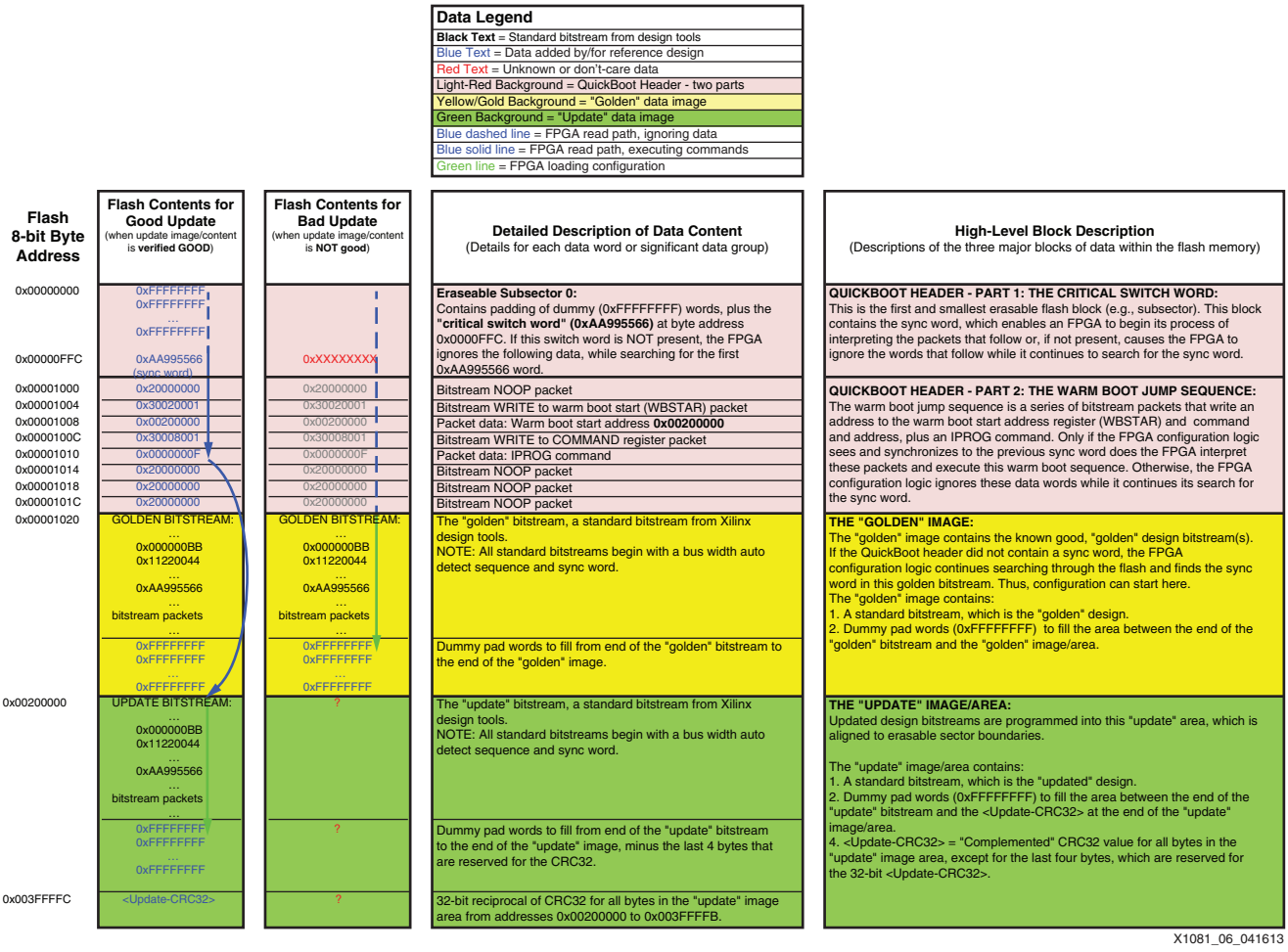

Figure 6: **QuickBoot SPI Flash Memory Map for Good and Bad Update Images**

Note: The addresses shown in Figure 6 for the update image are for a bitstream that can fit in a 16 Mb space. The update bitstream image addresses can be different for different sized FPGAs. See the reference design files for an Excel version of this information.

Figure 6 shows examples of a good and bad update bitstream. For the good update example, the critical switch word location contains the correct sync word `0xAA995566`. Therefore, the FPGA configuration logic goes through this sequence for the good update example:

1. Synchronizes at the critical switch word location.
2. Executes the warm boot jump sequence that follows in part 2 of the QuickBoot header.
3. Concludes by loading the update bitstream.

For the bad update example, the critical switch word location does not contain a valid sync word. Therefore, the FPGA configuration logic goes through this sequence for the bad update example:

1. Continues to search for the sync word, ignoring part 2 of the QuickBoot header.
2. Finds the standard sync word that begins the golden bitstream.
3. Concludes by loading the golden bitstream.

The update bitstream is located within its own set of erasable blocks. The start of the update bitstream is aligned to the beginning of its first erasable block at byte address `0x00200000`.

Note: The addresses of the update bitstream image for a specific implementation can be different from those shown in Figure 6 because the image update depends on the bitstream size.

Bitstream Image Size and Flash Memory Size Selection

In general, an estimate of the flash memory size is twice the size of the FPGA bitstream length after it is rounded up to the next power-of-2 number of megabits. These steps are required to obtain an estimate for the flash memory size in bits:

1. Find the FPGA bitstream length from the “Bitstream Length” table in *7 Series FPGAs Configuration User Guide* [Ref 1]. For example, the 7K70T bitstream length is approximately 23 Mb.
2. Compute an estimate for the image size by rounding the bitstream length up to the next power-of-2 number of megabits. For example, 23 Mb is rounded up to 32 Mb, which is 2^{25} bits.
3. Compute an estimate for the minimum flash size by multiplying by 2 the estimated image size from step 2 to accommodate a golden image and update image. For example, 32 Mb x 2 = 64 Mb.

The actual minimum image size is the sum of the following rounded up to the erase block size. The actual minimum flash size is the actual minimum image size multiplied by 2:

1. Sum these elements:
   a. First erasable segment size, which is the size of the erasable segment containing the critical switch word.
   b. QuickBoot header part 2 length.
   c. FPGA bitstream length.
2. Round the sum from step 1 up to the general erase segment size to obtain the minimum image size.
3. Multiply the result of step 2 by 2.
4. Convert the result of step 3 from bits to megabits to obtain the minimum flash size.

Table 2 gives a worksheet for computing the minimum image size and minimum flash size. KC705 Board Demonstrations, page 33 provides details on the BPI and SPI flash devices, respectively.

Table 2: **Worksheet for Minimum Image Size and Minimum Flash Size Computation**

Step/ Line	Instructions	Data Entry	BPI Flash and 7K70T FPGA	SPI Flash and 7K70T FPGA	Unit
1	QuickBoot header, part 1 size: Get the size of the first erasable segment for: • BPI flash: See BPI flash data sheet [Ref 2] for erase block size. • SPI flash: See SPI flash data sheet [Ref 3] for erase sub-sector size.		1,048,576	32,768	bit
2	QuickBoot header, part 2 length: • BPI mode: 384 bits. • SPI mode: 256 bits.		384	256	bit
3	Get the uncompressed bitstream length. See bitstream length table in *7 Series FPGAs Configuration User Guide* [Ref 1].		24,090,592	24,090,592	bit
4	Compute the size of the QuickBoot header and golden bitstream: = [Line 1] + [Line 2] + [Line 3]. (1)		25,139,552	24,123,616	bit
5	Get the size of a general erasable segment: • BPI flash: See BPI flash data sheet [Ref 2] for erase block size. • SPI flash: See SPI flash data sheet [Ref 3] for erase sector size.		1,048,576	524,288	bit
6	Use the block/sector size from line 5 to compute the number of blocks/sectors needed to fit the number of bits from line 4: = int(([Line 4] + [Line 5] – 1) / [Line 5]).		24	47	block
7	Compute the size of the needed blocks for line 5: = [Line 6] * [Line 5].		25,165,824	24,641,536	bit
8	Convert line 7 to Mb for minimum image size: = int(([Line 7] + 1,048,575) / 1,048,576).		24	24	Mb
9	Multiply the image size by 2 to fit two images, golden and update: = [Line 7] * 2.		50,331,648	49,283,072	bit
10	Convert line 9 to Mb for minimum flash size: = int(([Line 9] + 1,048,575) / 1,048,576).		48	48	Mb

Notes:

1. Additional bits can be added to line 4 to assure minimum separation between bitstreams. See Bitstream Separation for MMCM/PLL Lock or DCI Wait for details.

The flash memory must be at least as large as line 10 in Table 2. Each bitstream image size must be at least as large as line 8 in Table 2.

Bitstream Separation for MMCM/PLL Lock or DCI Wait

In the typical FPGA configuration procedure, the FPGA begins the startup procedure for its loaded design near the end of the bitstream loading process and stops reading data from flash memory before reaching the end of the bitstream. However, in some cases the FPGA startup procedure is extended. In these cases, the FPGA can continue to read data beyond the end of the bitstream in flash memory. For such cases, the next (i.e., update) bitstream in flash memory should be separated from the prior (i.e., golden) bitstream to avoid having the FPGA load the next bitstream while the FPGA waits to complete the startup procedure for the prior bitstream.

The extended startup wait can occur when the FPGA design has one or more of these design elements and bitstream settings:

- SelectIO™ interface with digitally-controlled impedance (DCI) and the BitGen -g Match_cycle is not set to NoWait.
- Mixed-mode clock manager (MMCM) has its STARTUP_WAIT attribute set to TRUE and the BitGen -g LCK_cycle is not set to NoWait.
- Phase-locked loop (PLL) has its STARTUP_WAIT attribute set to TRUE and the BitGen -g LCK_cycle is not set to NoWait.

For FPGA designs with the above condition(s), the bitstreams can be separated by increasing the image size reserved for the bitstream. Because the bitstream is placed near the beginning of the reserved image space, the increased reserved image size effectively adds empty space following the bitstream in the reserved image space. The minimum number of additional bits to add to the minimum bitstream image size can be computed using Equation 1.

$$\text{Additional bits} = (\text{configuration data width in bits}) \times (\text{configuration CCLK frequency}) \times (\text{wait time}) \quad \textit{Equation 1}$$

For the wait time in Equation 1, the longest of these applicable conditions should be used:

- For the SelectIO interface DCI, the wait time can be approximately 10 ms.
- For the MMCM maximum lock time, see MMCM_TLOCKMAX in the FPGA data sheet.
- For the PLL maximum lock time, see PLL_TLOCKMAX in the FPGA data sheet.

An example separation for a design using SelectIO interface DCI wait and BPI x16 configuration mode at 3 MHz is:

Additional bits = 480,000 bits = 16 bits/cycle $\times$ 3 MHz $\times$ 10 ms

An example separation for a design using SelectIO interface DCI wait and SPI x1 configuration mode at 20 MHz is:

Additional bits = 200,000 bits = 1 bit/cycle $\times$ 20 MHz $\times$ 10 ms

When the FPGA design has a condition that can cause the FPGA to extend its startup procedure, the user should compute the minimum additional bits for separating the bitstreams using Equation 1 and add the number of computed additional bits to line 4 in Table 2.

QuickBoot Configuration Time

The QuickBoot solution completes configuration in the same time as a standard FPGA configuration plus a small constant. The QuickBoot header incurs a small constant overhead at the beginning of the configuration process for all cases when the critical switch word determines whether to load either the update bitstream or golden bitstream. The initial constant is the time for the FPGA configuration logic to read through the first erasable segment to reach the critical switch word. If the update bitstream is selected, an additional small constant of time is taken for the warm boot jump program reset T_{PL}. For all configuration cases, the QuickBoot header overhead is no greater than the time interval required to read through the first erasable segment plus the warm boot jump program reset time.

Table 3 provides a worksheet for QuickBoot configuration time.

