Speedster7t Configuration User Guide (UG094)

Speedster FPGAs

Achronix
Data Acceleration

Copyrights, Trademarks and Disclaimers

Copyright © 2026 Achronix Semiconductor Corporation. All rights reserved. Achronix, Speedcore, Speedster, and ACE are trademarks of Achronix Semiconductor Corporation in the U.S. and/or other countries All other trademarks are the property of their respective owners. All specifications subject to change without notice.

Notice of Disclaimer

The information given in this document is believed to be accurate and reliable. However, Achronix Semiconductor Corporation does not give any representations or warranties as to the completeness or accuracy of such information and shall have no liability for the use of the information contained herein. Achronix Semiconductor Corporation reserves the right to make changes to this document and the information contained herein at any time and without notice. All Achronix trademarks, registered trademarks, disclaimers and patents are listed at http://www.achronix.com/legal.

Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

Website: www.achronix.com
E-mail : info@achronix.com

Table of Contents

Chapter 1 : Overview 1

Chapter 2 : Interface Performance 3

Chapter 3 : Bitstream Programming Modes for Speedster7t FPGAs 4

Bitstream Programming Time 5

Bitstream Programming Via CPU 5

CPU Mode Bitstream Programming Flow 5

Generating the CPU Mode Bitstream Files From ACE 5

How To Use the ACE-Generated CPU Bitstream File 6

CPU Mode Hardware Interface 7

Bitstream Programming via Flash Memories 9

Serial Flash Bitstream Programming Flow 9

Generating the Serial Flash Bitstream Files from ACE 9

Using ACE-Generated Serial Flash Bitstream Files 11

spi::program_bitstream Command 11

spi::program_all_bitstreams Command 12

Reading Back Data Stored In Flash 13

spi::read_bitstream Command 13

Serial Flash Hardware Interface 14

Flash Interface 15

Flash Device Configurations 16

1D Configuration *16*

4D Configuration *17*

Addressing Modes and Memory Organization 19

Address Range *20*

Flash Configuration Header (Page0 Header) *20*

Flash Configuration Protocol 21

Flash Modes 22

SPI Mode *23*

Dual Mode *23*

Quad Mode *24*

Octa Mode *24*

Flash Memory Size Requirements 25

Flash Configuration Using FTDI 25

FTDI Board-Level Device Connections *25*

Bitstream Programming via JTAG 28

Generating the JTAG Bitstream Files From ACE 28

How To Use the ACE-Generated JTAG Bitstream Files 28

JTAG Programming using the ACE Download View 28

ACE JTAG Connection Preference Page *28*

ACE JTAG Download View *29*

JTAG Programming using the ACE Flow Steps 31

JTAG Programming using the Tcl Library API 32

Variables Under ACE Tcl Console *33*

Tcl Command Tables *33*

Programming the Board Using JTAG and Read/Write Registers *40*

JTAG Hardware Overview 41

Introduction *41*

JTAG Configuration Overview *41*

JTAG Instructions *45*

JTAG Configuration Using FTDI 46

Overview *46*

FTDI Board-Level Device Connections *49*

FTDI Interface in ACE *54*

Programming Speeds and Requirements *55*

Known Device Limitations *56*

Software and Driver Install for FTDI *56*

Connecting the FTDI Device *58*

JTAG Configuration Using the Bitporter2 Pod 59

Software and Driver Install for Bitporter2 ..... *60*

Connecting the Bitporter2 Pod *62*

Bitstream Programming via PCIe 65

Design Requirements for Programming via PCIe 65

How to Generate the PCIe Bitstream Files with ACE 67

Bitstream Generation – Additional Outputs Section *67*

Bitstream Generation – Two Stage Programming Section *67*

How to Program a Device with the PCIe Bitstream Files 68

Standalone PCIe Programming Utility *68*

Programmatic Access Using Achronix SDK Library Functions *68*

Chapter 4 : FPGA Configuration Unit (FCU) 69

Overview 69

Configuration Pin Tables 70

Chapter 5 : Bitstream Generation Software Support in ACE 75

Bitstream Generation 75
Bitstream Output File Formats 75
Serial Flash Configuration Options 76
Encryption Options 78
Two-Stage Configuration Option 79
Partial Reconfiguration Configuration Options 79
FCU Configuration Options 80
Bitstream ID Configuration Options 81
CMEM Error Injection Options 81

Chapter 6 : Configuration Sequence and Power-Up 83

Device Power-Up 83

Chapter 7 : Configuration Error Correction and SEU Mitigation 85

Configuration Memory Architecture and Addressing 86
Error Injection and Reporting 88
ACE Implementation Options 88
Bitstream Error Injection 89
Bitstream Single-Bit Error Injection Example 90
Bitstream Dual-Bit Error Injection Example 91
Scrubbing Reset 92
Scrubbing FCU Registers 92

Chapter 8 : Design Security for Speedster7t FPGA 94

Bitstream Authentication 94

Bitstream Encryption 94
Generating Encrypted Bitstreams 95
Encrypting a Speedster7t AC7t1500 Bitstream 96
Using the ACE GUI 96
Using Tcl Commands 97
Encrypting a Speed AC7t1400 Bitstream 98
Using ACE GUI 98
Using Tcl Commands 99
Hardware Security 99
Physically Unclonable Function 100
Key Derivation Function 100
Rules for Encryption 100
Security Fuses 101
Fuses Set at Manufacturing 101
Fuses Set By Customer 101
Default Keys 102
Generating a Public and Private Key Pair on Speedster7t AC7t1500 102
Programming the Encryption Keys Into Speedster7t AC7t1500 eFuses 103
Programming the Encryption Keys into Speedster7t AC7t1400 eFuses 103
Loading Encrypted Bitstreams 104
Programming an AC7t1500 Encrypted Bitstream 104
Programming an AC7t1400 Encrypted Bitstream 105
Device DNA 106
ACE Placements to Read Device DNA 106
Top-Level Port Declaration 106
PDC 107

Chapter 9 : Partial Reconfiguration 108

Design Considerations 109

Using Partial Reconfiguration 110

ACE Implementation Options 110

Partial Reconfiguration Steps 110

Chapter 10 : Revision History 111

Chapter 1 : Overview

At startup, Speedster®7t FPGAs require configuration via a bitstream. This bitstream can be programmed through one of four available interfaces in the FPGA configuration unit (FCU), the logic controlling the configuration process.

The term FPGA configuration unit (FCU) refers to logic that controls the configuration (bitstream programming) process of the Speedster7t FPGA. This logic is responsible for the following:

- Receiving data on a variety of external interfaces (depending on the selected programming mode)
- Decoding instructions
- Sending configuration bit values to the appropriate destination (e.g., core configuration memory, the core boundary ring configuration memory, FCU registers, etc.)
- Controls the startup and shutdown sequences that drive resets to the on-chip logic
- Bitstream CRC checks
- SEU mitigation with CMEM scrubbing
- Bitstream Encryption Security
- Any core-level housekeeping that occurs on the de-assertion of reset (i.e., clearing of configuration memory)

Data from the configuration pins is brought into the FCU located in the core boundary logic. Depending on the configuration mode, this data passes through one of four interfaces and is then provided to the control logic and state machines in the FCU. At this point, the data bus is standardized to a configuration mode independent common interface. Data is processed and propagated to the configuration registers in the core boundary ring, to the core configuration memory, or to the hard IP blocks in the FPGA I/O ring.

When all of the configuration bits are successfully loaded, the FCU transitions the Speedster7t FPGA into user mode, allowing full operation of the custom design.

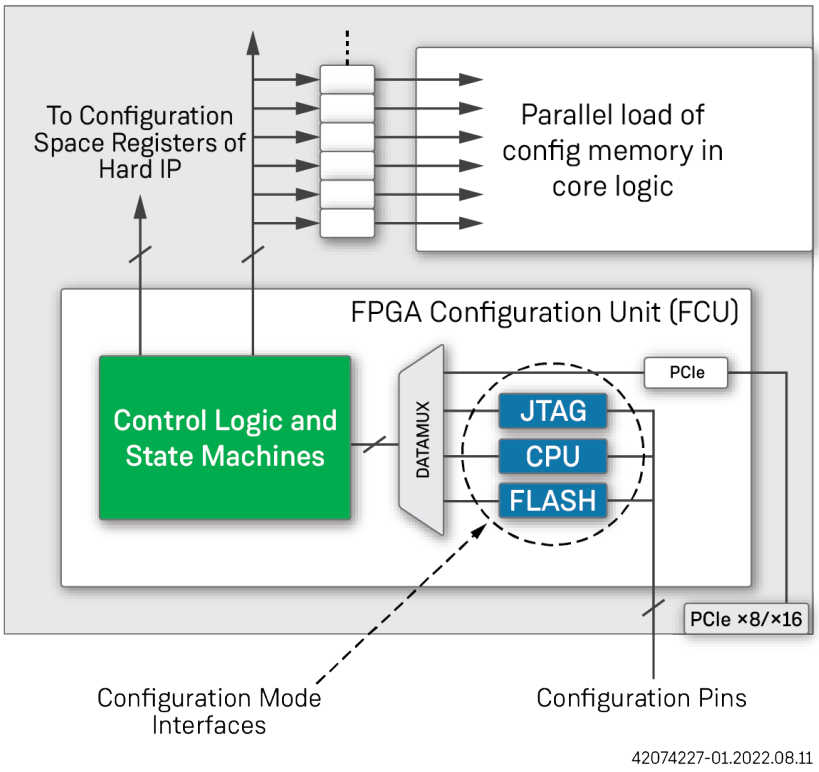


Figure 1 • Speedster7t Configuration Block

Chapter 2 : Interface Performance

The following table lists the various configuration interfaces supported by the Speedster7t FPGA and their corresponding maximum operating frequency.

Table 1 • Configuration Modes and Maximum Frequencies

Configuration Mode	Maximum Frequency
JTAG	50 MHz
CPU	250 MHz
Serial flash	62.5 MHz

All of the programming modes and interfaces are capable of running up to 250 MHz at the configuration pins. The FCU and all associated circuitry are also capable of running up to 250 MHz. Since the internal data bus in the FCU is 128 bits wide, and in most configuration modes, the data pin count is less than 128, the incoming data stream goes through a gearbox to reduce the throughput. This configuration ensures that the internal programming circuitry runs at less than 250 MHz to process the incoming data stream. In the widest data mode (CPU ×128), the gearbox is bypassed and the entire configuration interface can run at the full 250 MHz bandwidth. Depending on the mode and configuration data width, the total bandwidth varies, and the programming time changes accordingly.

> **Note**
> CPU ×128 mode is primarily for ATE use and not a recommended mode for design configuration.

Chapter 3 : Bitstream Programming Modes for Speedster7t FPGAs

Speedster7t FPGAs support four configuration modes:

- Flash
- JTAG
- CPU
- PCI Express

The selection between these modes is controlled by setting the FCU_CONFIG_MODESEL pins to the values shown in the following table. Both JTAG and PCIe modes are independent of the FCU_CONFIG_MODESEL pin setting. The JTAG mode overrides all other configuration modes except PCIe until disabled.

Table 2 • Pin Settings for Various Configuration Modes

Configuration Mode	Data Width	FCU_CONFIG_ MODESEL [3:0]	FCU_CONFIG_ SYSCLK_BYPASS (1)	FCU_CONFIG_ CLKSEL (1)
PCIe	-	XXXX	X	0
JTAG (2) (3)	-	XXXX	X	0/1
Flash single device (1D) (4)	1 (SPI)	0001	0/1	0
	2 (Dual)	1000		
	4 (Quad)	1010		
	8 (Octa)	1100		
Flash four devices (4D) (4)	1 (SPI)	0010		
	2 (Dual)	1001		
	4 (Quad)	1011		
	8 (Octa)	1101		
CPU	1	0011	1	0
	8	0100		
	16	0101		
	32	0110		
	128 (5)	0111		

Configuration Mode	Data Width	FCU_CONFIG_ MODESEL [3:0]	FCU_CONFIG_ SYSCLK_BYPASS (1)	FCU_CONFIG_ CLKSEL (1)

Table Notes

1. These straps select the configuration clock source:
   0 – on-chip oscillator clock
   1 – `FCU_CPU_CLK`
2. Always active. Enabled in the JTAG TAP controller.
3. If `FCU_CONFIG_MODELSEL[3:0]` pins are set such that flash or CPU configuration mode is selected, the JTAG override should be issued after flash programming has completed or the CPU mode interface is inactive.
4. In 1D mode, the flash bitstream is downloaded from one flash device. In 4D mode, the flash bitstream is downloaded from four flash devices.
5. Speedster7t FPGAs have 32 dedicated data I/O pins for the CPU interface supporting up to a ×32 interface. For ×128 mode, the upper 96 pins are shared with the DDR4 interface.

   Caution!
   CPU ×128 is primarily for ATE use and not a recommended mode for design configuration.

Bitstream Programming Time

Bitstream programming time is determined by the following formula in seconds:

(Total number of programming bits) / (programming data-width × clock frequency)

Note

When programming via JTAG, the clock frequency applied to the formula should be the frequency of JTAG_TCK.

Bitstream Programming Via CPU

CPU Mode Bitstream Programming Flow

Generating the CPU Mode Bitstream Files From ACE

1. In ACE, select the CPU mode additional output.
2. Select the CPU bus width to generate `.cpu` and `_cpu.bin` files for use in CPU programming mode as shown in the following example.
3. Run the Generate Bitstream flow step:

```
> run -step write_bitstream
```

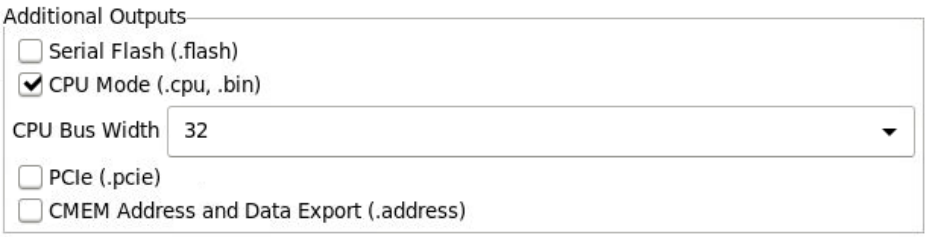


Figure 2 • ACE Additional Output Options Dialog

Table 3 • Bitstream Generation Implementation Options For CPU Mode

Option	ACE impl_option	Description
CPU Mode (`.cpu`)	`bitstream_output_cpu`	Enables generation of an additional CPU mode (`.cpu`) formatted output file with the same name as the `.hex` file. The file contains hexadecimal-formatted data organized with "CPU bus width" number of bits per file line. File data is sent to the FCU CPU interface line by line (one line per clock cycle), where the left-most bit on each line is the MSB and the right-most bit is the LSB. In simulation, this file can be loaded using the `readmemh` function. For convenience, an additional binary representation of the CPU mode output (`_cpu.bin`) file is written with the same name as the `.hex` file. It contains the same data in the same bit order as the `.cpu` file but in a binary format with no new-lines.
CPU Bus Width	`bitstream_output_cpu_width`	Controls the bit width of the CPU-mode formatted output file. When using the CPU interface in ×8 mode, set this value to 8. If using the CPU interface in ×32 mode, set this to 32. The value determines how many bitstream bits are printed per line in the `.cpu` output file. The bit sequence required by the FCU (and output in the generated bitstream file) might be different for each CPU bus width setting. It is important to set this option to match the actual CPU hardware interface width.

How To Use the ACE-Generated CPU Bitstream File

There are two different CPU mode output file formats generated by ACE:

1. The `*.cpu` file format. This file uses hexadecimal formatting and contains one CPU write per data line. If CPU bus width is set to 8, each line in the `*.cpu` file contains eight bits of hex-formatted data in big-endian format.
2. The `*_cpu.bin` format. This file is a pure binary formatted file (with no newlines) and is formatted according to the CPU bus width in little-endian format. If CPU bus width is set to 8, every eight bits of binary file data represents the eight bits of data needed for each sequential CPU write.

To use either file format to program the ACE-generated bitstream into the FPGA, simply loop over the bitstream file from start to end and perform a CPU write operation with (CPU bus width) bits of data from the file on the `FCU_CPU_DQ_IN_OUT` bus. Details on the hardware interface follow.

CPU Mode Hardware Interface

In CPU configuration mode, an external CPU controls the programming operations to the Speedster7t FPGA and offers a high-speed method for loading configuration data. Depending on the setting of the FCU_CONFIG_MODESEL pins, the CPU mode can be either a 1-, 8-, 16- or 32-bit wide parallel interface (128-bit wide is available for ATE test only). This interface is clocked using FCU_CPU_CLK with chip select support to indicate valid data. This mode supports a maximum clock rate of 250 MHz.

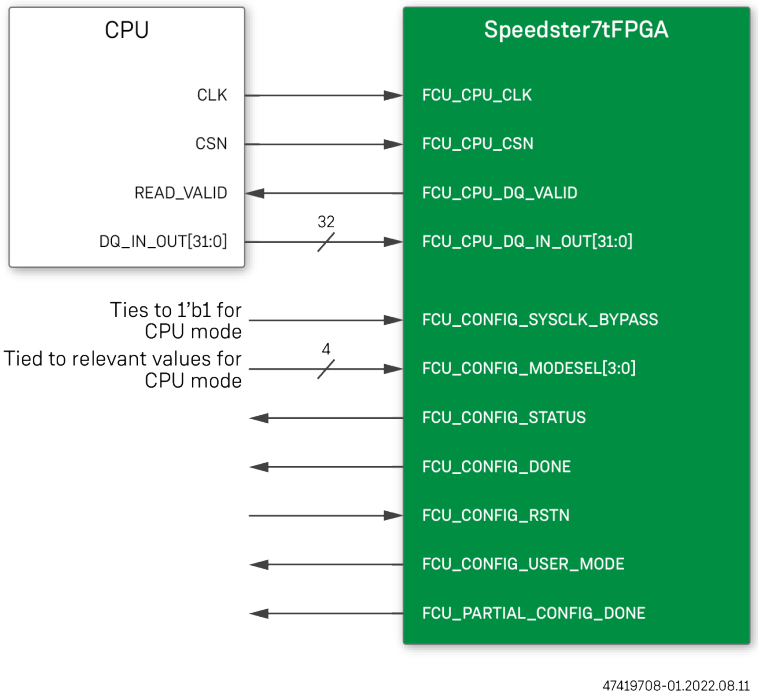


Figure 3 • External CPU Connectivity to a Speedster7t FPGA

> **Note**
>
> The CPU only needs to connect to the first 1, 8, 16 or 32 bits of FCU_CPU_DQ_IN_OUT depending on the CPU mode selected. All unused signals should be tied to weak pull-up resistors.

As described in the Configuration Sequence and Power-up (page 83) section, the configuration mode-specific operations occur between the release of FCU_CONFIG_STATUS (indicating that the configuration memory has been cleared and that the Speedster7t FPGA is ready to accept bitstream data) and the assertion of FCU_CONFIG_DONE (stating completion of configuration). The following example waveform for CPU×8 mode illustrates the sequence of events, clocking and control signal states needed for successful configuration in CPU mode:

1. After FCU_CONFIG_RSTN is de-asserted, FCU_CPU_CLK must continue to cycle to ensure that the FPGA cycles through the FCU states and the configuration memory is cleared. At that point, FCU_CONFIG_STATUS is driven high.

> **Note**
>
> After power-up, FCU_CONFIG_RSTN de-assertion must be done at least 1 ms after all power supplies have settled to their required DC voltage values. FCU_CONFIG_RSTN signal is used to initialize all the IP in the FPGA This wait time of 1 ms ensures that all IP are supplied with the right voltage values and see the FCU_CONFIG_RSTN de-assertion.

2. After at least 5 clock cycles of FCU_CONFIG_STATUS being driven high, FCU_CPU_CSN must be pulled low to begin writing the bitstream data into the Speedster7t FPGA. When the last set of data is written into the Speedster7t FPGA, FCU_CPU_CSN is pulled back high.
3. When FCU_CPU_CSN is pulled high, FCU_CPU_CLK must continue being clocked. When the FCU cycles through all of the configuration states, FCU_CONFIG_DONE is driven high to indicate that the Speedster7t FPGA was successfully programmed.
4. As the FCU_CPU_CLK toggles, the FCU cycles through its states to move the Speedster7t FPGA from programming mode into user mode, taking the fabric out of reset and performing operations to enable user-mode functions for all parts of the core. The FCU_CONFIG_USER_MODE signal is asserted to indicate when the Speedster7t FPGA has successfully transitioned into user mode.

At any point during the configuration, if FCU_CPU_CSN is asserted low, the FCU_CPU_DQ_IN_OUT bus should contain valid data or NOPs. A NOP is represented by logic 0 on all data pins. During this time, the FCU_CPU_DQ_VALID pin should be held low, indicating that the data pins can be driven by a device external to the FPGA as mentioned in **Configuration Pin Tables** (page 70). If FCU_CPU_CSN is high, the data on FCU_CPU_DQ_IN_OUT is ignored. When the bitstream is programmed, FCU_CPU_CSN can be held low while sending NOPs to the Speedster7t FPGA. This action does not affect the assertion of FCU_CONFIG_DONE or FCU_CONFIG_USER_MODE signals.

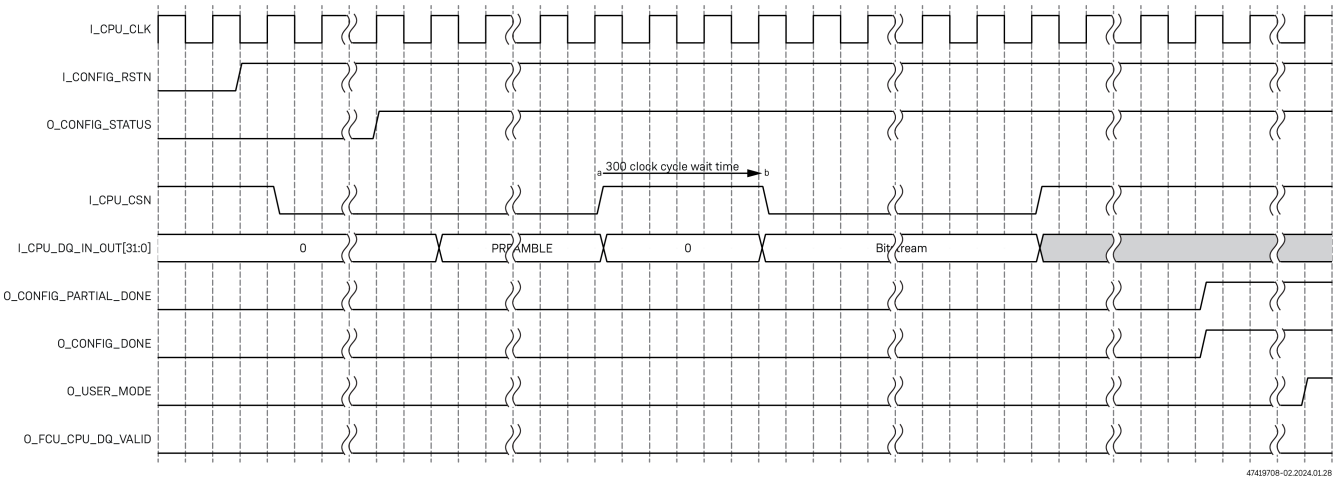


Figure 4 • Clocking and Control Signals for Successful Configuration

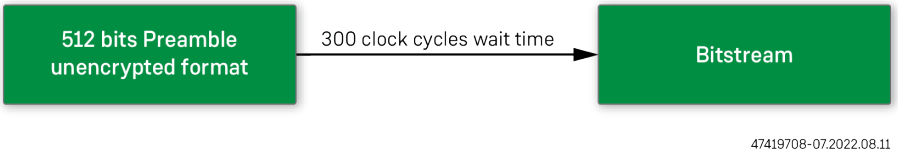


Figure 5 • Bitstream Data Sequence For Unencrypted Bitstream

> **Note**
>
> During the 300 clock cycle wait time, CPU_CSN is pulled high for encrypted bitstreams.
>
> When programming an encrypted bitstream, there are additional wait clock cycle requirements. Please refer to the details in the Design Security for Speedster7t FPGA (page 94) section of this document.

Bitstream Programming via Flash Memories

> **Caution!**
>
> Speedster7t FPGAs can interface to serial NOR flash devices only. Parallel NOR, NAND or other flash variants are *not* supported.

Serial Flash Bitstream Programming Flow

Generating the Serial Flash Bitstream Files from ACE

1. In ACE, select the Serial Flash additional output to generate the `.flash` and `_page0.flash` files for use in Serial Flash programming mode as shown in the following example.

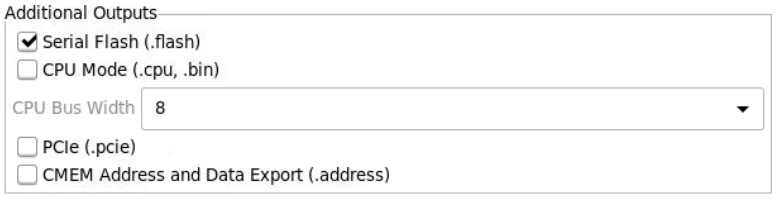


Figure 6 • ACE Additional Output Options Dialog

2. Run the Generate Bitstream flow step:

```
> run -step write_bitstream
```

Table 4 • Bitstream Generation Implementation Options For Serial Flash Mode

Option	ACE impl_option	Description
Serial Flash (`.flash`)	`bitstream_output_flash`	Enables generation of the serial flash-formatted `.flash` output file and the page0 header `_page0.flash` files in addition to, and with the same name as, the default `.hex` file. The `.flash` file contains a binary image that can be directly loaded into a single serial flash memory.

The following bitstream implementation options must be set correctly for the board serial flash hardware configuration.

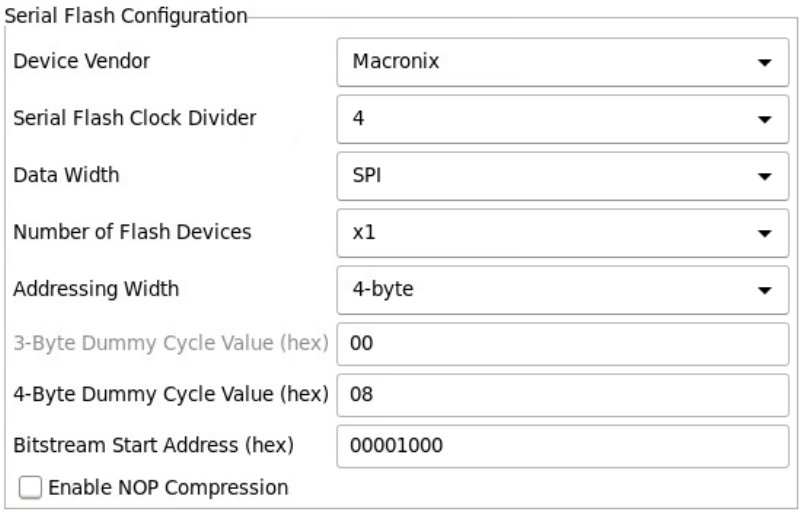


Figure 7 • ACE Serial Flash Configuration Options Dialog

Table 5 • Bitstream Generation Serial Flash Configuration Options

Option	ACE impl_option	Description
Device Vendor	`bitstream_page0_vendor`	Selects the flash device vendor. Allowed values: **Macronix** (`0`) **Micron** (`1`)
Serial Flash Clock Divider	`bitstream_page0_sf_clock_div`	Selects serial flash clock divider. Allowed values: **4** **8**
Data Width	`bitstream_page0_data_width`	Selects flash data readback width. Allowed values: **SPI** (`0`) **DUAL** (`1`) **QUAD** (`2`) **OCT** (`3`)

Option	ACE impl_option	Description
Number of Flash Devices	bitstream_page0_num_devices	Selects number of devices based on targeted x1 or x4 PROM. Allowed values: **x1** (0) **x4** (1)
Addressing Width	bitstream_page0_addr_width	Selects 3-byte or 4-byte addressing mode to support flash devices >1Gb. Allowed values: **3-byte** (0) **4-byte** (1)
3-Byte Dummy Cycle Value (hex)	bitstream_page0_dummy_cycle_3byte	Specifies the 3-byte addressing dummy cycle value. The default value is **00** and varies by device vendor. Must be specified as a 2-character hex value.
4-Byte Dummy Cycle Value (hex)	bitstream_page0_dummy_cycle_4byte	Specifies the 4-byte addressing dummy cycle value. The default value is **08** and varies by device vendor. Must be specified as a 2-character hex value.
Bitstream Start Address	bitstream_page0_start_addr	Specifies the bitstream start address. Should be a non-zero multiple of 4096. Must be specified as an 8-character hex value.
Enable NOP Compression	bitstream_page0_compress_nops	When unchecked (0), the *.flash file for I/O Ring programming is similar to other programming modes (CPU, JTAG, Hex, etc). When checked, the *.flash file bitstream contents are compressed, to help meet the 100ms PCIe link-up time. This results in a different bitstream for serial flash, which is dependent on the overall FCU data width (Number of Devices x Device Data Width).

Using ACE-Generated Serial Flash Bitstream Files

The flash device is programmed using Tcl with .flash and _page0.flash files. The spi::program_all_bitstreams ACE Tcl command is the recommended serial flash bitstream programming command because it automatically determines the offset and is useful for two-stage bitstream programming over PCIe as discussed in the **Bitstream Programming via PCIe** (page 65) section. If using spi::program_bitstream, the command must be issued twice to first program the stage0 header flash file followed by programming the stage1 or full flash file at the specified offset.

spi::program_bitstream Command

Example

```
spi::program_bitstream <board_config> <flash_file> <number_of_proms> -device_id
<pod_name/FTDI_device> -offset <int> -hz <int>
```

Arguments

Table 6 • spi::program_bitstream Command Arguments

Argument	Default	Description
<board_config>	–	The board or part name of the targeted Achronix device (e.g., MEP/FT2232H or VectorPath/FT4232H).
<flash_file>	–	The bitstream flash file to program from.
<number_of_proms>	–	The number of PROM devices: • **1** - single PROM • **4** - ×4 PROM
-device_id	–	Argument followed by the Bitporter 2 pod name or FTDI device connected to the FPGA board. Use the JTAG Tcl command, jtag::get_connected_devices, to query available FTDI devices when more than one device is connected.
-offset	0	Argument followed by the byte offset of the flash file within all of the PROMs. Must be specified when programming the .flash or _stage0.flash file.
-hz	30000	Lower SPI clock frequency to the value specified in this argument in KHz (optional) where 30000 is the maximum value.

spi::program_all_bitstreams Command

Example

```
spi::program_all_bitstreams <board_config> <page0_file> <flash_file> <number_of_proms>
-device_id <pod_name> -hz <int> -stage_1_header_file <file_name>
```

Arguments

Table 7 • spi::program_all_bitstreams Command Arguments

Argument	Default	Description
<board_config>	–	The board or part name of the targeted Achronix device (e.g., MEP/FT2232H or VectorPath/FT4232H).
<page0_file>	–	The bitstream page0 flash file to program from: *_page0.flash for full flash file, or *_stage0_page0.flash for two-stage bitstream programming.

Argument	Default	Description
<flash_file>	–	The bitstream flash file name: either .flash for full flash file, or *_stage0.flash for two-stage bitstream programming.
<number_of_proms>	–	The number of PROM devices: • **1** - single PROM • **4** - ×4 PROM
-device_id	–	Argument followed by the Bitporter 2 pod name or FTDI device connected to the FPGA board. Use the JTAG Tcl command, jtag::get_connected_devices, to query available FTDI devices when more than one device is connected.
-hz	30000	Lower SPI clock frequency to the value specified in this argument in KHz (optional) where 30000 is the maximum value.
-stage_1_header_file	–	Argument followed by the ACE-generated bitstream binary stage1 header flash file to program from: *_stage1_header.flash (used only with two-stage encrypted bitstream programming over PCIe).

