Using the Command-Line Jam STAPL Solution for Device Programming

2017.04.10

AN-425

Subscribe Send Feedback

The Jam™ Standard Test and Programming Language (STAPL) standard is compatible with all Altera devices that supports in-system programming (ISP) using JTAG. You can implement the Jam STAPL solution using the Jam STAPL players and the `quartus_jli` command-line executable.

You can simplify in-field upgrades and enhance the quality, flexibility, and life-cycle of your end products by using Jam STAPL to implement ISP. The Jam STAPL solution provides a software-level and vendor-independent standard for ISP using PCs or embedded processors. The Jam STAPL solution is suitable for embedded systems—small file size, ease of use, and platform independence.

Jam STAPL Players

Altera supports two types of Jam STAPL file formats. There are two Jam STAPL players to accommodate these file types.

- Jam STAPL Player—ASCII text-based Jam STAPL files (`.jam`)
- Jam STAPL Byte-Code Player—byte-code Jam STAPL files (`.jbc`)

The Jam STAPL players parse the descriptive information in the `.jam` or `.jbc`. The players then interprets the information as data and algorithms to program the targeted devices. The players do not program a particular vendor or device architecture but only read and understand the syntax defined by the Jam STAPL specification.

Alternatively, you can also use the `quartus_jli` command-line executable to program and test Altera® devices using `.jam` or `.jbc`. The `quartus_jli` command-line executable is provided with the Quartus® II software version 6.0 and later.

Differences Between the Jam STAPL Players and quartus_jli

A single `.jam` or `.jbc` can contain several functions such as programming, configuring, verifying, erasing, and blank-checking a device.

The Jam STAPL players are interpreter programs that read and execute the `.jam` or `.jbc` files. The Jam STAPL players can access the IEEE 1149.1 signals that are used for all instructions based on the IEEE 1149.1 interface. The players can also process user-specified actions and procedures in the `.jam` or `.jbc`.

The `quartus_jli` command-line executable has the same functionality as the Jam STAPL players but with additional capabilities:

- It provides command-line control of the Quartus II software from the UNIX or DOS prompt.
- It supports all programming hardware available in the Quartus II software version 6.0 and later.

Table 1: Differences Between Jam STAPL Players and quartus_jli Command-Line Executable

- You can download the Altera Jam STAPL players from the Altera website.
- You can find the `quartus_jli` command-line executable in the `<Quartus II system directory>\bin` directory.

Features	Jam STAPL Players	quartus_jli
Supported Download Cables	ByteBlaster™ II, ByteBlasterMV, and ByteBlaster parallel port download cables.	All programming cables are supported by the JTAG server such as the USB-Blaster™, ByteBlaster II, ByteBlasterMV, ByteBlaster, MasterBlaster™, and EthernetBlaster.
Porting of Source Code to the Embedded Processor	Yes	No
Supported Platforms	• 16-bit and 32-bit embedded processors. • 32-bit Windows. • DOS. • UNIX.	• 32-bit Windows. • 64-bit Windows. • DOS. • UNIX.
Enable or Disable Procedure from the Command-Line Syntax	• To enable the optional procedure, use the –d<*procedure*>=1 option. • To disable the recommended procedure, use the –d<*procedure*>=0 option.	• To disable the recommended procedure, use the –d<*procedure*> option. • To enable the optional procedure, use the –e<*procedure*> option.

Related Information

Altera Jam STAPL Software

Provides the Altera Jam STAPL software for download.

Jam STAPL Files

Altera supports two types of Jam STAPL files: `.jam` ASCII text files and `.jbc` byte-code files.

ASCII Text Files (.jam)

Altera supports the following formats of the ASCII text-based `.jam`:

- JEDEC JESD71 STAPL format. Altera recommends that you use this format for new projects. In most cases, you use `.jam` files in tester environments.
- Jam version 1.1 format (pre-JEDEC).

Byte-Code Files

The binary `.jbc` files are compiled versions of `.jam` files. A `.jbc` is compiled to a virtual processor architecture where the ASCII text-based Jam STAPL commands are mapped to byte-code instructions compatible with the virtual processor.

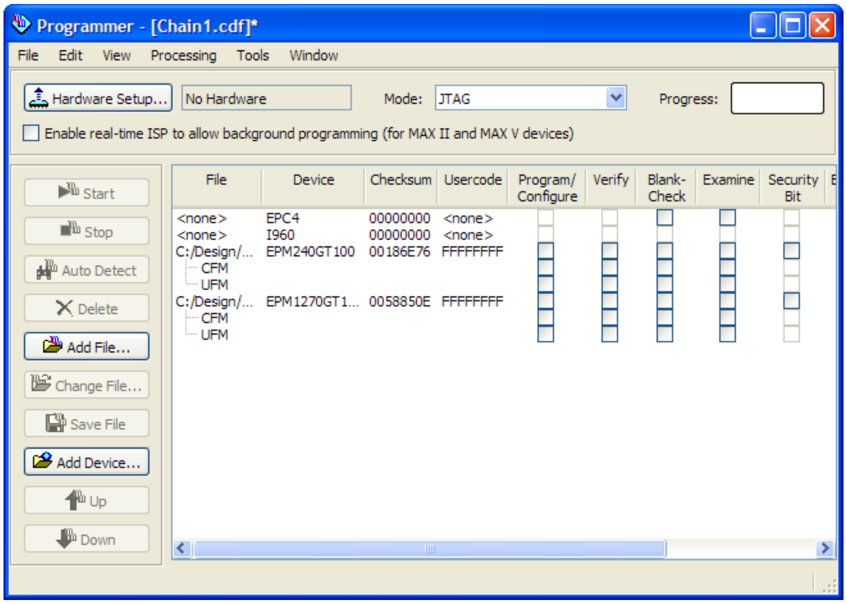
- Jam STAPL Byte-Code `.jbc` format—compiled version of the JEDEC JESD71 STAPL file. Altera recommends that you use this format in embedded application to minimize memory usage.
- Jam Byte-Code `.jbc` format—compiled version of the Jam version 1.1 format file.

Generating Byte-Code Jam STAPL Files

The Quartus II software can generate `.jam` and `.jbc` files. You can also compile a `.jam` into a `.jbc` with the stand-alone Jam STAPL Byte-Code Compiler. The compiler produces a `.jbc` that is functionally equivalent to the `.jam`.

The Quartus II software tools support programming and configuration of multiple devices from single or multiple `.jbc` files. You can include Altera and non-Altera JTAG-compliant devices in the JTAG chain. If you do not specify a programming file in the **Programming File Names** field, devices in the JTAG chain are bypassed.

Figure 1: Multi-Device JTAG Chain and Sequence Configuration in Quartus II Programmer



No Hardware

Mode: JTAG

Progress:

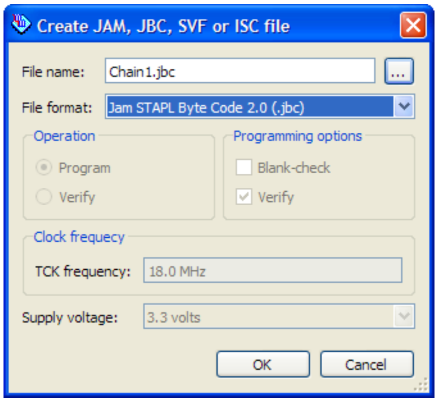
☐ Enable real-time ISP to allow background programming (for MAX II and MAX V devices)

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine	Security Bit
<none>	EPC4	00000000	<none>	☐	☐	☐	☐	☐
<none>	I960	00000000	<none>	☐	☐	☐	☐	☐
C:/Design/...	EPM240GT100	00186E76	FFFFFFFF	☐	☐	☐	☐	☐
CFM				☐	☐	☐	☐	☐
UFM				☐	☐	☐	☐	☐
C:/Design/...	EPM1270GT1...	0058850E	FFFFFFFF	☐	☐	☐	☐	☐
CFM				☐	☐	☐	☐	☐
UFM				☐	☐	☐	☐	☐

Note: If you convert JTAG chain files to `.jam`, the Quartus II Programmer options that you select for other devices in the JTAG chain are not programmed into the new `.jam`. The Quartus II Programmer ignores your programming options while you are creating a multi-device `.jam` or JTAG Indirect Configuration (`.jic`) file. However, you can choose the programming options to apply to the device when you use the Jam STAPL Player with the generated `.jam`. For a multi-device `.jam`, the programming options you choose are applied to each device that has a data file in the JTAG chain.

1. On the Quartus II menu, select **Tools** > **Programmer**.
2. Click **Add File** and select the programming files for the respective devices.
3. On the Quartus II Programmer menu, select **File** > **Create/Update** > **Create Jam, SVF, or ISC File**.
4. In the **File Format** list, select a `.jbc` format.