Table 3: **Worksheet for QuickBoot Configuration Time**

Line	Description	Data Entry	BPI x16 Example	SPI x1 Example	Unit
1	First erasable segment size.		65,536	32,768	BPI: Words SPI: bits
2	F_{MCCK_START} specification from FPGA data sheet.		3	3	MHz, Typ
3	Time to read through first erasable segment: = [Line 1] / [Line 2] / 1,000.		22	11	ms, Typ
4	Time for warm boot jump via IPROG. See FPGA data sheet for T_{PL} specification.		5	5	ms, Max
5	Time for QuickBoot header overhead: = [Line 3] + [Line 4].		27	16	ms, Typ
6	Bitstream length from *7 Series FPGAs Configuration User Guide* [Ref 1].		24,090,592	24,090,592	bit
7	F_{MCCK} bitstream-defined CCLK frequency.		6	26	MHz, Typ
8	Time to load standard bitstream: BPI mode: [Line 6] / (16 bits/word) / [Line 7] / 1,000 SPI mode: [Line 6] / [Line 7] / 1,000.		251	927	ms, Typ
9	Time to load QuickBoot bitstream (all cases) = [Line 5] + [Line 8].		278	943	ms, Typ

Line 8 in Table 3 shows the standard bitstream configuration time. Line 9 shows the QuickBoot configuration time, which is only different from the standard configuration time by time shown in line 5.

QuickBoot Verification of the Update Image

The robustness of the QuickBoot solution depends on the ability of the QuickBoot programming solution to reliably detect correct or incorrect programming of the update bitstream image. To verify the correctness of the update bitstream image in flash memory, the QuickBoot solution applies a cyclic redundancy check (CRC) method to verify the integrity of the update bitstream image.

QuickBoot Image Format for Cyclic Redundancy Check

The QuickBoot solution formats the update bitstream image in this sequence:

1. Standard FPGA bitstream from Xilinx design tools.
2. Dummy `0xFFFFFFFF` 32-bit pad words that fill the remainder of the reserved update area.
3. Complement of the CRC32 computed on data from step 1 and step 2.

A fixed region within the flash memory is allocated for the update bitstream image. For erase and programming convenience, the update region begins and ends at the boundaries of erasable segments that have been allocated for the update image.

The update bitstream begins at the start of the update region. After the end of the bitstream, dummy words fill the remainder of the update region, except for the region's last 32 bits. When the update image is generated as an MCS file, a CRC32 is computed for the entire contents of the update region, except for the last 32 bits. The complement of the computed CRC32 is placed in the last 32 bits of the update region. Thus, the integrity of the entire update region is protected by a CRC32 check code.

To verify the update image, the QuickBoot programming solution reads the entire update image/area, including the complement CRC32 value, and computes the CRC32. The resulting CRC32 value of the data and its complemented CRC32 value is expected to match a known

constant, called the residue, when the integrity of the update image is good. See *IEEE 802.3 Cyclic Redundancy Check* [Ref 4] for a detailed description of the CRC32, embedded complement of the CRC32, and constant residue.

QuickBoot Reference Designs

This application note is accompanied by two QuickBoot reference designs: one for SPI flash and the other for BPI flash. Each reference design consists of:

- Perl script for generating the QuickBoot flash memory images for configuration bitstreams
- Flash programmer module for updating the bitstream in the flash device

A specific Perl script and flash programmer module is provided for each configuration mode, SPI or BPI. Both the SPI and BPI flash programmer modules have a consistent interface and usage model.

Additional designs that use the flash programmer reference design modules are provided to demonstrate the reference design on the KC705 evaluation board using either the SPI flash or BPI flash.

QuickBoot Reference Design Implementation Guide

These steps describe how to implement the QuickBoot reference design:

1. QuickBoot hardware requirements:
   a. Select an FPGA configuration mode: BPI or SPI. The SPI mode is the lowest pin-count solution. The BPI mode supports the highest density flash devices. See *7 Series FPGAs Configuration User Guide* [Ref 1] for additional considerations.
   b. See Bitstream Image Size and Flash Memory Size Selection, page 10 to calculate the minimum required image size and flash size. Save the image size for the QuickBoot file generation step.
   c. Select a flash memory type and device based on the selected configuration mode and minimum required flash size (see Table 4).
   d. Check that a selected flash device is supported via the iMPACT tools indirect flash programming support list. See the iMPACT help page [Ref 5].
   e. See QuickBoot Reference Design – Hardware Requirements, page 15 for board connection requirements between the FPGA and flash device.
2. QuickBoot preliminary file generation for obtaining QuickBoot flash programmer parameters:
   a. Generate a preliminary test bitstream for the FPGA.
   b. Follow the instructions in Bitstream Generation and Formatting, page 16 to generate a standard MCS file for the test bitstream.
   c. Follow the instructions to generate the QuickBoot MCS image files and save the output report from the Perl script that generates the MCS files:
      - QuickBoot Flash Image File Generation for BPI Mode, page 17, or
      - QuickBoot Flash Image File Generation for SPI Mode, page 17
3. Instantiate the QuickBoot flash programmer module in every FPGA design:
   a. Instantiate the QuickBoot flash programmer modules corresponding to the selected flash memory type. Follow the instructions to connect the instantiated module to the FPGA pins that access the attached flash memory device, and configure the module per the report from the QuickBoot file generation step:
      - Connect the BPI flash programmer module as described in BPI Flash Programmer Reference Design Module, page 18 and configure the module as described in Configuring the `BpiFlashProgrammer.vhd` Parameters, page 20,

or

- Connect the SPI flash programmer module as described in SPI Flash Programmer Reference Design Modules, page 24 and configure the module as described in Configuring the `SpiFlashProgrammer.vhd` Parameters, page 27.

Note: Any alternate programmer implementation can be used as long as the programmer follows the required algorithm described in QuickBoot Flash Programming Algorithm, page 5.

4. Implement system interface to the QuickBoot flash programmer module according to QuickBoot Flash Programming Method, page 5.
5. Repeat step 2 using the FPGA design that has the integrated QuickBoot flash programmer. Depending on whether the FPGA design is for the golden/initial image or update image, go to step 6 or step 7.
6. For the golden bitstream image or for the initial FPGA implementation of the QuickBoot flash programmer, use the iMPACT tool or other convenient programming solution to program the flash with the `*_initial.mcs` file generated in step 5.

   Note: For the initial FPGA implementation of the QuickBoot flash programmer, check, test, and debug the QuickBoot flash programmer implementation using the tips from QuickBoot Flash Programmer Reference Design Checklist, Test, and Debug, page 32.
7. For the update bitstream image, send the `*_update.mcs` file generated in step 5 to the QuickBoot flash programmer that is integrated in the FPGA design.

QuickBoot Reference Design – Hardware Requirements

Table 4 gives the NOR flash memory devices supporting the QuickBoot reference design for 7 Series FPGAs on the KC705 evaluation board.

Table 4: **QuickBoot Supported FPGAs and Flash Memory Devices**

FPGA Series/Family	BPI x16 Flash Memories	SPI Flash Memories
7 Series FPGAs	Micron P30 NOR Flash	Micron N25Q NOR Flash

The designer should use the standard board-level connections between the FPGA and flash memory that are recommended for the chosen configuration mode in either the “Master BPI Configuration Interface” section or “Master SPI Configuration Mode” section in *7 Series FPGAs Configuration User Guide* [Ref 1].

Flash Reset Recommendation

When available, a flash device should be chosen with an active-Low reset pin. Most BPI flash devices have an active-Low reset pin. Some SPI flash devices are available with an active-Low reset pin. When an active-Low reset pin is available on a flash device, the flash active-Low reset pin should be connected to the FPGA INIT_B signal.

The FPGA INIT_B pin momentarily drives Low when the FPGA is reset via its PROGRAM_B pin. This ensures that when the FPGA is reset, the flash device is also reset and ready to output a configuration bitstream. Otherwise, if an FPGA reset occurs when the flash is busy with an internal erase or program operation, the flash is not reset from the erase or program operation and might not output configuration data upon the subsequent read request from the FPGA, which can result in FPGA configuration failure.

QuickBoot File Generation

The QuickBoot solution does not require any special options for the FPGA bitstreams. However, the QuickBoot solution requires a special QuickBoot header and specific image locations to be reserved within the flash memory map. See Figure 4 or Figure 5 for a detailed

memory map. Perl scripts are provided with the reference design that can generate the flash image (MCS) files for the QuickBoot solution.

Bitstream Generation and Formatting

The QuickBoot images begin with an FPGA bitstream. A standard bitstream should be generated for the chosen configuration mode. No special MultiBoot or Fallback configuration options are needed.

The bitstream(s) for the FPGA(s) to be configured must be converted to the MCS format.

For BPI mode, use this PROMGen command:

```
PROMGen -w -p mcs -data_width 16 -u 0 FPGA_design.bit
```

For SPI mode, use this PROMGen command:

```
PROMGen -w -p mcs -spi -u 0 FPGA_design.bit
```

Note: If multiple FPGAs are configured via a `BPIx16-to-SelectMAPx16` daisy-chain or via an `SPIx1-to-serial` daisy-chain, all `FPGA.bit` files must be listed in order on the PROMGen command line where `FPGA_design.bit` is shown above.

This is an example report from PROMGen:

```
>promgen -w -p mcs -spi -u 0 FPGA_design.bit
Release 14.4 - Promgen P.49d (nt64)
Copyright (c) 1995-2012 Xilinx, Inc. All rights reserved.
0x148da8 (1346984) bytes loaded up from 0x0
Using generated prom size of 2048K
Writing file "FPGA_design.mcs".
Writing file "FPGA_design.prm".
Writing file "FPGA_design.cfi".
```

PROMGen writes the `FPGA_design.mcs`, which is the input to the QuickBoot Perl script.

The example report shows the next power-of-2 PROM size that fits the bitstream is 2,048 KB. This power-of-2 PROM size, when converted to megabits, is typically a convenient size to which to set the QuickBoot image.

The example report also shows that the bitstream is 1,346,984 bytes long. This size, when converted to bits, can be used in the Table 2 worksheet for computing the minimum QuickBoot image size and minimum flash size.

QuickBoot Flash Image File Generation

A Perl script is provided with the reference design that takes the MCS file, generated by PROMGen, as input and generates the QuickBoot images as output. The Perl script always generates two kinds of images:

- Initial MCS flash memory image
- Updated MCS bitstream image

Depending on the stage of system programming, whether initial system configuration or remote update, one of the two images is used.

Initial MCS Flash Memory Image

The initial MCS flash memory image is the complete initial image for the flash memory that contains:

- QuickBoot header
- Golden bitstream, which is a copy of the given bitstream
- Update bitstream, which is a second copy of the given bitstream

The initial MCS is used to initially program the flash memory.

Update MCS Bitstream Image

The update MCS bitstream image is the data for remote update of the update image. The update MCS bitstream image contains the given bitstream, and its integrity is protected via a CRC32 code as described in QuickBoot Image Format for Cyclic Redundancy Check, page 13.

The update MCS bitstream image is sent to the QuickBoot flash programmer in the remote system for programming into the update bitstream area of the flash memory.

QuickBoot Flash Image File Generation for BPI Mode

For BPI mode, this is the Perl script command to generate the QuickBoot images:

```
xilPerl MakeBpiFlashProgrammerMcsFiles.pl FPGA_design.mcs
```

The Perl script follows this sequence:

1. Reads the given MCS file.
2. Computes by default the minimum image size for the QuickBoot golden image and update image.
3. Reports information about the BPI flash and update image location in flash memory.
4. Generates an initial MCS flash memory image file.
5. Updates MCS flash memory image file.