Flash Programming Example

```
cmd> spi::program_bitstream config1 <flash_file> <number_of_proms> -offset 4096
-device_id <pod_name>
Device:    config1  bringup board
Successfully opened SPI device: <pod_name>
  205056 of 1333712 blocks written
  410112 of 1333712 blocks written
  615168 of 1333712 blocks written
  820224 of 1333712 blocks written
  1025280 of 1333712 blocks written
  1230336 of 1333712 blocks written
Successfully programmed PROM devices with 1333968 bytes of data
```

Reading Back Data Stored In Flash

The contents stored in flash may be read back in ACE using the spi::read_bitstream Tcl command which reads back the bitstream from the connected PROMs and outputs the contents to a file.

spi::read_bitstream Command

Example

```
spi::read_bitstream <board_config> <flash_file> <number_of_proms> <number_of_bytes>
-device_id <pod_name> -offset <int>
```

Arguments

Table 8 • spi::read_bitstream Command Arguments

Argument	Default	Description
<board_config>	–	The board or part name of the targeted Achronix device (e.g., MEP/FT2232H or VectorPath/FT4232H).
<flash_file>	–	The bitstream binary flash file to be written. A single file is written for either a single PROM or a x4 PROM configuration.
<number_of_proms>	–	The number of PROM devices: • **1** - single PROM • **4** - ×4 PROM
<number_of_bytes>	–	The total number of bytes to read back from all PROM devices.
-device_id	–	Argument followed by the Bitporter 2 pod name or FTDI device connected to the FPGA board. Use the JTAG Tcl command, jtag::get_connected_devices, to query available FTDI devices when more than one device is connected.
-offset	0	Argument followed by the byte offset of the flash file within all of the PROMs.

Flash Readback Example

```
cmd> spi::read_bitstream config1 flash_readback 1 1333968 -device_id <pod_name>
Device:    config1  bringup board
Successfully opened SPI device: <pod_name>
  Prom 0 : file flash_readback_0
  205056 of 1333968 blocks read
  410112 of 1333968 blocks read
  615168 of 1333968 blocks read
  820224 of 1333968 blocks read
  1025280 of 1333968 blocks read
  1230336 of 1333968 blocks read
Successfully read back SPI device 0, into file flash_readback
```

Serial Flash Hardware Interface

Flash programming mode allows configuring Speedster7t FPGAs with flash memories. In this mode, the FPGA controls the programming operations and supplies the clock to the flash memory.

The clock supplied from the FPGA (on the FCU_FLASH_SCK pin) to the attached flash device(s) can be driven either by the FCU_CPU_CLK or the on-chip oscillator clock depending on the configuration options selected as described in the Bitstream Programming Modes for Speedster7t FPGAs (page 4) chapter. The frequency of this clock can be selected from one of the variants of the clock sources arriving at the FCU: the divide-by-4 or divide-by-8. This

selection is configured using the **Serial Flash Clock Divider** drop-down menu in the "Bitstream Generation Implementation Options" section of the ACE GUI. This setting ensures that only the flash state machine runs at the slower frequency. All other FCU and ACB logic continues to operate at the original input clock frequency. Details on downloading a bitstream into the flash devices via FTDI is presented in the Flash Configuration Using FTDI (page 25) chapter. It is very critical that the FTDI chip is used in combination with the FCU to write data to the flash devices. Therefore, all boards that intend to make use of flash configuration must have this component mounted accordingly. The following diagram details the appropriate connections needed to write to the flash devices and to subsequently program the Speedster7t FPGA.

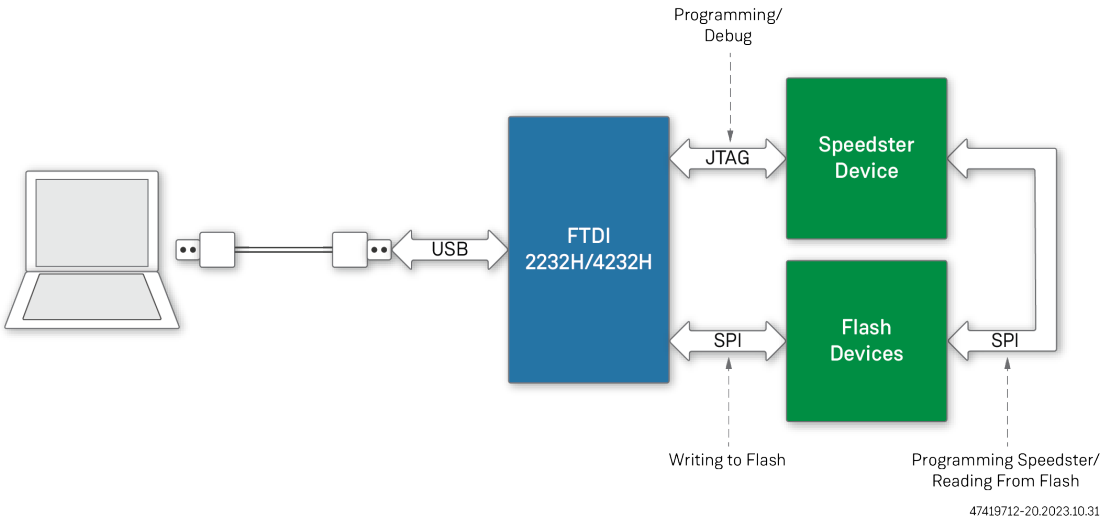


Figure 8 • FTDI Cable Connection Detail

> **Note**
>
> At power-on, the device defaults to the divide-by-8 setting. The FCU then sets the appropriate configuration register to control the clock divider based on the user selection in ACE. The transition from a divide-by-8 clock to any other selected clock frequency is glitch free. Also, flash write is always SPI only while read can be in SPI, DUAL, QUAD or OCTA mode as summarized in the following table.

Flash Interface

The configuration block is equipped with a flash interface that supports programming the FPGA from flash memory. A bitstream is assumed to be loaded into the flash memory using an external SPI interface. Flash registers within the configuration block assist with this process (refer to Registers and Addressing (page 0)). The complete feature list of the flash interface is described in the following table.

Table 9 • Flash Features

Feature	Description
Programming interface	SPI – JTAG, PCIe
Security mode	Double encryption.
Device mode	X1, X4.
Flash Read	SPI, Dual, Quad, Octa.
Page0 Header	Holds read address, read counts, and read commands.

Flash Device Configurations

Speedster7t FPGAs support two flash device configurations:

- Single flash device (1D).
- Four flash devices (4D).

1D Configuration

The 1D programming configuration is composed of a Speedster7t FPGA controlling communications with a single flash device. The `o_flash_sck` signal is used for clocking. The `o_flash_sdi` signal is the data output from the FPGA to communicate instructions to the flash device and `i_flash_sdo[0]` is the single-bit FPGA input pin which receives the bitstream from the flash in ×1 mode. The `o_flash_csn[0]` signal is pulled low as soon as communication between the FPGA and flash device begins, and stays low during the valid bitstream window.

The FPGA can communicate with the flash device in SPI, Dual, Quad or Octa modes in the 1D configuration.

The following figure provides a block diagram detailing how a serial flash device can be connected to a Speedster7t FPGA in Octa mode.

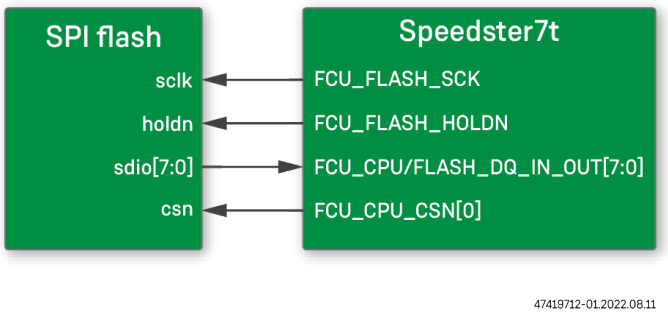


Figure 9 • Speedster7t FPGA 1D Octa Mode Flash Programming Configuration

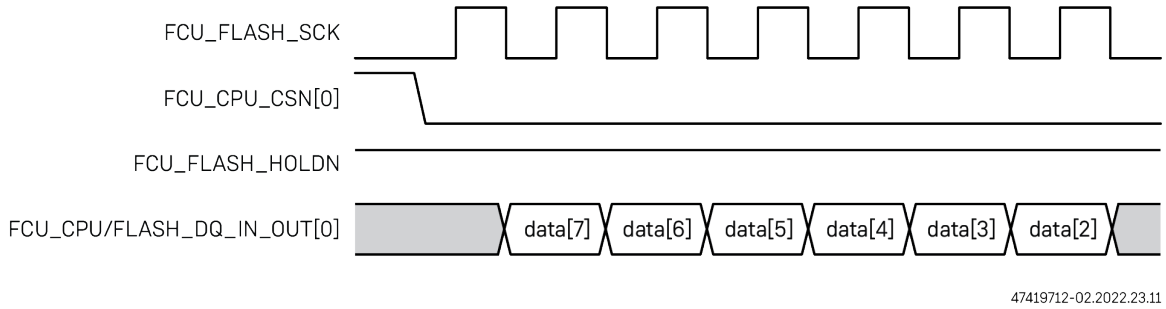


Figure 10 • 1D Flash SPI Read Data Ordering

4D Configuration

Serial 4D flash programming mode is essentially an enhanced and higher bandwidth implementation of the serial flash 1D configuration. The FPGA again controls communications and interfaces with not one but four flash memory devices to increase the data bandwidth four times.

When writing to the four flash memories, all four chip selects, `o_flash_csn[3:0]`, are pulled low simultaneously and 1-bit of bitsream data is sent to each flash device in SPI mode. When reading from the four flash memories, the FPGA pulls all of the `o_flash_csn[3:0]` signals low. Four-wide configuration data is read from the flash memories and transferred to the FPGA through the `i_flash_sdo` ports. When bitstream operations are complete (i.e., flash memory contents are read), transitioning from the end of the bitstream to user mode is performed as in CPU and flash 1D modes.

Each flash device can operate in SPI, Dual, Quad or Octa modes. The following figure provides a block diagram detailing how four serial flash devices (4D configuration) can be connected to a Speedster7t FPGA in Octa (x8) mode.

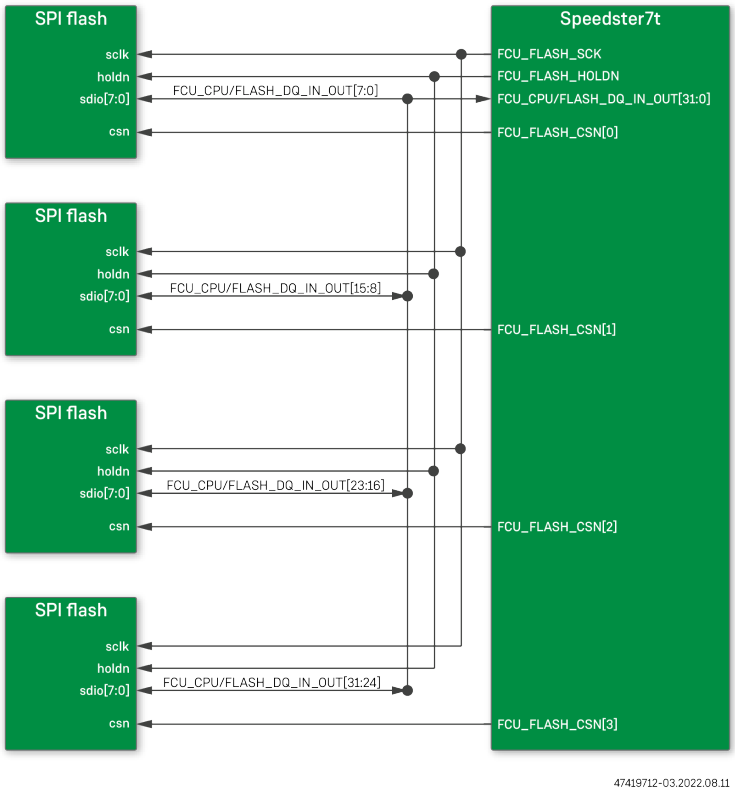


Figure 11 • Speedster7t FPGA 4D Flash Programming Configuration

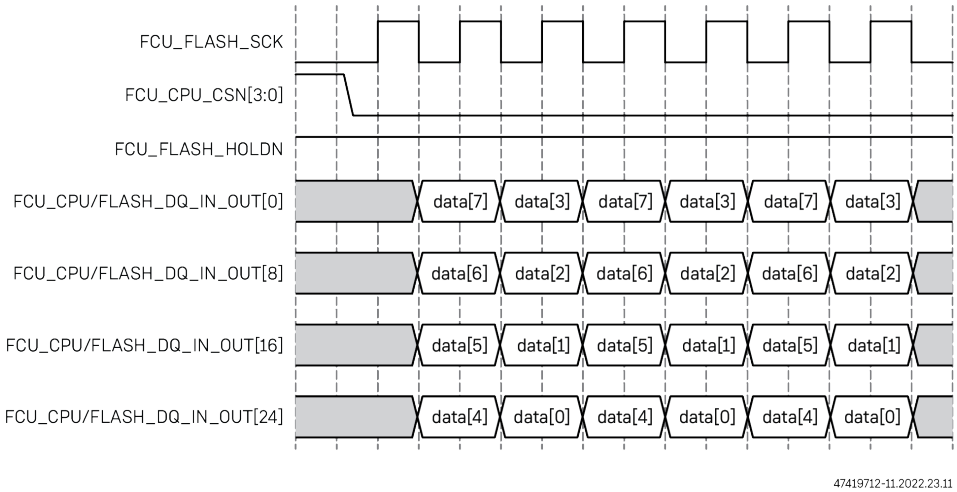


Figure 12 • 4D Flash SPI Read Data Ordering

> **Note**
>
> The FCU_FLASH_HOLDN signal must be held high at all times for both read and write accesses to the flash.

Addressing Modes and Memory Organization

Addressing modes for flash memory are based on the size of the device. A three-byte addressing mode is required for 128 Mb flash and smaller, and a four-byte addressing mode is required to support memory sizes above 128 Mb. Writes to the flash memory occur as pages, with each page consisting of 256 bytes. The following figure shows the memory organization.

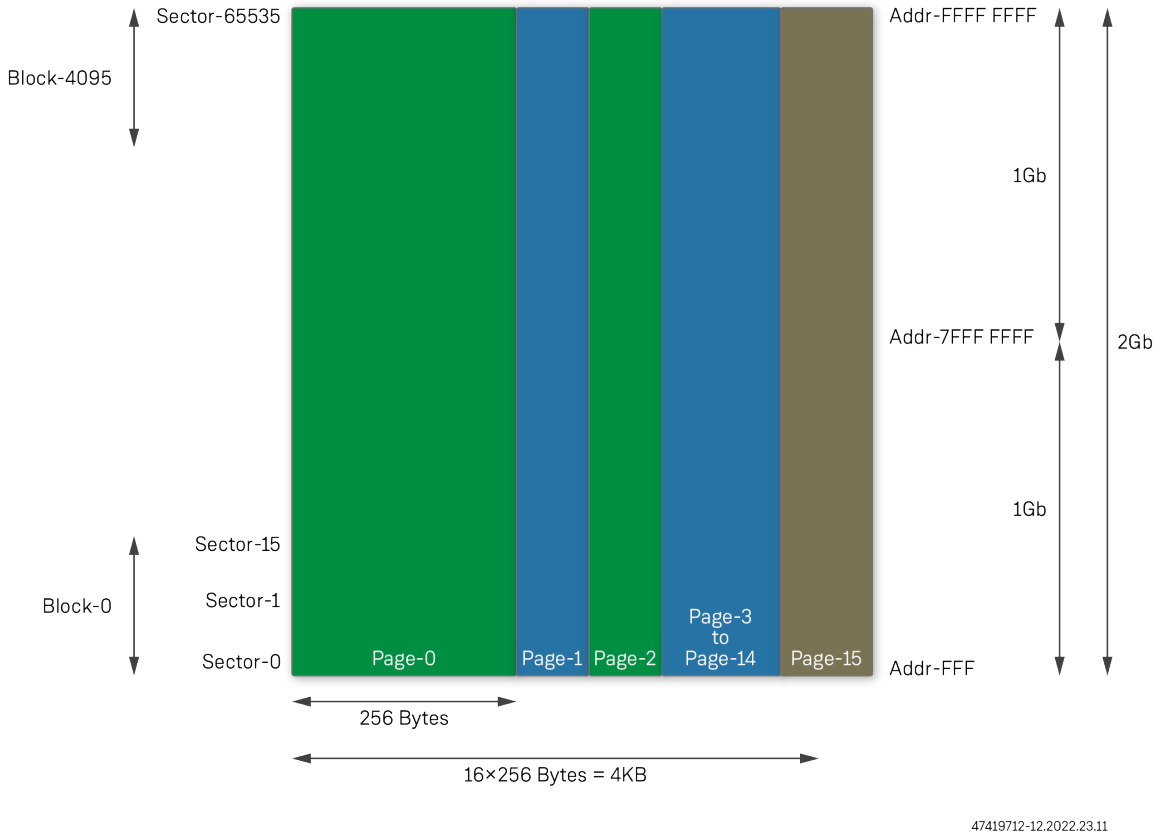


Figure 13 • Speedster7t FPGA Flash Memory Organization

Address Range

The following table shows the address ranges when two images are stored on a single flash device, assuming that each image is 1Gb in size.

Table 10 • Address Ranges For Two Bitstream Images on a Single Device

Address Range (32 bits)	Description	Configuration Details
0x0000_0000 to 0x0000_00FF	Page-0 address space. This range contains header information described in the flash configuration header section. This address range cannot be used for storing actual bitstreams.	These addresses are not user configurable.
0x0000_1000 to 0x07FF_FFFF	FPGA current address space.	The start address can be configured via the current address in the page-0 header. This example assumes the address starts at 0x0000_1000 for a 1 Gb bitstream.

Flash Configuration Header (Page0 Header)

> **Note**
>
> The page0 header information does not need to be manually created. ACE generates the _page0.flash file during bitstream generation.

The first 256 bytes in the flash memory (page0) holds control information that describes how the subsequent bitstream should be read from the flash device. This information can be written to the flash device in two ways:

- Via the JTAG interface along with the bitstream.
- Pre-programmed into the device by the manufacturer.

This space is not used for storing the device bitstream. It is formatted as described in the following table and is generated by ACE when the flash file output option is selected according to the flash configuration options previously described.

Table 11 • Page0 Header Format

Address	Bits	Value	Description	Device Specifics
0x0 to 0x3	32	Stage0 or full bitstream read address (new image).		

Address	Bits	Value	Description	Device Specifics
0x4 to 0x7	32	Bitstream read control.	Bit 0 – flash read enable. Bit 1 – flash fall back enable. Bit [5:2] – retry count. Bit [21:6] – timeout count. Bit 22 – enable 4-byte addressing. Bit [27:23] – dummy read cycles. Bit [30:28] – flash SCK div count. Bit [31]: 1'b1 – Micron, 1'b0 – Macronix.	Bit 1 is 1'b0 for AC7t1400/ AC7t1500
0x8 to 0xB	32	Flash configuration Stage0 or full bitstream read count.		
0xC to 0xF	32	Read command.		
0x18 to 0x20	24	Reserved.		
0x1C	8	Page0 Header version.	0x01, Version 1 (as of ACE 9.1.1).	
0x28	2	Bitstream mode.	Bit 0 - NOT encrypted. Bit 1 - full bitstream.	
0x30 to 0x33	32	Stage1 header read address (new image).	Calculated as stage0 read address + stage0 bitstream size (in bytes), aligned at 4kB boundary.	AC7t1400/AC7t1500 only.
0x38 to 0x3B	32	Flash configuration Stage1 header read count.		AC7t1400/AC7t1500 only.
0x40 to 0x43	32	Number of wait cycles between Stage0 and Stage1.		AC7t1400/AC7t1500 only.

Flash Configuration Protocol

With the FCU_CONFIG_MODESEL[3:0], FCU_CONFIG_CLKSEL and FCU_CONFIG_SYSCLK_BYPASS straps set for serial flash programming, operations begin as soon as the FPGA is powered up and the FCU receives the clock input. Immediately after reset is released, bitstream data is read out from the flash device through the flash interface (at this time the default is SPI (×1) mode). The bitstream read is performed in two stages described as follows:

Stage 1 – flash configuration header read from flash device:

- The FCU sends a default read command and address of 0x0000_0000 (32 bits) in SPI mode to the flash device and reads the flash configuration header.
- Internal registers are then updated, including the start address for the bitstream and flash read command.

Stage 2 – bitstream read from flash device:

- Based on the read mode obtained from the flash configuration header (×1/×2/×4/×8), the command and start address are sent to the flash device.
- The FCU reads the first 512 bits of bitstream data from the flash device and enters a wait state.

- If encryption is not enabled, the FCU reads the complete bitstream and configures the FPGA. If encryption is enabled and the efuse key is ready, the FCU reads the header segment0 data and sends it to the secure boot core. The flash read state machine then waits for 2.6 ms after which the FCU reads the complete bitstream and configures the FPGA.

Bitstream programming in all configuration modes is MSB to LSB. For transmitting a 32-bit FCU command, the ordering in the serial ×1 mode for 1D and 4D configuration is as follows:

- 1D flash configuration – the flash device transmits command bit 31 on the first clock and bits 30, 29, 28, etc. on subsequent clocks all the way down to bit 0 on the 32nd (last) clock.
- 4D flash configuration – the four flash devices transmit command bits [31:28] on the first clock, all the way down to bits [3:0] on the eighth (last) clock. The ordering within the 4-bit nibble corresponds to the flash device ordering. Specifically, on the first clock, flash[3] transmits bit 31, flash[2] transmits bit 30, flash[1] transmits bit 29 and flash[0] transmits bit 28

Flash Modes

The following section describes the various modes supported for read operations from attached flash device(s). Read operations from the flash device can be configured either as SPI, quad or octa modes for both 1D and 4D configurations. To write to the flash device(s), please consult the Flash Configuration Using FTDI (page 25) section.

> **Note**
>
> A flash write can be performed via either the JTAG or PCIe modes. The PCIe or JTAG port can access the data and command registers using an indirect addressing mode.

The following table describes the different combinations of the flash device configurations and modes supported in the Speedster7t FPGA.

Table 12 • Flash Device Configurations and Modes

Flash Programming Mode/Configuration	Flash Interface Width	No. of Flash Devices	Read Width SO[n:0] × No. of Flash Devices
SPI (1D)	1	1	1
SPI (4D)	1	4	4
Dual (1D)	2	1	2
Dual (4D)	2	4	8
Quad (1D)	4	1	4
Quad (4D)	4	4	16
Octa (1D)	8	1	8
Octa (4D)	8	4	32

Following are read operation timing diagrams for each of the flash interface widths.

Note

These diagrams pertain to Macronix devices. For details regarding other device vendors, please consult the flash device vendor datasheet.

SPI Mode

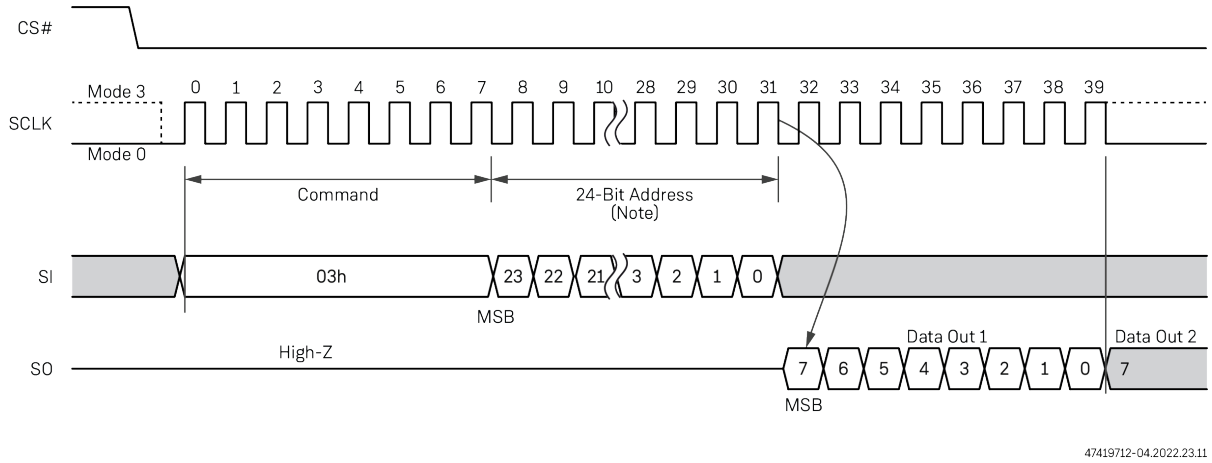


Figure 14 • SPI Read Mode Timing

Dual Mode

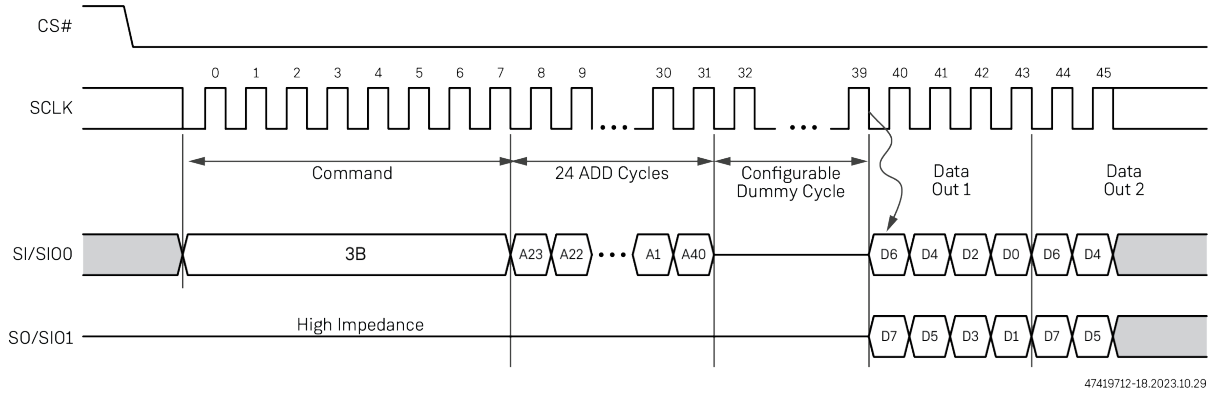


Figure 15 • Dual Read Mode (DREAD) Timing

Quad Mode

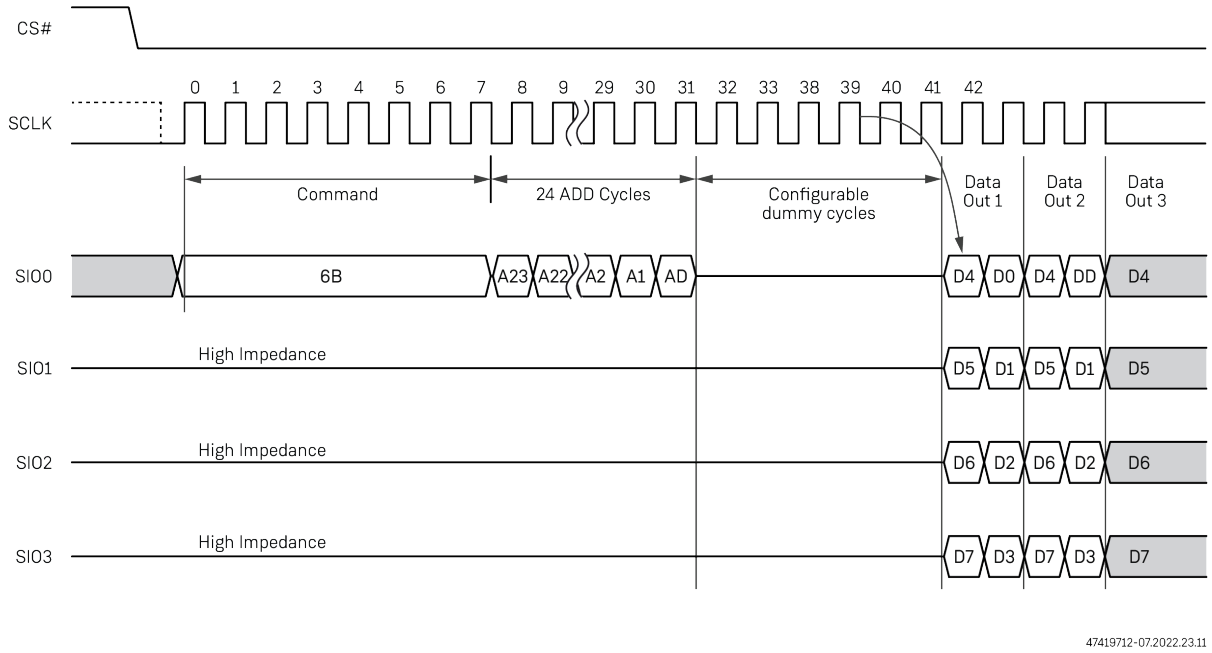


Figure 16 • Quad Read Mode (QREAD) Timing

Octa Mode

Warning!

Octa Mode Device Limitations:

For Octa Mode, Speedster7t devices currently only support flash devices in the 1-1-8 mode.

For context, normally flash devices have a SPI sequence that sets a configuration register value that fully converts everything to octa. Command,address,(return) data are all in octa: 8-bits on 8 lines throughout, or 8-8-8 mode. Some flash devices have a read command that works more like QSPI, and does not require writing a configuration register to enable. The command and address are sent on a single line (SDI) similar to SPI, and after a fixed number of dummy cycles, the data is read on all 8 lines. This is referred to as 1-1-8 mode.

The following figure represents 8-8-8 mode.

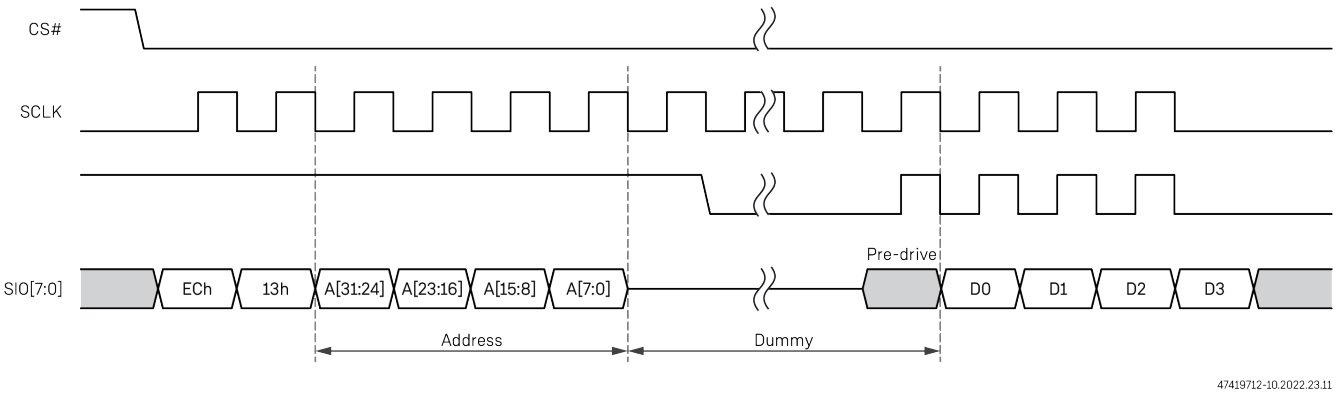


Figure 17 • Octa Read Mode (8READ) Timing

Flash Memory Size Requirements

As a general rule of thumb to cover the largest bitstream size, a 1 Gb flash memory is recommended to store one bitstream.

> **Note**
>
> Please work with the Achronix support team to evaluate the best Flash device size for any specific target device and end user design application.

Flash Configuration Using FTDI

The FTDI device multi-protocol synchronous serial engine (MPSEE) is configured for USB-to-SPI communication to program the flash device. SPI protocol lines are implemented using the MPSEE channels `ADBUS0–7`, `ACBUS0`, `ACBUS3`, and `BCBUS5`.

FTDI Board-Level Device Connections

ACE supports flash programming for three types of pin-out connections from FTDI to the flash device:

- Configuration 1
- Configuration 2
- Configuration 3

The FTDI devices supported by ACE are FT2232 and FT4232H. For pin-out connections config1 and config3, the signal `SPI_MODE_EN` is not required but can be used in situations where a multiplexer is placed between the FTDI device and another device that can access the flash memory. The three configuration connection types are described in the following tables.