Figure 2: Generating a .jbc for a Multi-Device JTAG Chain in the Quartus II Software



Create JAM, JBC, SVF or ISC file

File name: Chain1.jbc

File format: Jam STAPL Byte Code 2.0 (.jbc)

Operation: ● Program ○ Verify

Programming options: ☐ Blank-check ☑ Verify

Clock frequecy

TCK frequency: 18.0 MHz

Supply voltage: 3.3 volts

5. Click **OK**.

Related Information

Altera Jam STAPL Software

Provides the Altera Jam STAPL software for download.

List of Supported .jam and .jbc Actions and Procedures

A `.jam` or `.jbc` consists two types of statements: action and procedure.

- Action—a sequence of steps required to implement a complete operation.
- Procedure—one of the steps contained in an action statement.

An action statement can contain one or more procedure statements or no procedure statement. For action statements that contain procedure statements, the procedure statements are called in the specified order to complete the associated operation. You can specify some of the procedure statements as “recommended” or “optional” to include or exclude them in the execution of the action statement.

Table 2: Supported .jam or .jbc Actions and Optional Procedures for Each Action in Altera Devices

Devices	(.jam)/(.jbc) Action	Optional Procedures (Off by default)
MAX® 3000A MAX 7000B MAX 7000AE	Program	• do_blank_check • do_secure • do_low_temp_programming • do_disable_isp_clamp • do_read_usercode
	Blankcheck	do_disable_isp_clamp
	Verify	• do_disable_isp_clamp • do_read_usercode
	Erase	do_disable_isp_clamp
	Read_usercode	—

Devices	(.jam)/(.jbc) Action	Optional Procedures (Off by default)
MAX II MAX V MAX 10 FPGA	Program	• do_blank_check • do_secure • do_disable_isp_clamp • do_bypass_cfm • do_bypass_ufm • do_real_time_isp • do_read_usercode • do_verify • do_force_sram_download • do_bypass_icb(1) • do_bypass_cfm1(1)
	Blankcheck	• do_disable_isp_clamp • do_bypass_cfm • do_bypass_ufm • do_real_time_isp • do_force_sram_download • do_bypass_icb(1) • do_bypass_cfm1(1)
	Verify	• do_disable_isp_clamp • do_bypass_cfm • do_bypass_ufm • do_real_time_isp • do_read_usercode • do_force_sram_download • do_bypass_icb(1) • do_bypass_cfm1(1)
	Erase	• do_disable_isp_clamp • do_bypass_cfm • do_bypass_ufm • do_real_time_isp • do_force_sram_download • do_bypass_icb(1) • do_bypass_cfm1(1) • do_blank_check
	Read_usercode	—

---

(1) Applicable in MAX 10 FPGA only.

Devices	(.jam)/(.jbc) Action	Optional Procedures (Off by default)
Stratix® device family Arria® device family Cyclone® device family	Configure	• do_read_usercode • do_halt_on_chip_cc • do_ignore_idcode_errors
	Read_usercode	—
Enhanced Configuration Devices	Program	• do_blank_check • do_secure • do_read_usercode • do_init_configuration
	Blankcheck	—
	Verify	do_read_usercode
	Erase	—
	Read_usercode	—
	Init_configuration	—
Serial Configuration Devices	Configure	• do_read_usercode • do_halt_on_chip_cc • do_ignore_idcode_errors
	Program	• do_blank_check • do_epcs_unprotect
	Blankcheck	—
	Verify	—
	Erase	—
	Read_usercode	—

Definitions of .jam and .jbc Action and Procedure Statements

Table 3: Definitions of .jam Action Statements

Action	Description
Program	Programs the device.
Blankcheck	Checks the erased state of the device.
Verify	Verifies the entire device against the programming data in the `.jam` or `.jbc`.
Erase	Performs a bulk erase of the device.
Read_usercode	Returns the JTAG USERCODE register information from the device.

Action	Description
Configure	Configures the device.
Init_configuration	Specifies that the configuration device configures the attached devices immediately after programming.
Check_idcode	Compares the actual device IDCODE with the expected IDCODE generated in the `.jam` and `.jbc`.

Table 4: Definitions of .jam Procedure Statements

Procedure	Description
do_blank_check	When enabled, the device is blank-checked.
do_secure	When enabled, the security bit of the device is set.
do_read_usercode	When enabled, the player reads the JTAG USERCODE of the device and prints it to the screen.
do_disable_isp_clamp	When enabled, the ISP clamp mode of the device is ignored.
do_low_temp_programming	When enabled, the procedure allows the industrial low temperature ISP for MAX 3000A, 7000B, and 7000AE devices.
do_bypass_cfm	When enabled, the procedure performs the specified action only on the user flash memory (UFM).
do_bypass_ufm	When enabled, the procedure performs the specified action only on the configuration flash memory (CFM).
do_real_time_isp	When enabled, the real-time ISP feature is turned on for the ISP action being executed.
do_init_configuration	When enabled, the configuration device configures the attached device immediately after programming.
do_halt_on_chip_cc	When enabled, the procedure halts the auto-configuration controller to allow programming using the JTAG interface. The nSTATUS pin remains low even after the device is successfully configured.
do_ignore_idcode_errors	When enabled, the procedure allows configuration of the device even if an IDCODE error exists.
do_erase_all_cfi	When enabled, the procedure erases the common flash interface (CFI) flash memory that is attached to the parallel flash loader (PFL) of the MAX 10, MAX V, or MAX II device.
do_epcs_unprotect	When enabled, the procedure removes the protection mode of the serial configuration devices (EPCS).
do_verify	When Enabled, during Programming, the data is verified
do_bypass_icb	By default, operations will be targeted on fullchip (except read back). However if this procedure is enabled, ICB settings will be excluded.

Procedure	Description
do_bypass_cfm1	By default, operations will be targeted on fullchip (except read back). However if this procedure is enabled, CFM1 sector (if present) will be excluded.
do_force_sram_download	When this option is set, CRAM is upgraded (= internal reconfiguration) automatically on the timing pof was loaded to CFM. This option is used with real_time_isp.

Jam STAPL Player and quartus_jli Exit Codes

Exit codes are the integer values that indicate the result of an execution of a `.jam` or `.jbc`. An exit code value of zero indicates success. A non-zero value indicates failure and identifies the general type of failure that occurred.

Table 5: Exit Codes Defined in Jam STAPL Specification (JEST71)

Both the Jam STAPL Player and the `quartus_jli` command-line executable can return the exit codes listed in this table.

Exit Code	Description
0	Success
1	Checking chain failure
2	Reading IDCODE failure
3	Reading USERCODE failure
4	Reading UESCODE failure
5	Entering ISP failure
6	Unrecognized device ID
7	Device version is not supported
8	Erase failure
9	Blank-check failure
10	Programming failure
11	Verify failure
12	Read failure
13	Calculating checksum failure
14	Setting security bit failure
15	Querying security bit failure
16	Exiting ISP failure
17	Performing system test failure

Using the Jam STAPL Player

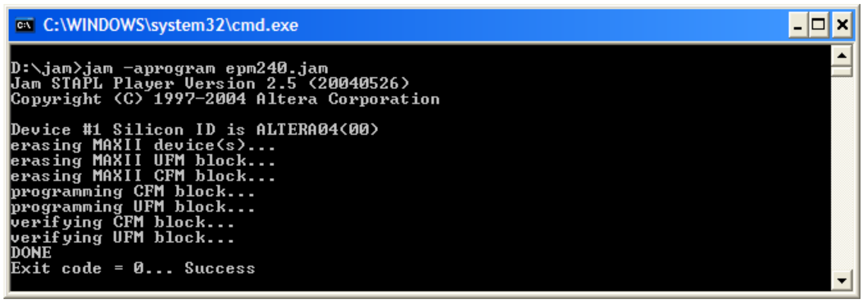
The Jam STAPL Player commands and parameters are not case-sensitive. You can write the option flags in any sequence.

To specify an action in the Jam STAPL Player command, use the -a option followed immediately by the action statement with no spaces. The following command programs the entire device using the specified `.jam`:

```
jam -aprogram <filename>.jam
```

Figure 3: Programming an EPM240 Device Using the Jam STAPL Player

This figure shows an example of a successful action with an exit code value of zero.