Alternatively, the image size can be set through the `imagesize IS` option, where `IS` is the image size in megabits. See this example:

```
xilPerl MakeBpiFlashProgrammerMcsFiles.pl –imagesize 16 FPGA_design.mcs
```

This is an example of the report from the `MakeBpiFlashProgrammerMcsFiles.pl` script:

```
>xilPerl MakeBpiFlashProgrammerMcsFiles.pl –imagesize 16 FPGA_design.mcs
INFO:  ##### Reading MCS file:  FPGA_design.mcs...
INFO:  BPI device type             = P30
INFO:  BPI flash (minimum) size    = 32 Mbits
INFO:  BPI block size              = 0x00010000 (65536) words
INFO:  Update image start address  = 0x00100000
INFO:  Update image end+1 address  = 0x00200000
INFO:  Update switch word address  = 0x0000FFFE
INFO:  Switch word                 = 0x000000BB
INFO:  Generating files...This can take several minutes....
INFO:  Output Update MCS file name = FPGA_design_update.mcs
INFO:  Output Initial MCS file name= FPGA_design_initial.mcs
```

QuickBoot Flash Image File Generation for SPI Mode

For SPI mode, this is the Perl script command to generate the QuickBoot images:

```
xilPerl MakeSpiFlashProgrammerMcsFiles.pl FPGA_design.mcs
```

The Perl script follows this sequence:

1. Reads the given MCS file.
2. Computes by default the minimum image size for the QuickBoot golden image and update image.
3. Reports information about the SPI flash and update image location in flash memory.
4. Generates an initial MCS flash memory image file.
5. Updates MCS flash memory image file.

Alternatively, the image size can be set through the `imagesize IS` option, where `IS` is the image size in megabits. For example:

```
xilPerl MakeSpiFlashProgrammerMcsFiles.pl –imagesize 16 FPGA_design.mcs
```

The output report from the Perl script lists basic parameter settings for the QuickBoot flash programmer. The Perl script report should be saved and the report values used to set the SPI flash programmer module settings as shown in Configuring the SpiFlashProgrammer.vhd Parameters, page 27.

This is an example of the report from the MakeSpiFlashProgrammerMcsFiles.pl script:

```
>xilPerl MakeSpiFlashProgrammerMcsFiles.pl -imagesize 16 FPGA_design.mcs
INFO:  ##### Reading MCS file:  FPGA_design.mcs...
INFO:  SPI Device Type             = N25Q
INFO:  SPI flash (minimum) size    = 32 Mbits
INFO:  SPI address width           = 24 bits
INFO:  SPI sector size             = 00010000 (65536) bytes
INFO:  SPI page size               = 00000100 (256) bytes
INFO:  Update image start address  = 0x00200000
INFO:  Update image end+1 address  = 0x00400000
INFO:  Update switch word address  = 0x00000FFC
INFO:  Switch word                 = 0xAA995566
INFO:  Generating files...This can take several minutes....
INFO:  Output Update MCS file name = FPGA_design_update.mcs
INFO:  Output Initial MCS file name= FPGA_design_initial.mcs
```

Option for Faster File Generation

The QuickBoot Perl scripts are available in the reference design package with its own CRC32 calculation subroutine. The calculation method is a direct copy of the VHDL CRC32 implementation and is not optimized for Perl.

A faster alternate CRC32 Perl subroutine is available as the standard Digest::CRC Perl package. A copy of the Perl package can be found on the Internet via a search using the key words “perl Digest CRC.”

If the Digest::CRC package is included with the Perl installation, then the commenting in the top lines of the MakeSpiFlashProgrammerMcsFiles.pl script can be changed to use the CRC32 function of the Digest::CRC package:

```
# Comment/uncomment one of the pairs of lines below,
# depending on whether you have the Digest::CRC
# Use the following two lines if you have the Digest::CRC
use Digest::CRC qw(crc32);
sub subCrc32{my $data=shift(@_);return(subFastCrc32($data));}
# Use the following two lines if you do NOT have the Digest::CRC
#sub crc32{return(0);}
#sub subCrc32{my $data=shift(@_);return(subSlowCrc32($data));}
```

BPI Flash Programmer Reference Design Module

The base reference design consists of one module called BpiFlashProgrammer.vhd.

BpiFlashProgrammer.vhd Module

BpiFlashProgrammer.vhd is the programming state machine that implements the QuickBoot programming algorithm for BPI flash. The BpiFlashProgrammer interfaces directly with the BPI flash.

Table 5 describes the BpiFlashProgrammer ports.

Table 5: **BpiFlashProgrammer Ports Description**

Port	Direction	Width	Description and Connection
inClk	Input	1	System clock input. See BPI Flash Programmer Clock Frequency, page 22 for the clock frequency recommendation.
inReset_EnableB	Input	1	Synchronous active-High reset and active-Low enable. When High, this module is held in reset. When Low, this module is activated to start the QuickBoot update image reprogramming procedure.
inCheckIdOnly	Input	1	Active-High check ID only signal. Debug input control signal. Set to 0 for normal operation. inCheckIdOnly is registered on the first rising edge of inClk when inReset_EnableB is Low. When registered as High, this input causes this module to stop after the check ID procedure.
inVerifyOnly	Input	1	Active-High verify only signal. Debug/advanced input control signal. Set to 0 for standard operation. inVerifyOnly is registered on the first rising edge of inClk when inReset_EnableB is Low. When registered as High, this input causes this module to skip erase/programming procedures and to perform only the verify procedure. See QuickBoot Verification of the Update Image, page 13 for a description of the verification procedure. This can be used for advanced applications.
inData32	Input	32	32-bit wide input data bus. The module can be configured via cDataWordWidth to accept data words on this bus that are 16 bits wide via inData32[15:0] or 32 bits wide via inData32[31:0]. By default, the module takes 32-bit words. See the inDataWriteEnable and outReady_BusyB ports for when input data is taken.
inDataWriteEnable	Input	1	Active-High data write enable signal. When outReady_BusyB is High and inDataWriteEnable is High, a data word is captured from the inData32 bus on the rising edge of inClk.
outReady_BusyB	Output	1	Active-High ready signal or active-Low busy signal. The module drives this signal High when it is ready to accept the next data word. Data can be accepted when the clock edge that immediately follows outReady_BusyB goes High.
outDone	Output	1	Active-High done signal. The module drives outDone High when programming is done or an error is detected. When outDone is High, outError should be checked for an error condition.
outError	Output	1	Active-High error signal. The module drives outError High when *any* error is detected. Additional details about the kind of error is provided via the signals below.
outErrorIdcode	Output	1	Active-High check ID error signal. Optional/debug signal connection: provides additional detail for when outError is High.
outErrorErase	Output	1	Active-High erase error signal. Optional/debug signal connection: provides additional detail for when outError is High.
outErrorProgram	Output	1	Active-High program error signal. Optional/debug signal connection: provides additional detail for when outError is High.
outErrorTimeOut	Output	1	Active-High erase or program timeout error signal. Optional/debug signal connection: provides additional detail for when outError is High. See module's cTimeOut* constants for timeout periods.
outErrorBlockLocked	Output	1	Active-High block locked error signal. Optional/debug signal connection: provides additional detail for when outError is High.
outErrorVpp	Output	1	Active-High VPP error signal. Optional/debug signal connection: provides additional detail for when outError is High.
outErrorCmdSequence	Output	1	Active-High command sequence error signal. Optional/debug signal connection: provides additional detail for when outError is High.
outStarted	Output	1	Active-High module started procedure signal. Optional/debug signal connection: provides additional detail on programmer progress.
outInitializeOK	Output	1	Active-High module completed initialization stage signal. Optional/debug signal connection: provides additional detail on programmer progress.
outCheckIdOK	Output	1	Active-High module successfully completed check ID stage signal. Optional/debug signal connection: provides additional detail on programmer progress.

Table 5: **BpiFlashProgrammer Ports Description *(Cont'd)***

Port	Direction	Width	Description and Connection
outEraseSwitchWordOK	Output	1	Active-High module successfully completed erase switch word stage signal. Optional/debug signal connection: provides additional detail on programmer progress.
outEraseOK	Output	1	Active-High module successfully completed update image erase stage signal. Optional/debug signal connection: provides additional detail on programmer progress.
outProgramOK	Output	1	Active-High module successfully completed update image program stage signal. Optional/debug signal connection: provides additional detail on programmer progress.
outVerifyOK	Output	1	Active-High module successfully completed verify stage signal. Optional/debug signal connection: provides additional detail on programmer progress.
outProgramSwitchWordOK	Output	1	Active-High module successfully completed switch word programming stage signal. Optional/debug signal connection: provides additional detail on programmer progress.
outAddress32	Output	32	Connect to BPI flash address[N:0] via the FPGA BPI address, A[N:00], pins. Connect as many of the address pins that are available on the BPI flash. The extra MSB addresses from this module can be left unconnected.
inoutData16	Output	16	Connect to BPI flash data[15:0] bus via the FPGA BPI data D[15:00] pins.
outFCSB	Output	1	Connect to BPI flash chip select signal via the FPGA FCS_B pin.

Configuring the `BpiFlashProgrammer.vhd` Parameters

`BpiFlashProgrammer.vhd` contains hard-coded addresses and commands to ensure that an implementation can only erase or program specific locations in the BPI flash memory array. The following section from the `BpiFlashProgrammer.vhd` module highlights the basic parameters that must be set for the specific system in which the module is implemented.

```
--##############################################################################
  -- BEGIN BASIC PARAMETER SETTINGS:
  -- THE FOLLOWING MUST MATCH THE OUTPUT FROM MakeBpiFlashProgrammerMcsFiles.pl
  -- BPI DEVICE TYPE: Only one of the following must be set to '1'. Other is '0'.
  constant  cFlashMicronP30   : std_logic := '1';
  constant  cFlashMicronG18   : std_logic := '0'; -- Not tested
  -- BPI BLOCK SIZE: Ensure setting is correct for the chosen device type.
  constant  cSizeBlockP30     : std_logic_vector(31 downto 0) := X"00010000"; -- For P30
  constant  cSizeBlockG18     : std_logic_vector(31 downto 0) := X"00020000"; -- For G18
  -- UPDATE IMAGE START ADDRESS
  constant  cAddrUpdateStart  : std_logic_vector(31 downto 0) := X"00100000";
  -- UPDATE IMAGE END+1 ADDRESS
  constant  cAddrUpdateEnd    : std_logic_vector(31 downto 0) := X"00200000";
  -- SWITCH WORD ADDRESS: Ensure setting is correct for the chosen device type
  constant  cAddrSwitchWordP30: std_logic_vector(31 downto 0) := X"0000FFFE"; -- For P30
  constant  cAddrSwitchWordG18: std_logic_vector(31 downto 0) := X"0001FFFE"; -- For G18
  -- SWITCH WORD
  constant  cSwitchWord       : std_logic_vector(31 downto 0) := X"000000BB";
  ------------------------------------------------------------------------------
  -- DATA WORD WIDTH (IN BYTES) FOR INPUT APPLIED TO THE MODULE'S 32-BIT inData32 BUS
  constant  cDataWordWidth    : integer := 4;  -- 4 = 32-bit word --> inData32[31:0]
  --                                           -- 2 = 16-bit word --> inData32[15:0]
  -- END BASIC PARAMETER SETTINGS
  --##############################################################################
```

Table 6 contains a description of the parameters from the BpiFlashProgrammer.vhd module.

Table 6: **BpiFlashProgrammer.vhd Parameters**

Parameter	Value	Output Report Line[1]	Description
cFlashMicronP30 cFlashMicronG18	1 or 0	BPI device type	BPI device type. These constants define the target flash type to be programmed. The constant that is set to 1 is the type of flash to be programmed. Only the supported cFlashMicronP30 constant should be set to 1. The rest are reserved and must be set to 0. The user should ensure this setting matches the BPI Device Type from the output report from the MakeBpiFlashProgrammerMcsFiles.pl Perl script.
cSizeBlockP30 cSizeBlockG18	Xhhhhhhhh (32-bit hex)	BPI block size	BPI block size (in words). This constant defines the general erasable block size of the BPI flash type. Typically, the preset values are correct and need NO adjustment. Check that the switch word address is correct for the target BPI flash and matches the BPI block size output report from the MakeBpiFlashProgrammerMcsFiles.pl script.
cAddrUpdateStart	Xhhhhhhhh (32-bit hex)	Update image start address	Update image start (word) address. Set to the starting word address for update image. The user should ensure this setting matches the Update image start address from the output report from the MakeBpiFlashProgrammerMcsFiles.pl Perl script.
cAddrUpdateEnd	Xhhhhhhhh (32-bit hex)	Update image end+1 address	Update image end+1 (word) address. Set to one more than the last word of the last block in the update image area. This is used as the ending value for the block or buffer address counters. The user should ensure this setting matches the Update image end+1 address from the output report from the MakeBpiFlashProgrammerMcsFiles.pl Perl script.
cAddrSwitchWordP30 cAddrSwitchWordG18	Xhhhhhhhh (32-bit hex)	Update switch word address	Switch word (word) address. These constants define the switch word first word address for each BPI flash type. Typically, the preset values are correct and need NO adjustment. The user should check that the switch word address is correct for the target BPI flash and matches the output report from the MakeBpiFlashProgrammerMcsFiles.pl script.
cSwitchWord	X000000BB (32-bit hex)	N/A	Critical switch word On value. Do NOT change.
cDataWordWidth	4 or 2	N/A	Input data word width (in bytes). This constant defines the width of the data presented on the inData32 bus. It should be set to 4 bytes for 32-bit data on inData32[31:0]. It should be set to 2 bytes for 16-bit data on inData32[15:0] only. inData32[31:16] are unused.