FTDI Flash Pinout To FT2232

Table 13 • FTDI FT2232-to-Flash Connections (Configuration 1)

FTDI Port Name	FTDI Device Pin Number	FTDI Net name	Flash Pin
ADBUS0	16	FTDI_AD0	SPI_CLK
ADBUS1	17	FTDI_AD1	FCU_DQ1
ADBUS2	18	FTDI_AD2	FCU_DQ9
ADBUS3	19	FTDI_AD3	FCU_DQ17
ADBUS4	21	FTDI_AD4	FCU_DQ25
ADBUS5	22	FTDI_AD5	SPI_SDI
ADBUS6	23	FTDI_AD6	SPI_CS_N[3]
ADBUS7	24	FTDI_AD7	SPI_CS_N[2]
ACBUS0	26	FTDI_ACBUS[0]	SPI_CS_N[0]
ACBUS3	29	FTDI_ACBUS[3]	SPI_CS_N[1]
BCBUS5	57	FTDI_BCBUS[5]	SPI_MODE_EN

FTDI Flash Pinout To FT4232H

Table 14 • FTDI FT4232-to-Flash Connections (Configuration 2)

FTDI Port Name	FTDI Device Pin Number	FTDI Net name	Flash Pin
ADBUS0	16	FTDI_AD0	SPI_CLK
ADBUS1	17	FTDI_AD1	NC
ADBUS2	18	FTDI_AD2	SPI_CS_N[1]
ADBUS3	19	FTDI_AD3	SPI_CS_N[2]
ADBUS4	21	FTDI_AD4	SPI_CS_N[3]

FTDI Port Name	FTDI Device Pin Number	FTDI Net name	Flash Pin
ADBUS5	22	FTDI_AD5	SPI_MOSI
ADBUS6	23	FTDI_AD6	SPI_MISO
ADBUS7	24	FTDI_AD7	SPI_CS_N[0]

Note

Configuration 2 is the hardware configuration used with the VectorPath S7t-VG6 accelerator card.

FTDI Flash Pinout To FT2232/4232H

Table 15 • FTDI FT2232/4232H-to-Flash Connections (Configuration 3)

FTDI Port Name	FTDI Device Pin Number	FTDI Net name	Flash Pin
ADBUS0	16	FTDI_AD0	SPI_CLK
ADBUS1	17	FTDI_AD1	SPI_MOSI
ADBUS2	18	FTDI_AD2	SPI_MISO
ADBUS3	19	FTDI_AD3	SPI_CS_N[2]
ADBUS4	21	FTDI_AD4	SPI_CS_N[3]
ADBUS5	22	FTDI_AD5	SPI_CS_N[0]
ADBUS6	23	FTDI_AD6	SPI_CS_N[1]
BCBUS5	57	FTDI_BCBUS[5]	SPI_MODE_EN

Note

Configuration 3 is the most optimized and recommended mode. If using configuration 3 with the FTDI FT4232H, the SPI_MODE_EN pin is not used because it is not available on that device.

Bitstream Programming via JTAG

Generating the JTAG Bitstream Files From ACE

The JTAG `.hex` file is the default file and is always generated when running the Generate Bitstream flow step (`run -step write_bitstream`). Advanced bitstream features are covered in the **Design Security for Speedster7t FPGA** (page 94), **Partial Reconfiguration** (page 108), and **Configuration Error Correction and SEU Mitigation** (page 85) sections.

How To Use the ACE-Generated JTAG Bitstream Files

There are three ways to program a JTAG bitstream `.hex` file created during the Generate Bitstream flow step to program over JTAG. These methods are detailed in the following pages.

- **JTAG Programming using the ACE Download View** (page 28)
- **JTAG Programming using the ACE Flow Steps** (page 31)
- **JTAG Programming using the Tcl Library API** (page 32)

JTAG Programming using the ACE Download View

ACE JTAG Connection Preference Page

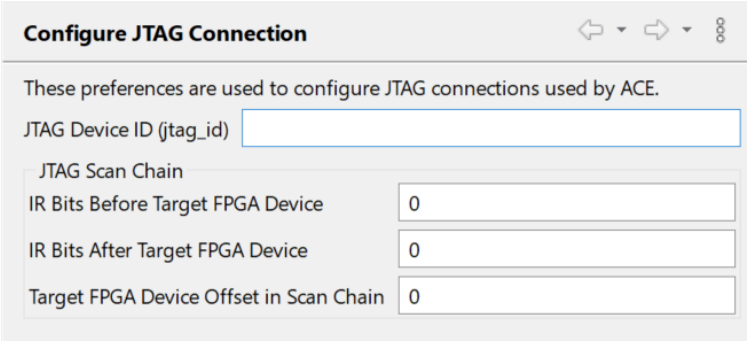


Figure 18 • Configure JTAG Connection Preference Page Example

Table 16 • Configure JTAG Connection Preference Page Options

Option	Description
JTAG Programmer Device Name (1)	The name of the JTAG device which should be used for all ACE JTAG interactions with the chosen FPGA or eFPGA. If the name is not specified, auto-detection of JTAG devices is attempted. **Performance Tip** Even if only one JTAG device is connected, specifying the JTAG device by name (instead of relying upon auto-detection) can save up to several seconds of initialization time on every JTAG connection.
JTAG Scan Chain	
IR Bits Before the Target FPGA Device (2)	Sets the (decimal) number of instruction register bits between the board JTAG TDI pin and the target device.
IR Bits After the Target FPGA Device (2)	Sets the (decimal) number of instruction register bits between the target device and the board JTAG TDO pin.
Target FPGA Device Offset in Scan Chain (2)	Sets the device count (in decimal) between the board JTAG TDI pin and target FPGA device.

Table Notes

1. Auto-detection can only be used safely when just one JTAG pod/device is connected. If more than one pod/device is automatically detected while no name is specified, JTAG interactions fail, stating that it is required to specify which pod/device to use. The Tcl command `jtag::get_connected_devices` provides a list of connected JTAG device names. See the *Speedster7t Configuration User Guide* (UG094) for more information.
2. The default value of zero is always correct for single-device JTAG scan chains. For multi-device scan chains, the default values of all zeros never work.

ACE JTAG Download View

The Download view provides a graphical interface for choosing and downloading (via JTAG) a bitstream `*.hex` file to an Achronix FPGA connected to the workstation using USB via a Bitporter2 pod or FTDI FT2232H or FT4232H device.

After downloading the bitstream, the JTAG connection status is retained, meaning that if the JTAG connection is open prior to programming, it stays open, and if the JTAG connection is closed prior to programming, it closes the connection when programming completes.

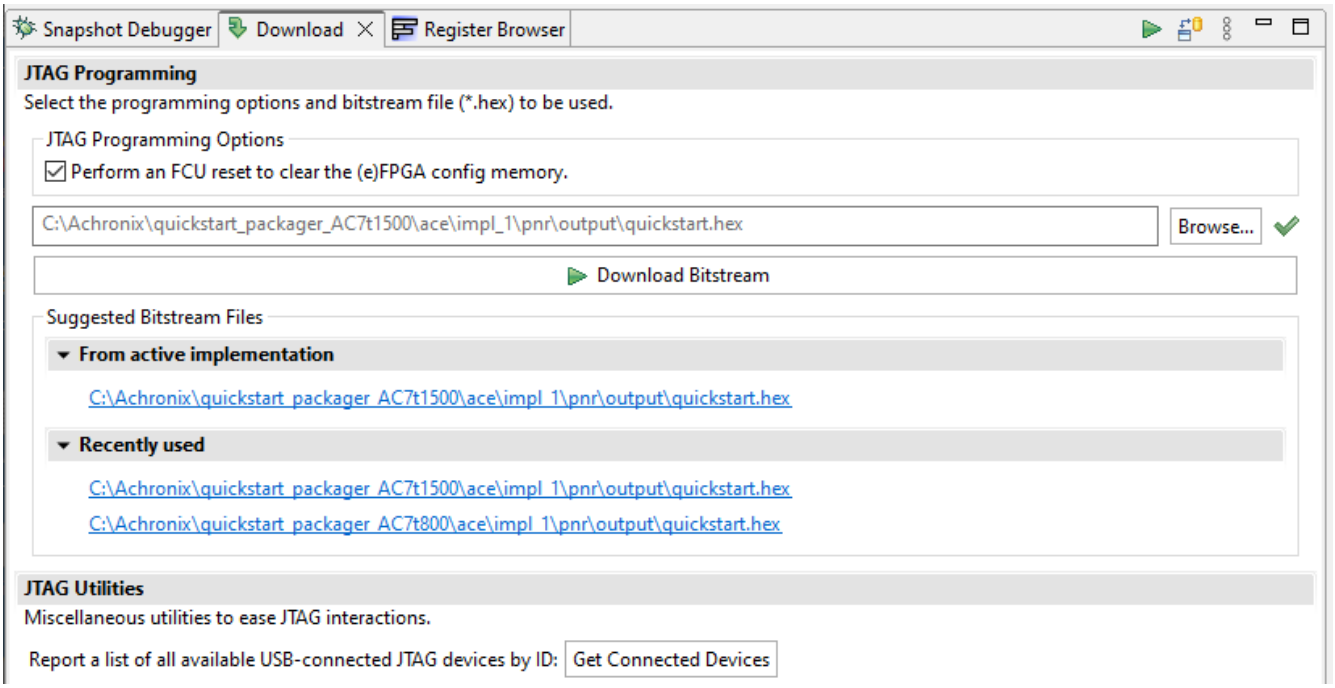


Figure 19 • Download View Example

Table 17 • Download View Options

Option	Description
JTAG Programming Options	
Perform an FCU reset to clear the (e)FPGA config memory.	When checked, performs a soft reset and clears all device configuration memory before beginning programming. This reset is typically only disabled for multi-stage programming (after stage 0 programming has completed, before programming later stages begins), or for "partial reconfig" when partial bitstreams are in use (see the chapter titled Partial Reconfiguration in the *Speedster7t Configuration User Guide* (UG094) for more details).
Browse	Allows choosing any `*.hex` bitstream file from the file system using a graphical file system browser.
Download Bitstream.	Clicking this button performs the actual download by calling the appropriate Tcl commands in the `jtag::` namespace. See also: *Speedster7t Configuration User Guide* (UG094).
Suggested Bitstream Files	

Option	Description
From active implementation.	A list of all `*.hex` bitstream files (shown as hyperlinks) found in the output directory of the current Active Implementation. Select any of these hyperlinks to choose that file for download.
Recently used.	A list of the most recently used `*.hex` bitstream files (shown as hyperlinks). Select any of these hyperlinks to choose that file for download.
JTAG Utilities	
Report a list of all available USB-connected JTAG devices by ID.	Press the button to run a Tcl command (`jtag::get_connected_devices`) to report a list of all connected JTAG devices in the Tcl Console view.

JTAG Programming using the ACE Flow Steps

ACE has a flow step that, upon completion of the "Generate Bitstream" flow step, allows downloading a generated bitstream into a connected Speedster7t FPGA. After downloading the bitstream, the JTAG connection status is retained, meaning that if the JTAG connection is open prior to programming, it will stay open, and if the JTAG connection is closed prior to programming, it will close the connection when programming completes.

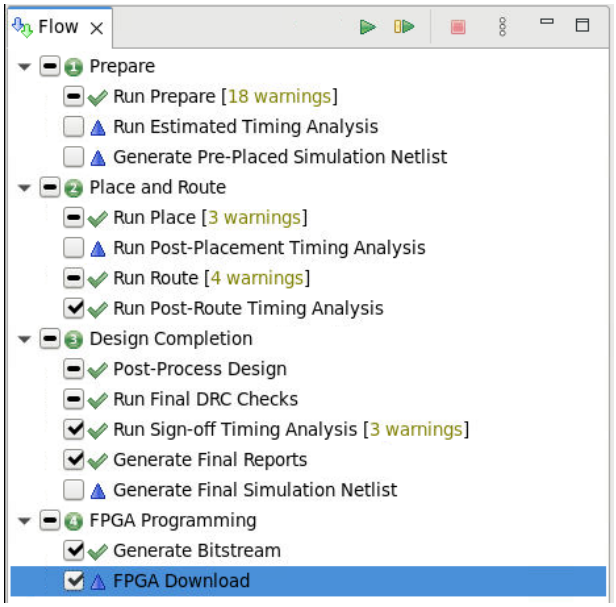


Figure 20 • ACE FPGA Download Flow Step

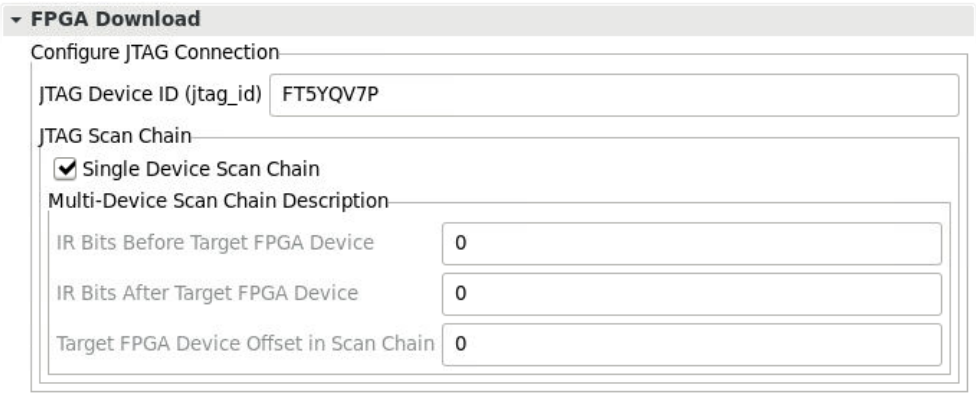


Figure 21 • ACE FPGA Download Options Dialog

Table 18 • FPGA Download Implementation Options - JTAG Scan Chain

Option	ACE impl_option	Description
JTAG Device ID (jtag_id)	`download_jtag_id`	Specifies the JTAG programming device name to attempt connecting to during FPGA download. If not populated, auto-detection of JTAG programming devices is attempted, and the download fails if more than one JTAG device is auto-detected.
Single Device Chain	`download_single_device`	This option should be enabled when the target is the only device on the JTAG scan chain (single-device JTAG scan chain). If this option is set to 0, the pre-IR, post-IR, and chain offset options are used to configure the scan chain.
IR Bits Before Target	`download_preir_padding`	Sets the (decimal) number of instruction register bits between the board JTAG TDI pin and the target device instruction register. Used for multi-device scan chains in order to pad the IR chain properly with ones, placing other devices in bypass mode.
IR Bits After Target	`download_postir_padding`	Sets the (decimal) number of instruction register bits between the target device and the board JTAG TDO pin. Used for multi-device scan chains in order to pad the IR chain properly with ones, placing other devices in bypass mode.
Chain Offset of Target	`download_chain_offset`	Sets the device count (in decimal) between the board JTAG TDI pin and target FPGA device. Setting this to 1 selects the second device on the chain, and so on.

JTAG Programming using the Tcl Library API

This section provides a list of JTAG Tcl commands for high-level general use and low-level specific use.

Variables Under ACE Tcl Console

When running in the ACE Tcl console, there are two types of Tcl variables used by the script: obligatory and optional. Both can be set in the Tcl console window before running a script.

> **Note**
> Starting with ACE 10.0, global variables are no longer just set to affect the ACE Tcl library.

```
# To set a TCL variable:
set var ACP1234X
# To clear a TCL variable:
unset var
# To determine the setting of a variable:
puts $var

#
# Accessor functions are available for the JTAG Tcl Library.

# To set the jtag-id used in subsequent operations:
set_global_jtag_id  ACP1234X
# To determine what value a variable is set to:
puts []
```

Table 19 • Variables Under ACE Tcl Console

Variable Name	Mandatory	Description
`jtag_id`	Yes	Must be set before scripts are run. Must match the JTAG ID value of the particular programming pod. To obtain a list of available programming pods, issue the `jtag::get_connected_devices` command.
`quiet_script`	No	If set to any value other than 0, the `jtag::apb_write()` and `jtag::apb_read()` commands are called without the -print option for scripts to run cleanly without excessive console logging.

Tcl Command Tables

Namespace Commands

There is a specific Tcl namespace within the API for each Speedster7t FPGA. The commands within each namespace are high-level commands such that each of these commands must be prefixed with its respective device namespace. For example, to read from a specific register in the Speedster7t AC7t1500 FPGA, the

csr_read_named command must be prefixed with the correct namespace as: ac7t1500::csr_read_named. The same action in the Speedster AC7t1400 FPGA requires the ac7t1400::csr_read_named command.

Acting On Read Returned Values

Under ACE, read commands operate as expected by interrogating the relevant registers and returning the values read to the ACE Tcl console. These values can then be used in further Tcl commands.

Commands

The following commands are specific to each device namespace.

Table 20 • High-Level Commands

Command	Arguments	Function	Description
Interrogate the Dictionaries			
get_dict_spaces (1)	None	Return the top address map spaces.	See list of available tokens under level 1.
get_dict_spaces (1)	<top level space token>	Return the level 2 tokens under the top level space.	See list of available tokens under level 2.
get_dict_spaces (1)	<level 1 token> <level 2 token>	Returns list of level 2 tokens. If using CSR_SPACE, these are the CSR IP areas under the specific IP.	See level 3 token descriptions.
get_dict_space	CSR_SPACE IP_NAME <IP area>	Returns a list of CSR registers under the specific IP and IP ID.	Returns register names (list can be long).
get_dict_space	CSR_SPACE IP_NAME <IP area> <register_name>	Gets the entry for the specified register name.	Returns an entry consisting of {addr[23:0] reg_size default_value}.
get_csr_reg_name (1)	address	Reverse dictionary lookup, providing the token hierarchy of the specified address.	Given the address (must be 11 hex digits), returns the tokens that specify that address. For example, given 08091340264, returns: `get_csr_reg_name() success. The address 08091340264 equates to CSR_SPACE DDR4 PHY MICRORESET`
Named CSR Register Accesses			
csr_write_named	CSR_SPACE IP_NAME IP_AREA REG_NAME <value>	Writes to selected register.	Value is treated as hex with or without the "0x" prefix.
csr_reset_named	CSR_SPACE IP_NAME IP_AREA REG_NAME	Resets selected register to its default value.	The default value is stored in dictionaries.
csr_read_named (2)	CSR_SPACE IP_NAME IP_AREA REG_NAME [expected value]	Reads the selected register.	Function returns register read (under ACE).

Command	Arguments	Function	Description
csr_verify_named	CSR_SPACE IP_NAME IP_AREA REG_NAME <value>	Verifies selected register is equal to value.	When run outside of ACE, function always returns 0.
csr_read_all_regs_named	CSR_SPACE IP_NAME IP_AREA	Reads all registers in an IP area.	Prints register name to console while reading the register.
csr_set_bits_named (2)	CSR_SPACE IP_NAME IP_AREA REG_NAME <low_bit> <high_bit> [expected value]	Sets bits [high_bit:low_bit] to 1'b1 in the selected register.	Performs a read-modify-write on the register. To set a single bit, assign high_bit = low_bit. The optional argument, expected value, is used in simulation only.
csr_clear_bits_named (2)	CSR_SPACE IP_NAME IP_AREA REG_NAME <low_bit> <high_bit> [expected value]	Clears bits [high_bit:low_bit] to 1'b0 in the selected register.	Performs a read-modify-write on the register. To clear a single bit, assign high_bit = low_bit. The optional argument, expected value, is used in simulation only.
Based CSR Register Accesses			
csr_named_base	CSR_SPACE IP_NAME [IP_AREA]	Declares the arguments to be the stateful base address values.	Supports 2 to 3 arguments. If IP_AREA is not specified, the stateful IP ID variable is set to BASE_IP (= 0).
csr_write_based	REG_NAME <value>	Writes to the selected register.	The value is treated as hex with or without the "0x" prefix.
csr_reset_based	REG_NAME	Resets the selected register to its default value.	The default value is stored in dictionaries.
csr_read_based (2)	REG_NAME [expected value]	Reads the selected register.	The function returns the register read (under ACE).
csr_verify_based	REG_NAME <value>	Verifies the selected register is equal to value.	When run outside of ACE, the function always returns 0.
Individual CSR Access			
csr_named_addr	CSR_SPACE IP_NAME [IP_AREA] [REG_NAME]	Returns the base address of the space.	Between 2 and 4 arguments are supported. The returned address is the base address of the provided arguments. If all four arguments are provided, the address is the full register address.
noc_write (3)	<addr value>	Writes to any location in the address map.	The addr value must be an eleven-character, 42-bit hex value and can be up to 32-bit hex. If csr_named_addr is used to obtain the base address, this function can be used by simply adding on the offsets to known registers.
noc_read (2) (3)	<addr> [expected value]	Reads from any location in the address map.	The addr value must be an eleven-character, 42-bit hex value. If csr_named_addr is used to obtain the base address, this function can be used by simply adding on the offsets to known registers.
noc_verify (3)	<addr> <value>	Reads and verifies the result from any location in the address map.	The addr value must be an eleven-character, 42-bit hex value. If csr_named_addr is used to obtain the base address, this function can be used by simply adding on the offsets to known registers.
set_bits_addressed	<addr> <low_bit> <high_bit>	Sets bits [high_bit:low_bit] to 1'b1 in the selected address.	Performs a read-modify-write on the address location. To set a single bit, assign high_bit = low_bit.

Command	Arguments	Function	Description
clear_bits_addressed	<addr> <low_bit> <high_bit>	Clears bits [high_bit:low_bit] to 1'b0 in the selected address.	Performs a read-modify-write on the address location. To clear a single bit, assign high_bit = low_bit.
NAP Access(4)			
nap_axi_write	NAP_SPACE <column row address> <value>	Creates an AXI write from the selected NAP.	The address and data are only 32-bits wide. The address is the AXI write address, awaddr and does not relate to selecting the NAP which uses column and row. The write data is relocated to the appropriate byte lane, selected by the address, in the 256-bit output from the NAP.
nap_axi_read (2)	NAP_SPACE <column row address> [expected value]	Creates an AXI read at the selected AXI NAP.	The address and data are only 32-bits wide. The address is the AXI read address, araddr and does not relate to selecting the NAP which uses column and row. The read data is relocated to the appropriate byte lane, selected by the address, in the 256-bit input from the NAP.
nap_axi_verify	NAP_SPACE <column row address> <value>	Creates an AXI read at the selected NAP. Compares the read value against the expected value.	The address and data are only 32-bits wide. The address is the AXI read address, araddr and does not relate to selecting the NAP which uses column and row. The read data is relocated to the appropriate byte lane, selected by the address, in the 256-bit input from the NAP.
GDDR6 and DDR4 Memory Access(5)			
memory_write	GDDR6_SPACE <controller> <channel> <address> <value>	Writes to the selected GDDR memory space.	<controller> is one of {GDDR_0 to GDDR_7}. <channel> is one of {CH_0 CH_1}. The address is up to a 33-bit hex field and the value is up to a 32-bit hex field.
memory_read	GDDR6_SPACE <controller> <channel> <address>	Reads from the selected GDDR memory space.	<controller> is one of {GDDR_0 to GDDR_7}. <channel> is one of {CH_0 CH_1}. The address is up to a 33-bit hex field and the returned value is 32-bit hex.
memory_write	DDR4_SPACE <address> <value>	Writes to the selected DDR4 memory space.	<address> is up to a 40-bit hex field. <value> is up to a 32-bit hex field.
memory_read	DDR4_SPACE <address>	Reads from the selected DDR4 memory space.	<address> is up to a 40-bit hex field The returned value is a 32-bit hex field.
Delay or Wait(6)			
wait_us	<wait value (decimal)>	Adds a wait value µs delay to the simulation command file.	The value is decimal, not hex. The wait is based on the FCU BFM cfg_clk. By default, this is 250MHz (4ns). If this clock is changed, this function must be updated. There is an associated ACE command only if the value exceeds 1,000 (>1ms).
wait_ns	<wait value (decimal)>	Adds a wait value ns delay to the simulation command file.	The value is decimal, not hex. The wait is based on the FCU BFM cfg_clk. By default this is 250MHz (4ns). If this clock is changed, this function must be updated. The delay is in multiples of 4ns. This command is only really applicable to simulation as the time between JTAG commands to the FCU in hardware is in the order of hundreds of µs or even ms. There is an associated ACE command only if the value exceeds 1,000,000 (>1ms).

Command	Arguments	Function	Description
Programming When Running Under ACE			
`program_hex_file`	`<hex filename` (with extension)`>` `[<optional arguments>]`	Programs a hex file. This operation opens the JTAG port and leaves it open.	The optional arguments are: • `-encrypted` – encrypted hex file. • `-do_not_enter_user_mode` – after programming, remain in configuration mode. Holds most of the I/O ring IP in reset. To use Windows backslashes, enclose the filename and path in braces "{}" (i.e., `program_hex_file {C:\Users\me\my_dir\test\my_hex.hex}`.

Table Notes

1. More details on available tokens can be found in the Application Note, *Runtime Programming of Speedster FPGAs* (AN025).[1]
2. The argument, `expected value`, is optional for functions that read from registers.
3. Excludes FCU registers. ACE has a specific command to access the FCU registers.
4. these commands access the NAP address space, not the CSR address space.
5. These functions rely on a stateful Tcl flow. The base addresses must be set first before the functions can make calls using the based address. These functions apply when a script is focused on a single IP block, saving the need to repeatedly re-enter values.
6. Inserts a wait into the command file. When run under ACE, there is at minimum an approximate 1 ms delay between commands

Low-level JTAG Tcl Commands

> **Warning!**
>
> These lower-level commands should only be used if no higher-level commands are available.

Table 21 • JTAG Tcl Commands in jtag:: namespace

Tcl Command	Description
`jtag::open <jtag_id>`	Opens a new connection to the JTAG device. Sets the initial clock frequency.
`jtag::get_connected_devices`	Gets the list of connected JTAG devices from the host machine. Returns the serial number (`jtag_id`) to be used with the `jtag::open` command.
`jtag::get_open_connections`	Returns the list of open connected JTAG devices in the ACE session.
`jtag::close <jtag_id>`	Closes an existing connection to a JTAG device.
`jtag::initialize_scan_chain <jtag_id> <pre_ir_bits> <post_ir_bits><target_device_offset> -single_device -target_device <string>`	Configures a scan chain. This function sets the initial clock frequency (based on the target device), checks the number of devices and IR length, sets preamble/ postamble IR/DR bits, and checks IDCODE.
`jtag::read <jtag_id> <bit_length> -irscan`	Performs a JTAG read. Specify `-irscan` to perform an IRSCAN. Returns a hex string of the read-back data.

1 https://www.achronix.com/documentation/runtime-programming-speedster-fpgas-an025

Tcl Command	Description
`jtag::write <jtag_id> <bit_length> <data> -irscan`	Performs a JTAG write to the connected JTAG device. Specify `-irscan` to perform an IRSCAN.
`jtag::write_read <jtag_id> <bit_length> <data> -irscan`	Performs a JTAG write and read (scan with capture) to the connected JTAG device. Specify `-irscan` to perform an IRSCAN. Returns a hex string of the read-back data.
`jtag::set_clock_frequency <jtag_id> <frequency>`	Sets the `TCK` frequency of the connected JTAG device. Returns the actual set frequency since the exact frequency cannot always be obtained. Waits in idle for 100 cycles.
`jtag::wait <jtag_id> <tap_state> <clock_ticks>` (1)	Waits in the specified TAP state for the specified number of `TCK` cycles.
`jtag::set_trst_n <jtag_id> <value>`	Asserts the specified value (0 or 1) on the `TRST_N` pin. A value of 0 asserts `TRSTN` and a value of 1 deasserts `TRSTN`.
`jtag::set_tap_state <jtag_id> <tap_state>` (1)	Sets the JTAG TAP state to one of the legal TAP states.
`jtag::get_tap_state <jtag_id>` (1)	Returns the current JTAG TAP state.
`jtag::read_config_rstn <jtag_id> -print`	Returns the value of FCU_CONFIG_RSTN.
`jtag::read_config_status <jtag_id> -print`	Returns the value of FCU_CONFIG_STATUS.
`jtag::read_config_done <jtag_id> -print`	Returns the value of FCU_CONFIG_DONE.
`jtag::initialize_fcu <jtag_id> -reset`	Performs initial setup. Must be run after the `jtag::initialize_scan_chain` command. The `-reset` option performs a soft reset and waits for configuration memory to be cleared. This action is required prior to bitstream programming with the `jtag::program_bitstream` command. When performing basic register/ cmem read/write operations, the `-reset` option should not be used (all registers are reset).
`jtag::exit_fcu <jtag_id>`	This command indicates to the FCU that it can exit JTAG mode and allow communication from other interfaces (such as the CPU).
`jtag::lock_fcu <jtag_id>`	This command locks the FCU after `jtag::program_bitstream` or locks the FCU after it is unlocked with `jtag::unlock_fcu`.
`jtag::unlock_fcu <jtag_id> -instance_id`	This command should be run after the `jtag::program_bitstream` command to unlock the FCU if locked after programming the bitstream.
`jtag::program_bitstream <jtag_id> <hex_file> -encrypted -stay_in_fcu_mode <bool>` (1)	Performs a bulk write to the Speedster7t FPGAs using data from the supplied hex file. Programs the bitstream and enters user mode. Must be run after `jtag::initialize_scan_chain` and `jtag::initialize_fcu -reset`. The optional `-encrypted` flag sends an encrypted bitstream hex file. This option adds the additional wait cycles needed for the Athena encryption engine. After the first 12,688 bytes of the encrypted bitstream, the code must wait in idle (pulsing `TCK`) 520,000 cycles. Refer to the **Design Security for Speedster7t FPGA (page 94)** section for details.
`jtag::read_cmem <jtag_id> <word_count> <word_step> <address> -print`	Reads a frame of 9344-bit data from the CMEM (Core/BRAM/LRAM/CFF/DFF) data space starting at frame address `<address>`. The address is the 24-bit frame address (upper 24-bits out of the 32-bit address). The read-back data is returned as a hex string to the caller. The `-print` option prints a message indicating the address and data read back.
`jtag::write_cmem <jtag_id> <address> -data <9344-bit string> -print`	Writes a frame of 9344-bit data to the CMEM (CORE/BRAM/LRAM/CFF/DFF) data space at address `<address>`. If no data is specified, the command writes all ones. The `-print` option prints a message indicating the write address.

Tcl Command	Description
`jtag::read_acb <jtag_id> <address> <word_count> <word_step> -print`	Reads `<word_count>` 32-bit words of data from the ACB data space (boundary ring) starting at word address `<address>`. The address is the full 24-bit ACB address. If multiple words are read back, the address increments by `<word_step>` for each read operation. The read-back data is returned as a Tcl list of hex strings to the caller. The `-print` option prints a message indicating the address and data read back.
`jtag::write_acb <jtag_id> <address> <data> -flush <int> -print`	Writes 32 bits of `<data>` into the register at address `<address>`. The address is the full 24-bit ACB address. The `-print` option prints a message indicating the write address.

`jtag::read_apb <jtag_id> <address> <word_count> <word_step> -print`	Read `<word_count>` 32-bit words of data from the full 42-bit address space, which can talk to CSR, GDDR6, DDR4, FCU, PCIE, or NAP data space starting at word address `<address>`. The address is the full 42-bit address. If multiple words are read back, the address increments by `<word_step>` for each read operation. The read-back data is returned as a Tcl list of hex strings to the caller. The `-print` option can be used to print a message indicating the address and data read back.
	Examples:
	`# Bulk read`
	`jtag::read_apb $jtag_id {08060000048 08070000050 0816000003c 08170000030} 1 1`
	`# Single read`
	`jtag::read_apb $jtag_id 08060000048 1 1`

`jtag::write_apb <jtag_id> <address> <data> -flush <int> -print`	Writes 32 bits of `<data>` into the register at address `<address>`. The address is the full 42-bit address. The `-flush` option adds `<int>` clock ticks to wait in IDLE after the write to flush it. The default is 300 clock cycles. The `-print` option prints a message indicating the write address.
`jtag::read_fcu <jtag_id> <address> <word_count> <word_step> -print`	Reads `<word_count>` 32-bit words of data from the FCU register space (Internal FCU Registers) starting at word address `<address>`. The address is the full 16-bit FCU address. If multiple words are read back, the address increments by `<word_step>` for each read operation. The read-back data is returned as a Tcl list of hex strings to the caller. The `-print` option prints a message indicating the address and data read back.
`jtag::write_fcu <jtag_id> <address> <data> -print`	Writes 32 bits of `<data>` into the FCU register at `<address>`. `<address>` is the full 16-bit FCU address. The `-print` option displays a message indicating the target address.

Table Notes

1. Legal JTAG TAP states:

   `CAPTURE_DR`, `CAPTURE_IR`, `EXIT1_DR`, `EXIT1_IR`, `EXIT2_DR`, `EXIT2_IR`, `IDLE`, `PAUSE_DR`, `PAUSE_IR`, `RESET`, `SELECT_DR_SCAN`, `SELECT_IR_SCAN`, `SHIFT_DR`, `SHIFT_IR`, `UPDATE_DR`, `UPDATE_IR`.