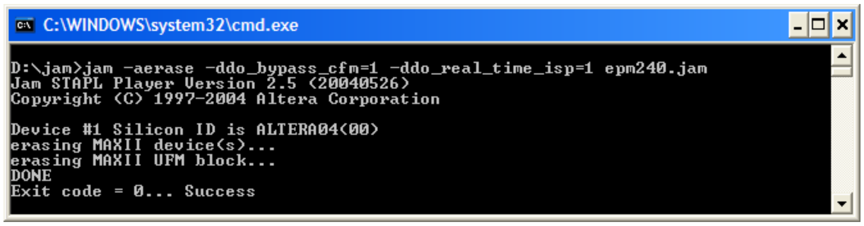


```
D:\jam>jam -aprogram epm240.jam
Jam STAPL Player Version 2.5 (20040526)
Copyright (C) 1997-2004 Altera Corporation

Device #1 Silicon ID is ALTERA04(00)
erasing MAXII device(s)...
erasing MAXII UFM block...
erasing MAXII CFM block...
programming CFM block...
programming UFM block...
verifying CFM block...
verifying UFM block...
DONE
Exit code = 0... Success
```

You can execute the optional procedures associated with each action using the –d option followed immediately by the procedure statement with no spaces. The following command erases only the UFM block of the device using real-time ISP:

```
jam -aerase -ddo_bypass_cfm=1 -ddo_real_time_isp=1 <filename>.jam
```

Figure 4: Erasing Only the UFM Block of the Device with the Real-Time ISP Feature Enabled



```
D:\jam>jam -aerase -ddo_bypass_cfm=1 -ddo_real_time_isp=1 epm240.jam
Jam STAPL Player Version 2.5 (20040526)
Copyright (C) 1997-2004 Altera Corporation

Device #1 Silicon ID is ALTERA04(00)
erasing MAXII device(s)...
erasing MAXII UFM block...
DONE
Exit code = 0... Success
```

Note: To run a .jbc, use the Jam STAPL Byte-Code Player executable name (`jbi`) with the same commands and parameters as the Jam STAPL Player.

Note: To program serial configuration devices with the Jam STAPL Player, you must first configure the FPGA with the Serial FlashLoader image. The following commands are required:

```
jam -aconfigure <filename>.jam
jam -aprogram <filename>.jam
```

Related Information

AN 370: Using the Serial FlashLoader With the Quartus II Software

Provides more information about generating .jam for serial configuration devices.

Using the quartus_jli Command-Line Executable

The `quartus_jli` command-line executable supports all Altera download cables such as the ByteBlaster, ByteBlasterMV, ByteBlaster II, USB-Blaster, MasterBlaster, and Ethernet Blaster.

Table 6: Command-Line Executable Options for quartus_jli Command-Line Executable

The `quartus_jli` commands and parameters are not case-sensitive. You can write the option flags in any sequence.

Option	Description
-a	Specifies the action to perform.
-c	Specifies the JTAG server cable number.
-d	Disables a recommended procedure.
-e	Enables an optional procedure.
-i	Displays information on a specific option or topic.
-l	Displays the header file information in a `.jam` or the list of supported actions and procedures in a `.jbc` file when the file is executed with an action statement.
-n	Displays the list of available hardware.
-f	Specifies a file containing additional command-line arguments.

Related Information

Differences Between the Jam STAPL Players and quartus_jli on page 1

Provides more information about download cables.

Command-line Syntax of quartus_jli Command-Line Executable

To specify which programming hardware or cable to use when performing an action statement, use this command syntax:

```
quartus_jli -a<action name> -c<cable index> <filename>.jam
```

To enable a procedure associated with an action statement, use this command syntax:

```
quartus_jli -a<action name> -e<procedure to enable> -c<cable index> <filename>.jam
```

To disable a procedure associated with an action statement, use this command syntax:

```
quartus_jli -a<action name> -d<procedure to disable> -c<cable index> <filename>.jam
```

To program serial configuration devices with the `quartus_jli` command-line executable, use the following commands:

```
quartus_jli -aconfigure <filename>.jam
quartus_jli -aprogram <filename>.jam
```

To get more information about an option, use this command syntax:

```
quartus_jli --help=<option|topic>
```

The following examples show the command-line syntax to run the `quartus_jli` command-line executable.

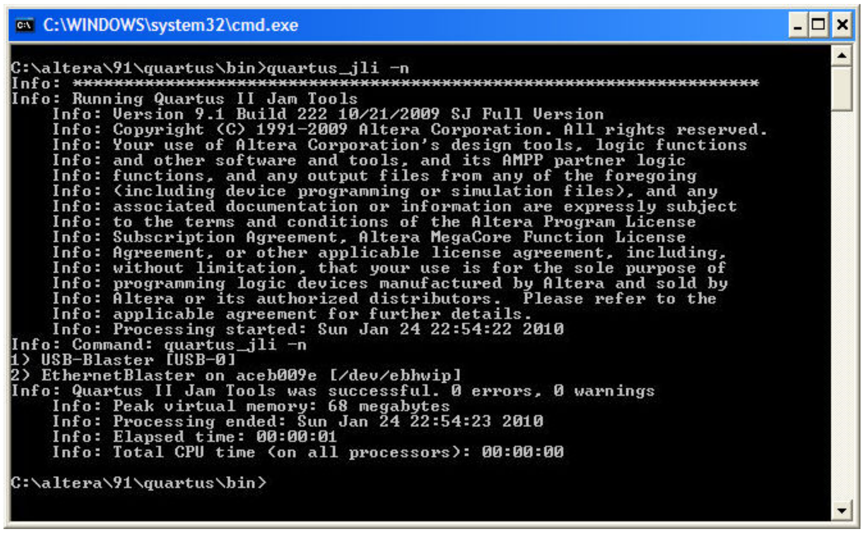
Example 1: Display a List of Available Download Cables in a Machine

To display a list of available download cables on a machine as shown in the following figure, at the command prompt, type this command:

```
quartus_jli -n
```

Figure 5: Display of the Available Download Cables

Numbers `1)` and `2)` in the figure are the cable index numbers. In the command, replace <*cable index*> with the index number of the relevant cable



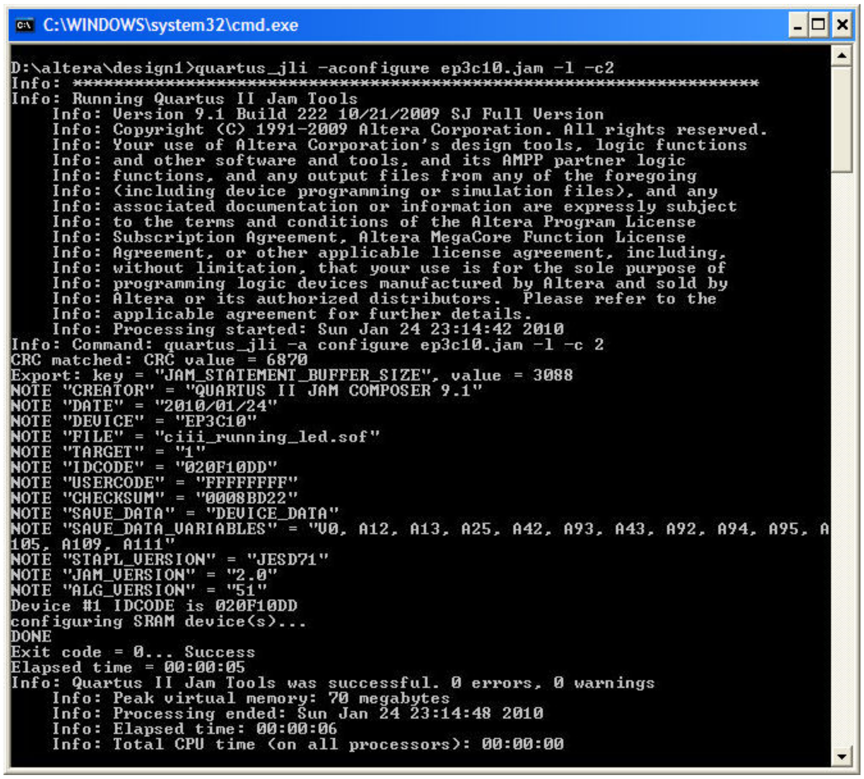
```
C:\altera\91\quartus\bin>quartus_jli -n
Info: *******************************************************************
Info: Running Quartus II Jam Tools
    Info: Version 9.1 Build 222 10/21/2009 SJ Full Version
    Info: Copyright (C) 1991-2009 Altera Corporation. All rights reserved.
    Info: Your use of Altera Corporation's design tools, logic functions
    Info: and other software and tools, and its AMPP partner logic
    Info: functions, and any output files from any of the foregoing
    Info: (including device programming or simulation files), and any
    Info: associated documentation or information are expressly subject
    Info: to the terms and conditions of the Altera Program License
    Info: Subscription Agreement, Altera MegaCore Function License
    Info: Agreement, or other applicable license agreement, including,
    Info: without limitation, that your use is for the sole purpose of
    Info: programming logic devices manufactured by Altera and sold by
    Info: Altera or its authorized distributors.  Please refer to the
    Info: applicable agreement for further details.
    Info: Processing started: Sun Jan 24 22:54:22 2010
Info: Command: quartus_jli -n
1) USB-Blaster [USB-0]
2) EthernetBlaster on aceb009e [/dev/ebhwip]
Info: Quartus II Jam Tools was successful. 0 errors, 0 warnings
    Info: Peak virtual memory: 68 megabytes
    Info: Processing ended: Sun Jan 24 22:54:23 2010
    Info: Elapsed time: 00:00:01
    Info: Total CPU time (on all processors): 00:00:00

C:\altera\91\quartus\bin>
```