Notes:

1. This is the MakeBpiFlashProgrammerMcsFiles.pl Perl script output report.

BpiFlashProgrammer.vhd also contains timeout limits for the flash erase and program commands. In general, the default timeout limits should work for most implementations. The following BpiFlashProgrammer.vhd code segment highlights the timeout limits.

```
-- Time out = Max time * inClk frequency Hz / 2 (cycles/polling iteration)
  constant  cClkFrequencyHz    : integer                       := 15000000;    -- Not used; Keep for reference
  constant  cClkCyclesPerTOLoop:integer                        := 2;           -- Not used; Keep for reference
  constant  cTimeOutErase      : std_logic_vector(31 downto 0) := X"02000000"; -- ~4 s*15,000,000/2
  constant  cTimeOutProgram    : std_logic_vector(31 downto 0) := X"00009000"; -- ~5ms*15,000,000/2
```

See Table 7 for a description of these parameters. Table 7 can be used to cross check the default timeout parameters against the maximum erase and program time specifications in the target flash data sheet and at the clock frequency used in the implementation.

Table 7: **BpiFlashProgrammer Timeout Limits**

Parameter	Default Value	Description
cClkFrequencyHz	15000000	The maximum expected inClk frequency. Default setting is 15 MHz.
cClkCyclesPerTOLoop	2	Number of inClk cycles per timeout loop.
cTimeOutErase	X02000000	Block erase – Maximum timeout loop count. Value = Max t_{ERS} time * cClkFrequencyHz / cClkCyclesPerTOLoop. Default X02000000 = 4 seconds * 15,000,000 Hz / 2. See flash data sheet for maximum t_{ERS} time.
cTimeOutProgram	X00009000	Buffer program – Maximum timeout loop count. Value = Max t_{PROG} time * cClkFrequencyHz / cClkCyclesPerTOLoop. Default X00009000 = 0.005 seconds * 15,000,000 Hz / 2. See flash data sheet for max t_{PROG} time.

BPI Flash Programmer Clock Frequency

The limiting factor for the BPI flash programmer is the flash read cycle time. The timing for each read cycle consists of the FPGA clock-to-address valid time, flash address valid to data out time, and FPGA data to clock setup time. The dominant time in the read cycle is the flash address valid to data out time, which can be approximately 120 ns. The BPI flash programmer uses two inClk cycles for each flash read cycle. Thus, the inClk period must be greater than 60 ns. In terms of frequency, the inClk frequency must be 15 MHz or slower. See Table 8 for example QuickBoot update times for different clock frequencies showing both typical and worst-case times.

Table 8: **Example QuickBoot Update Times for BPI Flash**

Description	Source/Computation	16 Mb Update @5 MHz (Typical)	16 Mb Update @15 MHz (Typical)	16 Mb Update @30 MHz (Typical)	16 Mb Update @5 MHz (Worst-Case)	16 Mb Update @15 MHz (Worst-Case)	16 Mb Update @30 MHz (Worst-Case)	Unit
Example update image size	From user design	16	16	16	16	16	16	Mb
Clock frequency	From user design	5	15	30	5	15	30	MHz
Erase block size	BPI flash data sheet [Ref 2]	65,536	65,536	65,536	65,536	65,536	65,536	Words
Sector erase time	BPI flash data sheet [Ref 3]	800	800	800	4,000	4,000	4,000	ms
Erase sectors for example update image	= (update_image_size) x 2^20 / 16 / (erase_block_size)	16	16	16	16	16	16	Blocks
Total internal flash erase time	= (erase_sectors) x (block_erase_time) / 1000	12.8	12.8	12.8	64	64	64	s
Program buffer size	BPI flash data sheet [Ref 2]	512	512	512	512	512	512	Words
Program buffer time	BPI flash data sheet [Ref 2]	900	900	900	3,016	3,016	3,016	µs
Program buffers for example update image	= (update_image_size) x 2^20 / 16 / (program_buffer_size)	2,048	2,048	2,048	2,048	2,048	2,048	Pages
Total flash internal program time	= (program_buffers) x (program_buffer_time) / 1000000	1.9	1.9	1.9	6.2	6.2	6.2	s
Total clock cycles to send update data for programming = update size bits / 16 bits/word *4.5 cycles/word	= (update_image_size) x 2^20 / 16 * 4.5	4,718,592	4,718,592	4,718,592	4,718,592	4,718,592	4,718,592	Cycles
Total program time	= (total_send_cycles) / (clock_frequency) /1000000 + (total_program_time)	2.9	2.3	2.1	7.2	6.6	6.4	s

Table 8: **Example QuickBoot Update Times for BPI Flash *(Cont'd)***

Description	Source/Computation	16 Mb Update @5 MHz (Typical)	16 Mb Update @15 MHz (Typical)	16 Mb Update @30 MHz (Typical)	16 Mb Update @5 MHz (Worst-Case)	16 Mb Update @15 MHz (Worst-Case)	16 Mb Update @30 MHz (Worst-Case)	Unit
Total clock cycles to read programmed data for verify CRC32 calculation = update size bits / 16 bits/word *2 cycles/word	= (update_image_size) x 2^20 / 16 * 2	2,097,152	2,097,152	2,097,152	2,097,152	2,097,152	2,097,152	Cycles
Total verify (read for CRC32) time	=(total_read_cycles) / (clock_frequency) /1000000	0.5	0.2	0.1	0.5	0.2	0.1	s
Total clock cycles for update data transfer = cycles to send update data for programming + cycles to receive update data for CRC32 calc	= (total_send_cycles) + (total_read_cycles)	6,815,744	6,815,744	6,815,744	6,815,744	6,815,744	6,815,744	Cycles
Total internal flash erase + program time	= (total_erase_time) + (total_program_time)	14.7	14.7	14.7	70.2	70.2	70.2	s
Total data transfer time	= (total_clock_cycles) / (clock_frequency MHz)/1000000	1.4	0.5	0.3	1.4	0.5	0.3	s
Total QuickBoot update time	= (total_erase_program_time) + (total_data_transfer_time)	16.1	15.2	15.0	71.6	70.7	70.5	s

Note: The total internal flash erase and program operations dominate the total QuickBoot update time. The clock frequency affects only the data transfer time. At 15 MHz, the total data transfer time is only 4% of the total QuickBoot update time. Thus, using clock frequencies greater than 15 MHz enables less than 4% improvement to the overall QuickBoot update time.

BpiFlashProgrammer Module Operation

All input signals to the BpiFlashProgrammer module are synchronous to the rising edge of inClk.

When inReset_EnableB = High, the BpiFlashProgrammer is held in reset/standby.

When inReset_EnableB = Low, the BpiFlashProgrammer module executes this update procedure:

1. Initialize: Set inReady_BusyB = Low and set outDone = Low.
2. Check ID: Read and check flash memory device identifier to double check connectivity.
3. Erase switch word: Erase the block that contains the critical switch word.
4. Erase update area: Erase the blocks that comprise the update bitstream image area.
5. Program update area: For each program buffer data, set in the update bitstream image area:
   a. Send the Buffered Program command.
   b. For each word in the buffer data set:
      - Set inReady_BusyB = High.
      - Wait for inDataWriteEnable = High.
      - Capture the data from the inData32 bus.
      - Set inReady_BusyB = Low.
      - Send data to the BPI flash.
      - If at the end of the current buffer data set, exit to step 5c.
   c. Enable the internal flash Buffered Program operation to program the data.
   d. Wait until the flash status is ready (i.e., the internal Buffered Program is complete).

6. Verify update area:
   a. Read all bytes from the update image area and compute the CRC32.
   b. Check that the resulting CRC32 matches an expected residue value.
7. Program the switch word.
8. Done: Set outDone = High.

Note: If at any step an error is detected, outDone and outError are set to High.

SPI Flash Programmer Reference Design Modules

The base reference design consists of two modules:

- `SpiFlashProgrammer.vhd`: QuickBoot flash programmer state machine for SPI flash
- `SpiSerDes.vhd`: Serializer/deserializer interface to the SPI bus

`SpiSerDes.vhd` Module

`SpiSerDes.vhd` is a utility module used by the main `SpiFlashProgrammer.vhd` module to access the SPI bus. The SPI bus has two serial data lines for bidirectional data flow. The `SpiSerDes.vhd` module provides a byte-wide interface to the SPI bus. The module serializes one byte of output data to an SPI bus and simultaneously deserializes one byte of input data from the SPI bus.

The `SpiSerDes.vhd` module is a basic building block and should be used as-is, without modification. The interface to the `SpiSerDes.vhd` module is provided in Table 9. Four ports of the interface must be connected to the FPGA pins that connect to the SPI flash.

Table 9: **SpiSerDes Port Descriptions**

Port	Direction	Width	Description and Connection
inClk	Input	1	System clock input. A gated version of inClk is forwarded to the SPI clock. Connect SpiSerDes.inClk and SpiFlashProgrammer.inClk to the same clock source. See SPI Flash Programmer Clock Frequency, page 29 for clock frequency considerations.
inReset_EnableB	Input	1	Synchronous active-High reset and active-Low enable. When High, this module is held in reset and outSpiCsB is held High. When Low, this module drives outSpiCsB Low, then waits for inStartTransfer to go High to initiate a serialized transfer. Connect to the SpiFlashProgrammer.outSSDReset_EnableB port.
inStartTransfer	Input	1	Active-High transfer start signal. Synchronous input initiates a serialized transfer. On the first rising-edge of inClk with inStartTransfer High, this module captures data from inData8Send to serialize to outSpiMosi. When the transfer is complete, this module drives outTransferDone High. Connect to the SpiFlashProgrammer.outSSDStartTransfer port.
outTransferDone	Output	1	Active-High transfer done signal. When a serial transfer has completed, this module drives outTransferDone High. When outTransferDone is High, the outData8Receive has the byte of deserialized data from inSpiMiso. Connect to the SpiFlashProgrammer.inSSDTransferDone port.
inData8Send	Input	8	8-bit data value to serialize through SPI MOSI output to the SPI flash. inData8Send is captured for serialization on the first rising edge of inClk when inStartTransfer is High. Connect to the SpiFlashProgrammer.outData8Send port.
outData8Receive	Output	8	8-bit data value captured from the SPI MISO input from the SPI flash. outData8Receive is valid when outTransferDone is High. Connect to the SpiFlashProgrammer.outData8Receive port.
outSpiCsB	Output	1	SPI chip-select (active-Low). Connect to SPI flash chip-select via FPGA FCS_B pin.
outSpiClk	Output	1	SPI clock. Connect to SPI flash clock via FPGA CCLK pin. See 7 Series FPGAs Access to the SPI Clock, page 25 for additional connection details.

Table 9: **SpiSerDes Port Descriptions *(Cont'd)***

Port	Direction	Width	Description and Connection
outSpiMosi	Output	1	SPI serial command/data master-out/slave-in. Connect to the SPI DQ0 pin via the FPGA D00_MOSI pin.
inSpiMiso	Input	1	SPI serial data master-in/slave-out. Connect to the SPI DQ1 pin via the FPGA D01_DIN pin.