Programming the Board Using JTAG and Read/Write Registers

Speedster7t Flow

To program a bitstream using JTAG Tcl, a `*.hex` file must be generated which appears in the `<design>/ace/<impl>/output/` directory. The implementation option `bitstream_output_hex` is enabled by default for Speedster7t FPGAs.

The following Tcl commands must be run to:

- Open the JTAG Connection
- Program the Bitstream
- Enter User Mode

```
set jtag_id [jtag::get_connected_devices]
jtag::open $jtag_id
jtag::initialize_scan_chain $jtag_id 0 0 0
jtag::initialize_fcu $jtag_id -reset
jtag::program_bitstream $jtag_id <*.hex file>
```

If the `bitstream_fcu_lock` impl option is set, the FCU locks at the end of programming. This prevents the FCU registers from being accessed. The FCU must be unlocked in order to regain access. However, the FCU cannot be unlocked on encrypted bitstreams. To unlock the FCU after entering user mode, add the following command:

```
jtag::unlock_fcu $jtag_id
```

If the device is already programmed, the `-reset` option must be skipped:

```
set jtag_id [jtag::get_connected_devices]
jtag::open $jtag_id
jtag::initialize_scan_chain $jtag_id 0 0 0
jtag::initialize_fcu $jtag_id
```

Following either code path, where appropriate, register reads and writes can be executed. The following are some example registers to read and write:

```
jtag::read_fcu $jtag_id 1000 -print
jtag::write_fcu $jtag_id 1064 00003001 -print
jtag::read_fcu $jtag_id 1064 1 4 -print
jtag::write_fcu $jtag_id 1064 00000000 -print
jtag::read_fcu $jtag_id 1064 1 4 -print
```

To release the JTAG lock, execute the following command:

```
jtag::close $jtag_id
```

JTAG Hardware Overview

Introduction

When a design has successfully completed the ACE design flow, it is ready for FPGA programming. ACE has a straightforward interface to generate the bitstream files required to implement all of the supported configuration modes. The bitstream files are generated during the the FPGA Programming – Generate Bitstream step of the compilation flow (see the Concepts, View, *Flow View* section in the ***ACE User Guide*** **(UG070)**[2] for more details).

The bitstream hex file needed for JTAG mode configuration is always generated by default. The "Bitstream Generation" section of the **Project Options** menu, also provides a menu selection to generate bitstream files for the other configuration modes.

The configuration options are unique to each device and ACE supports a number of settings for the features supported by each device.

JTAG Configuration Overview

The embedded programming and configuration logic in the Speedster7t FPGA is designed to support a variety of programming and debugging options. There are two external interfaces that can be used as communication channels between Achronix hardware and software:

- The Achronix Bitporter2 pod – provides a JTAG-only interface via USB to Achronix FPGAs. Device configuration must be completed via JTAG, along with communication with debug tools such as Snapshot. See **JTAG Configuration Using the Bitporter2 Pod** (page 59).
- An FTDI FT2232H/FT4232H device – provides a lower-cost JTAG interface to Achronix FPGAs through USB. This interface also allows debug tools to be accessible via JTAG. See **JTAG Configuration Using FTDI** (page 46).

The following figure outlines the basic block diagram of the programming and configuration logic. The configuration management unit is responsible for configuring the device with a bitstream and controls the startup and shutdown sequence from configuration mode to user mode and back.

2 https://www.achronix.com/documentation/ace-user-guide-ug070

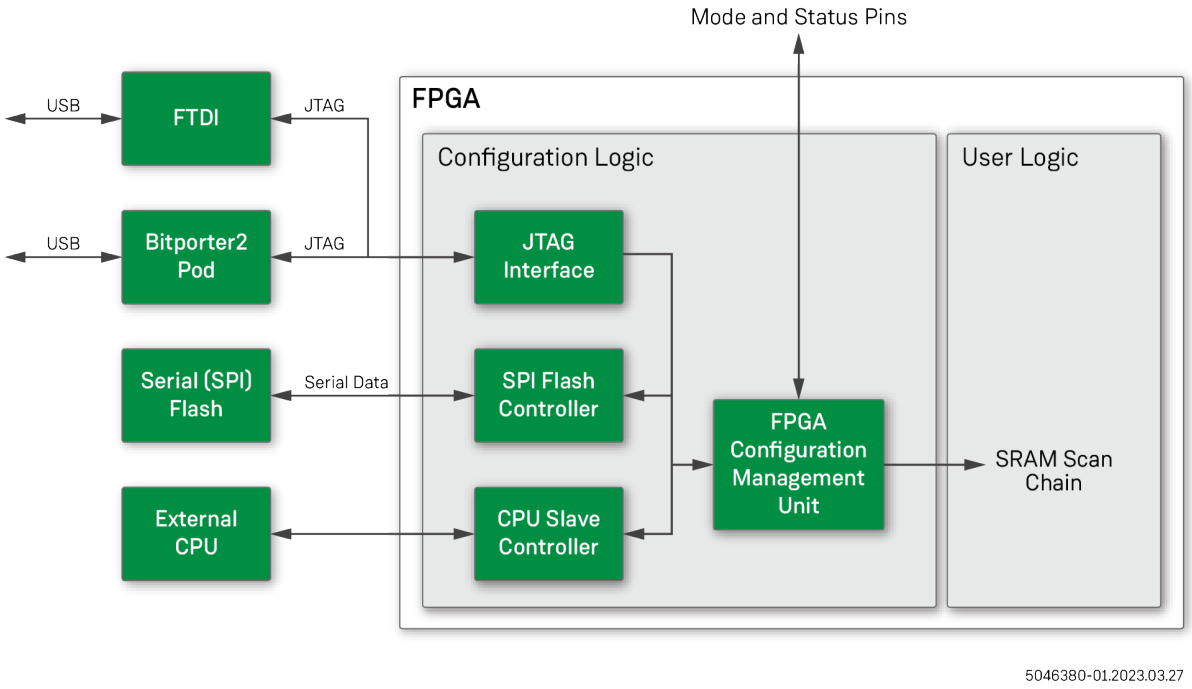


Figure 22 • Configuration Options Block Diagram

Board-Level Device Connections

The following figure details the board-level electrical connections to the JTAG header used to connect the Bitporter2. The subsequent figure provides the mechanical specifications (The value of V_{DDO_JTAG} is dependent on the I/O voltage of the JTAG target chip).

For board-level connection details, access board schematic documents on the Achronix support portal, **Board Level Issues**[3] (requires signed NDA to view board schematics).

3 https://support.achronix.com/hc/en-us/sections/360010558151-Board-Level-Issues

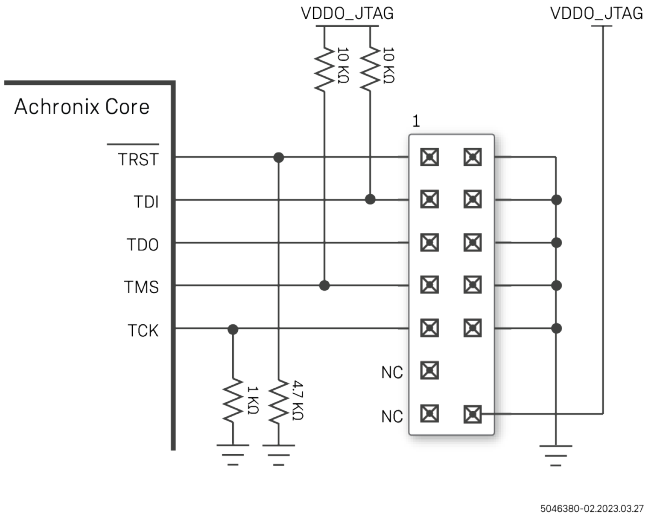


Figure 23 • JTAG Header Electrical Connections

Caution

The Tck signal produced by both the FT2232H/FT4232H device, and by all Bitporter 2 devices, is only present during programming. Further, its frequency accuracy and stability cannot be guaranteed. Therefore, it is not recommended to use this clock for any other purpose than JTAG programming of the device.

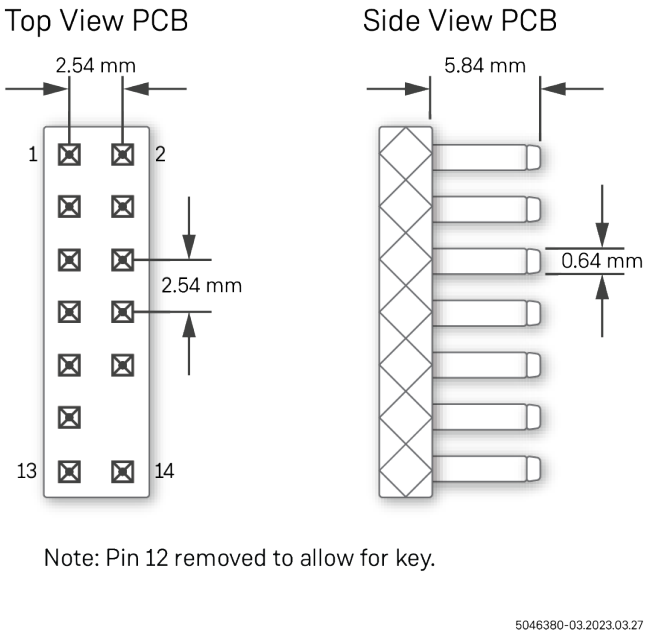


Figure 24 • JTAG Header Mechanical Specifications

JTAG TAP Controller Overview

The Speedster7t FPGA JTAG TAP controller is IEEE Std 1149.1 compliant and is used for programming the bitstream and debug via Snapshot in ACE. The JTAG_TMS and JTAG_TCK inputs determine whether an instruction register scan or data register scan is performed. JTAG_TMS and JTAG_TDI are sampled on the rising edge of JTAG_TCK, while JTAG_TDO changes on the falling edge. JTAG configuration and operation mode is independent of FCU_CONFIG_MODESEL settings.

JTAG implementation in Speedster7t FPGAs, which allows for bitstream programming as well as real-time in-system control and observation, is composed of the blocks shown in the following figure.

The external interface is a standard 5-pin JTAG interface, connected directly to the JTAG TAP controller. The TAP controller operates independently from the Speedster7t FPGA FCU. It is always active and uses JTAG_TCK for clocking. The TAP controller takes the data from the pins and converts it to DR instructions to communicate to the JTAG logic in the FCU. It also takes in data in the form of load/read requests, translating it to the appropriate signals to drive and expect on the JTAG pins.

The JTAG logic in the FCU interprets these DR instructions and generates input data in the standard 128-bit Speedster7t FPGA frame size format, along with a data valid indicator, to be forwarded to the FCU data mux and, ultimately, to the FCU state machine for configuration memory loading. The FCU data mux accepts 128-bit output data from the FCU, which has an associated valid signal for debug and read-back operations. The mux also provides an acknowledge signal to indicate to downstream circuitry that the data transfer was successful.

The FCU data mux simply selects between the configuration mode-specific data buses entering the FCU. This logic is controlled by the static FCU_CONFIG_MODESEL straps and the JTAG override logic from the JTAG TAP controller.

Finally, the FCU state machine accepts incoming data for loading the configuration memory. Conversely, it also provides output data from the configuration memory or Snapshot to forward upstream.

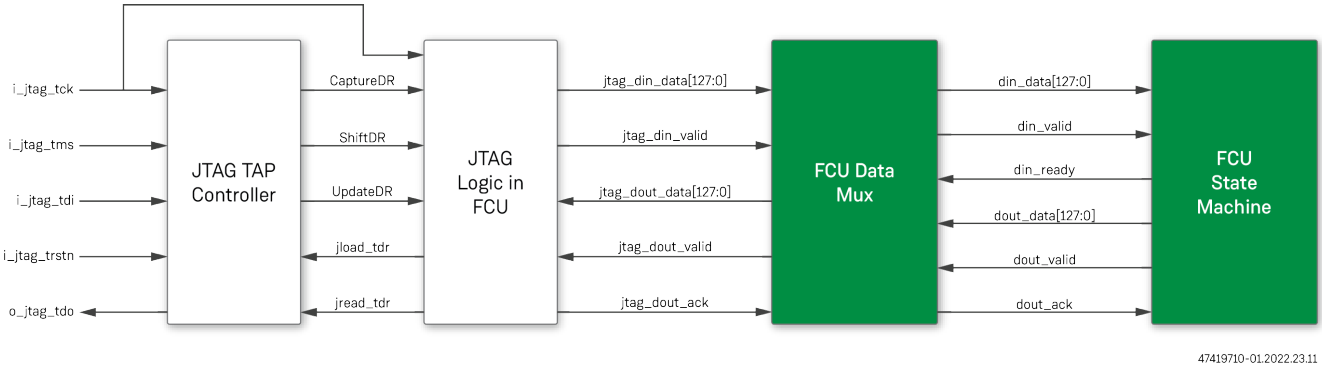


Figure 25 • Block Diagram for JTAG Instruction Processing in the FCU

JTAG Instructions

The following table lists all JTAG instructions supported by Speedster7t FPGAs.

Table 22 • JTAG Instructions

Instruction	Opcode	DR Width	Function
BYPASS	23'b00000000000000000000000	1	The required BYPASS instruction allows a Speedster7t FPGA to remain in a functional mode and selects the bypass register to be connected between JTAG_TDI and JTAG_TDO. This instruction allows serial data to be transferred through the FCU from JTAG_TDI to JTAG_TDO without affecting FPGA operation.
EXTEST	23'b11111111111111111101000	–	The required EXTEST instruction places the Speedster7t FPGA into an external boundary-test mode and selects the boundary-scan register to be connected between JTAG_TDI and JTAG_TDO. Output pins operate in test mode, driven from the contents of the boundary-scan update latch. Input data are captured in boundary-scan latches prior to shift operation. During this instruction, the boundary-scan register is accessed to drive test data outside the FPGA via the boundary outputs and receive test data from outside the FPGA via the boundary inputs.
EXTEST_PULSE	23'b11111111111111111101001	–	Generates a single pulse by entering and exiting the Run-Test/Idle state of the 1149.1 TAP controller.
EXTEST_TRAIN	23'b11111111111111111101010	–	Generates a stream of pulses while in the Run-Test/Idle state. A BSDL file for an 1149.6 device specifies the minimum number of pulses and the maximum time period allowed for pulse generation in the Run-Test/Idle state.
SAMPLE/PRELOAD	23'b11111111111111111111000	–	The required SAMPLE/PRELOAD instruction allows a Speedster7t FPGA to remain in its functional mode and selects the boundary-scan register to be connected between JTAG_TDI and JTAG_TDO. The output and input pins operate in normal mode. Input pin data and core logic output data are captured in the boundary-scan latches. During this instruction, the boundary-scan register can be accessed via a data scan operation to take a sample of the functional data entering and leaving the FPGA. This instruction is also used to preload test data into the boundary-scan register before loading an EXTEST instruction.
IDCODE	23'b11111111111111111111110	32	The optional IDCODE instruction allows a Speedster7t FPGA to remain in its functional mode and selects the optional device identification register to be connected between JTAG_TDI and JTAG_TDO. The IDCODE register appears between JTAG_TDI and JTAG_TDO after power-up, after the TAP has been reset using the optional TRST pin, or by otherwise moving to the Test-Logic-Reset state.

Instruction	Opcode	DR Width	Function
HIGHZ	23'b11111111111111111001111	–	The optional HIGHZ instruction sets all outputs (including two-state as well as three-state types) to a disabled (high-impedance) state and selects the bypass register to be connected between JTAG_TDI and JTAG_TDO.
CLAMP	23'b11111111111111111101111	–	Provides for "guarding" chip outputs during in-circuit test or boundary-scan functional test. Output pins operate in test mode, driven from the content of the boundary-scan update latch. The one-bit bypass register is selected for shifting.
INTDR	23'b00000000000000000111101	97	Provides access to the test data register implemented internal to the TAP controller. This internal register is used for global configuration and monitoring of global status signals.
JLOAD	23'b00000100000001100111010	128	Enables the scan-in of the configuration bitstream to the configuration logic (in this mode, the SHIFT-DR state is used to scan in the bitstream). For the read-back, the data register is read back. All of these operations are performed internally using a 128-bit parallel bus. Data is latched every 128 bits in the UPDATE-DR state.
JREAD	23'b00000100000001000111010	128	Enables the data register for read-back. When this instruction is decoded and CAPTURE-DR is executed, the data from the configuration logic is sampled as 32-bit data plus a valid bit. Multiple words of the configuration memory can be read back by cycling through the CAPTURE-DR/SHIFT-DR states. The 33-bit status register is selected between JTAG_TDI and JTAG_TDO.
JUSR1 (1)	23'b00000100000000100111010	User defined	Enables the USER1 TDR.
JUSR2 (1)	23'b00000100000000000111010	User defined	Enables the USER2 TDR.
JASYNCERR	23'b00000000000001110111010	–	Enables the connection to the fabric error status scan register.

Table Notes

1. This TDR is implemented in the fabric and is used for supporting debug functions in the fabric.

JTAG Configuration Using FTDI

Overview

The FTDI device provides a low-cost JTAG interface to Achronix Speedster7t FPGAs and Speedcore eFPGAs through a USB 2.0 (USB 1.0/3.0 compatible) interface, enabling both debug and configuration interfaces. Achronix supports interfacing only with FTDI FT2232H and FT4232H devices.

The following diagram shows how Speedcore/Speedster7t FPGAs interface to ACE via the FT2232H/FT4232H device. In this setup, the FTDI multi-protocol synchronous serial engine (MPSEE) is configured for single-chip USB-to-JTAG communication. The FTDI device interfaces to the host PC via USB. ACE allows configuring and debugging the FPGA using the built-in FTDI drivers.

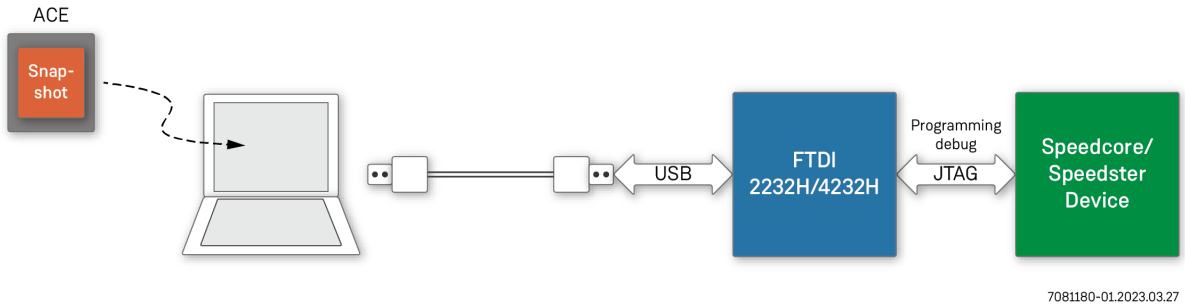


Figure 26 • FTDI Interface Example

The FT2232H has two independent 16-bit configurable interfaces while the FT4232H has four independent 8-bit configurable interfaces.

The FTDI configuration flow is as follows:

1. Generate a `design_name.hex` file from a placed-and-routed design within ACE.
2. Connect a USB cable between the on-board FTDI programming port and the USB port of the host PC.
3. Program the device through JTAG using one of three methods:
   - Using the FPGA download flow step after generating a bitstream
   - Using the ACE GUI Download view:

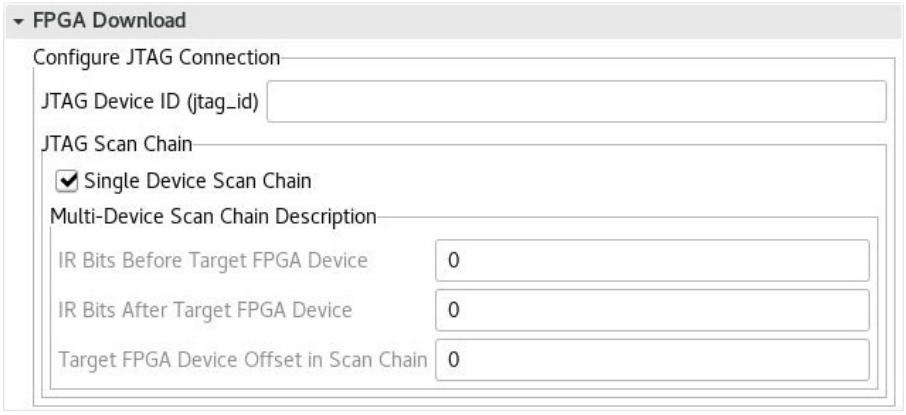


Figure 27 • Programming Device Connections In ACE Download View

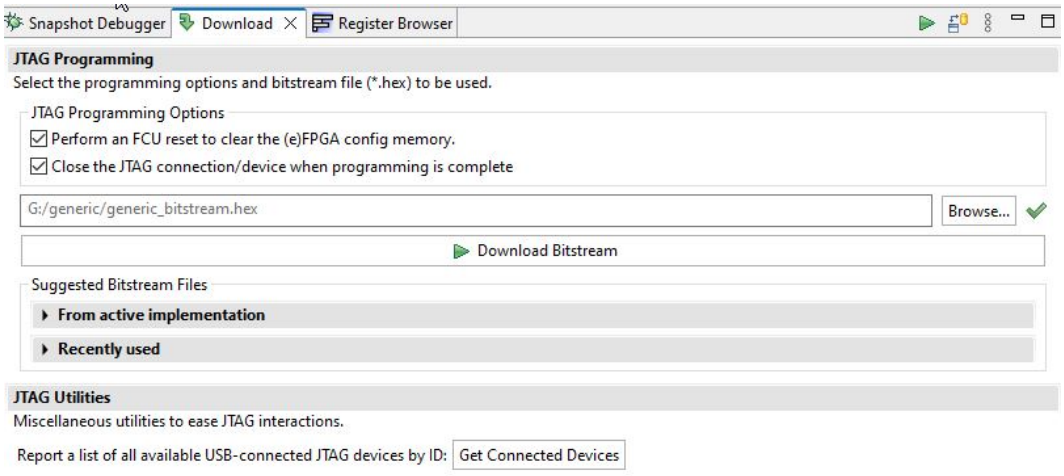


Figure 28 • Hex File Actions in ACE Download View

- Using the Tcl console with the `program_hex_file` command

The following figure shows the basic block diagram of the FTDI devices.

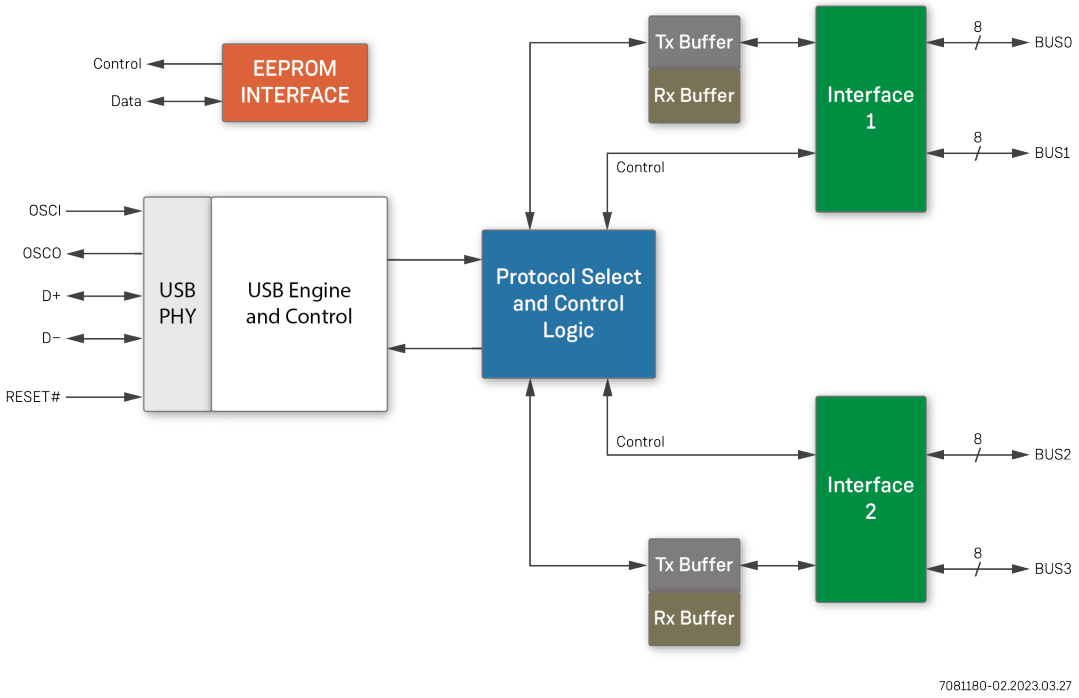


Figure 29 • FTDI FT2232H/FT4232H Basic Block Diagram

FTDI Board-Level Device Connections

FTDI JTAG Pinout

Achronix tools expect the BDBUS pins of the FTDI chip to be connected to JTAG as specified in the following table.

Table 23 • FTDI-to-FPGA JTAG Connections

JTAG Signal Name	JTAG Header Pin Number	FTDI Port Name	FT2232H Pin Number	FT4232H Pin Number
TRST_N	1	BDBUS[4]	43	30
TCK	9	BDBUS[0]	38	26
TMS	7	BDBUS[3]	41	29
TDI	3	BDBUS[1]	39	27
TDO	5	BDBUS[2]	40	28

Note

On Achronix boards, BDBUS[5] is connected to JTAG_MODE_EN_N, an active-low signal that is inverted into JTAG_MODE_EN, which is used as a mux between the FTDI chip and the JTAG header.

FTDI Voltage Compatibility

The FTDI devices have two voltage rails:

- V_{CORE}
- V_{CCIO}

V_{CORE} must be connected to 1.8V, while V_{CCIO} must be connected 3.3V. As a result, the output ports from the FTDI chips have a 3.3V range. However, Speedcore and Speedster7t devices both require 1.8V for the configuration signals, including JTAG. Therefore, it is necessary to insert voltage level shifters between the output of the FTDI and the JTAG input signals of the target device.

FTDI EEPROM Interface

An external EEPROM helps select the FTDI operating mode. Adding an external EEPROM allows each of the chip channels to be independently configured to one of three modes:

- Serial UART (RS232 mode)
- Parallel FIFO (245) mode
- Fast serial (opto-isolation) mode

The EEPROM *must* be programmed using the Achronix template file to allow the Achronix device drivers to find and communicate with the FTDI device. When used without an external EEPROM, the FTDI device defaults to a USB-to-dual-asynchronous-serial-port device. This mode is *not* supported by Achronix.

The external EEPROM can also be used to customize the following FTDI device USB parameters:

- VID
- PID
- Serial Number
- Product Description Strings
- Power Descriptor value

Other parameters controlled by the EEPROM include the following:

- Remote wake up
- Soft pull-down on power-off
- I/O pin drive strength

The following table summarizes modes that are configurable using the EEPROM:

Table 24 • EEPROM Configuration Modes

Configuration Method	ASYNC Serial UART	ASYNC FIFO (245)	SYNC FIFO (245)	ASYNC Bit-bang	SYNC Bit-bang	MPSSE	Fast Serial Interface	CPU-Style FIFO	Host Bus Emulation
EEPROM Configured	YES	YES	YES				YES	YES	
Application Software Configured			YES	YES	YES	YES			YES

Programming the EEPROM

The FTDI utility, FT_PROG[4], can be used to program the EEPROM. A generic template file, Achronix_EEPROM_Template_for_FTDI4232.zip[5], is available on the Achronix support portal for programming the EEPROM.

Unzip the file to a local folder. The archive contains a template file: `FT4232_VP3.xml`.

The following table lists the values of the parameters in the Achronix generic EEPROM file.

4 http://www.ftdichip.com/Support/Utilities.htm#FT_PROG
5 https://support.achronix.com/hc/en-us/articles/4496793147668

Table 25 • Generic Achronix EEPROM File Contents

Variable	Value	Programmable	Comments
Manufacturer	BittWare	Yes	Specifies the Vendor name.
Product Description	Achronix VectorPath	Yes	String "Achronix" is required anywhere in the value.
Serial Number		Yes	Must be a non-zero/non-null value starting with "AC".
Vendor ID	`0x0403`	No	Do not modify this value.
Product ID	`0x6011`	No	Do not modify this value.

Follow these steps to program the EEPROM:

1. Launch FT_PROG and open the example Achronix Speedcore EEPROM template `.xml` file:

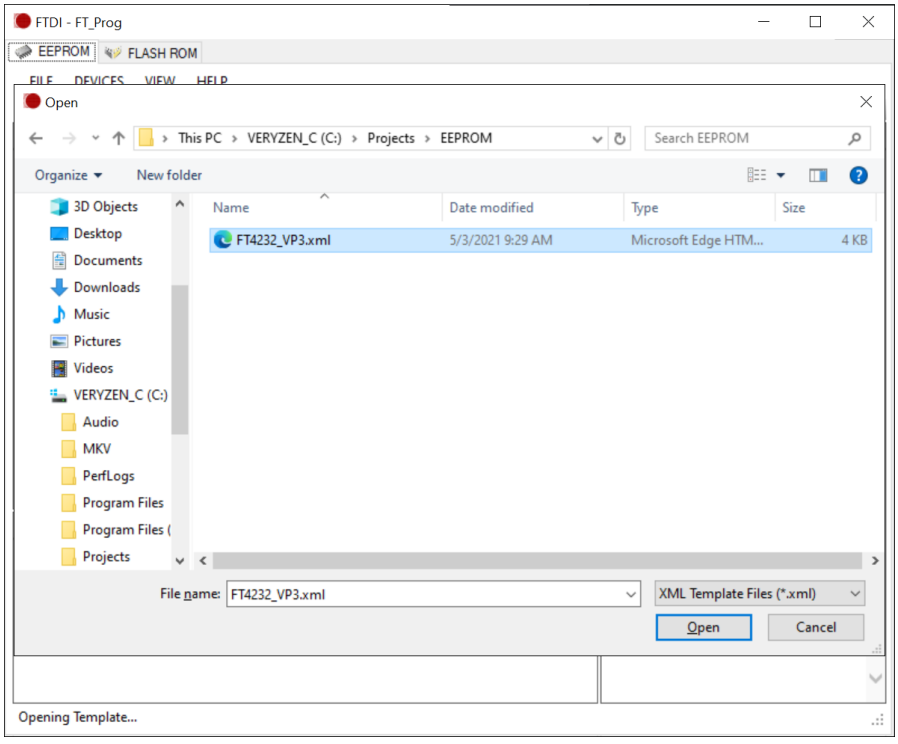


Figure 30 • Opening FT4232_VP3.xml

The parameter list can be seen on the left side under **Device Tree**. The **Vendor ID** and **Product ID** fields are under **USB Device Descriptor**. These fields should not be modified; otherwise, ACE cannot recognize the FTDI device.

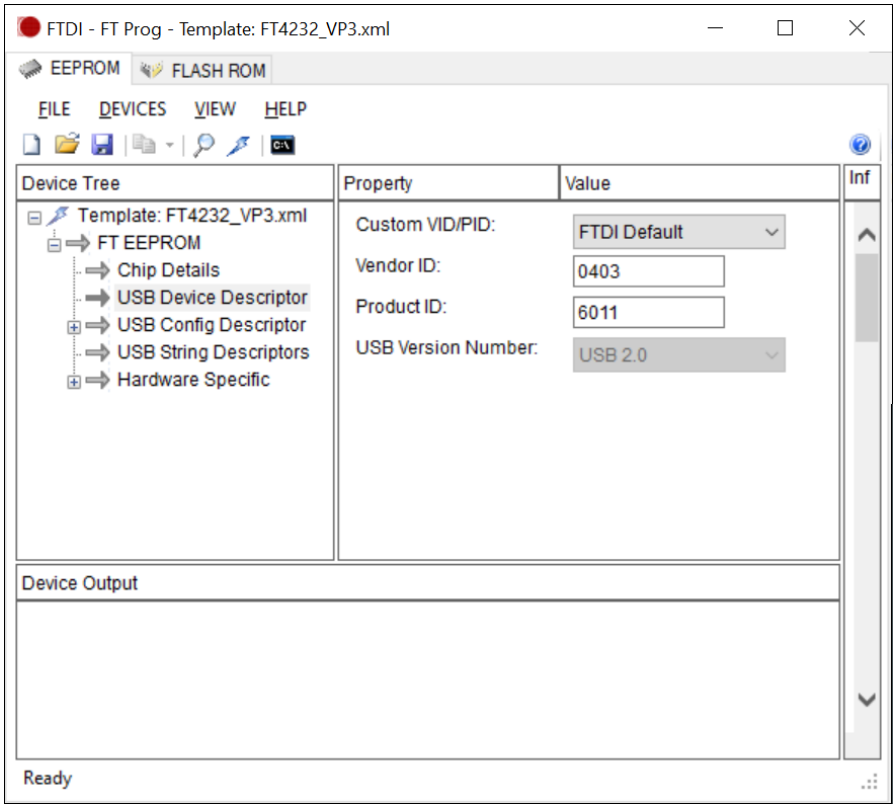


Figure 31 • Reviewing FT4232_VP3.xml

2. If required, modify the **Manufacturer** and **Product Description** fields under **USB String Descriptors**. A serial number can be specified manually or auto-generated.