Example 2: Display Header File Information in a Jam File

To display the header file information in a .jam when executing an action statement, use this command syntax:

```
quartus_jli -a<action name> <filename>.jam -l
```

Figure 6: Header File Information of a Jam File when Executing an Action Statement



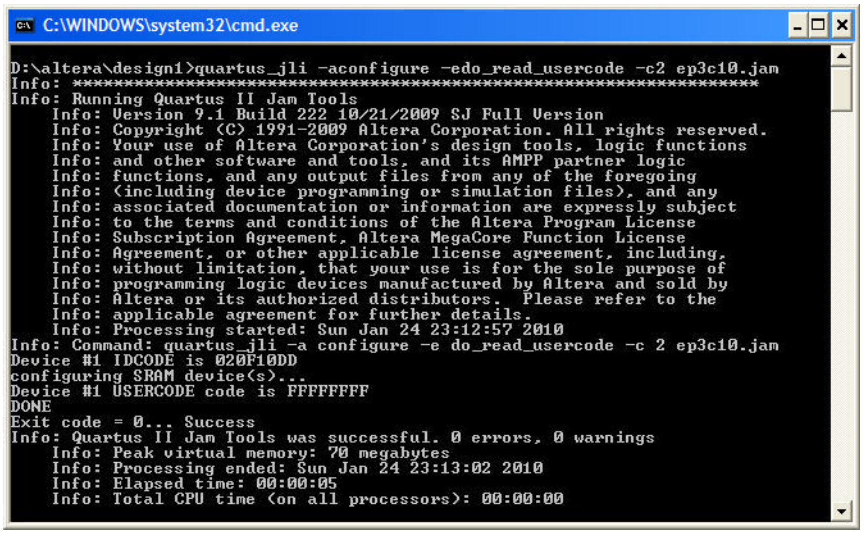
```
D:\altera\design1>quartus_jli -aconfigure ep3c10.jam -l -c2
Info: ********************************************************************
Info: Running Quartus II Jam Tools
    Info: Version 9.1 Build 222 10/21/2009 SJ Full Version
    Info: Copyright (C) 1991-2009 Altera Corporation. All rights reserved.
    Info: Your use of Altera Corporation's design tools, logic functions
    Info: and other software and tools, and its AMPP partner logic
    Info: functions, and any output files from any of the foregoing
    Info: (including device programming or simulation files), and any
    Info: associated documentation or information are expressly subject
    Info: to the terms and conditions of the Altera Program License
    Info: Subscription Agreement, Altera MegaCore Function License
    Info: Agreement, or other applicable license agreement, including,
    Info: without limitation, that your use is for the sole purpose of
    Info: programming logic devices manufactured by Altera and sold by
    Info: Altera or its authorized distributors.  Please refer to the
    Info: applicable agreement for further details.
    Info: Processing started: Sun Jan 24 23:14:42 2010
Info: Command: quartus_jli -a configure ep3c10.jam -l -c 2
CRC matched: CRC value = 6870
Export: key = "JAM_STATEMENT_BUFFER_SIZE", value = 3088
NOTE "CREATOR" = "QUARTUS II JAM COMPOSER 9.1"
NOTE "DATE" = "2010/01/24"
NOTE "DEVICE" = "EP3C10"
NOTE "FILE" = "ciii_running_led.sof"
NOTE "TARGET" = "1"
NOTE "IDCODE" = "020F10DD"
NOTE "USERCODE" = "FFFFFFFF"
NOTE "CHECKSUM" = "0008BD22"
NOTE "SAVE_DATA" = "DEVICE_DATA"
NOTE "SAVE_DATA_VARIABLES" = "V0, A12, A13, A25, A42, A93, A43, A92, A94, A95, A
105, A109, A111"
NOTE "STAPL_VERSION" = "JESD71"
NOTE "JAM_VERSION" = "2.0"
NOTE "ALG_VERSION" = "51"
Device #1 IDCODE is 020F10DD
configuring SRAM device(s)...
DONE
Exit code = 0... Success
Elapsed time = 00:00:05
Info: Quartus II Jam Tools was successful. 0 errors, 0 warnings
    Info: Peak virtual memory: 70 megabytes
    Info: Processing ended: Sun Jan 24 23:14:48 2010
    Info: Elapsed time: 00:00:06
    Info: Total CPU time (on all processors): 00:00:00
```

Example 3: Configure and Return JTAG USERCODE of an FPGA Device

To configure and return the JTAG USERCODE of an FPGA device using the second download cable on the machine with a specific .jam, at the command prompt, type this command:

```
quartus_jli -aconfigure -edo_read_usercode -c2 <filename>.jam
```

Figure 7: Configuring and Reading the JTAG USERCODE of the EP2C70 Device Using the USB-Blaster Cable



```
D:\altera\design1>quartus_jli -aconfigure -edo_read_usercode -c2 ep3c10.jam
Info: *******************************************************************
Info: Running Quartus II Jam Tools
    Info: Version 9.1 Build 222 10/21/2009 SJ Full Version
    Info: Copyright (C) 1991-2009 Altera Corporation. All rights reserved.
    Info: Your use of Altera Corporation's design tools, logic functions
    Info: and other software and tools, and its AMPP partner logic
    Info: functions, and any output files from any of the foregoing
    Info: (including device programming or simulation files), and any
    Info: associated documentation or information are expressly subject
    Info: to the terms and conditions of the Altera Program License
    Info: Subscription Agreement, Altera MegaCore Function License
    Info: Agreement, or other applicable license agreement, including,
    Info: without limitation, that your use is for the sole purpose of
    Info: programming logic devices manufactured by Altera and sold by
    Info: Altera or its authorized distributors.  Please refer to the
    Info: applicable agreement for further details.
    Info: Processing started: Sun Jan 24 23:12:57 2010
Info: Command: quartus_jli -a configure -e do_read_usercode -c 2 ep3c10.jam
Device #1 IDCODE is 020F10DD
configuring SRAM device(s)...
Device #1 USERCODE code is FFFFFFFF
DONE
Exit code = 0... Success
Info: Quartus II Jam Tools was successful. 0 errors, 0 warnings
    Info: Peak virtual memory: 70 megabytes
    Info: Processing ended: Sun Jan 24 23:13:02 2010
    Info: Elapsed time: 00:00:05
    Info: Total CPU time (on all processors): 00:00:00
```

Using Jam STAPL for ISP with an Embedded Processor

Embedded systems contain both hardware and software components. When you are designing an embedded system, lay out the PCB first. Then, develop the firmware that manages the functionality of the board.

Methods to Connect the JTAG Chain to the Embedded Processor

You can connect the JTAG chain to the embedded processor in two ways:

- Connect the embedded processor directly to the JTAG chain
- Connect the JTAG chain to an existing bus using an interface device

In both JTAG connection methods, you must include space for the MasterBlaster or ByteBlasterMV header connection. The header is useful during prototyping because it allows you to quickly verify or modify the contents of the device. During production, you can remove the header to save cost.