7 Series FPGAs Access to the SPI Clock

For FPGA configuration, the FPGA CCLK pin drives the SPI flash clock (C) pin. The FPGA CCLK pin is a dedicated pin that is not directly accessible from fabric. The SPI flash programmer modules can access the FPGA CCLK pin via the STARTUPE2.USRCCLKO primitive and port.

Note: The STARTUPE2.USRCCLKTS port must also be driven Low to ensure the CCLK output is enabled. See Figure 7 for the STARTUPE2 access path from the FPGA logic to the SPI clock signal.

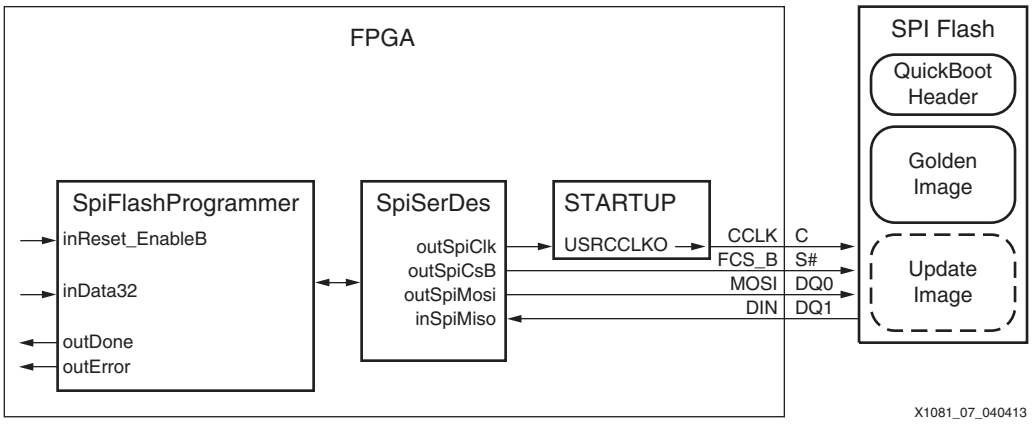

Figure 7: **SPI Clock Access Via STARTUP Primitive**

`SpiFlashProgrammer.vhd` is the programming state machine that implements the QuickBoot programming algorithm. `SpiFlashProgrammer.vhd` uses the `SpiSerDes.vhd` module to send commands/data to the SPI flash and to receive data from the SPI flash.

Table 10 describes the SpiFlashProgrammer ports.

Table 10: **SpiFlashProgrammer Port Descriptions**

Port Name	Direction	Width	Description and Connection
inClk	Input	1	System clock input. Connect SpiSerDes.inClk and SpiFlashProgrammer.inClk to the same clock source. See the SpiSerDes.inClk description for the clock frequency considerations.
inReset_EnableB	Input	1	Active-High reset and active-Low enable. When High, this module is held in reset. When Low, this module is activated to start the QuickBoot update image reprogramming procedure.
inCheckIdOnly	Input	1	Active-High check ID only signal. Debug input control signal. Set to `0` for normal operation. inCheckIdOnly is registered on the first rising edge of inClk when inReset_EnableB is Low. When registered as High, this input causes this module to stop after the check ID procedure.

Table 10: **SpiFlashProgrammer Port Descriptions *(Cont'd)***

Port Name	Direction	Width	Description and Connection
inVerifyOnly	Input	1	Active-High verify only signal. Debug/advanced input control signal. Set to 0 for standard operation. inVerifyOnly is registered on the first rising edge of inClk when inReset_EnableB is Low. When registered as High, this input causes this module to skip erase/programming procedures and to perform only the verify procedure. See QuickBoot Verification of the Update Image, page 13 for a description of the verification procedure. This can be used for advanced applications.
inData32	Input	32	32-bit wide input data bus. The module can be configured via cDataWordWidth to accept data words on this bus that are 8 bits wide via inData32[7:0], 16 bits wide via inData32[15:0], or 32 bits wide via inData32[31:0]. By default, the module takes 32-bit words. See inDataWriteEnable and outReady_BusyB ports for when input data is taken.
inDataWriteEnable	Input	1	Active-High data write enable signal. When outReady_BusyB is High and inDataWriteEnable is High, a data word is captured from the inData32 bus on the rising edge of inClk.
outReady_BusyB	Output	1	Active-High ready signal or active-Low busy signal. The module drives this signal High when it is ready to accept the next data word. Data can be accepted as soon as the clock edge that immediately follows outReady_BusyB goes High.
outDone	Output	1	Active-High done signal. The module drives outDone High when programming is done or an error is detected. When outDone is High, check outError for an error condition.
outError	Output	1	Active-High error signal. The module drives outError High when *any* error is detected. Additional details about the kind of error are provided via the signals below.
outErrorIdcode	Output	1	Active-High check ID error signal. Optional/debug signal connection: Provides additional detail for when outError is High.
outErrorErase	Output	1	Active-High erase error signal. Optional/debug signal connection: Provides additional detail for when outError is High.
outErrorProgram	Output	1	Active-High program error signal. Optional/debug signal connection: Provides additional detail for when outError is High.
outErrorTimeOut	Output	1	Active-High erase or program timeout error signal. Optional/debug signal connection: Provides additional detail for when outError is High. See module's cCmd*TimeOut constants for timeout periods.
outErrorCrc	Output	1	Active-High verify CRC error signal. Optional/debug signal connection: Provides additional detail for when outError is High.
outStarted	Output	1	Active-High module started procedure signal. Optional/debug signal connection: Provides additional detail on programmer progress.
outInitializeOK	Output	1	Active-High module completed initialization stage signal. Optional/debug signal connection: Provides additional detail on programmer progress.
outCheckIdOK	Output	1	Active-High module successfully completed check ID stage signal. Optional/debug signal connection: Provides additional detail on programmer progress.
outEraseSwitchWordOK	Output	1	Active-High module successfully completed erase switch word stage signal. Optional/debug signal connection: Provides additional detail on programmer progress.
outEraseOK	Output	1	Active-High module successfully completed update image erase stage signal. Optional/debug signal connection: Provides additional detail on programmer progress.
outProgramOK	Output	1	Active-High module successfully completed update image program stage signal. Optional/debug signal connection: Provides additional detail on programmer progress.
outVerifyOK	Output	1	Active-High module successfully completed verify stage signal. Optional/debug signal connection: Provides additional detail on programmer progress.
outProgramSwitchWordOK	Output	1	Active-High module successfully completed switch word programming stage signal. Optional/debug signal connection: Provides additional detail on programmer progress.
outSSDReset_EnableB	Output	1	Connect to SpiSerDes.inReset_EnableB port.
outSSDStartTransfer	Output	1	Connect to SpiSerDes.inStartTransfer port.

Table 10: **SpiFlashProgrammer Port Descriptions *(Cont'd)***

Port Name	Direction	Width	Description and Connection
outSSDTransferDone	Output	1	Connect to SpiSerDes.outTransferDone port.
outSSDData8Send	Output	8	Connect to SpiSerDes.inData8Send port.
inSSDData8Receive	Input	8	Connect to SpiSerDes.outData8Receive port.

Configuring the `SpiFlashProgrammer.vhd` Parameters

`SpiFlashProgrammer.vhd` contains hard-coded addresses and commands to ensure that an implementation can erase or program only specific locations in the SPI flash memory array. The following code segment from the `SpiFlashProgrammer.vhd` module highlights the basic parameters that must be set for the specific system in which the module is implemented.

```
--#############################################################################
  -- BEGIN BASIC PARAMETER SETTINGS:
  -- THE FOLLOWING MUST MATCH THE OUTPUT FROM MakeSpiFlashProgrammerMcsFiles.pl
  -- SPI DEVICE TYPE: Only one of the following must be set to '1'. Other is '0'.
  constant  cMicronN25Q       : std_logic := '1'; -- Micron N25Q serial NOR flash
  constant  cMicronNP5Q       : std_logic := '0'; -- Micron NP5Q phase change memory
  constant  cAtmelDataFlash   : std_logic := '0'; -- Atmel DataFlash or Spartan-3AN with power-of-2 page size. (NOT TESTED)
  constant  cWinbondW25Q      : std_logic := '0'; -- Winbond W25Q (NOT TESTED)
  constant  cSpansionS25FS    : std_logic := '0'; -- Spansion S25FS (NOT TESTED)
  -- SPI SECTOR SIZE
  constant  cSizeSector       : std_logic_vector(31 downto 0) := X"00010000"; -- 65536 bytes
  -- SPI PAGE SIZE: Ensure setting is correct for the chosen device type.
  constant  cSizePage         : std_logic_vector(31 downto 0) := X"00000100"; -- 256 (std)
  constant  cSizePageNP5Q     : std_logic_vector(31 downto 0) := X"00000040"; -- 64
  -- SPI COMMAND ADDRESS SIZE (IN BITS): Ensure setting is correct for the target flash
  constant  cAddrWidth        : integer   := 24;  -- 24 or 32 (address width in bits)
  -- UPDATE IMAGE START ADDRESS
  constant  cAddrUpdateStart  : std_logic_vector(31 downto 0) := X"00200000";
  -- UPDATE IMAGE END+1 ADDRESS
  constant  cAddrUpdateEnd    : std_logic_vector(31 downto 0) := X"00400000";
  -- SWITCH WORD ADDRESS: Ensure setting is correct for the chosen device type
  constant  cAddrSwitchWordN25Q : std_logic_vector(31 downto 0) := X"00000FFC"; -- Typically=(Subsector or page) size - 4.
  constant  cAddrSwitchWordNP5Q : std_logic_vector(31 downto 0) := X"0000003C"; -- Typically=(Subsector or page) size - 4.
  constant  cAddrSwitchWordAtmel: std_logic_vector(31 downto 0) := X"000000FC"; -- Typically=(Subsector or page) size - 4.
  -- SWITCH WORD
  constant  cSwitchWord         : std_logic_vector(31 downto 0) := X"AA995566";
  ------------------------------------------------------------------------------
  -- DATA WORD WIDTH (IN BYTES) FOR INPUT APPLIED TO THE MODULE'S 32-BIT inData32 BUS
  constant  cDataWordWidth    : integer := 4;   -- 4 = 32-bit word --> inData32[31:0]
  --                                            -- 2 = 16-bit word --> inData32[15:0]
  --                                            -- 1 = 8-bit word --> inData32[7:0]
  -- END BASIC PARAMETER SETTINGS
  --#############################################################################
```

Table 11 provides a description of the `SpiFlashProgrammer.vhd` parameters.

Table 11: **SpiFlashProgrammer.vhd Parameters**

Parameter	Value	Output Report Line[1]	Description
cMicronN25Q cMicronNP5Q cAtmelDataFlash cWinbondW25Q cSpansionS25FS	`1` or `0`	SPI Device Type	SPI device type. These constants define the target flash type to be programmed. The constant that is set to `1` is the type of flash to be programmed. Only one of these five constants must be set to `1` and the rest must be set to `0`. Only the supported cMicronN25Q constant should be set to `1`. The rest are reserved and must be set to `0`. Ensure this setting matches the SPI Device Type from the output report of the `MakeSpiFlashProgrammerMcsFiles.pl` Perl script.
cAddrWidth	`24` or `32` (integer)	SPI address width	SPI command address width (in bits). This constant defines the address width for SPI flash commands. Typically, SPI flash of 128 Mb or smaller use 24-bit addresses, and SPI flash greater than 128 Mb use 32-bit addresses.
cSizeSector	Xhhhhhhhh (32-bit hex)	SPI sector size	SPI sector size (in bytes). This constant defines the general erasable sector size of the SPI flash. Ensure this setting matches the SPI sector size from the output report of the `MakeSpiFlashProgrammerMcsFiles.pl` Perl script.