> **Note**
>
> The value set for **Product Description** must contain the string "Achronix" to ensure proper operation. Achronix software uses the serial number to uniquely identify JTAG connections. Thus, it is highly recommended that the serial number be set to auto-generate. If the Achronix software cannot read a serial number, or finds it to be null/blank/empty, the Achronix software ignores the connected FT2232H/FT4232H device.

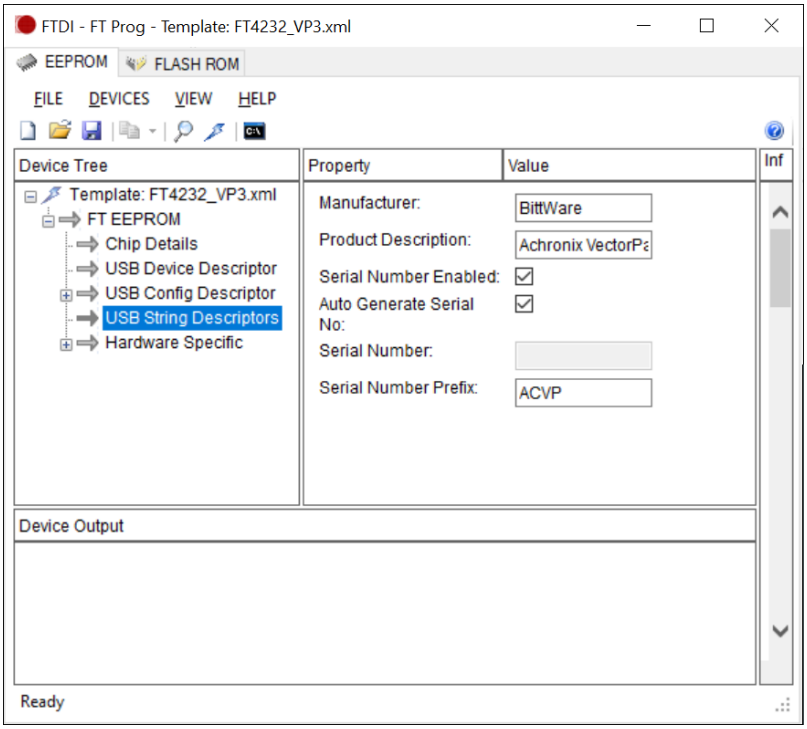


Figure 32 • Modifying The FT4232_VP3.xml File

EEPROM Interface – Board Implementation

The following figure shows the connection between EEPROM and FTDI chip on board.

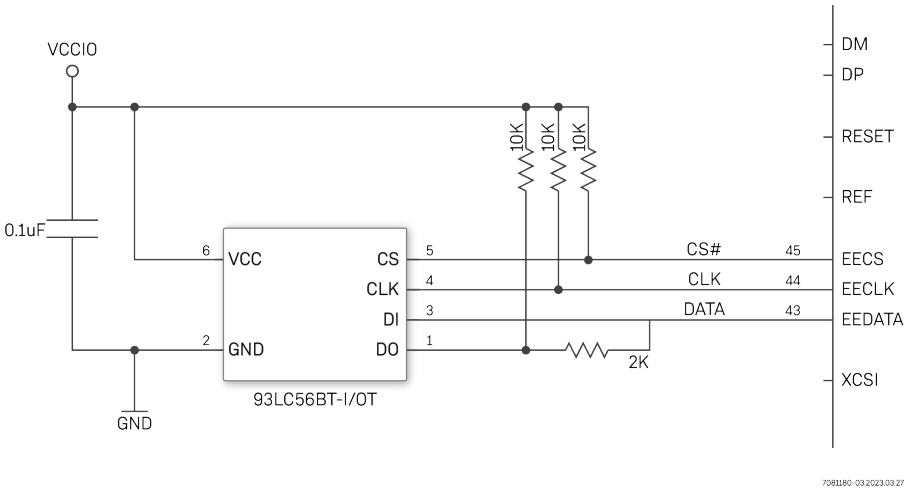


Figure 33 • EEPROM FTDI Board-Level Connection

FTDI Crystal Requirements

A 12 MHz crystal should be connected to the OSCI and OSCO pins of the FTDI 2232H chip. A 12KΩ resistor should be connected between REF and GND on the PCB. The value for the loading capacitors should be selected as per manufacturer recommendation.

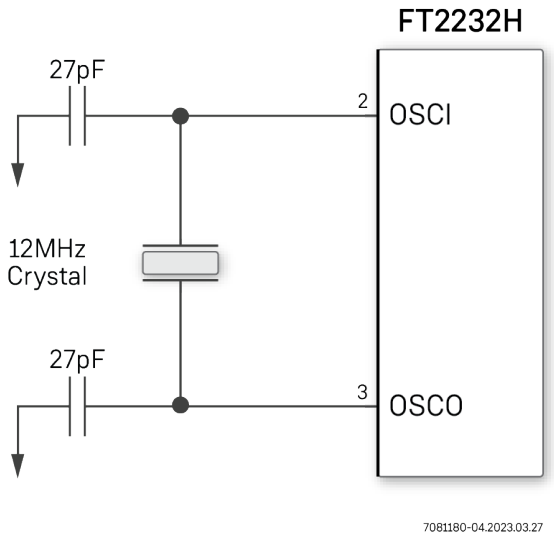


Figure 34 • FTDI Crystal Board-Level Connection

FTDI Interface in ACE

To use the FTDI interface in ACE, select **Window → Preferences → Configure JTAG Connection** and input the relevant information for the programmer device name and the scan chain details. ACE then knows to use FTDI for Snapshot, Download View (bitstream programming), and JTAG browser (and even SerDes link tuning). If no name is entered, ACE/STAPL player autodetects to select the programming device.

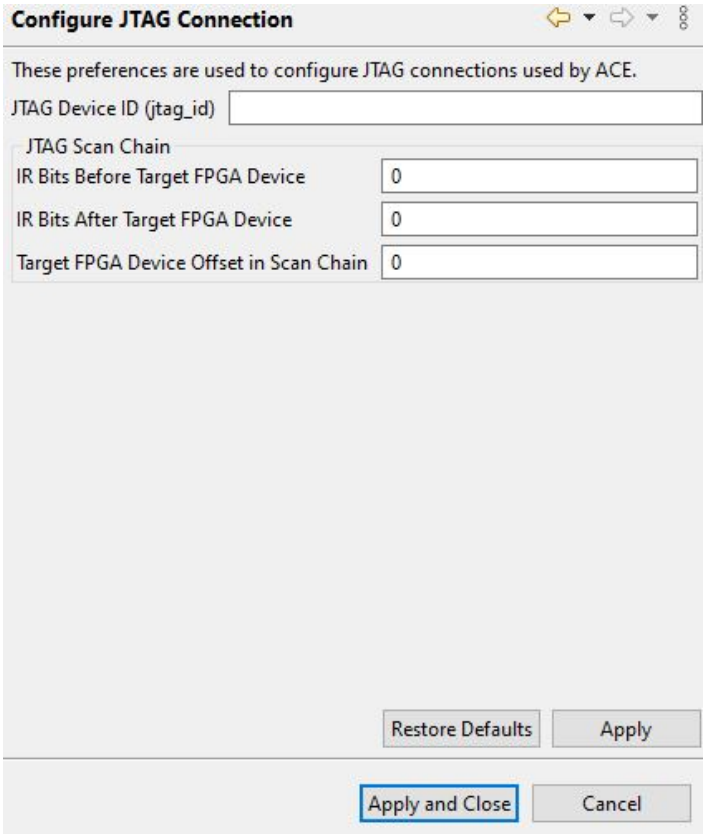


Figure 35 • Configuring the FTDI Interface in ACE

Programming Speeds and Requirements

JTAG Interface

The possible FT2232H/FT4232H frequencies are limited by FTDI to:

$$F = 60\ \text{MHz} / ((1 + \text{clkDiv}) \times 2)$$

Where clkDiv must be an integer ranging from 0 to 0xFFFF, thus providing an effective frequency range from 30 MHz (maximum) to 457.763 Hz (minimum).

The STAPL JTAG tools allow arbitrary frequencies to be requested (in integer Hz), but the drivers then choose the fastest frequency which is still less than or equal to the requested frequency.

> **Note**
>
> The STAPL frequency is presently not a user-editable value and is hard-coded in the STAPL player by Achronix in all current use cases.

Caution!

The `Tck` produced by the FTDI device is only present during programming. Further, its frequency accuracy and stability cannot be guaranteed. Therefore, it is not recommended to use this clock for any other purpose than JTAG programming of the device.

Known Device Limitations

While the Achronix JTAG tools can support multi-device JTAG scan chains, the Speedster7t AC7t1500ES0 FPGA JTAG test access port (TAP) does not support the JTAG BYPASS instruction. Because JTAG BYPASS is not supported, affected Speedster7t AC7t1500ES0 PPGAs must be the only JTAG device on its own JTAG scan chain. The JTAG BYPASS feature us fully supported for Speedster7t AC7t1500/AC7t1450 production FPGAs.

Software and Driver Install for FTDI

Introduction

Prior to device configuration, the FTDI USB drivers must be installed on the host system. The JTAG Tcl library and the USB drivers are included as part of the ACE software suite. Intended for general use, ACE includes a graphical download tool, the Snapshot debugging tool, the JTAG Browser tool, the HW Demo tool, as well as the JTAG Tcl library to aid in command-line configuration from ACE.

ACE and the Components Installation

When the ACE software suite is installed, a copy of the FTDI USB drivers is included. ACE installation is covered in a separate document, the ***ACE Installation and Licensing Guide*** (UG002)[6].

Windows

Near the end of the install, the ACE package prompts to install the FTDI CDM USB drivers:

6 https://www.achronix.com/documentation/ace-installation-and-licensing-guide-ug002

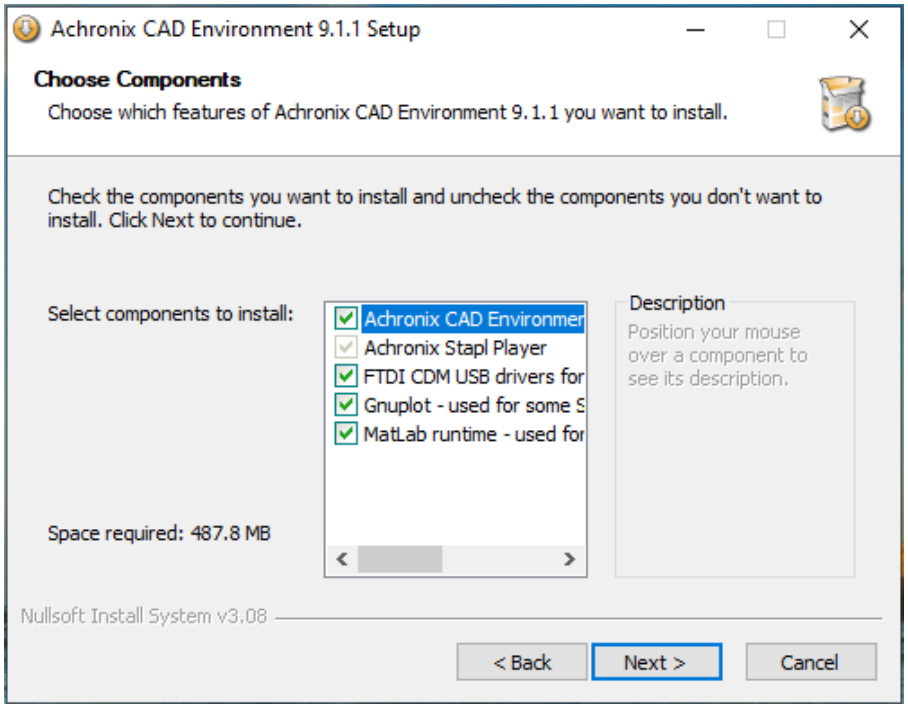


Figure 36 • ACE Installation Choose Components Dialog Example

Linux

> **Note**
>
> When using the FTDI FT2232H connection from Linux, RHEL/CentOS 7.4 and up have been successfully tested.

Linux USB Driver Installation

In Linux, the USB driver installation script can be found in the `system/cmd/` directory. Special udev rules must be created to set the permissions so that regular users may write to the FT2232H device. To update these rules, execute the following as root:

```
% system/cmd/install_acx_bitporter_usb.pl
```

Note

If using Ubuntu, it is recommended to use the following syntax:

```
% sudo system/cmd/install_acx_bitporter_usb.pl
```

The USB cables might need to be disconnected and re-connected after the install script is run. Whether or not the new rules are already applied depends upon implementation details within the Linux distribution.

Supported Operating Systems

JTAG interactions are currently supported with FTDI Interface or Bitporter 2 under the following operating systems.

- 64-bit Red Hat Enterprise Linux Release 7.9+, 8.x, and 9.x
- 64-bit CentOS 7.9-2009
- Rocky Linux 8.x, and 9.x
- 64-bit Microsoft Windows 10, 11
- Ubuntu 20.04 LTS, 22.04 LTS
- SUSE 15.4+

Minimum Hardware Requirements

- Pentium-class PC with a minimum of 512 MB of memory (2 GB for Windows 10)
- A USB 2.0 port if configuring through FTDI interface
- A powered USB 2.0 port if configuring through the Bitporter 2 pod

Note

1. USB 1.0 and 1.1 ports may be used for the Bitporter 2 and FTDI interfaces, but USB 2.0 is strongly recommended for performance reasons.
2. USB 3.x ports may be used for Bitporter2 or FTDI (both Linux and Windows) interfacing, but performance is limited to USB 2.0 speeds.

Connecting the FTDI Device

Connecting to the FT2232H or FT4232H via USB

Before connecting the FTDI USB port to the host PC, ensure that the software installation has completed (refer to Software and Driver Install for FTDI (page 56)).

Note

Depending upon the specific configuration of the FTDI chip on the board, in some cases the USB cable must be connected to a powered USB port on the host PC. In other configurations, an unpowered USB port suffices. Consult the board documentation for details.

Disconnecting the FT2232H or FT4232H Interface

To end a programming session and disconnect the FTDI USB cable from the target hardware:

1. Wait until ACE Tcl Console finishes running.
2. Close an existing connection to a JTAG device using the following command:

```
jtag::close <jtag_id>
```

3. Disconnect the USB cable from the target hardware.

Verifying the Setup

Connectivity Self Test

To verify that the USB drivers and FTDI JTAG interface are functioning together correctly:

1. Open a Tcl console in ACE.
2. At the command prompt, execute:

```
jtag::get_connected_devices
```

   The command gets the list of connected JTAG devices from the host machine along with the serial number (`jtag_id`).
3. Use the following command to open a new connection to the JTAG device using the `jtag_id` returned by the previous command.

```
jtag::open <jtag_id>
```

Handling Multiple FTDI Devices Connected to the Same PC

The ACE Tcl Console can support multiple users sharing a collection or pool of FTDI devices connected to a single PC via USB.

> **Warning!**
> When multiple JTAG devices are connected to a single PC, the desired JTAG connection must be specified.

If no specific connection is open, use the mentioned `jtag::open` command to open the connection. If a connection is already identified, close the current connection and open the connection to the desired device using the commands mentioned in the JTAG Programming using the Tcl Library API (page 32) section.

JTAG Configuration Using the Bitporter2 Pod

The Bitporter2 pod (pictured below) connects between a host PC via USB (1.x, 2.x, or 3.x) connection and a JTAG-compliant connector on the target system. When connected, the Bitporter2 pod supports device configuration and debug.

Note

USB 1.0 through 3.1 are supported, but are limited to USB 2.0 "High-Speed" or lower.



Figure 37 • Bitporter2 Pod

The JTAG configuration flow is as follows:

1. Generate a `design_name.hex` file from a placed-and-routed design within ACE.
2. Connect the Bitporter pod to the USB port of the host PC and to the JTAG port of the target system.

Software and Driver Install for Bitporter2

Introduction

The Bitporter2 pod utilizes the FTDI 2232 USB→JTAG interface chip. Prior to device configuration, the FTDI USB drivers must be installed on the host system. Intended for general use, ACE includes:

- A graphical download tool
- The Snapshot debugging tool
- The HW Demo tool

ACE and the Components Installation

When the ACE software suite is installed, a copy of the FTDI USB drivers is included. ACE installation is covered in a separate document, the ***ACE Installation and Licensing Guide*** (UG002)[7].

7 https://www.achronix.com/documentation/ace-installation-and-licensing-guide-ug002

Windows

Near the end of the install, the ACE package prompts to install the FTDI CDM USB drivers:

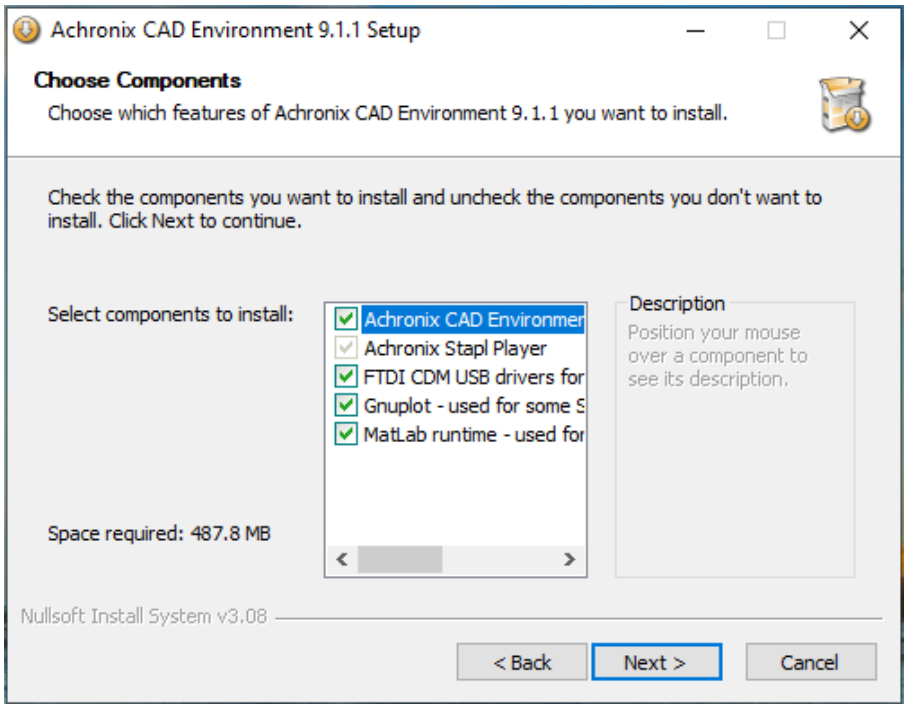


Figure 38 • ACE Installation Choose Components Dialog Example

Linux

> **Note**
>
> When using the FTDI FT2232H connection from Linux, RHEL/CentOS 7.4 and up have been successfully tested.

Linux USB Driver Installation

In Linux, the USB driver installation script can be found in the `system/cmd/` directory. Special udev rules must be created to set the permissions so that regular users may write to the FT2232H device. To update these rules, execute the following as root:

```
% system/cmd/install_acx_bitporter_usb.pl
```

Note

If using Ubuntu, it is recommended to use the following syntax:

```
% sudo system/cmd/install_acx_bitporter_usb.pl
```

The USB cables might need to be disconnected and re-connected after the install script is run. Whether or not the new rules are already applied depends upon implementation details within the Linux distribution.

Supported Operating Systems

JTAG interactions are currently supported with FTDI Interface or Bitporter 2 under the following operating systems.

- 64-bit Red Hat Enterprise Linux Release 7.9+, 8.x, and 9.x
- 64-bit CentOS 7.9-2009
- Rocky Linux 8.x, and 9.x
- 64-bit Microsoft Windows 10, 11
- Ubuntu 20.04 LTS, 22.04 LTS
- SUSE 15.4+

Minimum Hardware Requirements

- Pentium-class PC with a minimum of 512 MB of memory (2 GB for Windows 10)
- A USB 2.0 port if configuring through FTDI interface
- A powered USB 2.0 port if configuring through the Bitporter 2 pod

Note

1. USB 1.0 and 1.1 ports may be used for the Bitporter 2 and FTDI interfaces, but USB 2.0 is strongly recommended for performance reasons.
2. USB 3.x ports may be used for Bitporter2 or FTDI (both Linux and Windows) interfacing, but performance is limited to USB 2.0 speeds.

Connecting the Bitporter2 Pod

The Bitporter2 pod has two labeled jacks and two LED indicators:

- JTAG – used by the 14-pin JTAG ribbon cable
- USB – USB mini-B jack for communication with the host computer
- PWR LED – lights to indicate power from USB interface is present
- ACT LED – flashes to indicate data transfer to/from target

The Bitporter2 is powered by its USB interface.

Since the pod requires power, it does not work if it is not connected to a powered USB port.

Bitporter2 Board-Level Device Connections

JTAG Pinout

Table 26 • Bitporter2 Connections

Signal	Pin
TRST_N	1
TCK	9
TMS	7
TDI	3
TDO	5
V_JTAG	14
Ground	2, 4, 6, 8, 10

Bitporter2 Voltage Compatibility

The Bitporter2 derives power from the USB interface at 5V. Internally, it regulates this input to two voltage rails: 1.8V and 3.3V. The pod includes level shifters in order to match the JTAG interface voltage to that of the target. The target must supply the proper voltage on the JTAG interface pin 14.

Connecting the Bitporter2 Pod via USB

Caution!

Before connecting the Bitporter2 pod: Do not connect the Bitporter2 USB cable until after the software is installed (see **Software and Driver Install for Bitporter2** (page 60)). If the Bitporter2 USB cable is connected to the workstation during USB driver installation, the USB driver might not install correctly.

1. Turn off the power to the target hardware.
2. Connect one end of the JTAG flat ribbon cable to the target JTAG connector. Pin 1 is indicated by the red strip.

   Note

   If the target JTAG connector is not keyed, the target user guide should specify the location of pin 1 on the target JTAG connector.
3. Connect the other end of the JTAG flat ribbon cable to the Bitporter2 pod. The connector is keyed.
4. Connect one end the USB cable to the host PC.
5. Connect the other end of USB cable to the Bitporter2 pod.

6. Pod initialization:
   - During the pod initialization, the Bitporter2 pod power LED turns on and the ACT LED might flash.
   - When pod initialization completes successfully, the power LED remains lit, and the ACT LED turns off.

   > **Note**
   >
   > In Windows, after pod initialization is complete, a temporary notification might appear indicating that device drivers were successfully installed.

7. Turn on the power to the target hardware.

Verifying the Setup

Bitporter2 Connectivity Self Test

To verify that the USB drivers and Bitporter2 pod are correctly functioning together:

1. Open a Tcl console in ACE.
2. At the command prompt, enter:

   ```
   jtag::get_connected_devices
   ```

   The command returns a listing of all correctly connected and currently available pods (those not actively in use).
3. Execute the following command to open a new connection to the JTAG device using the `jtag_id` returned by the previous command.

   ```
   jtag::open <jtag_id>
   ```

Bitporter2-to-Target-Device Connectivity Test

After the Bitporter2 connectivity self test has successfully completed, it is still useful to ensure the Bitporter2 is properly connected to the target device via the JTAG ribbon cable. See the previous information in this chapter which describes the proper way to connect the Bitporter2.

1. Open a Tcl console in ACE.
2. Enter the following command:

   ```
   jtag::initialize_scan_chain <jtag_id> <pre_ir_bits> <post_ir_bits>
   <target_device_offset> -target_device <string>
   ```

   This command configures a scan chain. It sets the initial clock frequency (based on the target device), checks the number of devices and IR length, sets preamble/postamble IR/DR bits, and checks IDCODE.

   > **Note**
   >
   > The `<project_name>.hex` file can be generated from the user design or by using the quick-start design.

3. After successfully starting communications with the Bitporter Pod, the program returns the device ID code of the target device.

   > **Note**
   >
   > The actual text output, including the ID code, varies slightly by device type and revision.

   The following is an example of the text output including the IDCODE:

```
Checking JTAG device chain:
IR bits before target device: 0
IR bits after target device: 0
Target device offset: 0
Number of devices detected on scan chain: 1
Instruction Register Length is: 23
Set pre-IR padding to: 0
Set post-IR padding to: 0
Set pre-DR padding to: 0
Set post-DR padding to: 0
Verifying expected device is at the expected location...
Found JTAG IDCODE: 30400641
Achronix device detected: AC7t1500
...Verified Achronix device found at expected location.
```

Bitstream Programming via PCIe

Bitstream programming via PCI Express, also called config via protocol or CvP, requires a two-stage process in which part of the I/O ring is first programmed via flash, CPU, or JTAG, while the remainder of the I/O ring and the core is programmed later via PCIe. The two-stage process is required because the PCIe controller must be configured before the device can be enumerated by the host server as a PCIe endpoint.

In stage 0, the entire I/O ring does not need to be programmed – only the PLLs, related I/O, and the PCIe subsystem. Other subsystems (e.g., Ethernet/DDR4/GDDR6, clock networks to fabric, reset networks, NoC components, and the core fabric) are all programmed in stage 1. The size of the bitstream for the core fabric is significantly larger than that for the I/O ring, therefore, the stage 1 bitstream benefits the most from PCIe's fast download speeds.

This document explains how to configure the PCIe controller in a design to support CvP, which options to use in ACE to generate two-stage bitstream files, and the Achronix SDK tools that are available to download bitstreams from a host machine to an FPGA card over PCIe.

Design Requirements for Programming via PCIe

There are few requirements on the PCIe subsystem configuration when using CvP. Stage 1 bitstream download requires access to the PCIe controller's configuration and status registers (CSRs) in order to set up direct memory access (DMA) transactions. Therefore, one of the PCIe base address registers (BARs) must be configured to allow access to the PCIe controller's direct bus interface (DBI).

During normal operation, most FPGA designs will map a BAR to the DBI Gateway, a soft logic macro that provides synchronization for the DBI CSRs. For more information see the knowledge base article, **What is the DBI Gateway and How Can I Use It?**[8] Full bitstreams may configure the PCIe controller with either the "full" or "compressed" DBI Gateway BAR mapping. However, the DBI Gateway soft logic is not available during two-stage CvP because the FPGA core is not configured by the stage 0 bitstream. Therefore, the stage 0 bitstream must configure a PCIe BAR to map to the low-level hard DBI that is internal to the PCIe controller.

The Achronix SDK hides the complexity of using the hard DBI, but it does require a different PCIe BAR configuration. Instead of the "full" or "compressed" mappings to the DBI Gateway, the stage 0 bitstream must map a BAR directly to the start of PCIe CSR space, which is at NOC address `0x081_9100_0000`.
The `acx_pcie_program_bitstream` tool in the Achronix SDK, described below, takes the DBI BAR number as a command line argument.

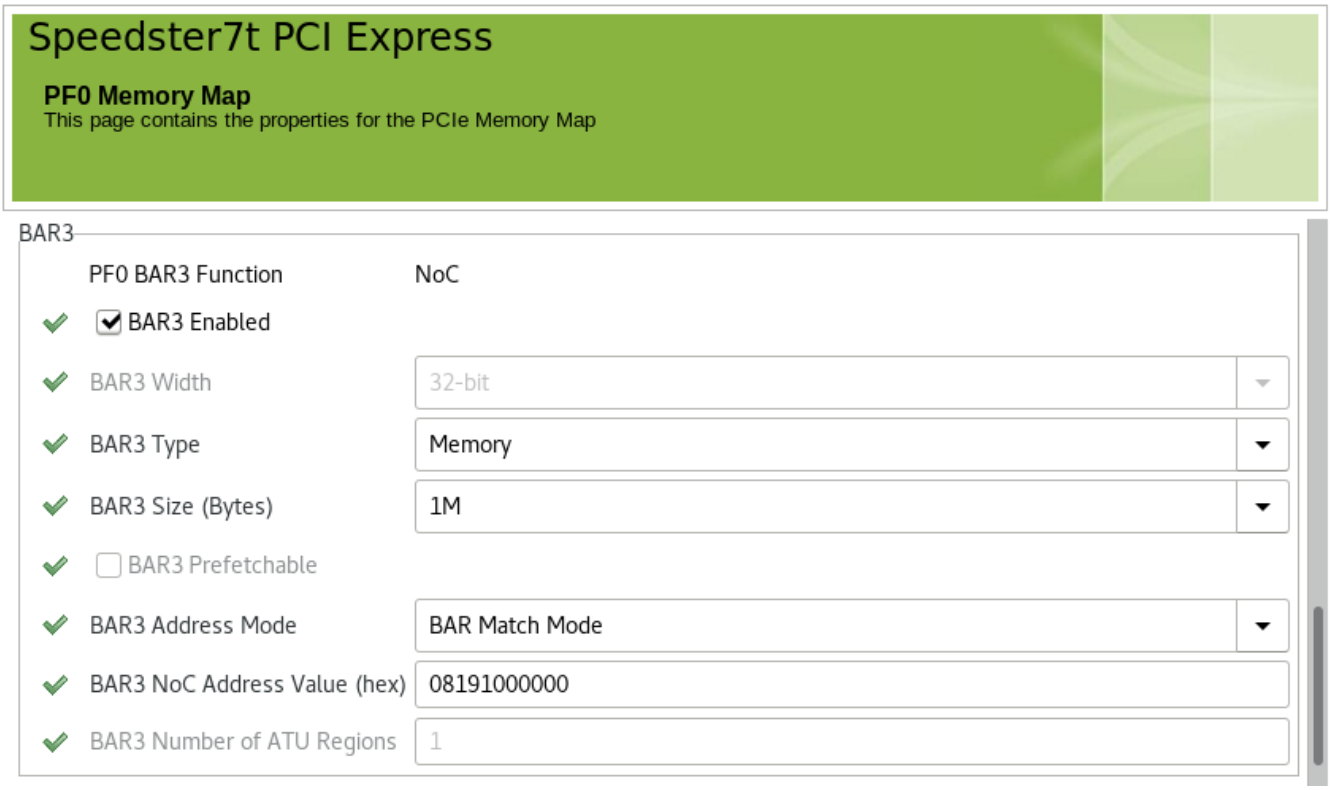


Figure 39 • Example of BAR3 Mapped to PCIe CSR Space for Hard DBI

Because the hard DBI is not synchronized, Achronix does not recommend using the hard DBI for general applications. It should only be used within the scope of CvP. If an application needs access to the DBI outside of CvP, then it is recommended to map a separate BAR to the Gateway's NoC address. After CvP is complete the runtime application may repurpose the hard DBI BAR for other uses. How to dynamically reconfigure BARs at runtime is beyond the scope of this document. See the *Software Development Kit User Guide* (UG107)[9] for more information.

8 https://support.achronix.com/hc/en-us/articles/34117665702292-What-is-the-DBI-Gateway-and-How-Can-I-Use-It
9 https://achronix.com/documentation/software-development-kit-user-guide-ug107

How to Generate the PCIe Bitstream Files with ACE

Two sets of implementation options must be changed to enable two-stage bitstream programming over PCIe.

Bitstream Generation – Additional Outputs Section

In the Additional Outputs sub-section of the Bitstream Generation section of the ACE Options View, select **the PCIe (.pcie)** option to generate the `.pcie` file for stage 1 config over PCIe. Optionally, select the **Serial Flash (.flash)** option to generate the `_stage0.flash` file for two-stage bitstream programming and the `.flash` file for one-stage bitstream programming via flash memory. A `_stage0.hex` file for two-stage programming and a `.hex` file for one-stage programming over JTAG is always output. The bitstream(s) will be exported during the Generate Bitstream flow step in ACE.