Connecting the Embedded Processor Directly to the JTAG Chain

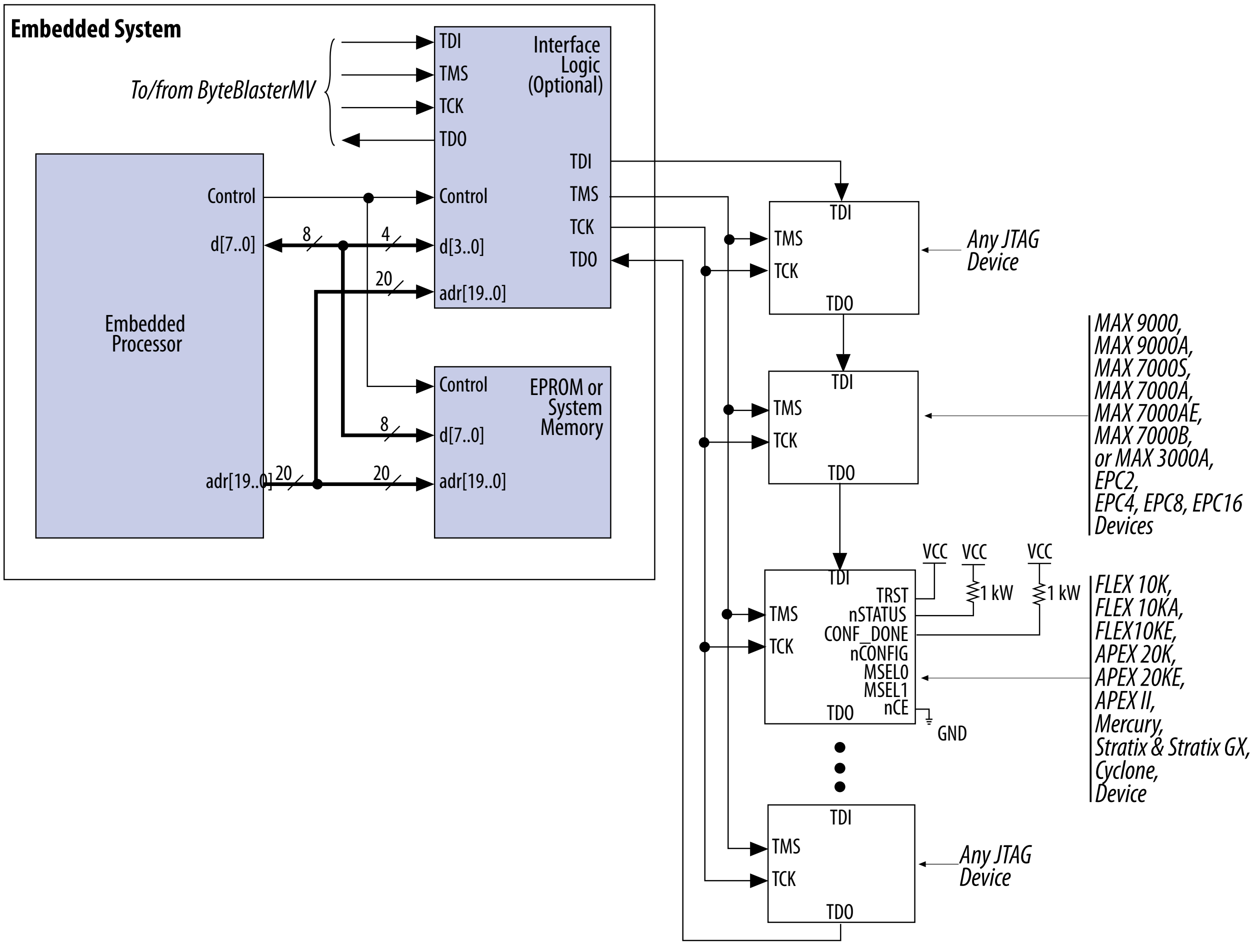
In this method, four of the processor pins are dedicated to the JTAG interface.

This method is the most straightforward. This method saves board space but reduces the number of available embedded processor pins.

Connecting the JTAG Chain to an Existing Bus Using an Interface Device

In this method, the JTAG chain is represented by an address on the existing bus and the processor performs read and write operations on this address.

Figure 8: Connecting the JTAG Chain to the Embedded System



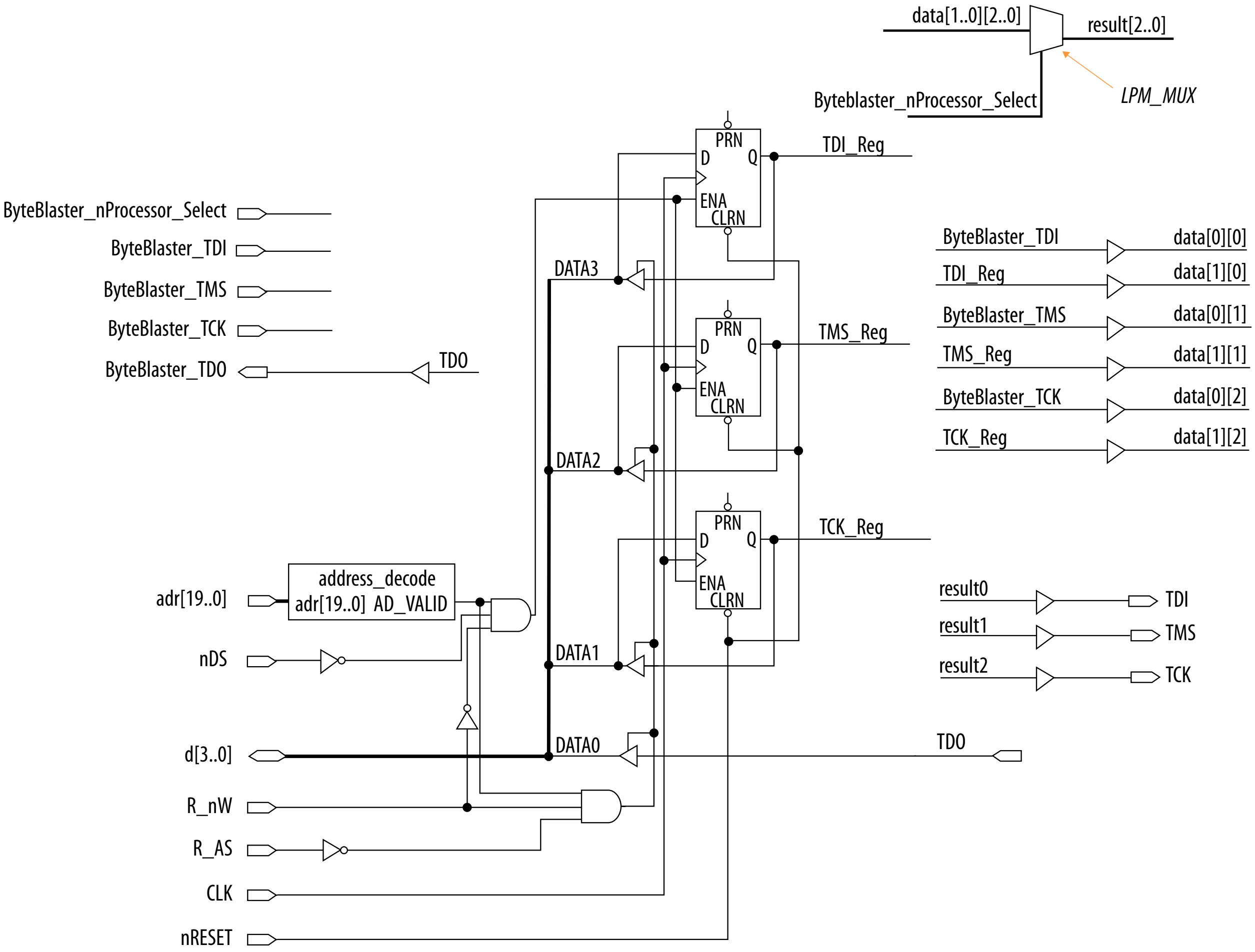
Example 4: Design Schematic of Interface Device

The following figure shows an example design schematic of an interface device. This example design is for your reference only. If you use this example, you must ensure that:

- `TMS`, `TCK`, and `TDI` are synchronous outputs
- Multiplexer logic is included to allow board access for the MasterBlaster or ByteBlasterMV download cable

Figure 9: Interface Logic Design Example

Except for the data[3..0] data path, all other inputs in this figure are optional. These inputs are included only to illustrate how you can use the interface device as an address on an embedded data bus.



The embedded processor asserts the JTAG chain's address. You can set the `R_nW` and `R_AS` signals to notify the interface device when you want the processor to access the chain.

- To write—connect the `data[3..0]` data path to the JTAG outputs of the device using the three `D` registers that are clocked by the system clock (`CLK`). This clock can be the same clock used by the processor.
- To read—enable the tri-state buffers and let the `TDO` signal flow back to the processor.

This example design also provides a hardware connection to read back the values in the `TDI`, `TMS`, and `TCK` registers. This optional feature is useful during the development phase because it allows the software to check the valid states of the registers in the interface device.

In addition, the example design includes multiplexer logic to permit a MasterBlaster or ByteBlasterMV download cable to program the device chain. This capability is useful during the prototype phase of development when you want to verify the programming and configuration.

Board Layout

When you lay out a board that programs or configures the device using the IEEE Std. 1149.1 JTAG chain, you must observe several important elements.

Treat the TCK Signal Trace as a Clock Tree on page 16
The `TCK` signal is the clock for the entire JTAG chain of devices. Because these devices are edge-triggered by the `TCK` signal, you must protect the `TCK` signal from high-frequency noise and ensure that the signal integrity is good.

Use a Pull-Down Resistor on the TCK Signal on page 16
You must hold the `TCK` signal low using a pull-down resistor to keep the JTAG test access port (TAP) in a known state at power-up.

Make the JTAG Signal Traces as Short as Possible on page 16
Short JTAG signal traces help eliminate noise and drive-strength issues.

Add External Resistors to Pull the Outputs to a Defined Logic Level on page 17
During programming or configuration, you must add external resistors to the output pins to pull the outputs to a defined logic level.