Table 11: **SpiFlashProgrammer.vhd Parameters *(Cont'd)***

Parameter	Value	Output Report Line[1]	Description
cSizePage cSizePageNP5Q	Xhhhhhhhh (32-bit hex)	SPI page size	SPI page size (in bytes). These constants define the program page size. Typically, the preset values are correct and need NO adjustment. Check that the page size is correct for the target SPI flash. The cSizePageNP5Q constant is selected as the page size when the cMicronNP5Q device type is selected. Otherwise, for all other SPI flash, cSizePage is selected as the page size. Ensure this setting matches the SPI page size from the output report of the `MakeSpiFlashProgrammerMcsFiles.pl` Perl script.
cAddrUpdateStart	Xhhhhhhhh (32-bit hex)	Update image start address	Update image start (byte) address. Set to the starting byte address for update image. Ensure this setting matches the Update image start address from the output report of the `MakeSpiFlashProgrammerMcsFiles.pl` Perl script.
cAddrUpdateEnd	Xhhhhhhhh (32-bit hex)	Update image end+1 address	Update image end+1 (byte) address. Set to one more than the last byte of the last sector in the update image area. This is used as the ending value for the sector or page address counters. Ensure this setting matches the Update image end+1 address from the output report from the `MakeSpiFlashProgrammerMcsFiles.pl` Perl script.
cAddrSwitchWordN25Q cAddrSwitchWordNP5Q cAddrSwitchWordAtmel	Xhhhhhhhh (32-bit hex)	Update switch word address	Switch word (byte) address. These constants define the switch word first byte address for each SPI flash type. Typically, the preset values are correct and need NO adjustment. Check that the switch word address is correct for the target SPI flash and matches the output report from the `MakeSpiFlashProgrammerMcsFiles.pl` script.
cSwitchWord	XAA995566 (32-bit hex)	N/A	Critical switch word ON value. Do NOT change.
cDataWordWidth	4, 2, or 1	N/A	Input data word width (in bytes). This constant defines the width of the data presented on the inData32 bus. Set to 4 bytes for 32-bit data on inData32[31:0]. Set to 2 bytes for 16-bit data on inData32[15:0] only. inData32[31:16] are unused. Set to 1 byte for 8-bit data on inData32[7:0] only. inData32[31:8] are unused.

Notes:

1. This is the `MakeSpiFlashProgrammerMcsFiles.pl` Perl script output report.

`SpiFlashProgrammer.vhd` also contains timeout limits for the flash erase and program commands. In general, the default timeout limits should work for most implementations. The following code segment from the `SpiFlashProgrammer.vhd` module highlights the timeout limits.

```
-- Device timeouts. timeout = max_time (s) X inClk (Hz) / 22 or X"00000000" = ultimate MAX count
-- where 22 is polling loop cycle count.
constant  cClkFrequencyHz  : integer                        := 20000000;    -- Not used; Keep for reference
constant  cClkCyclesPerTOLoop:integer                       := 22;          -- Not used; Keep for reference
constant  cCmdSETimeOut    : std_logic_vector(31 downto 0) := X"00299D69"; -- max tSE * cClkFrequencyHz / 22
constant  cCmdSSETimeOut   : std_logic_vector(31 downto 0) := X"000B18E9"; -- max tSSE* cClkFrequencyHz / 22
constant  cCmdPPTimeOut    : std_logic_vector(31 downto 0) := X"000011C2"; -- max tPP * cClkFrequencyHz / 22
```

Table 12 provides a description of the SpiFlashProgrammer timeout limits. Table 12 should be used to cross-check the default timeout parameters against the maximum erase and program time specifications in the target flash data sheet and at the clock frequency used in the implementation.

Table 12: **SpiFlashProgrammer Timeout Limits**

Parameter Name	Default Value	Description
cClkFrequencyHz	20000000	The maximum expected inClk frequency. Default setting is 20 MHz.
cClkCyclesPerTOLoop	22	Number of inClk cycles per timeout loop.
cCmdSETimeOut	X00299D69	Sector erase: Maximum timeout loop count. Value = Max t_{SE} time in seconds * cClkFrequencyHz / cClkCyclesPerTOLoop. Default X00299D69 = 3 seconds * 20,000,000 Hz / 22. See flash data sheet for maximum t_{SE} time.
cCmdSSETimeOut	X000B18E9	Sub-sector erase: Maximum timeout loop count. Value = Max t_{SSE} time in seconds * cClkFrequencyHz / cClkCyclesPerTOLoop. Default X000B18E9 = 0.8 seconds * 20,000,000 Hz / 22. See flash data sheet for maximum t_{SSE} time.
cCmdPPTimeOut	X000011C2	Page program: Maximum timeout loop count. Value = Max t_{PP} time in seconds * cClkFrequencyHz / cClkCyclesPerTOLoop. Default X000011C2 = 0.005 seconds * 20,000,000 Hz / 22. See flash data sheet for maximum t_{PP} time.

SPI Flash Programmer Clock Frequency

The SPI flash programmer modules must operate from the same system clock source. The clock source is forwarded to the SPI bus clock line. Therefore, the input system clock frequency must be within the capabilities of the SPI bus. The SPI bus works on half a clock period: data is output on the falling edge of the SPI clock and is captured on the rising edge of the SPI clock. Maximum system frequency is less than the maximum clock frequency specified in the SPI flash data sheet due to a combination of device clock-to-out time, device setup time, trace delays, and FPGA fabric delays.

A conservative recommendation that is compatible with most SPI flash is to use a system clock frequency of 20 MHz or slower for the clock (inClk port) to the SPI flash programmer modules. Table 13 provides example QuickBoot update times for different clock frequencies. Both typical and worst-case times are shown.

Table 13: **Example QuickBoot Update Times for SPI Flash**

Description	Source/Computation	16 Mb Update @10 MHz (Typical)	16 Mb Update @20 MHz (Typical)	16 Mb Update @40 MHz (Typical)	16 Mb Update @10 MHz (Worst-Case)	16 Mb Update @20 MHz (Worst-Case)	16 Mb Update @40 MHz (Worst-Case)	Unit
Example update image size	User design	16	16	16	16	16	16	Mb
Clock frequency	User design	10	20	40	10	20	40	MHz
Erase sector size	From SPI flash data sheet [Ref 3]	65,536	65,536	65,536	65,536	65,536	65,536	Byte
Sector erase time	From SPI flash data sheet [Ref 3]	700	700	700	3,000	3,000	3,000	ms
Erase sectors for example update image	= (update_image_size) x 2^20 / 8 / (erase_sector_size)	32	32	32	32	32	32	sector
Total internal flash erase time	= (erase_sectors) x (sector_erase_time) / 1000	22.4	22.4	22.4	96	96	96	s
Program page size	SPI flash data sheet [Ref 3]	256	256	256	256	256	256	Byte
Page program time	SPI flash data sheet [Ref 3]	0.5	0.5	0.5	5	5	5	ms
Program pages for example update image	= (update_image_size) x 2^20 / 8 / (program_page_size)	8,192	8,192	8,192	8,192	8,192	8,192	Page

Table 13: **Example QuickBoot Update Times for SPI Flash *(Cont'd)***

Description	Source/Computation	16 Mb Update @10 MHz (Typical)	16 Mb Update @20 MHz (Typical)	16 Mb Update @40 MHz (Typical)	16 Mb Update @10 MHz (Worst-Case)	16 Mb Update @20 MHz (Worst-Case)	16 Mb Update @40 MHz (Worst-Case)	Unit
Total flash internal program time	= (program_pages) x (page_program_time) / 1000	4.1	4.1	4.1	41	41	41	s
Total clock cycles to send update data for programming. = update size in bits / 8 bits/byte * 11 cycles/byte	= (update_image_size) x 2^20 / 8 * 11	23,068,672	23,068,672	23,068,672	23,068,672	23,068,672	23,068,672	Cycle
Total program time	= (total_send_cycles) / (clock_frequency) /1000000 + (total_program_time)	6.5	5.3	4.7	43.4	42.2	41.6	s
Total clock cycles to read programmed data for verify CRC32 calculation. = update size in bits / 8 bits/byte * 11 cycles/byte	= (update_image_size) x 2^20 / 8 * 11	23,068,672	23,068,672	23,068,672	23,068,672	23,068,672	23,068,672	Cycle
Total verify (read for CRC32) time	= (total_read_cycles) / (clock_frequency) /1000000	2.4	1.2	0.6	2.4	1.2	0.6	s
Total clock cycles for update data transfer = cycles to send update data for programming + cycles to receive update data for CRC32 calc	= ((total_send_cycles) + (total_read_cycles)	46,137,344	46,137,344	46,137,344	46,137,344	46,137,344	46,137,344	Cycle
Total internal flash erase + program time	= (total_erase_time) + (total_program_time)	26.5	26.5	26.5	137	137	137	s
Total data transfer time	= (total_clock_cycles)/ (clock_frequency)/1000000	4.7	2.4	1.2	4.7	2.4	1.2	s
Total QuickBoot Update Time	=(total_erase_program_time) + (total_data_transfer_time)	31.2	28.9	27.7	141.7	139.4	138.2	s

Note: The total internal flash erase and program operations dominate the total QuickBoot update time. The clock frequency affects only the data transfer time. At 20 MHz, the total data transfer time is only 9% of the total QuickBoot update time. Thus, using clock frequencies greater than 20 MHz enables less than 9% improvement to the overall QuickBoot update time.

SpiFlashProgrammer Module Operation

All input signals to the SpiFlashProgrammer module are synchronous to the rising edge of inClk.

When inReset_EnableB = High, the SpiFlashProgrammer is held in reset/standby.

When inReset_EnableB = Low, the SpiFlashProgrammer module executes this update procedure:

1. Initialize: Set inReady_BusyB = Low and set outDone = Low.
2. Check ID: Read and check flash memory device identifier to double-check connectivity.
3. Erase switch word: Erase the sub-sector that contains the critical switch word.
4. Erase update area: Erase the sectors that comprise the update bitstream image area.
5. Program update area: For each page in the update bitstream image area:
    a. Send the Page Program command.
    b. For each word in the current page:
        - Set inReady_BusyB = High.
        - Wait for inDataWriteEnable = High.

- Capture the data from the inData32 bus.
- Set inReady_BusyB = Low.
- Send data to the SPI flash.
- If at the end of the current page, exit to step 5c.

c. Enable the internal flash Page Program operation to program the page data.

d. Wait until the flash status is ready (i.e., internal Page Program is complete).

6. Verify the update area:
   a. Read all bytes from the update image area and compute the CRC32.
   b. Check that the resulting CRC32 matches an expected residue value.
7. Program the switch word.
8. Done: Set outDone = High.

Note: If an error is detected at any step, outDone and outError are set to High.

Using the QuickBoot Flash Programmer Module

Both the SPI and BPI flash programmer reference designs have the same user interface. Figure 3 illustrates using the QuickBoot flash programmer module in a system.

Table 5 and Table 10 list the ports of the QuickBoot flash programmer module for BPI mode and SPI mode, respectively. Important ports and their typical user connections are listed here:

- **inClk**: Connect a clock to the inClk port. See BPI Flash Programmer Clock Frequency, page 22 or SPI Flash Programmer Clock Frequency, page 29 for a clock frequency recommendation.
- **inReset_EnableB**: Connect the output from a writable control register to the inReset_EnableB port.
- **inData32**: Connect the source data bus to the inData32 port.
- **outReady_BusyB** and **inDataWriteEnable**: Connect the source data flow control logic to the outReady_BusyB and inDataWriteEnable ports.
- **outDone** and **outError**: Connect to a readable register to monitor for done and errors.

Typically, a FIFO or packet feeder is used to stream update bitstream image data to the inData32 port using the outReady_BusyB and inDataWriteEnable flow control signals.

When a new update MCS flash memory image is available, Table 14 shows the steps for controlling the QuickBoot flash programmer module and for sending the update MCS data to update the flash memory for three kinds of data delivery mechanisms: streaming data via FIFO, streaming data via packets, and repeating data register writes.

Table 14: **Steps for Programming the Update Image via the QuickBoot Flash Programmer Module**

Step	Streaming FIFO to inData32	Streaming Packets to inData32	Repeating Register Write to inData32
Reset/Standby	Hold inReset_EnableB = High	Hold inReset_EnableB = High	Hold inReset_EnableB = High
Step 1	Set inReset_EnableB = Low	Set inReset_EnableB = Low	Set inReset_EnableB = Low
Step 2	Optional: Explicitly wait sufficient time while SpiFlashProgrammer erases the switch word and update area.	Optional: Explicitly wait sufficient time while SpiFlashProgrammer erases the switch word and update area.	Optional: Explicitly wait sufficient time while SpiFlashProgrammer erases the switch word and update area.