Figure 40 • The Additional Outputs sub-section of the Bitstream Generation Section in the ACE Options View

Option	ACE impl_option	Description
PCIe (.pcie)	`bitstream_output_pcie`	Enables the generation of an PCIe-formatted output file, with the same name as the `.hex` file, but having a `.pcie` extension. This file is binary formatted and can only be used with two-stage programming or partial reconfiguration.
Serial Flash (.flash)	`bitstream_output_flash`	Enables the generation of an additional serial flash formatted output files, with the same name as the .hex file, but having a `.flash` extension. This file is binary formatted. Only the `_stage0.flash` file is needed when stage1 will be loaded over PCIe.

Bitstream Generation – Two Stage Programming Section

The **Two-Stage Programming** option must be selected to generate the bitstream files needed for two-stage CvP. With the **Two-Stage Programming** option enabled, ACE will export a `_stage0.hex` file and optionally, if the Additional Outputs option **Serial Flash (.flash)** is enabled, a `_stage0.flash` file. When generating encrypted AC71400/AC7t1500 bitstreams, additional files `_stage1_header.hex` and `_stage1_header.flash` are also generated. An error will be thrown it a user enables the Additional Outputs option **PCIe (.pcie)** but does not enable the **Two-Stage Programming** option.



Figure 41 • ACE Bitstream Generation Options Dialog

Table 27 • Bitstream Generation Implementation Options — Two-Stage Programming

Option	ACE impl_option	Description
Enable Two-Stage Programming	`bitstream_two_stage`	If checked (`1`), enables two-stage programming. This generates files with `*_stage0.*` and `*_stage1.*` naming. This option only creates stage0 files for flash and stage1 files for PCIe when enabled.

How to Program a Device with the PCIe Bitstream Files

The FPGA must be configured with a stage 0 bitstream (`.hex` or `.flash`) before the host machine can enumerate the device as a PCIe endpoint. When using encryption (for AC7t1500/AC7t1400), the stage 1 header must also be programmed into the flash memory or over JTAG following the stage 0 bitstream. Refer to the sections, "**Bitstream Programming via JTAG** (page 28)" and "**Bitstream Programming via Flash Memories** (page 9)" for information on how to program the stage 0 bitstream, and optionally the stage 1 header bitstream, over JTAG or flash, respectively.

Once the device has been enumerated as a PCIe endpoint, there are two methods available to download the `.pcie` bitstream file to the device. Both methods are made available as part of the Achronix software development kit (SDK). See the knowledge base article **How do I Download the Achronix SDK?**[10] (requires login to access) and the ***Software Development Kit User Guide*** **(UG107)**[11] for more information about the SDK.

Standalone PCIe Programming Utility

Use the acx_pcie_program_bitstream utility which is available in the `SDK/tools/acx_pcie_program_bitstream` directory of the Achronix SDK. After SDK installation, the executable is also available in `/opt/achronix/sdk/bin`. Call `acx_pcie_program_bitstream --help` for usage information, or see the ***Software Development Kit User Guide*** **(UG107)**[12]

Programmatic Access Using Achronix SDK Library Functions

There is also the option to integrate CvP into the user application by calling the function `acx_pbs_program_bitstream()`, which is available in the `Achronix_program_bitstream.h` header file and Achronix SDK shared-object library. See the SDK man pages and ***Software Development Kit User Guide*** **(UG107)**[13] for more information. Source code for the acx_pcie_program_bitstream utility, discussed in the previous section, is available in the SDK download, and may be used as a reference when creating user software applications.

10 https://support.achronix.com/hc/en-us/articles/15234194799252-How-do-I-Download-the-Achronix-SDK
11 https://achronix.com/documentation/software-development-kit-user-guide-ug107
12 https://achronix.com/documentation/software-development-kit-user-guide-ug107
13 https://achronix.com/documentation/software-development-kit-user-guide-ug107

Chapter 4 : FPGA Configuration Unit (FCU)

The term FPGA configuration unit (FCU) refers to logic that controls the configuration (bitstream programming) process of the Speedster7t FPGA. This logic is responsible for the following:

- Receiving data on a variety of external interfaces (depending on the selected programming mode)
- Decoding instructions
- Sending configuration bit values to the appropriate destination (e.g., core configuration memory, the core boundary ring configuration memory, FCU registers, etc.)
- Controls the startup and shutdown sequences that drive resets to the on-chip logic
- Bitstream CRC checks
- SEU mitigation with CMEM scrubbing
- Bitstream Encryption Security
- Any core-level housekeeping that occurs on the de-assertion of reset (i.e., clearing of configuration memory)

Overview

The following features are supported by the FCU:

- Multiple bitstream programming configuration modes (see **Bitstream Programming Modes for Speedster7t FPGAs** (page 4))
- Bitstream **CRC** (page 0) checks. Bitstream errors are detected by CRC, and the bitstream fails to program if a CRC error is detected.
- AES encryption/decryption and bitstream security (**Design Security for Speedster7t FPGA** (page 94))
- Configuration memory scrubbing and SEU mitigation (single-bit error correction, dual-bit error detection) (**Configuration Error Correction and SEU Mitigation** (page 85))
- Read-back (**JTAG Programming using the Tcl Library API** (page 32))

The FCU has thee main operating modes:

- **Power-on** – triggered after the input signal `FCU_CONFIG_RSTN` is driven high. When the FCU state machine starts, it progresses through a number of housekeeping activities, including the clearing of the configuration memory if needed. This housekeeping happens without any additional user input. All instructions sent via one of the programming interfaces during this time are ignored. At the end of this mode, the output pin `FCU_CONFIG_STATUS` (earlier driven low) is driven high, and the FCU returns to the instruction processing mode.
- **Bitstream Programming** – in this mode, the FCU functions as a simple CPU, processing incoming instructions and sending control signals downstream as directed. Instructions are received on 128-bit boundaries but processed 32 bits per clock cycle. The FCU can request data from the host or stall when it is processing the previous instruction. Depending on the programming interface being used, a set of output status signals generated by the FCU are used to determine how to proceed. Refer to **Bitstream Programming Modes for Speedster7t FPGAs** (page 4) for additional details. While the bitstream is being programmed, `FCU_CONFIG_STATUS` is driven high. When all of the bitstream data is loaded, `FCU_CONFIG_DONE` is asserted if the bitstream download was successful. The FCU then signals the startup state machine to release

resets and enter user mode. If there were any errors programming the bitstream, FCU_CONFIG_DONE stays low and the FCU_CONFIG_ERR_ENC bus can be checked to determine the error.

- **User Mode** – if the bitstream programming sequence completed successfully, and the startup state machine has completed release of all startup sequenced resets, FCU_CONFIG_DONE remains high and FCU_CONFIG_USER_MODE is raised.

Configuration Pin Tables

Warning!

Bidirectional FCU pins, including FCU_CONFIG_DONE, FCU_PARTIAL_CONFIG_DONE, and FCU_CONFIG_STATUS, *must be properly pulled up*. Missing or incorrect pull-ups will prevent the FPGA from operating correctly .

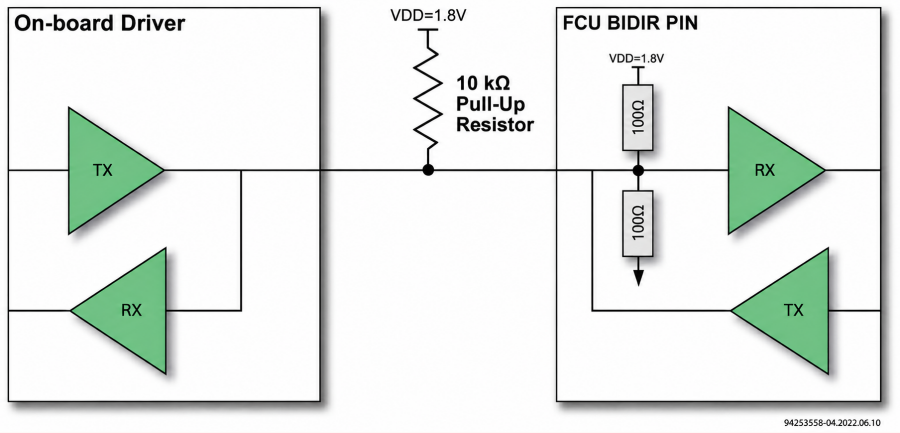


Table 28 • Interface Pin Table

Pin Name	Direction	Usage	
Configuration Interface			
FCU_CONFIG_MODESEL[3:0	Input	FPGA configuration unit (FCU) configuration mode selection inputs.	
		Configuration Mode	**CFG_MODESEL[3:0]**
		CPU x1	0011
		CPU x8	0100
		CPU x16	0101
		CPU x32	0110
		CPU x128	0111

		Flash SPI (x1)-1D	0001
		Flash SPI (x1)-4D	0010
		Flash Dual (x2)-1D	1000
		Flash Dual (x2)-4D	1001
		Flash Quad (x4)-1D	1010
		Flash Quad (x4)-4D[9]	1011
		Flash Octa (x8)-1D	1100
		Flash Octa (x8)-4D[9]	1101
		JTAG	Always active mode
FCU_CONFIG_STATUS [1]	Inout [2]	Active-high configuration status open-drain output signal indicating that the FCU has completed initial start-up, has cleared the CMEM, and is awaiting FCU commands for bitstream programming. When Hi-Z, it remains Hi-Z until the FCU is power-cycled, reset for a re-initialization sequence, or a CRC error is seen during bitstream load.	
FCU_CONFIG_DONE [1]	Inout [2]	Active-high configuration done open-drain output signal indicating that bitstream loading completed successfully and that the device is ready to enter user mode. When Hi-Z, it remains Hi-Z until the FCU is power-cycled or reset for a re-initialization sequence. If a device configuration error occurs, the CONFIG_DONE output remains low. Holding this pin low on the board must be used as a method to synchronize the start-up of multiple devices. **Warning!** This pin must be tied High (even if not externally driven) as the FCU proceeds only if a high value is present on this pin (see note 6).	
FCU_CONFIG_RSTN [3]	Input	Asynchronous active-low reset input clearing the configuration memory in the device and the logic in the FCU.	
FCU_CONFIG_USER_MODE [1]	Output	Active-high output indicating the device has transitioned to user mode. When high, it remains asserted until the FCU is power-cycled or reset for a re-initialization sequence.	
FCU_CONFIG_SYSCLK_BYPASS FCU_CONFIG_CLKSEL [4]	Input	Active-high bypass configuration system clock setting. Along with CFG_CLKSEL, this setting allows for clock selection during programming. Setting SYSCLK_BYPASS high and CFG_CLKSEL low is the most recommended setting since it is compatible with all programming modes.	
		SYSCLK_BYPASS	**CFG_CLKSEL** / **CFG_MODESEL[3:0]** / **Configuration Clock**
		0	0 / 0000, 0001, 0010, 1000 to 1101 / On-chip Oscillator [5]
		1	0 / 0000, 0001, 0010, 1000 to 1101 / CPU Clock [5]
		X	0 / 0011, 01XX / CPU Clock [5]
		X	1 / XXXX / JTAG TCK
FCU_CONFIG_BYPASS_CLEAR	Input	Active-high input pin to bypass configuration memory clear during device initialization.	
FCU_CONFIG_ERR_ENC[2:0]	Output [6]	Error status	

		FCU_CONFIG_ERR_ENC[2:0]	Status	Priority
		000	No error	
		010	CRC Error.	0 (Lowest)
		001	Single-bit/multiple-bit scrubbing error.	1
		011	Secure Boot Failure OR Security error.	2
		100	Efuse PUF enrollment error.	3
		101	Asserted when the AXI interface of the IP configuration space register block does not receive a ready from the initiator.	4
		110	Secure boot authorization error.	5 (Highest)
		Other	Undefined.	
FCU_LOCK	Output	AC7t1500 – this pin is set high when the FCU is unlocked and low when FCU is locked. AC7t800 – this pin is set high when stage0 bitstream programming is complete.		
FCU_OSC_CLK	Output	This clock is internally generated from a ring oscillator. For debug purposes, it can be bypassed and the external clock, CPU_CLK, can be used. This is an internal ring oscillator used to provide a free-running clock to the FCU and its frequency can vary by process and temperature.		
FCU_PARTIAL_CONFIG_DONE	Inout (2)	Active-high configuration done open-drain output signal indicating that bitstream loading completed successfully for partial reconfiguration of the FPGA and that it is ready to enter user mode. **Warning!** This pin must be tied High (even if not externally driven) as the FCU only proceeds when a high value is present.		
FCU_STAP_SEL	Input	When asserted high, this signal enables the JTAG interface pins to be directly connected to the JTAG controller in the SerDes PMA blocks allowing SerDes configuration, debug, and performance monitoring directly from the JTAG interface. For bitstream download and design debug using the JTAG interface, this pin must be held low. For SerDes PMA debug only mode, this pin must be held high.		
FCU_STATUS[1:0]	Output	Status bits showing the FCU state.		
		FCU_STATUS	State	
		11	fcu_locked	
		10	sync_found	
		01	ID found	
		00	instance ID found / FCU unlocked	
FCU_STRAP[2:0]	Inputs	Test mode input pins. When FCU_STRAP[0] is 0, FCU is in functional mode and when FCU_STRAP[0] is 1, FCU_CPU_CSN is gated and only controllable via test logic.		
JTAG Interface				
JTAG_TCK	Input	Clock input to the FCU JTAG controller.		

JTAG_TRSTN	Input	Active-low reset input to the FCU JTAG controller.
JTAG_TDI	Input	Serial data input to the FCU JTAG controller. Synchronous to JTAG_TCK.
JTAG_TDO	Output	Serial data output from the FCU JTAG controller. Synchronous to JTAG_TCK.
JTAG_TMS	Input	Mode select input to the FCU JTAG controller. Synchronous to JTAG_TCK.
Flash Memory Interface		
FCU_FLASH_SCK	Output	Clock output from FCU to flash memory device(s).
FCU_FLASH_HOLDN	Output	Active-low hold output to flash memory device(s). This signal is used to pause serial communications between the Speedster7t FPGA and the flash device without deselecting the device or stopping the serial clock. Synchronous to FLASH_SCK.
FCU_FLASH_CSN[3:0]	Output	Active-low chip select to enable/disable one or more of the attached flash memory devices. For x1 mode, only CSN[0] is used. For x4 mode, connect each CSN[3:0] to a flash device.
CPU Interface		
FCU_CPU_CLK (7)(10)	Input	Input clock from external CPU. The data/address bus is synchronous to this clock. This signal must be driven continuously, regardless of programming mode and also during user mode. This clock must operate at 100 MHz when programming any design bitstream that uses the Achronix Device Manager. A high-quality, stable clock for this required. Recommended V_{IL}=300 mV, V_{IH}=1500 mV, Duty Cycle=45-55%, Total Jitter < 0.1 UI.
FCU_CPU_CSN (8)	Input	Active-low CPU mode chip select.
FCU_CPU_DQ_IN_OUT[31:0]	Input/ Output	Data Input/Output pins shared between the CPU and Flash interfaces. The CPU interface is inaccessible when the Flash mode is in use and vice-versa. This bus is only 8-bits wide for the AC7t800/AC7t700 devices.
FCU_CPU_DQ_VALID	Output	Active-high control bit to indicate to the CPU the clock cycles when the CPU_DQ bus has valid read-back data. Synchronous to FCU_CPU_CLK.

Table Notes

1. Refer to the **Configuration Sequence and Power-Up** (page 83) section of the user guide for details.
2. This output is an open-drain signal. In the default mode of operation, it is recommended that this signal be connected to an LED as an indicator on the board. In this case, use an external 10kΩ ±5% pull-up resistor to 3.3V to drive a 1kΩ resistor to the input of a FET to turn on the LED. If LED usage is not desired, this signal must be pulled-up to 1.8V (`FCU_CB_VDDIO`) instead using the same 10kΩ pull-up resistor. The I/O standard for this is SSTL-18 and the recommended minimum pull-up impedance is 68Ω.
3. `FCU_CONFIG_RSTN` must be held low, and cannot glitch during device power-up. All other input pins need only be stable when i_config_rstn is ready to be released after power-up.
4. Regardless of configuration mode, `FCU_CONFIG_CLKSEL` must be set to "0" and `FCU_CONFIG_SYSCLK_BYPASS` must be set to "1" when programming any design bitstream that uses the Achronix Device Manager. This is also necessary in order for CSR access after a bitstream has been downloaded into the FPGA. Achronix recommends this as the default setting as well.
5. In JTAG configuration mode, soft control is used to select the JTAG clock as the configuration clock for the duration of the bitstream download.
6. All configuration status related output signals are driven from registers. The reset value for these registers is "0", and the transition from "0" to "1" is glitch free after reset de-assertion and when reaching the appropriate FCU states.
7. `FCU_CPU_CLK` can either start with a rising or falling edge. An SMA connector or input circuit must be utilized to apply this clock signal. An `FCU_CPU_CLK` input provision is required in order to utilize the Achronix Device Manager. For more details about the Achronix Device Manager, consult the ***Speedster7t Soft IP User Guide*** (UG103)[14].
8. Refer to the `FCU_CPU_CSN` **Bitstream Programming Via CPU** (page 5) section of the user guide for details.
9. Not supported for AC7t700/AC7t800 devices.
10. Although SSTL standards only require V_{IH} and V_{IL} values of ±200 mV, Achronix places a more stringent requirement on this clock to ensure a robust incoming clock from the PCB.

14 https://www.achronix.com/documentation/speedster7t-soft-ip-user-guide-ug103

Chapter 5 : Bitstream Generation Software Support in ACE

Bitstream Generation

ACE has a straightforward interface to generate the bitstream files required to implement all of the supported configuration modes. The bitstream file is generated during the the "FPGA Programming – Generate Bitstream" step of the compilation flow (see "Flow View" in the *ACE User Guide* (UG070)[15] for more details). This page is a consolidated list of all implementation options for reference. For specific use cases, refer to the sections above.

Bitstream Output File Formats

ACE provides a set of implementation options to generate the bitstream in various file formats for each of the supported bitstream programming modes. The `.hex` file needed for JTAG mode configuration is always generated by default. The **Additional Outputs** section of the **Project Options** menu, shown in the following figure, also provides a menu option to generate bitstream files for the other configuration modes.



Figure 42 • ACE Additional Output Options Dialog

Table 29 • Bitstream Generation Implementation Options — Additional Outputs

Option	ACE impl_option	Description
Serial Flash (.flash)	`bitstream_output_flash`	Enables the generation of an additional serial flash-formatted output file, and the page0 header file, having the same name as the `.hex` file, but with a `.flash` extension. The file contains a binary image that can be directly loaded into a single serial flash memory.

15 https://www.achronix.com/documentation/ace-user-guide-ug070

Option	ACE impl_option	Description
CPU Mode (.cpu)	`bitstream_output_cpu`	Enables the generation of an additional CPU-Mode-formatted output file, having the same name as the `.hex`file, but with a `.cpu` extension. The file contains hexadecimal-formatted data organized with (CPU bus width) number of bits per file line. Data from this file is sent to the FCU CPU interface line by line (one line per clock cycle) from the top to the bottom of the file, where the left-most bit on each line is the MSB and the right-most bit is the LSB. In simulation, this file can be loaded using the `readmemh` function. For convenience, an additional binary representation of the CPU Mode output file is written, having the same name as the `.hex` file, but with a `_cpu.bin` extension. It contains the same data in the same bit order as the `.cpu` file, but in binary format with no new-lines.
CPU Bus Width	`bitstream_output_cpu_width`	Controls the bit width of the CPU-mode-formatted output file. When using the CPU interface in ×8 mode, set this value to **8**. If using the CPU interface in ×32 mode, set this to **32**. The value determines how many bitstream bits are printed per line in the `.cpu` output file. The bit sequence required by the FCU (and output in the generated bitstream file) might be different for each CPU bus width setting. Therefore, it is important to set this option to match the actual CPU hardware interface width.
PCIe (.pcie)	`bitstream_output_pcie`	Enables the generation of an additional PCIe formatted output file, having the same name as the `.hex` file, but with a `.pcie` extension. This option is binary-formatted and can only be used with two-stage programming or partial reconfiguration.
CMEM Address and Data Export (.address)	`bitstream_output_address`	Enables an additional CMEM address and data export output file. All addresses listed in this file are "used" in the bitstream. The data in this file can be compared against readback data. The file has the same name as the `.hex` file, but with the `.address` extension.

Serial Flash Configuration Options

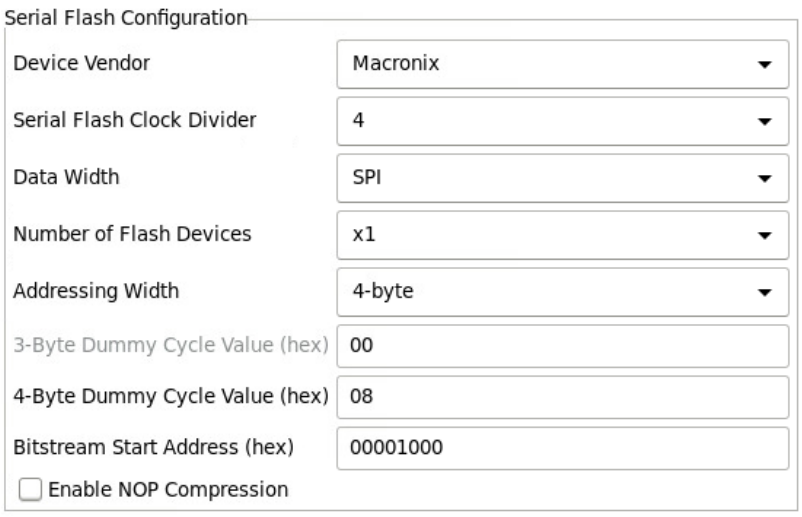


Figure 43 • ACE Serial Flash Configuration Options Dialog

Table 30 • Bitstream Generation Serial Flash Configuration Options

Option	ACE impl_option	Description
Device Vendor	`bitstream_page0_vendor`	Selects the flash device vendor. Allowed values: **Macronix** (`0`) **Micron** (`1`)
Serial Flash Clock Divider	`bitstream_page0_sf_clock_div`	Selects serial flash clock divider. Allowed values: **4** **8**
Data Width	`bitstream_page0_data_width`	Selects flash data readback width. Allowed values: **SPI** (`0`) **DUAL** (`1`) **QUAD** (`2`) **OCT** (`3`)
Number of Flash Devices	`bitstream_page0_num_devices`	Selects number of devices based on targeted x1 or x4 PROM. Allowed values: **x1** (`0`) **x4** (`1`)
Addressing Width	`bitstream_page0_addr_width`	Selects 3-byte or 4-byte addressing mode to support flash devices >1Gb. Allowed values: **3-byte** (`0`) **4-byte** (`1`)
3-Byte Dummy Cycle Value (hex)	`bitstream_page0_dummy_cycle_3byte`	Specifies the 3-byte addressing dummy cycle value. The default value is **00** and varies by device vendor. Must be specified as a 2-character hex value.
4-Byte Dummy Cycle Value (hex)	`bitstream_page0_dummy_cycle_4byte`	Specifies the 4-byte addressing dummy cycle value. The default value is **08** and varies by device vendor. Must be specified as a 2-character hex value.
Bitstream Start Address	`bitstream_page0_start_addr`	Specifies the bitstream start address. Should be a non-zero multiple of 4096. Must be specified as an 8-character hex value.
Enable NOP Compression	`bitstream_page0_compress_nops`	When unchecked (0), the `*.flash` file for I/O Ring programming is similar to other programming modes (CPU, JTAG, Hex, etc). When checked, the `*.flash` file bitstream contents are compressed, to help meet the 100ms PCIe link-up time. This results in a different bitstream for serial flash, which is dependent on the overall FCU data width (Number of Devices x Device Data Width).

Encryption Options

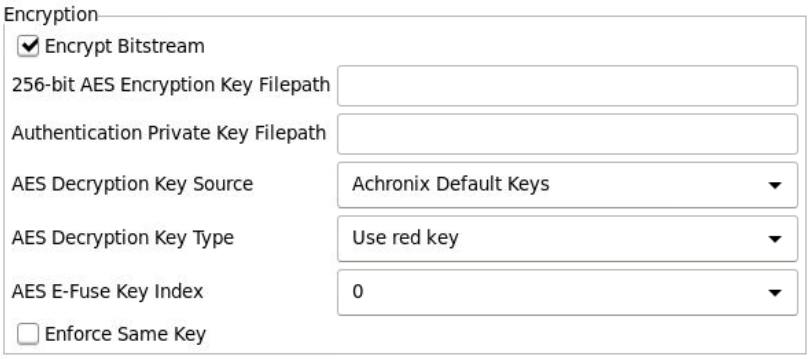


Figure 44 • ACE Encryption Options Dialog (Showing AC7t1500 Options)

Table 31 • Bitstream Generation Implementation Options — Encryption

Option	ACE impl_option	Description
Encrypt Bitstream	`bitstream_encrypted`	Check the box if bitstream should be encrypted. This option is always enabled for certain devices.
256-bit AES Encryption Key Filepath (Device Specific)	`bitstream_encryption_aes_key_file`	If the Achronix default Keys are not selected as the key source, enter a file name and path in the box to encrypt the bitstream data. The file must be: • an absolute or relative path to the current ACE project • a .txt file type • an AES hexadecimal value within the `.txt` file: ◦ For ACE installations 9.0 or later, any 256-bit or 64-character value. ◦ For ACE installations before 9.0, any 256-bit or 64-character value with a new line character at the end of the AES sequence. The total file would then be 260 bits or 65 characters.
Authentication Private Key Filepath (Device Specific)	`bitstream_encryption_pem_key_file`	If **Achronix Default Keys** are not selected as the decryption key source, enter a file name and path in the box to encrypt the bitstream data. This must be an absolute or relative path to the current ACE project. This should be a `.pem` file type created in the **Generating a Public and Private Key Pair** section.
AES Decryption Key Source	`bitstream_encryption_key_source`	Specifies which keys on the target device are used for decryption. Allowed values: **E-Fuse keys** (`0`) **Achronix Default Keys** (`1`)
AES Decryption Key Type [1]	`bitstream_encryption_key_type`	Specifies which key to use as the source during encryption. Allowed values: **Use PUF black key** (`0`) – create the red key from the PUF black key. **Use red key** (`1`) – treat the red key as the source.

Option	ACE impl_option	Description
AES E-Fuse Key Index	`bitstream_encryption_key_index`	Specifies which key to use. Bitstreams may be encrypted with 4 different AES keys. This is the index whose key value should be assigned to the 256-bit AES encryption key filepath. This also requires **AES Decryption Key Source** to be set to **E-Fuse Keys**, specifying to ACE to use E-fuse keys. The index value can be **0**, **1**, **2**, or **3**. In order to decrypt the target FPGA, the AES key must be written to the corresponding key index in the FCU.
Enforce Same Key	`bitstream_encryption_same_key`	Specifies whether subsequent bitstreams can be programmed with the same encryption types and keys without resetting the FPGA. If checked (1), subsequent bitstreams must use the same key source, key type, and key index unless the FPGA has been reset.

Table Notes

1. When using the Achronix default keys as the decryption key source, the red keys must be treated as the source, and PUF is not allowed.

For more details regarding encryption, refer to Design Security for Speedster7t FPGA (page 94).

Two-Stage Configuration Option



Figure 45 • ACE Bitstream Generation Options Dialog

Table 32 • Bitstream Generation Implementation Options — Two-Stage Programming

Option	ACE impl_option	Description
Enable Two-Stage Programming	`bitstream_two_stage`	If checked (1), enables two-stage programming. This generates files with `*_stage0.*` and `*_stage1.*` naming. This option only creates stage0 files for flash and stage1 files for PCIe when enabled.

For more details regarding two-stage programming, refer to Bitstream Programming via PCIe (page 65).

Partial Reconfiguration Configuration Options

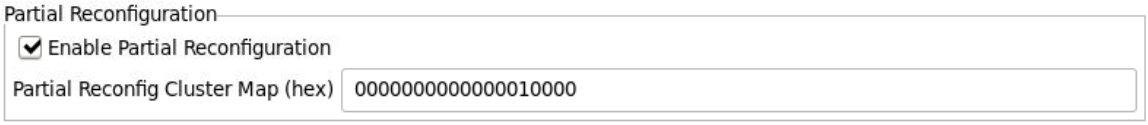


Figure 46 • ACE Partial Reconfiguration Options Dialog

Table 33 • Bitstream Generation Implementation Options — Partial Reconfiguration (Device Specific)

Option	ACE impl_option	Description
Enable Partial Reconfiguration	`bitstream_partial_reconfig`	When checked (1), enables partial reconfiguration.
Partial Reconfig Cluster Map (hex)	`bitstream_partial_reconfig`	20-character hexadecimal value specifying the target fabric cluster. The ACE Clusters view should be used to find the appropriate value.

For more details regarding partial reconfiguration, refer to Partial Reconfiguration (page 108).

FCU Configuration Options

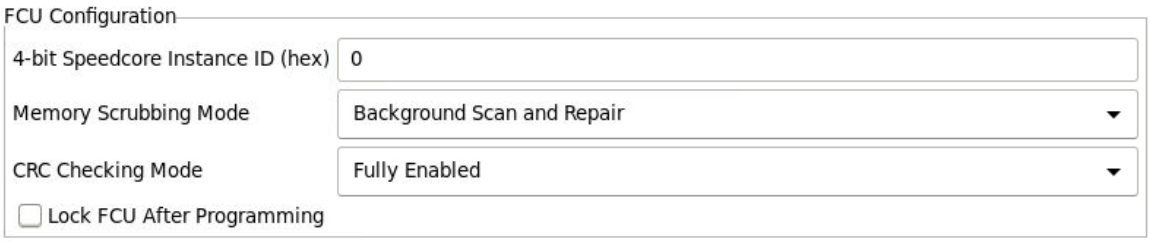


Figure 47 • ACE FCU Configuration Options Dialog

Table 34 • Bitstream Generation Implementation Options — FCU Configuration

Option	ACE impl_option	Description
4-bit Speedcore Instance ID (hex)	`bitstream_instance_id`	Specifies the 4-bit instance ID of the Speedcore device. Must be specified as a one-character hexadecimal value. **Not used for Speedster7t**.
Memory Scrubbing Mode	`bitstream_scrub_mode`	Selects the CMEM scrubbing mode. Allowed values: **Disabled** (0) **Background Scan** (1) **Background Scan and Repair** (2).
CRC Checking Mode	`bitstream_crc_mode`	Selects the CRC checking mode. Allowed values: **Fully Enabled** (0) **Partially Enabled** (1) **Bypassed** (2).
Lock FCU After Programming	`bitstream_fcu_lock`	When checked (1), locks the FCU of the target device after programming. This option is automatically set when bitstream encryption is enabled.

Bitstream ID Configuration Options



Figure 48 • ACE Bitstream ID Configuration Options Dialog

Table 35 • Bitstream Generation Implementation Options — Bitstream ID Configuration

Option	ACE impl_option	Description
Bitstream ID Type	`bitstream_id_type`	Inserts the selected type into an FCU register in the bitstream. If **None** (`0`) is selected, all zeros are inserted. If **Timestamp** (`1`) is selected, the epoch time during bitstream generation is inserted. If **User Defined** (`2`) is selected, the value in `bitstream_id_value` is inserted.
31-bit User Defined Bitstream ID (hex)	`bitstream_id_value`	User-defined 8 hexadecimal character value set when **Bitstream ID Type** is **User Defined**. MSB is set to `1` while the other 31 bits are set by the user.