Treat the TCK Signal Trace as a Clock Tree

The `TCK` signal is the clock for the entire JTAG chain of devices. Because these devices are edge-triggered by the `TCK` signal, you must protect the `TCK` signal from high-frequency noise and ensure that the signal integrity is good.

Ensure that the `TCK` signal meets the rise time (t_R) and fall time (t_F) parameters specified in the data sheet of the relevant device family.

You may also need to terminate the signal to prevent overshoot, undershoot, or ringing. This step is often overlooked because the signal is software-generated and originated at a processor general-purpose I/O pin.

Use a Pull-Down Resistor on the TCK Signal

You must hold the `TCK` signal low using a pull-down resistor to keep the JTAG test access port (TAP) in a known state at power-up.

A missing pull-down resistor can cause a device to power-up in the state of JTAG and its boundary-scan test (BST). This situation can cause conflicts on the board.

A typical resistor value is 1 kΩ.

Make the JTAG Signal Traces as Short as Possible

Short JTAG signal traces help eliminate noise and drive-strength issues.

Give special attention to the `TCK` and `TMS` pins. Because `TCK` and `TMS` signals are connected to every device in the JTAG chain, these traces see higher loading than the `TDI` or `TDO` signals.

Depending on the length and loading of the JTAG chain, you may require additional buffering to ensure the integrity of the signals that propagate to and from the processor.

Add External Resistors to Pull the Outputs to a Defined Logic Level

During programming or configuration, you must add external resistors to the output pins to pull the outputs to a defined logic level.

The output pins tri-state during programming or configuration. Additionally, on MAX 7000, FLEX® 10K, APEX™ 20K, and all configuration devices, the pins are pulled up by a weak internal resistor—for example, 50 kΩ.

However, not all Altera devices have weak pull-up resistors during ISP or in-circuit reconfiguration. For information about which device has weak pull-up resistors, refer to the data sheet of the relevant device family.

Note: Altera recommends that you tie the outputs that drive sensitive input pins to the appropriate level using an external resistor on the order of 1 kΩ. You may need to analyze each of the preceding board layout elements further, especially signal integrity. In some cases, analyze the loading and layout of the JTAG chain to determine whether you need to use discrete buffers or a termination technique.

Related Information

AN100: In-System Programmability Guidelines

Embedded Jam STAPL Players

The embedded Jam STAPL Player is able to read `.jam` that conforms to the standard JEDEC file format and is backward compatible with legacy Jam version 1.1 syntax. Similarly, the Jam STAPL Byte-Code Player can play `.jbc` compiled from Jam STAPL and Jam version 1.1 `.jam`.

The Jam STAPL Byte-Code Player

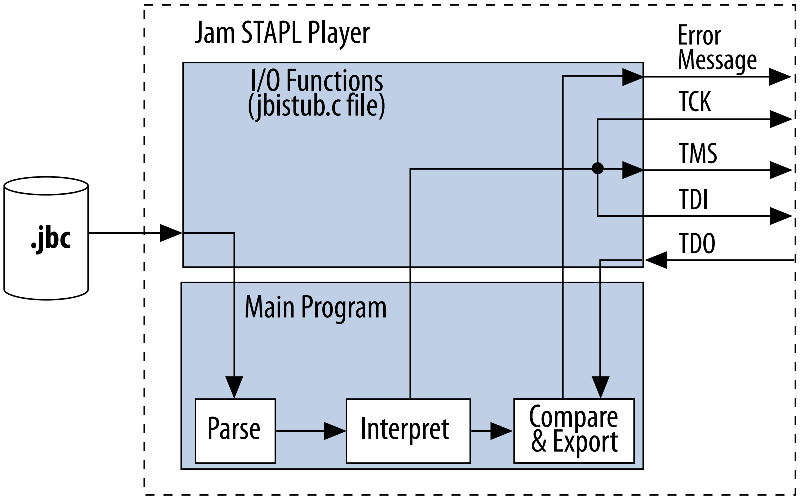
The Jam STAPL Byte-Code Player is coded in the C programming language for 16 bit and 32 bit processors. A specific subset of the player source code also supports some 8 bit processors.

The source code for the 16 bit and 32 bit Jam STAPL Byte-Code Player is divided into two categories:

- `jbistub.c`—platform-specific code that handles I/O functions and applies to specific hardware.
- All other C files—generic code that performs the internal functions of the player.

Figure 10: Jam STAPL Byte-Code Player Source Code Structure

This shows the organization of the source code files by function. The process of porting the Jam STAPL Byte-Code Player to a particular processor is simplified because the platform-specific code is all kept inside `jbistub.c`.



Related Information

AN 111: Embedded Programming Using the 8051 and Jam Byte-Code

Provides more information about Altera's support for 8 bit processors.

Steps to Port the Jam STAPL Byte-Code Player

The default configuration of `jbistub.c` includes the code for DOS, 32 bit Windows, and UNIX. Because of this configuration, the source code is compiled and evaluated for the correct functionality and debugging on these operating systems.

For embedded environments, you can remove this code with a single `#define` preprocessor statement. In addition, porting the code involves making minor changes to specific parts of the code in `jbistub.c`.

Table 7: Functions Requiring Customization

This table lists the `jbistub.c` functions that you must customize to port the Jam STAPL Byte-Code Player.

Function	Description
`jbi_jtag_io()`	Provides interfaces to the four IEEE 1149.1 JTAG signals, `TDI`, `TMS`, `TCK`, and `TDO`.
`jbi_export()`	Passes information, such as the user electronic signature (UES), back to the calling program.
`jbi_delay()`	Implements the programming pulses or delays needed during execution.
`jbi_vector_map()`	Processes signal-to-pin map for non-IEEE 1149.1 JTAG signals.
`jbi_vector_io()`	Asserts non-IEEE 1149.1 JTAG signals as defined in the VECTOR MAP.

Perform the steps in the following sections to ensure that you customize all the necessary codes.

1. **Step 1: Set the Preprocessor Statements to Exclude Extraneous Code** on page 19
   To eliminate DOS, Windows, and UNIX source code and included libraries, change the default `PORT` parameter to `EMBEDDED`.
2. **Step 2: Map the JTAG Signals to the Hardware Pins** on page 19
   The `jbi_jtag_io()` function in `jbistub.c` contains the code that sends and receives the binary programming data. By default, the source code writes to the parallel port of the PC. You must remap all four JTAG signals to the pins of the embedded processor.
3. **Step 3: Handle Text Messages from jbi_export()** on page 20
   The `jbi_export()` function uses the `printf()` function to send text messages to `stdio`.
4. **Step 4: Customize Delay Calibration** on page 20
   The `calibrate_delay()` function determines how many loops the host processor runs in a millisecond. This calibration is important because accurate delays are used in programming and configuration.

Step 1: Set the Preprocessor Statements to Exclude Extraneous Code

To eliminate DOS, Windows, and UNIX source code and included libraries, change the default `PORT` parameter to `EMBEDDED`.

Add this code to the top of `jbiport.h`:

```
#define PORT EMBEDDED
```

Step 2: Map the JTAG Signals to the Hardware Pins

The `jbi_jtag_io()` function in `jbistub.c` contains the code that sends and receives the binary programming data. By default, the source code writes to the parallel port of the PC. You must remap all four JTAG signals to the pins of the embedded processor.

Figure 11: Default PC Parallel Port Signal Mapping

This figure shows the `jbi_jtag_io()` signal mapping of the JTAG pins to the parallel port registers of the PC. The PC parallel port hardware inverts the most significant bit: `TDO`.

7	6	5	4	3	2	1	0	I/O Port
0	TDI	0	0	0	0	TMS	TCK	OUTPUT DATA - Base Address
TDO	X	X	X	X	---	---	---	INPUT DATA - Base Address + 1

Example 5: PC Parallel Port Signal Mapping Sample Source Code for jbi_jtag_io()

```
int jbi_jtag_io(int tms, int tdi, int read_tdo)
{
    int data = 0;
    int tdo = 0;
```

```
        if (!jtag_hardware_initialized)
        {
            initialize_jtag_hardware();
            jtag_hardware_initialized = TRUE;
        }
        data = ((tdi ? 0x40 : 0) | (tms ? 0x2 : 0));    /*TDI,TMS*/
        write_byteblaster(0, data);

        if (read_tdo)
        {
            tdo = (read_byteblaster(1) & 0x80) ? 0 : 1; /*TDO*/
        }
        write_blaster(0, data | 0x01);                  /*TCK*/
        write_blaster(0, data);

        return (tdo);
    }
```

- The PC parallel port inverts the actual value of `TDO`. Because of this, the `jbi_jtag_io()` function in the preceding code inverts the value again to retrieve the original data in the following line:

  ```
  tdo = (read_byteblaster(1) & 0x80) ? 0 : 1;
  ```

- If your target processor does not invert `TDO`, use the following code:

  ```
  tdo = (read_byteblaster(1) & 0x80) ? 1 : 0;
  ```

- To map the signals to the correct addresses, use the left shift (<<) or right shift (>>) operator. For example, if `TMS` and `TDI` are at ports 2 and 3, respectively, use this code:

  ```
  data = (((tdi ? 0x40 : 0) >> 3) | ((tms ? 0x02 : 0) << 1));
  ```

- Apply the same process to `TCK` and `TDO`.