Step 3	Stream the update MCS data to a FIFO that feeds the inData32 port.
	Assumes streaming-FIFO subsystem has flow control for busy periods and can tolerate the busy periods when the flash is being erased or pages are being programmed.
		Stream the update MCS data via packets to a module that can feed each packet to the inData32 port.
		Assumes streaming-packet subsystem has flow control for busy periods and can tolerate the busy periods when the flash is being erased or pages are being programmed.
			For each update MCS data word do
			begin
			Write data to register that drives inData32.
			Wait for inReady_BusyB = High;
			Write inDataWriteEnable = High;
			Write inDataWriteEnable = Low;
			end;

Table 14: **Steps for Programming the Update Image via the QuickBoot Flash Programmer Module *(Cont'd)***

Step	Streaming FIFO to inData32	Streaming Packets to inData32	Repeating Register Write to inData32

Step 4	Poll for outDone = High.	Poll for outDone = High.	Repeat begin
			Read register that receives outDone;
			if outDone=High, then exit loop;
			end;

Step 5	Check for outError = High. If outError = High, then an error occurred.	Check for outError = High. If outError = High, then an error occurred.	Read register that receives outError; if outError = High, then an error occurred.

Step 2 is an optional wait for the update image to be erased. For NOR flash memory, the erase operation takes considerable time. See Table 8 or Table 13 to calculate the total internal flash erase time for BPI or SPI modes, respectively. Calculation methods for the typical and worst-case erase times are shown. If the beginning of step 3 cannot tolerate waiting for the total internal flash erase time, the optional step 2 should be implemented with an explicit and sufficient wait before beginning step 3.

The system data delivery flow control mechanism must also be capable of handling the busy periods during step 3. During step 3, the QuickBoot flash programmer is periodically busy. The period at which the programmer is busy is whenever the flash programmer has received and filled the data buffer for BPI flash or delivered a page of data for SPI flash. The programmer then has a busy status while the flash memory performs its internal page program or internal buffered program operation. See Table 8 for the BPI flash buffer size and internal buffered programming time. See Table 13 for the SPI flash page size and internal page programming time.

QuickBoot Flash Programmer Reference Design Checklist, Test, and Debug

This is a checklist of implementation tips, test actions, and debug:

- Basic FPGA-flash check: Can the iMPACT tool program a standard, single bitstream into the attached flash?
- Can the standard, single bitstream in the attached flash programmed by the iMPACT tool configure the FPGA?
- Can the iMPACT tool program the attached flash with the `*_initial.mcs` file, and does the FPGA configure from the attached flash with the design contained within the `*_initial.mcs`?

  Note: The iMPACT tool should be used to readback the flash contents. The readback MCS should be saved for later comparison against the initial flash contents.
- Can the iMPACT tool program the attached flash with the `*_update.mcs` file, and does the FPGA configure from the attached flash with the design contained within the `*_update.mcs`?

  Note: The iMPACT tool should be used to readback the flash contents. The readback MCS should be saved for later comparison against the expected contents of an updated flash.
- For the QuickBoot SPI flash programmer only: Is the SpiSerDes.inClk and SpiFlashProgrammer.inClk connected to the same system clock source?
- Is the QuickBoot flash programmer inClk source clock frequency within the recommended range shown in either BPI Flash Programmer Clock Frequency, page 22 or SPI Flash Programmer Clock Frequency, page 29?
- Basic system/connectivity check: Does the QuickBoot flash programmer module with inCheckIdOnly = High complete without error? Can this be repeated multiple times without error?

- Do the cAddrSwitchWord, cAddrUpdateStart, and cAddrUpdateEnd values set in the flash programmer module match the values reported by the Perl script for the `*_update.mcs` output?
- After attempting a QuickBoot flash update process, use the iMPACT tool with a JTAG cable to readback the attached flash contents. Analyze the readback contents to check what bytes have been changed by the QuickBoot flash programmer update process. Compare against readback files that were saved earlier for expected contents:
  - Check the value in the readback MCS file at the critical switch word location.
  - Compare the entire readback MCS file to the `*_initial.mcs` file. Only the critical switch word segment and update image areas should have changes.
  - Compare the update image area in the readback MCS file to the `*_update.mcs` file and compare the bit and byte orders of the values within the file.
- Monitor the outError* signals from the QuickBoot flash programmer for errors that might have occurred.
- Monitor the out*OK signals for stages that the QuickBoot flash programmer successfully completed or for stages that were not reached.

KC705 Board Demonstrations

See Table 4 for information on the NOR flash devices on the KC705 evaluation board. QuickBoot demonstration designs are provided for each flash type, BPI and SPI, respectively.

For the demonstration designs, an auxiliary `JtagToBpiFlashProgrammer.vhd` or `JtagToSpiFlashProgrammer.vhd` module is added to each reference design to bridge the FPGA JTAG port with the corresponding QuickBoot BPI or SPI flash programmer module. For convenience, SVF files are provided for controlling and delivering data to the QuickBoot flash programmer via the iMPACT tool and JTAG cable.

Note: SVF files have a known limitation and thus are not optimized for programming time. The SVF programming time via the iMPACT tool should not be considered representative of the QuickBoot flash programmer's update time. Instead, see Table 8 or Table 13 for QuickBoot update times.

See *KC705 Evaluation Board for the Kintex-7 FPGA User Guide* [Ref 6] for KC705 board references within this section.

KC705 Board Demonstration for QuickBoot Using BPI Flash

The BPI flash on the KC705 board can accommodate many XC7K325T FPGA bitstreams. For the purpose of the demonstration, two compressed bitstreams are used. Both bitstreams contain the BpiFlashProgrammer module. One bitstream triggers the GPIO_LED_7 to blink once per period. The second bitstream triggers the GPIO_LED_7 to blink twice per period.

The demonstration design includes additional modules that drive the BpiFlashProgrammer from the JTAG interface. Perl scripts are provided that generate SVF or STAPL files for controlling the BpiFlashProgrammer module and for sending the update MCS data via JTAG.

The demonstration design uses two files. Copies of the following two files are available in the `PrebuiltDemoFiles` directory of the reference design:

- `BpiQuickBootDemo1_initial.mcs`: Contains the complete initial QuickBoot flash image with the QuickBoot header, one copy of the `BpiQuickBootDemo1.bit` design in the golden image location, and a second copy of the `BpiQuickBootDemo1.bit` design in the update image location.
- `BpiQuickBootDemo2_update.svf`: Contains JTAG commands and data for updating the flash update image with the `BpiQuickBootDemo2.bit` design from the `BpiQuickBootDemo2_update.mcs` file.

To run the demonstration:

1. Turn off the KC705 board power.

2. Connect the USB cable from a computer to the KC705 USB JTAG module.
3. Set the KC705 switch and jumper settings according to the “Default Switch and Jumper Settings” section in *KC705 Evaluation Board for the Kintex-7 FPGA User Guide* [Ref 6].
4. Set the J17 mode switch settings to BPI mode:
   - M0 = OFF
   - M1 = ON
   - M2 = OFF
5. Turn on the KC705 board power.
6. Use iMPACT to program `BpiQuickBootDemo1_initial.mcs` into the BPI flash that is attached to the XC7K325T FPGA. (See the iMPACT Help “Indirect Programming” section [Ref 5] for detailed instructions.) When attaching the BPI flash to the FPGA in the iMPACT tool, make these selections:
   - **BPI PROM**
   - **28F00AP30**
   - Data Width: **16**
   - RS[1:0]b Pin Address Bits: **25:24**.

   After successful programming and FPGA reconfiguration, GPIO_LED_7 blinks once per period, showing that `BpiQuickBootDemo1.bit` is loaded.
7. Delete the XC7K325T device from the iMPACT boundary-scan window.
8. Add the `BpiQuickBootDemo2_update.svf` file.
9. To verify a successful update:
   a. Execute the `BpiQuickBootDemo2_update.svf` file to completion to program `BpiQuickBootDemo2_update.mcs` into the update image area of the BPI flash. (See the iMPACT Help “Executing XSVF/SVF Files” section [Ref 5] for detailed instructions.)
   b. After successful execution of the SVF and reconfiguration of the FPGA, GPIO_LED_7 should blink twice per period, showing that the update image is loaded.
10. Example of a failed update and FPGA configuration recovery:
    a. Execute the `BpiQuickBootDemo2_update.svf` again to reprogram the update image, and interrupt the programming midway by turning off the power to the KC705 board.
    b. Turn on power to the KC705 board. GPIO_LED_7 should blink once per period, showing that the original `BpiQuickBootDemo1.bit` is loaded from the golden image location.
11. Example of a successful update: Execute the `BpiQuickBootDemo2_update.svf` file to completion to restore the update image. GPIO_LED_7 blinks twice per period, showing that the update image is restored in flash and loaded in the FPGA.

Note: On the KC705 board, the FPGA RS[1:0] pins drive the most significant address bits of the BPI flash for the purpose of selecting evaluation designs from the default KC705 board setup. Although the QuickBoot reference design is compatible with the FPGA-RS-to-BPI-address pin connections on the KC705 board, the recommendation for the QuickBoot solution is a direct set of FPGA-address-to-BPI-address pin connections.

KC705 Board Demonstration for QuickBoot Using SPI Flash

Normally, the SPI flash on the KC705 board can accommodate one uncompressed XC7K325T FPGA bitstream. For the purpose of the demonstration, two compressed bitstreams are used that can both fit within the SPI flash. Both bitstreams contain the SpiFlashProgrammer module. One bitstream triggers GPIO_LED_7 to blink once per period. The second bitstream triggers GPIO_LED_7 to blink twice per period.

The demonstration design includes additional modules that drive the SpiFlashProgrammer from the JTAG interface. Perl scripts are provided that generate SVF or STAPL files for controlling the SpiFlashProgrammer module and for sending the update MCS data via JTAG.

The demonstration design uses two files. Copies of the following two files are available in the `PrebuiltDemoFiles` directory of the reference design:

- `SpiQuickBootDemo1_initial.mcs`: Contains the complete initial QuickBoot flash image with the QuickBoot header, one copy of the `SpiQuickBootDemo1.bit` design in the golden image location, and a second copy of the `SpiQuickBootDemo1.bit` design in the update image location.
- `SpiQuickBootDemo2_update.svf`: Contains JTAG commands and data for updating the flash update image with the `SpiQuickBootDemo2.bit` design from the `SpiQuickBootDemo2_update.mcs` file.

To run the demonstration:

1. Turn off the board power.
2. Connect the USB cable from a computer to the KC705 USB JTAG Module.
3. Set the KC705 switch and jumper settings according to the “Default Switch and Jumper Settings” section in *KC705 Evaluation Board for the Kintex-7 FPGA User Guide* [Ref 6].
4. Override the default J17 mode switch settings to SPI mode:
   - M0 = ON
   - M1 = OFF
   - M2 = OFF
5. Turn on the KC705 board power.
6. Use iMPACT to program `SpiQuickBootDemo1_initial.mcs` into the SPI flash that is attached to the XC7K325T FPGA. (See the iMPACT Help “Indirect Programming” section [Ref 5] for detailed instructions.) When attaching the SPI flash to the FPGA in the iMPACT tool, make these selections:
   - **SPI PROM**
   - **N25Q128 1.8/3.3V**
   - Data Width: **1**

   After successful programming and FPGA reconfiguration, GPIO_LED_7 blinks once per period, showing that `SpiQuickBootDemo1.bit` is loaded.
7. Delete the XC7K325T device from the iMPACT boundary-scan window.
8. Add the `SpiQuickBootDemo2_update.svf` file.
9. Example of a successful update:
   a. Execute the `SpiQuickBootDemo2_update.svf` file to completion to program `SpiQuickBootDemo2_update.mcs` into the update image area of the SPI flash. (See the iMPACT Help “Executing XSVF/SVF Files” section [Ref 5] for detailed instructions.)
   b. After successful execution of the SVF and reconfiguration of the FPGA, GPIO_LED_7 blinks twice per period showing that the update image is loaded.