CMEM Error Injection Options

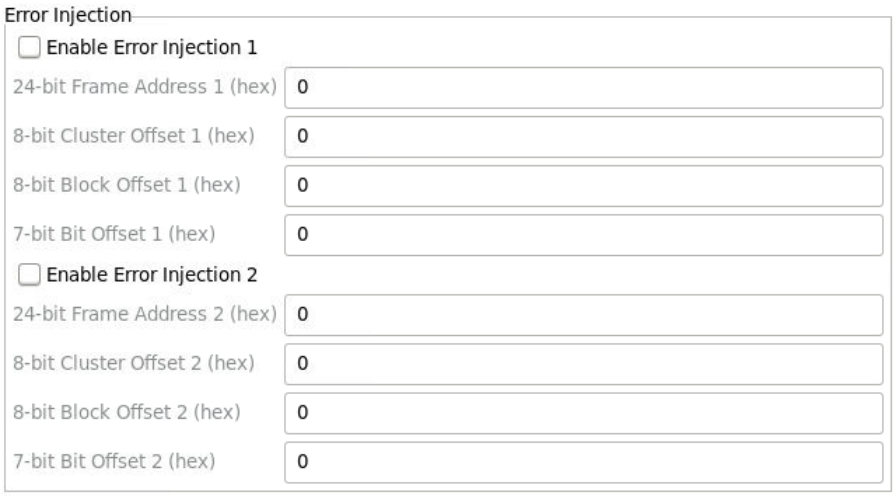


Figure 49 • Error Injection

Table 36 • Bitstream Generation Implementation Options — Error Injection

Option	ACE impl_option	Description
Enable Error Injection 1	`bitstream_error_inject_en1`	When checked (1), enables the first bit error injection for the following address and bit offsets.
24-bit Frame Address 1 (hex)	`bitstream_error_inject_addr1`	Specifies the 24-bit frame address for the first bit error to be injected into the bitstream. (1)
8-bit Cluster Offset 1 (hex)	`bitstream_error_inject_cluster_offset1`	Specifies the cluster offset into the target Frame for the first bit error to be injected into the bitstream. (2)
8-bit Block Offset 1 (hex)	`bitstream_error_inject_block_offset1`	Specifies the block offset into the target cluster for the first bit error to be injected into the bitstream. (2)
7-bit Bit Offset 1 (hex)	`bitstream_error_inject_bit_offset1`	Specifies the 7-bit bit offset into the target block for the first bit error to be injected into the bitstream. (3)
Enable Error Injection 2	`bitstream_error_inject_en2`	When checked (1), enables the second bit error injection for the address and bit offsets that follow.
24-bit Frame Address 2 (hex)	`bitstream_error_inject_addr2`	Specifies the 24-bit frame address for the second bit error to be injected into the bitstream. (1)
8-bit Cluster Offset 2 (hex)	`bitstream_error_inject_cluster_offset2`	Specifies the cluster offset into the target frame for the second bit error to be injected into the bitstream. (2)
8-bit Block Offset 2 (hex)	`bitstream_error_inject_block_offset2`	Specifies the block offset into the target cluster for the second bit error to be injected into the bitstream. (2)
7-bit Bit Offset 2 (hex)	`bitstream_error_inject_bit_offset2`	Specifies the 7-bit bit offset into the target block for the second bit error to be injected into the bitstream. (3)

Note

1. Must be specified as a 6-character hexadecimal value.
2. Must be specified as a 2-character hexadecimal value. The valid range depends on the target device.
3. Must be specified as a 2-character hexadecimal value.

For more information on how to insert up to 2 bit errors, refer to Configuration Error Correction and SEU Mitigation (page 85).

Chapter 6 : Configuration Sequence and Power-Up

The power-up and configuration sequence for Speedster7t FPGAs is as follows:

1. Configure the board to set FCU_CONFIG_MODESEL.
2. Power up the board.
3. After power-up, wait for **sufficient time** for the power supplies to settle to their final DC values.
4. Ensure that JTAG_TRSTN is stable at 0.
5. Ensure that FCU_CONFIG_RSTN is stable at '0'.
6. Ideally JTAG_TCK is pulled down to '0' and is stable. If a pull-up resistor is used on JTAG_TCK, JTAG_TCK has to be stable at '1' and should not have ringing.
7. Start toggling the FCU_CPU_CLK. Ensure that the clock is stable and has good signal quality.

   > **Note**
   >
   > FCU_CONFIG_RSTN de-assertion must be done at least 1 ms after all power supplies have settled to their required DC voltage values. FCU_CONFIG_RSTN signal is used to initialize all IP in the FPGA. This 1 ms wait time ensures that all IP are supplied with the correct voltage values and see the FCU_CONFIG_RSTN de-assertion.

8. If the above conditions are satisfied, release FCU_CONFIG_RSTN (transition from '0' to '1').
9. Wait for FCU_CONFIG_STATUS.
10. Program the bitstream.
11. Check FCU_CONFIG_USER_MODE and FCU_CONFIG_DONE. If not in user mode, check FCU_CONFIG_ERR_ENC.

Device Power-Up

The first step in bringing up a Speedster7t FPGA is to appropriately power it up. The ***Speedster7t Power User Guide (UG087)***[16] details how the power supplies and configuration-related pins and signals must be asserted to ensure a successful power-up. To summarize these requirements:

1. Drive FCU_CONFIG_RSTN low.
2. Power-up all supplies to full rail while keeping FCU_CONFIG_RSTN low to ensure that the Speedster7t FPGA powers up in a reset state. The FCU clock need not be running at this time.
3. If the FCU_CONFIG_MODESEL pins are not statically set (tied off to ground/V_{DD} using the resistor loading options), drive them to set the desired configuration mode using the external interface.
4. Drive FCU_CONFIG_RSTN high to release the reset. Start providing clocks on the FCU clock.
5. Ensure that all clocks used by the Speedster7t FPGA are stable when reset is released.

16 https://www.achronix.com/documentation/speedster7t-power-user-guide-ug087

Note

FCU_CPU_CLK is limited to 250 MHz in all configuration modes.

There are no signaling or sequencing requirements for powering down. The supplies can simply be turned off.

Chapter 7 : Configuration Error Correction and SEU Mitigation

As with all SRAM devices, a single-event upset (SEU) is a potential issue within the Speedster7t FPGA. To assist in mitigating this effect, the FCU can be instructed to scrub the configuration memory (CMEM) of the FPGA fabric during user mode.

The following ACE implementation options pertain to scrubbing.

Table 37 • Bitstream Generation Implementation Options – Scrubbing

Option	ACE impl_option	Description
Memory Scrubbing Mode	`bitstream_scrub_mode`	Selects the CMEM scrubbing mode. Allowed values include: • Disabled (`0`) • Background Scan (`1`) • Background Scan and Repair (`2`)

Scrubbing first reads the logic cluster data and parity bits from configuration memory cells. If an SEU occurs, the error is detected (scrub_mode=1) or corrected (scrub_mode=2) and scrubbing continues to the next frame. During a multiple bit error, no correction is possible, and the scrubbing state machine goes into ERROR state, as the integrity of the CMEM has been compromised. The configuration block also has dedicated registers that hold the scrub address, block offset, bit offset, and the number of single and multiple error counts. The number of single/multiple error count is also encoded and routed to an I/O pad. refer to **Configuration Pin Tables** (page 70) for details.

If scrubbing is enabled, FCU clocks *must* be supplied even after entering user mode as the scrubbing state machine must be clocked continuously. FCU clocks need not be provided after entering user mode if scrubbing is disabled. In this case, the state machine does not run at all as there is no value in running if errors cannot be reported.

The scrubbing state machine starts when the device enters user mode as indicated by the `FCU_CONFIG_USER_MODE` output pin. Scrubbing cannot occur before user mode is entered.

Scrubbing can be driven by `cfg_clk` or `sys_clock` based on the setting highlighted in the following table.

Table 38 • Scrub Clock Settings

cfg_ipcsr_clk_sw (1)	scrub_clk
0	cfg_clk (2)
1	sys_clk (3)

cfg_ipcsr_clk_sw [1]	scrub_clk

Table Notes

1. Set high by setting bit 8 of the FCU register `0x00EC`. See table Scrub Register Definitions (page 92) for details about this register
2. This signal is defined by the configuration clock selected based on the pins `FCU_CONFIG_SYSCLK_BYPASS` and `FCU_CONFIG_CLKSEL` as shown in Configuration Pin Tables (page 70).
3. This signal is defined as the clock driven by the on-chip oscillator.

Configuration Memory Architecture and Addressing

The Speedster7t FPGA configuration memory (CMEM) is organized as shown in the following table:

Table 39 • Configuration Memory Components

Component	Description
Cluster	Clusters come in two forms, sequentially numbered from south to north within a column, starting at 0: • **Logic clusters** – the arrays of RLB, BRAM, MLP, NoC tiles, and branch tiles that form the building blocks of the programmable logic fabric. They are connected by abutment to build the fabric of the desired size. • **Delimiter custers** – do not contain any user logic and are used for clock routing. These clusters do contain configuration bits set by ACE software based on the user design. Since they contain configuration bits, they have ECC logic and, consequently, they are scrubbed.
Frame	A vertical column of configuration memory cells spanning the entire array of clusters in the Y direction. Bitstream programming and scrubbing both operate on a frame basis with the 24-bit frame address indicating which specific column is being targeted. There are many configuration memory frames for each IP tile column (e.g., RLB, BRAM, LRAM, MLP and NoC). Frames are numbered west to east across the core, starting at 0.
Block	Every frame within a cluster is composed of blocks. Each block consists of 128 bits of CMEM cells. Block addressing is handled via offsets relative to where the block is located in a cluster. Each cluster block offset starts at 0, numbered south to north. Similarly, bit offsets are relative to the block, starting with bit 0 at the south end of the block. It is this addressing scheme that is used for the reporting of bit error locations.

The error information and state machine behavior across the different scrubbing operating modes are as follows:

Table 40 • Error Reporting Based On Scrubbing Mode

Mode	Operation
Disabled	Nothing happens on any kind of error (nothing is reported). Registers storing error counts are not modified.
Background scan	Regardless of error type, error bits and address/offset bits are set accordingly. The scrubbing state machine halts operation after the first error detection. Refer to **Scrubbing Reset** (page 92) for details. Registers storing total error counts independently for single and multiple errors are incremented.
Background scan and repair	In the case of a single-bit error, the error is corrected and reported via the single-error signal pin. The scrubbing state machine continues to run. However, the address and offset bits for subsequent errors are not recorded. Only information for the first error is preserved. The mechanism described in **Scrubbing Reset** (page 92) is necessary to reset the error and address/offset bits. If the state machine sees multiple errors, the error and address/offset bits are set accordingly, and the scrubbing state machine halts operation after detection. Refer to **Scrubbing Reset** (page 92) for details. Registers storing total error counts independently for single and multiple errors are incremented.

Error Injection and Reporting

ACE Implementation Options

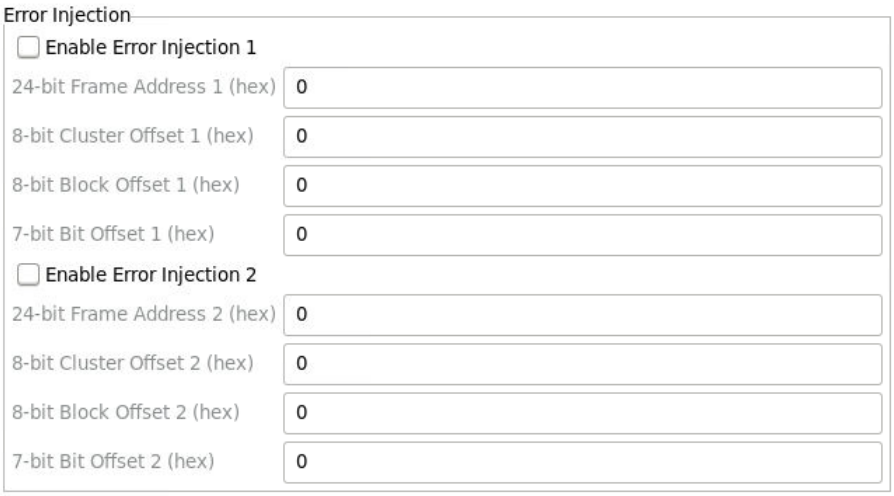


Figure 50 • Error Injection

Table 41 • Bitstream Generation Implementation Options — Error Injection

Option	ACE impl_option	Description
Enable Error Injection 1	bitstream_error_inject_en1	When checked (1), enables the first bit error injection for the following address and bit offsets.
24-bit Frame Address 1 (hex)	bitstream_error_inject_addr1	Specifies the 24-bit frame address for the first bit error to be injected into the bitstream. (1)
8-bit Cluster Offset 1 (hex)	bitstream_error_inject_cluster_offset1	Specifies the cluster offset into the target Frame for the first bit error to be injected into the bitstream. (2)
8-bit Block Offset 1 (hex)	bitstream_error_inject_block_offset1	Specifies the block offset into the target cluster for the first bit error to be injected into the bitstream. (2)
7-bit Bit Offset 1 (hex)	bitstream_error_inject_bit_offset1	Specifies the 7-bit bit offset into the target block for the first bit error to be injected into the bitstream. (3)
Enable Error Injection 2	bitstream_error_inject_en2	When checked (1), enables the second bit error injection for the address and bit offsets that follow.
24-bit Frame Address 2 (hex)	bitstream_error_inject_addr2	Specifies the 24-bit frame address for the second bit error to be injected into the bitstream. (1)
8-bit Cluster Offset 2 (hex)	bitstream_error_inject_cluster_offset2	Specifies the cluster offset into the target frame for the second bit error to be injected into the bitstream. (2)
8-bit Block Offset 2 (hex)	bitstream_error_inject_block_offset2	Specifies the block offset into the target cluster for the second bit error to be injected into the bitstream. (2)
7-bit Bit Offset 2 (hex)	bitstream_error_inject_bit_offset2	Specifies the 7-bit bit offset into the target block for the second bit error to be injected into the bitstream. (3)

Note

1. Must be specified as a 6-character hexadecimal value.
2. Must be specified as a 2-character hexadecimal value. The valid range depends on the target device.
3. Must be specified as a 2-character hexadecimal value.

Bitstream Error Injection

ACE provides a set of previously referenced project implementation options to automatically insert up to dual-bit errors into the generated bitstream output file. This feature is helpful when simulating the Speedster7t FPGA CMEM scrubbing interface. The error injection implementation options allow bit errors to be inserted, but generate bitstream CRC values that still allow the bitstream programming to enter user mode. The CRC value does not account for the inserted errors, so it is not necessary to disable CRCs to inject errors using this feature. If manually editing the generated bitstream file to inject ECC scrubbing errors, the bitstream CRC mode option must be set to bypass the CRCs.

To successfully inject an error into the ACE-generated bitstream output file, first understand which frame addresses are being programmed and know the valid cluster, block, and bit offset ranges for the target device. The cluster offset only affects the Y dimension (i.e., it is only a function of the number of rows of clusters). For example, with a

Speedster7t FPGA with 8 rows, the valid range for `bitstream_error_inject_cluster_offset` options is 0 to 16. Even numbers inject errors into delimiter clusters while odd numbers inject errors into rows:

- Each logic cluster has eight 128-bit blocks of addressable space. Thus, for logic clusters, the valid range for `bitstream_error_inject_block_offset` options is 0 to 7 (hex).
- Each delimiter cluster has one 128-bit block of addressable space. Thus, for delimiter clusters, the valid range for `bitstream_error_inject_block_offset` options is only 0 (hex).

The valid range of `bitstream_error_inject_bit_offset` is always 0 to 0x7F.

Bitstream Single-Bit Error Injection Example

In this example, a single-bit error is injected into a valid frame address at a valid location inside cluster 9 (a logic cluster) at block offset 1 and bit offset 6. To inject this single-bit error:

1. Start the ACE GUI and load the ACE project.
2. In the Options view, under the Bitstream Generation section, check the box to enable the **CMEM Address and Data Export (.address)** output file. Ensure the check-boxes, **Enable Error Injection 1** and **Enable Error Injection 2,** are *not* checked. Before enabling error injection, a valid frame address must first be found. Configure any other bitstream options to meet the test requirements.

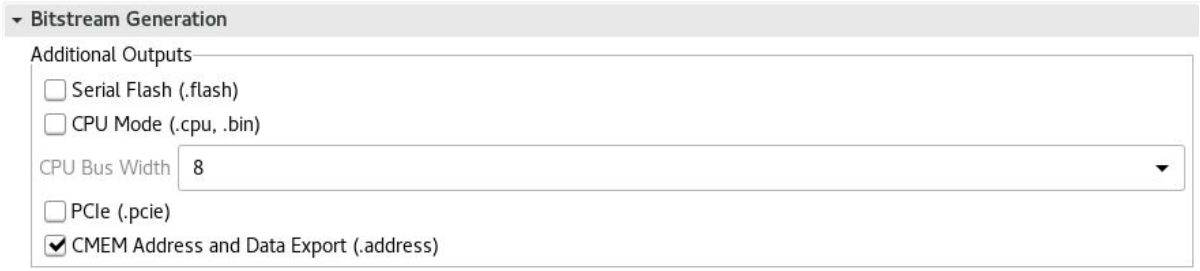


Figure 51 • CMEM Address and Data Export

3. In the Flow view, double-click the **Generate Bitstream** flow step to generate the initial bitstream output files.
4. With a text editor, open the `.address` output file located in `<ace_project_dir>/<impl_dir>/output/<design>.address`.
5. Locate a valid 24-bit frame address by finding any line that begins with "CMEM Address:". The leftmost six characters (upper 24-bits) of the 32-bit hexadecimal value on that line represents a valid 24-bit frame address to use with the ACE bitstream error injection feature. This example assumes the following frame address from inside the `.address` output file: `50801100`. The resulting 24-bit frame address is `508011`.
6. In the Options view, under the Bitstream Generation section, set the Memory Scrubbing Mode option to either **Background Scan** or **Background Scan and Repair** to enable detection of the inserted ECC bit error, and check the option **Enable Error Injection 1**.
7. Enter the following values:
   - `508011` for the 24-bit Frame Address 1 (hex) field.
   - `9` for the 8-bit Cluster Offset 1 (hex) field
   - `1` for the 8-bit Block Offset 1 (hex) field
   - `6` for the 7-bit Bit Offset 1 (hex) field.

8. (Optional) to save copies of the original bitstream output file prior to injecting an error, make copies of the files now. The next step overwrites the current bitstream output files on the file system.
9. In the Flow view, double-click the **Generate Bitstream** flow step to generate the new error-injected bitstream output files.
10. When simulating programming the Speedster7t FPGA instance with this error-injected bitstream, some time after entering, the Speedster7t FPGA pin interface indicates a single-bit scrubbing error in address `508011` in block offset 1 and bit offset 6. Ensure to set the cluster select pins to a value of `4'h9` to select cluster 9 and tie the scrubbing enable pin high to enable scrubbing.

Bitstream Dual-Bit Error Injection Example

In this example, a dual-bit error is inserted into a valid frame address at a valid location inside cluster 9 (a logic cluster) with the first bit error at block offset 1 and bit offset 6, and the second bit error at block offset 5 and bit offset `1A`. To insert this dual-bit error:

1. Repeat Steps 1 to 7 from the Bitstream Single-Bit Error Injection Example (page 90).
2. In the Options view, under the Bitstream Generation section, check the option **Enable Error Injection 2** to enable the second error bit to be injected.
3. Enter the following values:
   - `508011` for the 24-bit Frame Address 2 (hex) option.
   - `9` for the 8-bit Cluster Offset 1 (hex) option.
   - `5` for the 8-bit Block Offset 1 (hex) option.
   - `1A` for the 7-bit Bit Offset 1 (hex) option.
4. (Optional) to save copies of the original bitstream output file prior to injecting an error, make copies of the files now. The next step overwrites the current bitstream output files on the file system.
5. In the Flow view, double-click to **Generate Bitstream** flow step to generate the new error-injected bitstream output files.
6. When simulating programming a Speedster7t FPGA instance with this error-injected bitstream, some time after entering user mode, the Speedster7t FPGA pin interface indicates a dual-bit scrubbing error in address `508011` (block and bit offset outputs should be ignored in this case). Ensure to set the cluster select pins to a value of `4'h9` to select cluster 9 and tie the scrubbing enable pin high to enable scrubbing.

> **Note**
>
> Configuration memory scrubbing operates independently on each cluster within a given frame. To cause a dual-bit error, both errors must be injected into the same cluster offset. If two errors are inserted into the same frame but at different cluster offsets, they are treated as two independent single-bit errors, and both may be corrected if **Background Scan and Repair** is selected for Memory Scrubbing Mode.
>
> If dual-bit errors are injected into the same cluster offset in the same frame, the dual-bit error can only be detected and not repaired. The Speedster7t FPGA memory scrubbing interface outputs indicate the frame address and cluster offset of the dual-bit error, but do not report the block offset and bit offset. The block and bit offset outputs should be ignored in this case.

Scrubbing Reset

The registers feeding the error address/offset values, as well as the FCU error counter registers, can be cleared simultaneously in one of three ways:

1. A complete power-cycle of the Speedster7t FPGA instance, involving powering the core down and powering it back up.
2. A re-initialization sequence which can be accomplished by toggling `FCU_CONFIG_RSTN`.
3. Loading a new ACE-generated bitstream image.

After reset and reconfiguration, scrubbing can begin again (if enabled).

Scrubbing FCU Registers

The following table lists the FCU registers pertaining to scrubbing. For more information on accessing the FCU registers, consult the "Speedster7t Tcl Commands" section in **JTAG Programming using the Tcl Library API** (page 32).

Table 42 • Scrub Register Definitions

Scrub Register	Address	Description
CONFIG_REG_ADDR_SCRUB_CONTROL	16'h00EC	• 1:0 – scrub enable. • 2 – scrub mode. • 16:12 – scrub cluster select.
CONFIG_REG_ADDR_SCRUB_SINGLE_ERROR_STATUS	16'h11f8	• Bit 2:0 – internal scrubbing state. • Bit 31:3 – indicates scrub single error status of all delimiter and logic clusters. The lowest-order bit physically represents the southern most cluster. Any MSBs greater than the number of clusters in a given fabric are tied to 0.
CONFIG_REG_ADDR_SCRUB_MULTIPLE_ERROR_STATUS	16'h11fc	Indicates scrub multiple error status of all delimiter and logic clusters. The lowest-order bit physically represents the southern most cluster. Any MSBs greater than the number of clusters in a given fabric are tied to 0.

Chapter 8 : Design Security for Speedster7t FPGA

Achronix recognizes the importance of protecting the sensitive IP placed onto the FPGA. To provide a high level of protection, Speedster7t FPGAs have a number of features to support bitstream encryption as well as authentication. These features ensure that the design configuration on the FPGA cannot be accessed and also ensures that the design is the one intended. Speedster7t FPGAs provide this high level of security through the following features:

- Support for ECDSA authenticated and AES-GCM encrypted bitstream
- Dynamic power analysis (DPA) protection to prevent side-channel attacks
- Physically unclonable function (PUF) for tamper-proof protection
- Securely stores both public and encrypted private keys

With this security solution deployed, customer designs are secure. Even with possession of the device, the underlying design cannot be extracted, cannot be reverse engineered, nor can the design be altered in any way.

Bitstream Authentication

Authentication of a bitstream ensures that the FPGA is configured with the intended design. Achronix provides a two-step authentication process that first authenticates an encrypted bitstream before decrypting it, and then performs authentication a second time on the decrypted bitstream before configuring the device:

1. A bitstream is encrypted using AES-GCM, which provides authenticated encryption.
2. The user provides an asymmetric private key to sign the encrypted bitstream using ECDSA.
3. When the FPGA is configured with an encrypted and signed bitstream, it uses the public key stored in an internal electronic fuse (eFuse) to authenticate the bitstream using the public key.
4. When authenticated, the bitstream decryption is enabled, and the bitstream is authenticated a second time while decrypting with AES-GCM.
5. After the second authentication, the bitstream is used to configure the FPGA.

> **Note**
>
> AC7t1400 has a predefined authentication key that is built into ACE and stored into the eFuse during manufacturing. Any attempt to change the device public authentication key renders the device inoperable.

Bitstream Encryption

Bitstreams consist of the sensitive intellectual property of the designer. Achronix provides tools to generate bitstreams that are encrypted and signed using very strong encryption with hardware designed to be resilient to side-channel attacks, such as dynamic power analysis (DPA). Additionally, the key derivation function (KDF) inside the secure boot portion of the FPGA, along with the physically unclonable function (PUF) ensure protection of the secret keys to decode and authenticate the bitstreams. Together these systems provide a solution that is safe from attacks such that even with possession of the device, an adversary cannot extract the underlying design, cannot change the system to perform another task other than the intended task, and cannot reverse engineer the core intellectual property.

The following figure shows an overview of the security system and how elements work together to protect the bitstream. Blocks shown in grey represent encryption/decryption elements. Blocks shown in blue are authentication elements and green blocks handle authenticated and encrypted bitstreams.

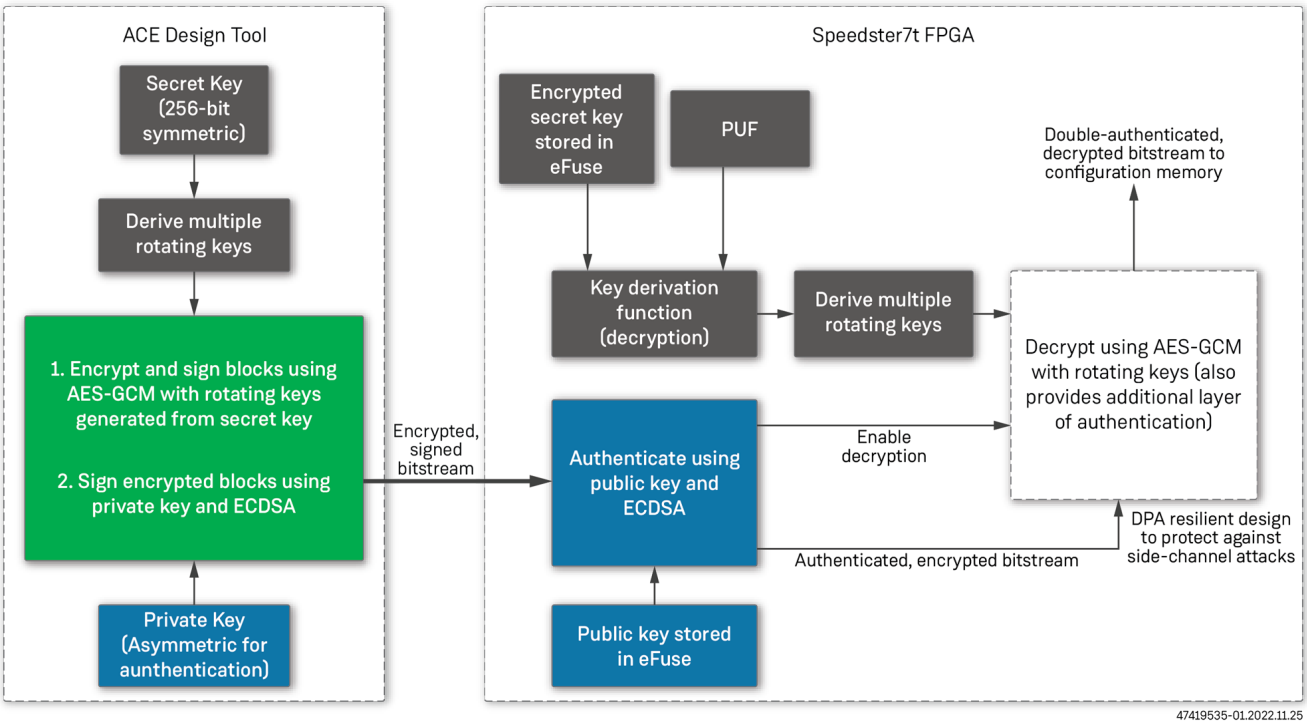


Figure 52 • Bitstream Encryption/Authentication Block Diagram

Generating Encrypted Bitstreams

To generate an encrypted bitstream, a 256-bit secret key is provided to ACE. In order to provide better protection against side-channel attacks, ACE does not simply use this secret key to encrypt the entire bitstream. Instead, the secret key is used as an initial key. ACE then generates new derived keys based on the initial secret key to encrypt smaller segments of the bitstream, each with a different derived key and a new nonce. Here the nonce, also known as an initialization vector (IV), is a random number only used once per segment such that the same pattern is not generated while replaying or encrypting the same bitstream. Bitstream encryption is performed using the highly secure 256-bit AES-GCM encryption standard. Galois/counter mode (GCM) is an advanced form of symmetric-key block encryption which enhances the 256-bit advanced encryption standard (AES) by using a nonce (one-time use random value) and a counter mode so that each segment of data is uniquely encrypted. ACE also uses a Galois message authentication code (GMAC) to simultaneously sign and authenticate the data, including the unencrypted preamble section of the bitstream to guarantee the bitstream has not been altered. To further protect the bitstream, ACE also signs each segment of the encrypted bitstream using ECDSA. See the section on **Bitstream Authentication** (page 94) for more details on the ECDSA authentication.

Caution!

The AC7t1400 comes pre-programmed from the factory with one 256-bit secret key used to encrypt the entire bitstream within ACE, which has the same secret key. No attempt should be made to program this AES private key. Three additional programmable AES private key slots are provided.

Encrypting a Speedster7t AC7t1500 Bitstream

Using the ACE GUI

1. Go to **Bitstream Generation** in the ACE options panel.

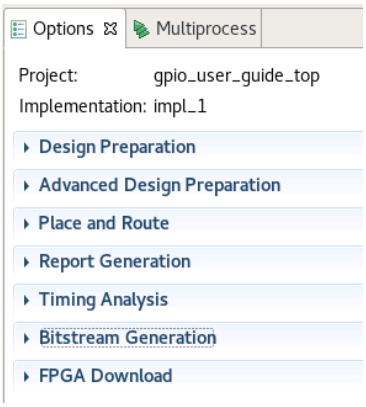


Figure 53 • ACE Options Panel

2. Configure the **AES Encryption** options.

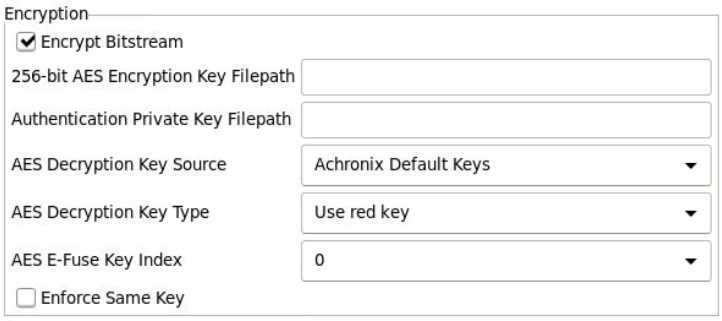


Figure 54 • AC7t1500 AES Encryption Configuration Options

   a. Check the **Encrypt Bitstream** option.
   b. If **Achronix Default Keys** is not selected for the **AES Decryption Key Source**, enter a file name and path in the **256-bit AES Encryption Key Filepath** box to encrypt the bitstream data.

This file must be:

- An absolute or relative path to the current ACE project
- A `.txt` file type
- An AES hexadecimal value within the `.txt` file:
    1. For ACE installations 9.0 or later, any 256-bit or 64-character value.
    2. For ACE installations before 9.0, any 256-bit or 64-character value with a new line character at the end of the AES sequence. This would make the total file 260 bits or 65 characters.

c. If **Achronix Default Keys** is not selected as the **AES Decryption Key Source**, enter a file name and path in the **Authentication Private Key Filepath** box to encrypt the bitstream data.

This file must be:

- An absolute or relative path to the current ACE project
- A `.pem` file type that was created in the Generating a Public and Private Key Pair (page 102) section

d. For **AES Decryption Key Source**, select whether to use **E-Fuse Keys** (`0`) or the **Achronix Default Keys** (`1`) for decryption.

e. For **AES Decryption Key Type**, select whether to **Use PUF black key** (`0`) to create the red key from the black key or **Use red key** (`1`) to treat the red key as the source during encryption.

> **Note**
>
> When using **Achronix Default Keys** as the **AES Decryption Key Source**, the red keys must be treated as the source, and PUF is not allowed.

f. For **AES E-Fuse Key Index**, select which key to use. Bitstreams have the ability to be encrypted with 4 different AES keys. This is the AES key index whose key value should be assigned to the data in the **256-bit AES Encryption Key Filepath**. This also requires **AES Decryption Key Source** to be set to **E-Fuse Keys** (`0`). In order to decrypt the target FPGA bitstream, the AES key must be written to the corresponding key index in the FCU. The **AES E-Fuse Key Index** value can be **0**, **1**, **2**, or **3**. Additional details on using the key index are in the Programming the Encryption Keys (page 103) section.

g. Check the **Enforce Same Key** option if programming multiple encrypted bitstreams. This option specifies whether subsequent bitstreams can be programmed with the same encryption types and keys without resetting the FPGA. If checked, subsequent bitstreams must use the same key source, key type, and key index unless the FPGA has been reset.