The `read_byteblaster` and `write_byteblaster` signals use the `inp()` and `outp()` functions from the `conio.h` library, respectively, to read and write to the port. If these functions are not available, you must substitute them with equivalent functions.

Step 3: Handle Text Messages from jbi_export()

The `jbi_export()` function uses the `printf()` function to send text messages to `stdio`. The Jam STAPL Byte-Code Player uses the `jbi_export()` signal to pass information, for example, the device UES or USERCODE, to the operating system or software that calls the Jam STAPL Byte-Code Player. The function passes text and numbers as strings and decimal integers, respectively.

If there is no `stdout` device available, the information can be redirected to a file or storage device, or passed back as a variable to the program that called the player.

Related Information

AN 39: IEEE 1149.1 JTAG Boundary-Scan Testing in Altera Devices

Step 4: Customize Delay Calibration

The `calibrate_delay()` function determines how many loops the host processor runs in a millisecond. This calibration is important because accurate delays are used in programming and configuration.

By default, this number is hardcoded as 1,000 loops per millisecond and represented as:

```
one_ms_delay = 1000
```

If this parameter is known, adjust it accordingly. Otherwise, use code similar to the code included for Windows and DOS platforms that counts the number of clock cycles it takes to execute a single loop. This code has been sampled over multiple tests and, on average, produces an accurate delay result. The advantage to this approach is that calibration can vary based on the speed of the host processor.

After the Jam STAPL Byte-Code Player is ported and working, verify the timing and speed of the JTAG port at the target device. Timing parameters for the supported Altera devices must comply with the JTAG timing parameters and values provided in the data sheet of the relevant device family.

If the Jam STAPL Byte-Code Player does not operate within the timing specifications, you must optimize the code with the appropriate delays. Timing violations can occur in powerful processors that can generate `TCK` at a rate faster than 10 MHz.

Note: To avoid unpredictable Jam STAPL Byte-Code Player operation, Altera strongly recommends keeping the source code files other than `jbistub.c` in their default state.

Jam STAPL Byte-Code Player Memory Usage

The Jam STAPL Byte-Code Player uses memory in a predictable manner. You can estimate the ROM and RAM usage.

Estimating ROM Usage

Figure 12: Equation to Estimate the Maximum Required ROM Size

Use this equation to estimate the maximum amount of ROM required to store the Jam STAPL Byte-Code Player and the `.jbc`.

$$ROM\ Size = .jbc\ Size + Jam\ STAPL\ Byte\text{-}Code\ Player\ Size$$

The `.jbc` size can be separated into these categories:

- The amount of memory required to store the programming data.
- The space required for the programming algorithm.

Figure 13: Equation to Estimate `.jbc` Size

This equation provides a `.jbc` size estimate that may vary by ±10%, depending on device utilization. If device utilization is low, `.jbc` sizes tend to be smaller because the compression algorithm used to minimize file size will more likely find repetitive data.

This equation also indicates that the algorithm size stays constant for a device family but the programming data size grows slightly as more devices are targeted. For a given device family, the increase in the `.jbc` size caused by the data component is linear.

$$.jbc\ Size = Alg + \sum_{k=1}^{N} Data$$

- *Alg* stands for space used by the algorithm
- *Data* stands for space used by the compressed programming data
- *k* stands for the index representing the device being targeted
- *N* stands for the number of target devices in the chain

Algorithm File Size Constants

Table 8: Algorithm File Size Constants Targeting a Single Altera Device Family

Device	Typical `.jbc` Algorithm Size (KB)
Stratix device family	15
Cyclone device family	15
Arria device family	15
Mercury™	15
EPC16	24
EPC8	24
EPC4	24
EPC2	19
MAX 7000AE	21
MAX 7000	21
MAX 3000A	21
MAX 9000	21
MAX 7000S	25
MAX 7000A	25
MAX 7000B	17
MAX II	24.3
MAX V	24.3
MAX 10	24.3(2)

Table 9: Algorithm File Size Constants Targeting Multiple Altera Device Families

This table lists the algorithm file size constants for possible combinations of Altera device families that support the Jam language.

Devices	Typical .jbc Algorithm Size (KB)
FLEX 10K, MAX 7000A, MAX 7000S, MAX 7000AE(3)	31
FLEX 10K, MAX 9000, MAX 7000A, MAX 7000S, MAX 7000AE	45
MAX 7000S, MAX 7000A, MAX 7000AE	31
MAX 9000, MAX 7000A, MAX 7000S, MAX 7000AE	45

---

(2) Size is preliminary.

(3) If you are configuring FLEX or APEX devices, and programming MAX 9000 and MAX 7000 devices, the FLEX or APEX algorithm adds negligible memory.

Compressed and Uncompressed Data Size Constants

Table 10: Data Constants for Altera Devices Supporting the Jam Language (for ISP)

In this table, the enhanced configuration devices (EPC) data sizes use a compressed Programmer Object File (`.pof`).

Device	Typical Jam STAPL Byte-Code Data Size (KB)	
	Compressed	Uncompressed[4]
EP1S10	105	448
EP1S20	188	745
EP1S25	241	992
EP1S30	320	1310
EP1S40	369	1561
EP1S60	520	2207
EP1S80	716	2996
EP1C3	32	82
EP1C6	57	150
EP1C12	100	294
EP1C20	162	449
EPC4[5]	242	370
EPC8[5]	242	370
EPC8[6]	547	822
EPC16[5]	242	370
EPC16[7]	827	1344
EP1SGX25	243	992
EP1SGX40	397	1561
EP1M120	30	167
EP1M350	76	553
EP20K30E	14	48
EP20K60E	22	85
EP20K100E	32	130

---

(4) For more information about how to generate `.jbc` with uncompressed programming data, refer to **www.altera.com/mysupport**.

(5) The programming file targets one EP1S10 device.

(6) The programming file targets one EP1S25 device.

(7) The programming file targets one EP1S40 device.

Device	Typical Jam STAPL Byte-Code Data Size (KB)	
	Compressed	Uncompressed[4]
EP20K160E	56	194
EP20K200E	53	250
EP20K300E	78	347
EP20K400E	111	493
EP20K600E	170	713
EP20K1000E	254	1124
EP20K1500E	321	1509
EP2A15	107	549
EP2A25	163	788
EP2A40	257	1209
EP2A70	444	2181
EPM7032S	8	8
EPM7032AE	6	6
EPM7064S	13	13
EPM7064AE	8	8
EPM7128S, EPM7128A	5	24
EPM7128AE	4	12
EPM7128B	4	12
EPM7160S	10	28
EPM7192S	11	35
EPM7256S, EPM7256A	15	51
EPM7256AE	11	18
EPM7512AE	18	37
EPM9320, EPM9320A	21	57
EPM9400	21	71
EPM9480	22	85
EPM9560, EPM9560A	23	98
EPF10K10, EPF10K10A	12	15

---

[4] For more information about how to generate `.jbc` with uncompressed programming data, refer to **www.altera.com/mysupport**.

Device	Typical Jam STAPL Byte-Code Data Size (KB)	
	Compressed	Uncompressed[4]
EPF10K20	21	29
EPF10K30	33	47
EPF10K30A	36	51
EPF10K30E	36	59
EPF10K40	37	62
EPF10K50, EPF10K50V	50	78
EPF10K50E	52	98
EPF10K70	76	112
EPF10K100, EPF10K100A, EPF10K100B	95	149
EPF10K100E	102	167
EPF10K130E	140	230
EPF10K130V	136	199
EPF10K200E	205	345
EPF10K250A	235	413
EP20K100	128	244
EP20K200	249	475
EP20K400	619	1,180
EPC2	136	212
EPM240	12.4[8]	12.4
EPM570	11.4	19.6
EPM1270	16.9	31.9
EPM2210	24.7	49.3
MAX V	[9]	[9]
MAX 10	[9]	[9]

---

[4] For more information about how to generate `.jbc` with uncompressed programming data, refer to **www.altera.com/mysupport**.

[8] There is a minimum limit of 64 kilobits (Kb) for compressed arrays with the `.jbc` compiler. Programming data arrays that are smaller than 64 Kb (8 kilobytes (KB)) are not compressed. The EPM240 programming data array is below the limit, which means that the `.jbc` files are always uncompressed. A memory buffer is needed for decompression. For small embedded systems, it is more efficient to use small uncompressed arrays directly rather than to uncompress the arrays.