10. Example of a failed update and FPGA configuration recovery:
    a. Execute `SpiQuickBootDemo2_update.svf` again to reprogram the update image, and interrupt the programming midway by turning off the power to the KC705 board.
    b. Turn on power to the KC705 board. GPIO_LED_7 blinks once per period, showing that the original `SpiQuickBootDemo1.bit` is loaded from the golden image location.
11. Example of a successful update: Execute the `SpiQuickBootDemo2_update.svf` file to completion to restore the update image. GPIO_LED_7 blinks twice per period, showing that the update image is restored in flash and loaded in the FPGA.

Note: Although the Micron N25Q128 flash can be ordered with a reset pin, the SPI flash on the KC705 board does not have a reset pin. In step 10, if the KC705 FPGA_PROG_B SW14 pushbutton is pressed instead of turning the power off to interrupt the flash programming, the FPGA might not configure after release of the FPGA_PROG_B pushbutton. See Flash Reset Recommendation, page 15 for details.

Summary of the QuickBoot Method

The QuickBoot method is a remote update solution for FPGA bitstreams that provides protection against errors or interrupts during the bitstream update process. The QuickBoot method provides a fast, deterministic configuration time for all conditions of a remotely updatable system. QuickBoot has several additional advantages compared to traditional remote update configuration methods, as shown in Table 15.

Table 15: **Comparison of QuickBoot Method Versus Other Remote Update Configuration Methods**

Features	QuickBoot Method	Traditional MultiBoot+Fallback	Configure, Lookahead, and MultiBoot(1)	Simple Overwrite of Single Image
Supports fallback configuration of a golden bitstream for failed remote updates	Yes	Yes	Yes	No
Supports single FPGA configuration	Yes(2)	Yes	Yes	Yes
Supports single stacked silicon interconnect (SSI) FPGA configuration	Yes(2)	Yes	Yes	Yes
Supports daisy-chain FPGA configuration	Yes(2)	No	Yes	Yes
Configuration time for good update image (As an approximate multiple of the time for a standard configuration procedure)	1X + small constant to read through QuickBoot header	1X	3X: 1. Load first stage bitstream 2. Read/verify update bitstream 3. Load update design	1X standard bitstream load time
Configuration time for fallback image (i.e., when the update image is bad) (As an approximate multiple of the time for a standard configuration procedure)	1X + small constant to read through QuickBoot header	2X: 1. Attempt load of update bitstream. 2. Fallback and load golden bitstream	3X: 1. Load first stage bitstream 2. Read/verify update bitstream 3. Load fallback design	N/A
Requires unique bitstream generation options for golden bitstream versus update bitstream	No	Yes	No	N/A
Update error detection responsibility	Error detection during programming, with automatic error when programming is incomplete	Error detection during configuration	Error detection during first stage lookahead run-time	None
Supports update of the golden image	Yes(3)	No	Yes	No

Notes:
1. The lookahead method first loads and runs an initial bitstream that looks ahead to validate other bitstreams in flash and then chooses to reconfigure from another bitstream.
2. FPGA support is limited by the capacity of the flash device.
3. See Remote Update for the Golden Image, page 38.

Reference Design Files

The reference design files for this application note can be downloaded from:

https://secure.xilinx.com/webreg/clickthrough.do?cid=323710

Table 16 gives the reference design matrix.

Table 16: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Randal Kuramoto
Target devices (stepping level, ES, production, speed grades)	7 Series FPGAs
Source code provided	Yes
Source code format	VHDL
Design uses code and IP from existing Xilinx application note and reference designs, CORE Generator software, or third party	Yes, *IEEE 802.3 Cyclic Redundancy Check* [Ref 4]
Simulation	
Functional simulation performed	Yes
Timing simulation performed	No
Test bench used for functional and timing simulations	Yes
Test bench format	VHDL
Simulator software/version used	ISim 14.4
SPICE/IBIS simulations	N/A
Implementation	
Synthesis software tools/version used	ISE® Design Suite 14.4 or higher
Implementation software tools/versions used	ISE Design Suite 14.4 or higher
Static timing analysis performed	Yes
Hardware Verification	
Hardware verified	Yes
Hardware platform used for verification	KC705 board

Appendix: Advanced Applications

Distinguishing the Loaded Golden or Update Image

When the system has a choice of loading a golden image or update image, one concern of such a remote update system is determining the version of the design that is actually loaded and running. *Bitstream Identification with USR_ACCESS* [Ref 7] can be used to place a special identifier in the bitstreams. Two options are:

1. Reserve one fixed known identifier value for the golden image bitstream. All later bitstreams must have a different identifier value. If the identifier value matches the reserved golden identifier value, the loaded image is the golden image.
2. Use an incrementing or date-value identifier for determining whether an incoming design is newer than the running design.

Quicker Start for QuickBoot Configuration

The initial start time can be reduced through modifications of the QuickBoot header. See QuickBoot Configuration Time, page 12 for the initial configuration delay introduced by the basic QuickBoot solution. The modifications are:

- Reserve the first erasable segment for an initial warm boot jump sequence that jumps directly to the critical switch word. This bypasses the sequential read through the first erasable segment.
- Reserve the second erasable segment for the critical switch word and place the critical switch word at the end of this segment.
- The QuickBoot flash programmer module must be modified to set the cAddrSwitchWord location to correspond to the modified switch word address.
- The Perl script must be given the -quickjump option such that it produces MCS files with the modified QuickBoot Header.

The -quickjump modification can reduce the initial start time to the time the FPGA takes to execute a warm boot jump sequence. The warm boot jump sequence can take up to T_{PL} time. See the *Kintex-7 FPGAs Data Sheet* [Ref 8] for the T_{PL} time.

Remote Update for the Golden Image

The standard QuickBoot solution keeps a golden image that is constant throughout the life of the system. However, it is possible to cascade the QuickBoot solution in a way that enables the golden image to also be updated. Figure 5 illustrates how in place of the golden image of the standard QuickBoot memory map, a secondary QuickBoot structure is placed. Figure 8 shows the cascaded QuickBoot solution that supports update to the golden image.

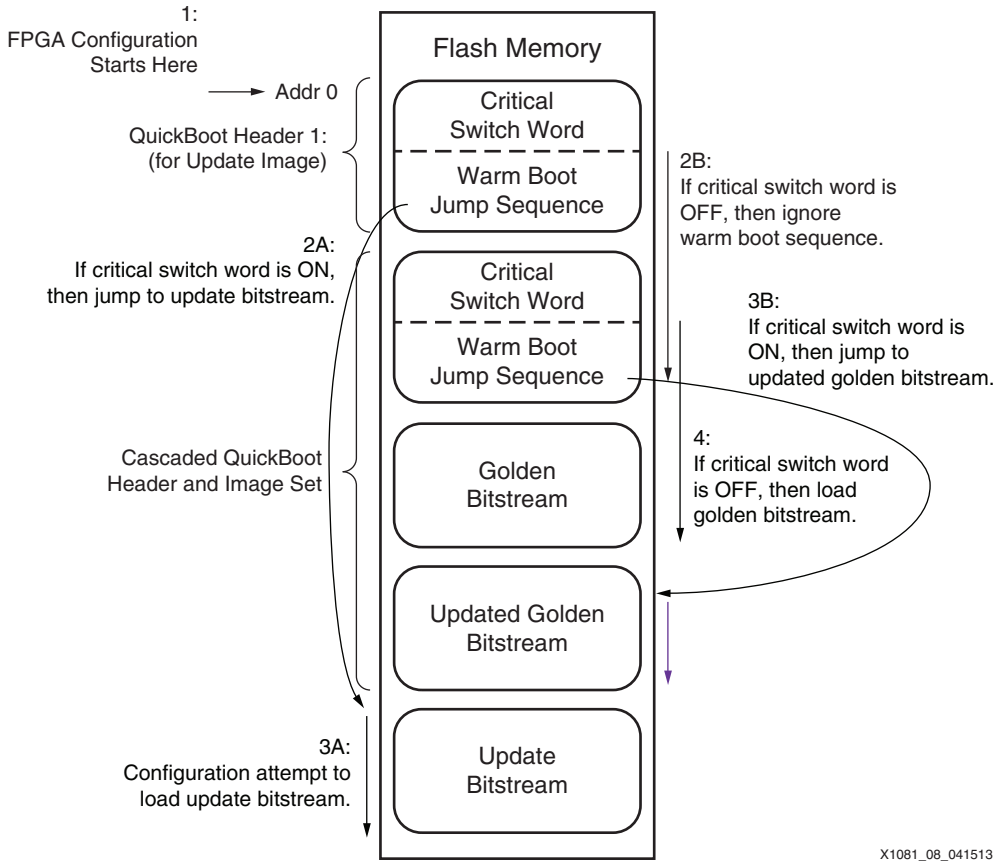

Figure 8: **Cascaded QuickBoot for an Updatable Golden Image**

The base reference design is hard-coded to ensure only one specific switch word location and only one specific update image area is modified during the update process. By default, the base reference design does not support update of multiple update images.

Pre-Verified MultiBoot Images

The lookahead technique loads an initial FPGA design. The initial FPGA design looks ahead to read and verify other FPGA bitstreams that can be loaded. The QuickBoot CRC verification technique can be used to perform the lookahead verify. The Perl script can generate images with the CRC32. The QuickBoot flash programmer modules have a verify-only option to compute the CRC and validate images in flash memory.

References

This document uses the following references:

1. *7 Series FPGAs Configuration User Guide* (UG470)
2. BPI flash, Micron Parallel NOR P30 Flash Memory data sheet http://www.micron.com/products/nor-flash/parallel-nor-flash#p30-p33
3. SPI flash, Micron Serial NOR N25Q Flash Memory data sheet http://www.micron.com/products/nor-flash/serial-nor-flash#n25q
4. *IEEE 802.3 Cyclic Redundancy Check* (XAPP209)
5. iMPACT Flash Device Support Table http://www.xilinx.com/cgi-bin/docs/rdoc?v=latest_ise;d=isehelp_start.htm;a=pim_c_introduction_indirect_programming.htm
6. *KC705 Evaluation Board for the Kintex-7 FPGA User Guide* (UG810)
7. *Bitstream Identification with USR_ACCESS* (XAPP497)
8. *Kintex-7 FPGAs Data Sheet: DC and AC Switching Characteristics* (DS182)

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
05/09/2013	1.0	Initial Xilinx release.
06/07/2013	1.1	Updated reference design filenames throughout. Added Note 1 to Table 2. Added Bitstream Separation for MMCM/PLL Lock or DCI Wait. Replaced “SPI flash” with “BPI flash” in step 9a of KC705 Board Demonstration for QuickBoot Using BPI Flash. Updated second bullet in KC705 Board Demonstration for QuickBoot Using SPI Flash.
01/24/2014	1.2	Added “SPI” to Figure 6 title. In Table 5, updated column heading to “Description and Connection” and updated outErrorTimeOut description. Updated column heading to “Features” in Table 15.
03/18/2014	1.3	Updated description of Line 8 in Table 3.

Notice of Disclaimer

Automotive Applications Disclaimer

XILINX PRODUCTS ARE NOT DESIGNED OR INTENDED TO BE FAIL-SAFE, OR FOR USE IN ANY APPLICATION REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS APPLICATIONS RELATED TO: (I) THE DEPLOYMENT OF AIRBAGS, (II) CONTROL OF A VEHICLE, UNLESS THERE IS A FAIL-SAFE OR REDUNDANCY FEATURE (WHICH DOES NOT INCLUDE USE OF SOFTWARE IN THE XILINX DEVICE TO IMPLEMENT THE REDUNDANCY) AND A WARNING SIGNAL UPON FAILURE TO THE OPERATOR, OR (III) USES THAT COULD LEAD TO DEATH OR PERSONAL INJURY. CUSTOMER ASSUMES THE SOLE RISK AND LIABILITY OF ANY USE OF XILINX PRODUCTS IN SUCH APPLICATIONS.