Using Tcl Commands

If preferred, these options may be specified by entering the following commands into the ACE project file, or directly into the ACE Tcl console:

```
set_impl_option -project <ace project name> -impl impl_1 bitstream_encrypted "1"
set_impl_option -project <ace project name> -impl impl_1
bitstream_encryption_aes_key_file "key_files/aes.txt"
set_impl_option -project <ace project name> -impl impl_1
bitstream_encryption_pem_key_file "key_files/my_eckey.privkey.pem"
```

```
set_impl_option -project <ace project name> -impl impl_1 bitstream_encryption_key_source
"0"
set_impl_option -project <ace project name> -impl impl_1 bitstream_encryption_key_type
"1"
set_impl_option -project <ace project name> -impl impl_1 bitstream_encryption_key_index
"0"
set_impl_option -project <ace project name> -impl impl_1 bitstream_encryption_same_key
"0"
```

Encrypting a Speed AC7t1400 Bitstream

Using ACE GUI

1. Go to **Bitstream Generation** in the ACE options panel.

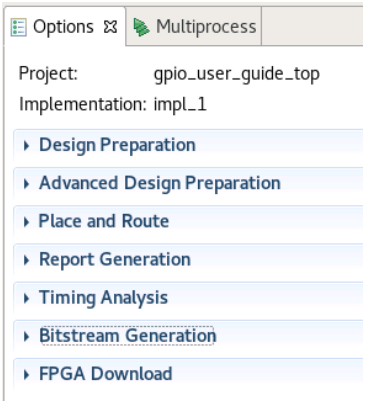


Figure 55 • ACE Options Panel

2. Configure the **AES Encryption** options:

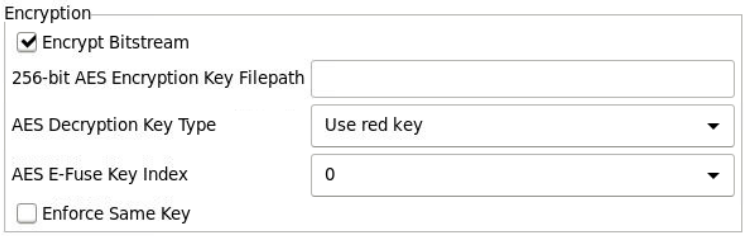


Figure 56 • AC7t1400 AES Encryption Configuration Options

1. Check the **Encrypt Bitstream** option.
2. Enter a file name and path in the **256-bit AES Encryption Key Filepath** box to encrypt the bitstream data. The file must be:

- An absolute or relative path to the current ACE project
- A `.txt` file type
- An AES hexadecimal value within the `.txt` file:
  i. For ACE installations 9.0 or later, any 256-bit or 64-character value
  ii. For ACE installations before 9.0, any 256-bit or 64-character value with a newline character at the end of the AES sequence. This would make the total file 260 bits or 65 characters.

3. For **AES Decryption Key Type**, select whether to **Use PUF black key** (`0`) to create the red key from the black key or **Use red key** (`1`) to treat the red key as the source during encryption.
4. For **AES E-Fuse Key Index**, select which key to use. Bitstreams have the ability to be encrypted with 4 different AES keys. This is the AES key index whose key value should be assigned to the data in the **256-bit AES Encryption Key Filepath**. In order to decrypt the target FPGA bitstream, the AES key must be written to the corresponding key index in the FCU. The **AES E-Fuse Key Index** value can be **0**, **1**, **2**, or **3**. Additional details on using the key index are in the Programming the Encryption Keys (page 103) section.

   Note

   The key value at E-Fuse Key Index "0" has already been populated by Achronix and cannot be altered. If the user wishes to utilize the predefined key value offered by Achronix, choose key index "0".

5. Check the **Enforce Same Key** option if programming multiple encrypted bitstreams. This option specifies whether subsequent bitstreams can be programmed with the same encryption types and keys without resetting the FPGA. If checked, subsequent bitstreams must use the same key index unless the FPGA has been reset.

Using Tcl Commands

If preferred, these options may be specified by entering the following commands into the ACE project file, or directly into the ACE Tcl console:

```
set_impl_option -project <ace project name> -impl impl_1 bitstream_encrypted "1"
set_impl_option -project <ace project name> -impl impl_1
bitstream_encryption_aes_key_file "key_files/aes.txt"
set_impl_option -project <ace project name> -impl impl_1 bitstream_encryption_key_type
"1"
set_impl_option -project <ace project name> -impl impl_1 bitstream_encryption_key_index
"0"
set_impl_option -project <ace project name> -impl impl_1 bitstream_encryption_same_key
"0"
```

Hardware Security

There are several security features available in the hardware to support decryption of encrypted bitstreams, safe storage of secret keys, and strict rule enforcement which locks the device if security rules are violated. The main features for decryption and safe storage of keys use the physically unclonable function (PUF) which provides a unique secret value per individual FPGA, and the key derivation function (KDF) which uses the PUF as the key to encrypt/decrypt the actual secret keys from the encrypted keys that are stored in an electronic fuse (eFuse).

Physically Unclonable Function

The PUF generates a unique secret identifier for each individual FPGA. It is created from random physical variations that occur during the semiconductor manufacturing process, such that the same logic on an FPGA creates completely different and unique values on each individual FPGA, even those on the same wafer. The value of the PUF is random per individual FPGA, but remains constant over the lifetime of that device. The PUF value is not known to Achronix or the manufacturer, and the value cannot be observed without destroying or altering the value of the PUF. This PUF value can be used to encrypt the user secret key and store an encrypted version of the secret key in an eFuse. Then when an encrypted bitstream is loaded into the FPGA, the PUF value is used to temporarily decrypt the stored encrypted secret key. This secret key is then used to generate the multiple rotating keys to decrypt the bitstream blocks that configure the FPGA.

Key Derivation Function

The KDF uses 256-bit AES encryption in conjunction with the PUF to create an encrypted version of the user secret key that can be stored in an eFuse. While it is theoretically possible to observe the contents of the eFuse if an adversary is in possession of the device and has access to advanced reverse engineering equipment, the stored key is an encrypted version of the secret key that uses the PUF value as the master key for encryption. Again, the PUF value cannot be known and is unique to each individual device, thus making the stored key safe. Additionally, when the KDF needs to decrypt an encrypted bitstream, it loads the encrypted key from the eFuse along with the PUF value and temporarily decrypts the secret key. The secret key is then used as the initial key for the module that generates the multiple derived keys for AES-GCM decryption of the bitstream prior to loading it into the configuration memory in the FPGA.

The following two figures show how the PUF and KDF are used to generate a secure encrypted key to store in an eFuse, and how they are used to recreate the secret key to decrypt the bitstream.

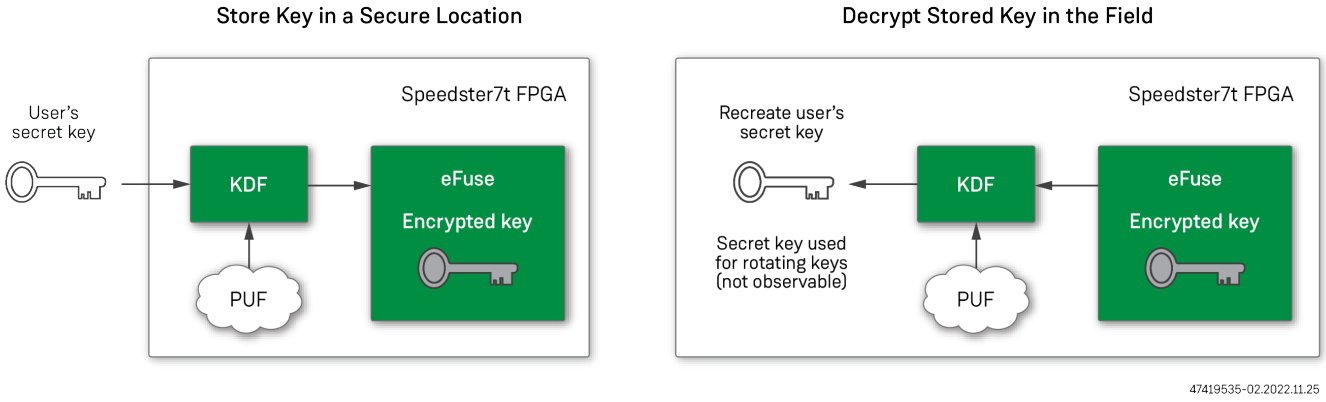


Figure 57 • Safe Secret Key Storage

Rules for Encryption

When using encrypted bitstreams, the FPGA enforces a set of rules. If the security rules are violated, the FPGA locks up and cannot be used in any way without powering down the device. First, there is an ordering rule determining how bitstreams are to be loaded. Speedster7t FPGA bitstreams have three phases and must follow these ordering rules:

1. Zero, one, or multiple pre-configuration (stage0) bitstreams.

2. One, and only one, full configuration bitstream.
3. Zero, one, or multiple partial reconfiguration bitstreams.

Additionally, there are rules to determine which keys can be used for the encryption. The eFuses can store up to four secret keys — bitstreams can be encrypted using up to four different initial keys. These rules must be followed to prevent locking the device:

AC7t1400 allows three of the four AES keys to be programmable by customers. The first AES key is predefined by Achronix and cannot be modified.

1. If the `encrypted_bitstreams_only` eFuse bit has been set for the FPGA, the device only accepts encrypted bitstreams.
2. If any pre-configuration bitstream is encrypted, all pre-configuration bitstreams must be encrypted using the same key.
3. If either the pre-configuration bitstream or the full bitstream are encrypted, they both must be encrypted and both must use the same key.
4. Any partial reconfiguration bitstreams may use a different key if and only if the previous bitstream sets the `same_key` bit to 0 in the preamble, and the partial reconfiguration bitstream also sets that same bit to 0 in its preamble.

> **Note**
>
> It is acceptable to load an unencrypted bitstream after a previous encrypted bitstream. It is *not* acceptable to load an encrypted bitstream after a previous unencrypted bitstream.

Security Fuses

There are several eFuses that are related to the security features in Speedster7t FPGAs. Some of these are set during manufacturing and cannot be changed by the customer, and others are available for customer use. See the eFuse chapter for details.

Fuses Set at Manufacturing

There are two fuses that can be set at manufacturing time to limit the features of the FPGA (The part number of the device indicates if these limitations exist in a part):

- **Bitstream decrypt disable** – if set, the FPGA cannot accept encrypted bitstreams
- **DPA disable for bitstream decrypt** – if set, the FPGA still supports encrypted bitstreams, but there is limited hardware protection for differential power analysis (DPA) side-channel attacks that can potentially expose secret keys

This product only supports encrypted Bitstreams. **Bitstream decrypt disable** is not applicable to the Speedster7t AC7t1400.

Fuses Set By Customer

There are several eFuses that can be set by the customer if using encrypted bitstreams:

- **Bitstream authentication key** – this fuse contains a 768-bit hash of the public key used for first-level authentication of encrypted bitstreams. This fuse is not readable.
- **Bitstream decryption key** – these fuses contain the four 256-bit secret keys that can be used for decryption and authentication of encrypted bitstreams. These fuses can contain the actual secret keys or the encrypted version of the secret keys (using PUF and KDF). These fuses are not readable.
- **Bitstream user register** – this fuse contains the 32-bit value set by the user to identify the key version used. The secret key itself cannot be read back, but the user register value can be read. The user keeps a mapping of key versions to keys.
- **Bitstream user lock** – this one-bit fuse, if set, disables further updates to the authentication key, decryption key, and user register.
- **Encrypted bitstreams only** – this one-bit fuse, if set, forces the FPGA to only accept encrypted bitstreams that use one of the keys stored in the fuses.

In the Speedster7t AC7t1400, the **Bitstream authentication key** is predefined and should never be set by the customer. Only three of the **Bitstream decryption keys** are available for customer use, the first one is reserved for Speedster7t AC7t1400 use.

The **Encrypted bitstreams only** fuse is set at manufacturing for the Speedster7t AC7t1400.

Default Keys

Achronix provides a default public key for authentication and a default secret key for encryption/decryption of the bitstream. These keys are available for testing to provide confidence the security system works. The default keys should not be used to protect sensitive designs — they are only made available for testing purposes. Additionally, when the eFuse is set to accept encrypted bitstreams only, the FPGA no longer accepts the default keys.

The default keys are not available in the Speedster7t AC7t1400. It comes with a pre-programmed authentication key and one secret key. These keys cannot be modified.

Generating a Public and Private Key Pair on Speedster7t AC7t1500

The Athena key generator is a tool delivered to ACE users and can be found within the installation directory. Use the Athena authentication key generator to create unique asymmetric public and private key pairs. The key generator, geneckey, is located at `<ACE_INSTALL_DIR>/system/cmd64/geneckey`. To use geneckey, provide an output base file name. It either creates 3 files in the current directory, or if a path before the base file name is provided, it creates these files in the chosen directory.

```
$ cd <ACE_INSTALL_DIR>
<ACE_INSTALL_DIR>$ system/cmd64/geneckey /path/to/output/file/<base_file_name>
```

For example, if the base file name is "my_eckey", the key generator outputs three files:

- The private key, `my_eckey.privkey.pem`. This is used with the bitstream implementation option `bitstream_encryption_pem_key_file` in ACE. The `.pem` file type is typically used for secure protocols such as with encryption. The following is an example `.pem` file generated with the the Athena authentication key generator:

```
-----BEGIN PRIVATE KEY-----
```

```
< PEM
key
value
here >
-----END PRIVATE KEY-----
```

- A 768-bit hash of the public key, `my_eckey.pubkey.txt`. This is used when writing to the FPGA eFuses before programming.
- The public key, `my_eckey.pubkey.pem`. This is only created by the key generator and the user does not need to write this value during bitstream creation or when writing to the eFuses.

Programming the Encryption Keys Into Speedster7t AC7t1500 eFuses

The following eFuse programming steps are a one-time process per part, as an eFuse can only be programmed once per part. These steps only need to be repeated if it is desired to write a new key value in an eFuse that was not previously programmed with encryption keys.

1. Apply power to the Speedster7t FPGA from a powered-off state or initiate a FCU reset by asserting the `FCU_CONFIG_RSTN` pin. Refer to the ***Speedster7t 7t1500 Pin Table***[17]for the specific ball number.
2. Establish a JTAG connection. For detailed instructions on how to establish a JTAG connection and issue commands, please refer to the ***ACE User Guide*** (UG070)[18].
3. If using the Achronix default keys, there is no need to use the following commands. If the user selects to use their own E-Fuse keys, then the following commands must be issued in the ACE Tcl console with the AES and PEM key values.

   ```
   jtag::write_ecdsa_authentication_key_efuse $jtag_id <Public PEM key>
   jtag::write_aes_encryption_key_efuse $jtag_id <E-Fuse Key Index> <256-bit AES
   Encryption Key>
   ```

   - Public PEM key is the value in the `my_eckey.pubkey.txt` file generated in the first section with geneckey and the Athena key generator
   - The E-Fuse key index and 256-bit AES encryption key values those set in the ACE options while encrypting a bitstream
4. Reset the FPGA by cycling the power.

Programming the Encryption Keys into Speedster7t AC7t1400 eFuses

A future version of ACE is to enable users to program the AES keys into Speedster7t AC7t1400 eFuses.

17 https://achronix.com/documentation/speedster7t-7t1500-pin-table
18 https://www.achronix.com/documentation/ace-user-guide-ug070

Loading Encrypted Bitstreams

Loading an encrypted bitstream is similar to loading an unencrypted bitstream. However, the most important difference is that when the unencrypted 512-bit preamble of the bitstream is loaded, the FPGA disables all data read-out, thus securing the device containing a sensitive user IP and protecting it from being known, reverse engineered, or altered in any way. Encrypted bitstreams are loaded following these steps:

1. When the hardware detects the loading of an encrypted bitstream, all readout and debug features are disabled, preventing the reading of any internal state related to the FPGA fabric or the FCU.
2. Security rules for loading encrypted bitstreams are checked. If the rule checker fails, the FPGA enters a locked state and can only be re-enabled with a power cycle or FCU reset.

   If a board management controller is used to load in the bitstreams, there are additional requirements to be aware of:

   a. After the 512-bit preamble of the bitstream, the board management controller must pause and wait for 300 clock cycles before sending the next portion of the encrypted bitstream.
   b. After the first 12,688 bytes of the encrypted bitstream, the board management controller must pause and wait at least 520,000 FCU clocks, or about 2 ms (assuming a 32-bit data path and 250 MHz FCU clock).
   c. For encrypted bitstreams, the board management controller is limited to sending 32-bits per FCU clock. For unencrypted bitstreams, the controller can send data at a rate up to 128-bits per FCU clock.

   When using encrypted bitstreams, it is *not* possible to use any debug features of the FPGA. Debug features are *only* available when using unencrypted bitstreams.

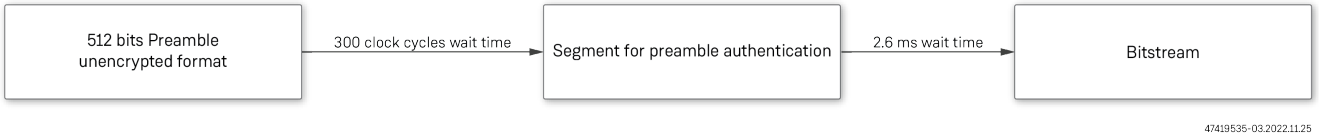


Figure 58 • Encrypted Bitstream Loading Sequence

Programming an AC7t1500 Encrypted Bitstream

After writing the eFuses, the encrypted bitstream programming can proceed following these steps:

1. Establish a JTAG connection. For detailed instructions on establishing a JTAG connection and issuing commands, refer to the ***ACE User Guide*** **(UG070)**[19].
2. Program the encrypted bitstream using the `-encrypted` switch:

   ```
   ac7t1500::program_hex_file <path to bitstream> -encrypted
   ```

3. Verify the encrypted bitstream has been configured:
   a. Check that the `FCU_CONFIG_USER_MODE` ball is asserted high, indicating that the device has transitioned into user mode.

19 https://www.achronix.com/documentation/ace-user-guide-ug070

> **Note**
>
> FCU_CONFIG_USER_MODE only transitions from 0 to 1 when the encrypted bitstream is full and not a stage0 or partial reconfiguration. Refer to the **Speedster7t 7t1500 Pin Table**[20] for the specific ball number.

Additionally, VectorPath card users can verify that the FPGA is configured by reading the configuration status register in the BMC. To read the BMC on the VectorPath card, the BittWare software development kit must be installed.

Refer to the Knowledge Base article,**Where Can I Download the Software Development Kit for a VectorPath Card?**[21] for additional information.

b. After installing the SDK, run the following command in the ACE Tcl console:

```
bw_bmc_configure fpga
```

4. If the device has successfully configured and entered user mode, the console displays the following message:

```
FPGA Configuration: Configured Normal
FPGA Boot Source:   User
```

Figure 59 • Successful Configuration Status From the BMC

Programming an AC7t1400 Encrypted Bitstream

After writing the eFuses, the encrypted bitstream programming can proceed following these steps:

1. Establish a JTAG connection. For detailed instructions on establishing a JTAG connection and issuing commands, refer to the ***ACE User Guide*** **(UG070)**[22].
2. Program the encrypted bitstream:

```
ac7t1400::program_hex_file <path to bitstream>
```

> **Note**
>
> In this command, the `-encrypted` switch is not needed as it is built into the Speedster7t AC7t1400 library.

3. Verify the encrypted bitstream has been configured:
   a. Check that the FCU_CONFIG_USER_MODE ball is asserted high indicating that the device has transitioned into user mode.

20 https://achronix.com/documentation/speedster7t-7t1500-pin-table
21 https://support.achronix.com/hc/en-us/articles/4415140267156
22 https://www.achronix.com/documentation/ace-user-guide-ug070

> **Note**
>
> FCU_CONFIG_USER_MODE only transitions from 0 to 1 when the encrypted bitstream is full and not a stage0 or partial reconfiguration. Refer to the **Speedster7t 7t1500 Pin Table**[23] for the specific ball number.

Additionally, VectorPath card users can verify that the FPGA is configured by reading the configuration status register in the BMC. To read the BMC on the VectorPath card, the BittWare software development kit must be installed. Refer to the Knowledge Base article,**Where Can I Download the Software Development Kit for a VectorPath Card?**[24] for additional information.

b. After installing the SDK run the following command in the ACE Tcl console:

```
bw_bmc_configure fpga
```

4. If the device has successfully configured and entered user mode, the console displays the following message:

```
FPGA Configuration: Configured Normal
FPGA Boot Source:   User
```

Figure 60 • Successful Configuration Status From the BMC

Device DNA

The Speedster7t family of FPGAs contains a set of internal fuses that can be opened during production, resulting in stored read only logical values. One of these fuses stores a 32-bit sequence known as Device_DNA. The device DNA is similar to a serial number in that it is a value that uniquely identifies a specific part. This 32-bit sequence, which can be read from the fabric via IPINs after the FPGA has been programmed, consists of a randomly-generated string of 16 ones and 16 zeroes. Each code is unique.

ACE Placements to Read Device DNA

To connect the fabric inputs to the IPINs and read the device DNA, three elements in the design are required:

- The top-level port declaration
- PDC file to create and place IPINs for those ports
- Soft logic that routes the IPIN outputs to a user-visible register. For example, an instance of the register_control_block macro.

Top-Level Port Declaration

The top-level port can be given any name desired, but for this example it is named `i_dna`.

23 https://achronix.com/documentation/speedster7t-7t1500-pin-table
24 https://support.achronix.com/hc/en-us/articles/4415140267156

```
input  wire [31 : 0] i_dna
```

PDC

The PDC file creates an IPIN instance for each port and place it on a top-level GPIO input pin using the pin's symbolic hardware port name. Those port names may be different on different fabrics.

AC7t1500

```
# ------------------------------------------------------------------------
# AC7t1500 Connection to E-Fuses
# ------------------------------------------------------------------------
for {set i 0} {$i < 32} {incr i} {
    create_boundary_pins "p:i_dna\[$i\]" "i_dna_ipin\[$i\]" -purpose CFG
    set_placement -fixed -batch "p:i_dna\[$i\]" "d:cfg2efuse_scratch_fuse\[$i\]"
}
```

AC7t800

```
# ------------------------------------------------------------------------
# AC7t800 Connection to E-Fuses
# ------------------------------------------------------------------------
for {set i 0} {$i < 32} {incr i} {
    create_boundary_pins "p:i_dna\[$i\]" "i_dna_ipin\[$i\]" -purpose CFG
    set_placement -fixed -batch "p:i_dna\[$i\]" "d:i_efuse2core_scratch_fuse\[$i\]"
}
```

Chapter 9 : Partial Reconfiguration

Partial reconfiguration enables reprogramming a portion of the fabric with a smaller bitstream while leaving the remaining configuration intact. Each region that can be reconfigured independently is called a fabric cluster, or simply, "cluster". The Speedster7t AC7t1500 FPGA has 80 clusters which can be reconfigured independently. Partial reconfiguration can only be initiated after the FPGA has entered user-mode.

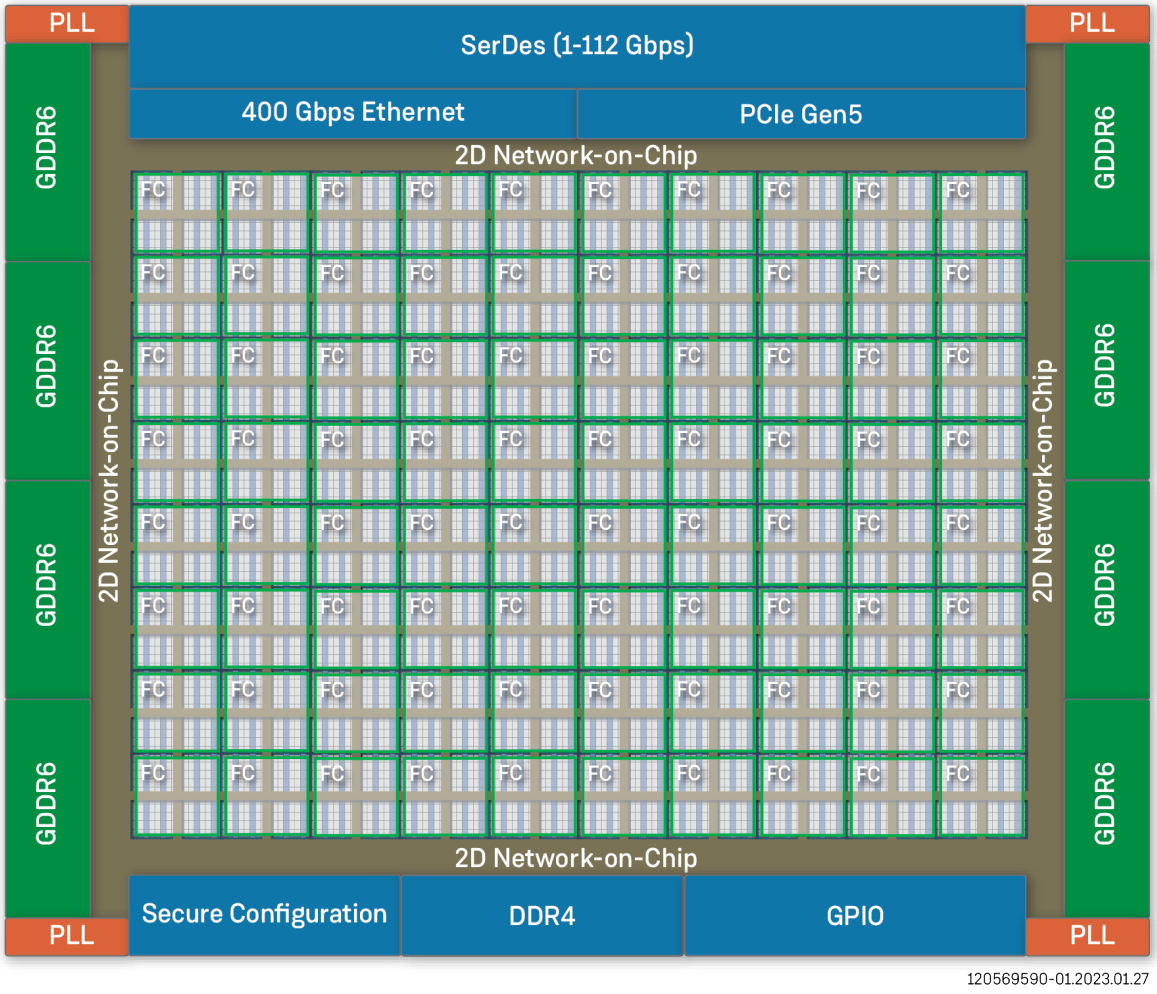


Figure 61 • AC7t1500 FPGA Partial Reconfiguration Fabric Cluster Layout

> **Note**
> The Speedster7t AC7t800 FPGA has 36 clusters.

There are many advantages to partial-reconfiguration:

- Enable dynamic functions for certain blocks in the design
- Dynamic load balancing of accelerator cores
- Smaller FPGA logic functions can be loaded into the FPGA only when needed (hardware overlays)
- Faster programming times

Design Considerations

Partial-reconfiguration introduces additional complexity into the design. Defining correct functional hierarchy is very important for designs that utilize dynamically-reconfigured modules. In order to avoid functional issues while performing partial reconfiguration, system-level logic must be implemented to stop and start communications with the portions of the user design being reconfigured while the remainder of the design is operating.

The partial reconfiguration flow is greatly simplified by leveraging the 2D NoC. All data and communication between a partial reconfiguration module and the remainder of the user design and I/O ring IP (PCIe, Ethernet, GDDR6, DDR4, etc) can be via NAP connections inside the fabric clusters you intend to reconfigure. These connections allow the module to talk to any node in the system without any logic or data signal routing crossing between the partial reconfiguration fabric cluster boundary and the remainder of the design. Utilizing NAP connections allows the partial reconfiguration module to be self-contained and portable — it can be moved to any other set of fabric clusters within the FPGA.

The only signals not self-contained are clocks and (optionally) any global resets or enables. Partial reconfiguration bitstreams do not reconfigure the global clock trunk, the clock branches, or the clock minitrunks. Clocks and global signals must be pre-routed as part of the base bitstream.

All clocks in the system must continue to run during partial reconfiguration since other logic in the fabric also continues to run live and may be using those clocks. Partial reconfiguration bitstreams do configure the connections between the clock stems and the clock branch, and all other logic and routing in the fabric clusters. To connect the local clock stems to the clock branch within the context of a running base design, the partial bitstream must connect to the proper clock branch tracks.

To manage this, ACE provides a set of tools and constraints for pre-routing clocks and global signals. Up-front planning must be performed for clocks and other global signals (e.g., resets and enables) which must be routed into partial reconfiguration regions. See the ***ACE User Guide*** **(UG070)**[25] for details and a full tutorial.

It is also important to define the correct placement constraints so that the target module is completely contained within the cluster marked for partial reconfiguration. The resources for the module cannot exceed the available resources for a cluster and optimizations across the cluster should be disabled.

Currently, there is no support in software for virtual pin placement constraints for the router, which would be needed to support any data nets going into or out of the partially-reconfigured clusters (with the exception of pre-routed clocks and global signals which use the clock network).

25 https://www.achronix.com/documentation/ace-user-guide-ug070

Using Partial Reconfiguration

ACE Implementation Options

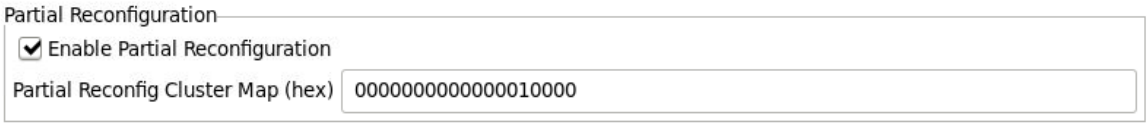


Figure 62 • ACE Partial Reconfiguration Options Dialog

Table 43 • Bitstream Generation Implementation Options — Partial Reconfiguration (Device Specific)

Option	ACE impl_option	Description
Enable Partial Reconfiguration	`bitstream_partial_reconfig`	When checked (1), enables partial reconfiguration.
Partial Reconfig Cluster Map (hex)	`bitstream_partial_reconfig`	20-character hexadecimal value specifying the target fabric cluster. The ACE Clusters view should be used to find the appropriate value.

Partial Reconfiguration Steps

To leverage partial reconfiguration, first program a base bitstream, which at least configures the I/O ring, clocks, and and global signal pre-routes.

A partial bitstream can be generated for a given module in ACE by setting the Partial Reconfiguration Cluster Map and Partial Reconfiguration implementation options shown above prior to generating the bitstream. This generates a partial bitstream which can be programmed on top of (after) the base bitstream. Multiple partial bitstreams can be programmed sequentially. Please see the *ACE User Guide* (UG070)[26] for details.

The partial reconfiguration bitstream generated from ACE includes the following sequence:

1. Write a value of `32'h1000_0000` to `CONFIG_REG_CRC` and a value of `32'h0000_0000` to the `CONFIG_REG_CRC2` register. This brings the partial state machine to shutdown state and asserts the partial reset.
2. Send the SYNC, JTAG ID and the preamble header, then program the selected clusters using the partial bitstream.
3. Write a value of `32'h0200_0000` to `CONFIG_REG_CRC` and a value of `32'h0000_0000` to the `CONFIG_REG_CRC2` register. This releases the reset to the partial clusters and generates `partial_config_done`.

26 https://www.achronix.com/documentation/ace-user-guide-ug070

Chapter 10 : Revision History

Version	Date	Description
1.0	20 Apr 2021	• Initial Achronix release
1.0.1	26 Apr 2021	• Change images Read from FCU Register and Read from ACB Register to execution order
1.0.2	16 Sep 2021	• Minor changes to reflect new FCU features
1.0.3	01 Feb 2022	• Correction to **FLASH configuration header** (page 20) bit[31]
1.0.4	22 Feb 2022	• Removed **original (divide-by-1), divide-by-2** (page 9) support for the flash clock
1.0.5	25 Sep 2023	• Deprecate references to Speedster AC7t1550 device
2.0	02 Apr 2024	• Added information on two-stage programming via PCIe • Added information on dual-mode flash programming • Added information on design security • Added information on device DNA • Removed references to .stapl files • Merged user guide with JTAG Configuration User guide
2.1	26 Mar 2025	• Clean-up of device names for change to AC7t1400 • Updates to **Bitstream Programming via PCIe** (page 65) • Minor edits to wording and corrections
2.2	22 Oct 2025	• Updates to clarify the power on reset sequence and signal stability requirements in **Configuration Sequence and Power-Up.** (page 83) • Requirements for FCU_CPU_CLK updated in **Configuration Pin Tables** (page 70). • **Device DNA** (page 106) updated.
2.3	20 Nov 2025	• Added condition for FCU_CONFIG_RSTN de-assertion after power up
2.4	23 Nov 2026	• Add FCU bidirectional pin warning message to the section, **Configuration Pin Tables** (page 70).