[9] The file size is design dependent. Refer to the generated `.jbc` file for the file size.

Jam STAP Byte-Code Player Size

Table 11: Jam STAPL Byte-Code Player Binary Size

Use the information in this table to estimate the binary size of the Jam STAPL Byte-Code Player

Build	Description	Size (KB)
16 bit	Pentium/486 using the MasterBlaster or ByteBlasterMV download cables	80
32 bit	Pentium/486 using the MasterBlaster or ByteBlasterMV download cables	85

Estimating Dynamic Memory Usage

Figure 14: Equation to Estimate Maximum Required DRAM

Use this equation to estimate the maximum amount of DRAM required by the Jam STAPL Byte-Code Player.

$$RAM\ Size = .jbc\ Size + \sum_{k=1}^{N} Data\ (Uncompressed\ Data\ Size)_k$$

The `.jbc` size is determined by a single-device or multi-device equation.

The amount of RAM used by the Jam STAPL Byte-Code Player is the total size of the `.jbc` and the sum of the data required for each targeted device. If the `.jbc` file is generated using compressed data, then some RAM is used by the player to uncompress and temporarily store the data.

If you use an uncompressed `.jbc`, the RAM size is equal to the uncompressed `.jbc` size.

Note: The memory requirements for the stack and heap are negligible in terms of the total amount of memory used by the Jam STAPL Byte-Code Player. The maximum depth of the stack is set by the `JBI_STACK_SIZE` parameter in `jbimain.c`.

Related Information

- **Estimating ROM Usage** on page 21
  Provides the equation to estimate the .jbc size.
- **Compressed and Uncompressed Data Size Constants** on page 23
  Lists the uncompressed data sizes.

Example of Calculating DRAM Required by Jam STAPL Byte-Code Player

To determine memory usage, first determine the amount of ROM required and then estimate the RAM usage.

This example uses a 16-bit Motorola 68000 processor to program EPM7128AE and EPM7064AE devices in an IEEE Std. 1149.1 JTAG chain using a compressed `.jbc`.

1. Use the multi-device equation to estimate the `.jbc` size.

Figure 15: Multi-Device Equation to Estimate `.jbc` Size

$$.jbc\ Size = Alg + \sum_{k=1}^{N} Data$$

- Because the `.jbc` file contains compressed data, use the compressed data file size constants to determine the data size. Refer to the related information.
- In this example, *Alg* is 21 KB and *Data* is the sum of EPM7064AE and EPM7128AE data sizes (8 KB + 4 KB = 12 KB).
- The the `.jbc` file size is 33 KB.

2. Estimate the Jam STAPL Byte-Code Player size—this example uses a Jam STAPL Byte-Code Player size of 62 KB because the Motorola 68000 processor is a 16 bit processor. Use the following equation to determine the amount of ROM required. In this example, the ROM size is 95 KB.

Figure 16: Equation to Estimate the Maximum Required ROM Size

$$ROM\ Size = .jbc\ Size + Jam\ STAPL\ Byte\text{-}Code\ Player\ Size$$

3. Estimate the RAM usage using the following equation. In this example, the `.jbc` size is 33 KB.

Figure 17: Equation to Estimate Maximum Required DRAM

$$RAM\ Size = .jbc\ Size + \sum_{k=1}^{N} Data\ (Uncompressed\ Data\ Size)_k$$

- Because the `.jbc` uses compressed data, add up the uncompressed data size for each device to find the total amount of RAM usage. Refer to the related information.
- The uncompressed data size constants for EPM7064AE and EPM7128AE are 8 KB and 12 KB, respectively.
- The total DRAM usage in this example is calculated as RAM Size = 33 KB + (8 KB + 12 KB) = 53 KB.

In general, `.jam` files use more RAM than ROM. This characteristic is desirable because RAM is cheaper. In addition, the overhead associated with easy upgrades becomes less of a factor when programming a large number of devices. In most applications, the importance of easy upgrades outweigh memory costs.

Related Information

Compressed and Uncompressed Data Size Constants on page 23
Lists the compressed data sizes.

Updating Devices Using Jam

To update a device in the field, download a new `.jbc` and run the Jam STAPL Byte-Code Player, in most cases, with the program action statement.

The main entry point for the Jam STAPL Byte-Code Player is `jbi_execute()`. This routine passes specific information to the player. When the player finishes, it returns an exit code and detailed error information for any run-time errors. The interface is defined by the routine's prototype definition in `jbimain.c`:

```
JBI_RETURN_TYPE jbi_execute
(
    PROGRAM_PTR program
    long program_size,
    char *workspace,
    long workspace_size,
    char *action,
    char **init_list,
    int reset_jtag
    long *error_address,
    int *exit_code,
    int *format_version
)
```

The code within `main()` in `jbistub.c` determines the variables that are passed to `jbi_execute()`. In most cases, this code is not applicable to an embedded environment. Therefore, you can remove this code and set up the `jbi_execute()` routine for the embedded environment.

Before calling the `jbi_execute` function, construct `init_list` with the correct arguments that correspond to the valid actions in `.jbc`, as specified in the JEDEC standard JESD71 specification. The `init_list` is a null-terminated array of pointers to strings.

An initialization list tells the Jam STAPL Byte-Code Player the types of functions to perform—for example, program and verify—and this list is passed to `jbi_execute()`. The initialization list must be passed in the correct manner. If an initialization list is not passed or the initialization list is invalid, the Jam STAPL Byte-Code Player simply checks the syntax of the `.jbc` and if there is no error, returns a successful exit code without performing the program function.

Example 6: Code to Set Up init_list for Performing Program and Verify Operation

Use this code to set up `init_list` that instructs the Jam STAPL Byte-Code Player to perform a program and verify operation.

```
char CONSTANT_AREA init_list[][] = "DO_PROGRAM=1", "DO_VERIFY=1";
```

The default code in the Jam STAPL Byte-Code Player sets `init_list` differently and is used to give instructions to the Jam STAPL Byte-Code Player from the command prompt.

The code in this example declares the `init_list` variable while setting it equal to the appropriate parameters. The `CONSTANT_AREA` identifier instructs the compiler to store the `init_list` array in the program memory.

After the Jam STAPL Byte-Code Player completes a task, the player returns a status code of type JBI_RETURN_TYPE or integer. A return value of "0" indicates a successful action. The `jbi_execute()` routine can return any of the exit codes as defined in the Jam STAPL Specification.

Related Information

Jam STAPL Player and quartus_jli Exit Codes on page 8

jbi_execute Parameters

Table 12: Parameters in the jbi_execute() Routine

You must pass the mandatory parameters for the Jam STAPL Byte-Code Player to run.

Parameter	Status	Description
`program`	Mandatory	A pointer to the `.jbc`. For most embedded systems, setting up this parameter is as easy as assigning an address to the pointer before calling `jbi_execute()`.
`program_size`	Mandatory	Amount of memory (in bytes) that the `.jbc` occupies.
`workspace`	Optional	A pointer to dynamic memory that can be used by the Jam STAPL Byte-Code Player to perform its necessary functions. The purpose of this parameter is to restrict the player memory usage to a predefined memory space. This memory must be allocated before calling `jbi_execute()`. If the maximum dynamic memory usage is not a concern, set this parameter to null, which allows the player to dynamically allocate the necessary memory to perform the specified action.
`workspace_size`	Optional	A scalar representing the amount of memory (in bytes) to which workspace points.
`action`	Mandatory	A pointer to a string (text that directs the Jam STAPL Byte-Code Player). Example actions are `PROGRAM` or `VERIFY`. In most cases, this parameter is set to the string `PROGRAM`. The text can be in upper or lower case because the player is not case-sensitive. The Jam STAPL Byte-Code Player supports all actions defined in the Jam STAPL Specification. Take note that the string must be null-terminated.
`init_list`	Optional	An array of pointers to strings. Use this parameter when applying Jam version 1.1 files, or when overriding optional sub-actions. Altera recommends using the STAPL-based `.jbc` with `init_list`. When you use a STAPL-based `.jbc`, `init_list` must be a null-terminated array of pointers to strings.

Parameter	Status	Description
`error_address`	—	A pointer to a long integer. If an error is encountered during execution, the Jam STAPL Byte-Code Player records the line of the `.jbc` where the error occurred.
`exit_code`	—	A pointer to a long integer. Returns a code if there is an error that applies to the syntax or structure of the `.jbc`. If this kind of error is encountered, the supporting vendor must be contacted with a detailed description of the circumstances in which the exit code was encountered.

Related Information

- **List of Supported .jam and .jbc Actions and Procedures** on page 4
- **Definitions of .jam and .jbc Action and Procedure Statements** on page 6

Running the Jam STAPL Byte-Code Player

Calling the Jam STAPL Byte-Code Player is like calling any other subroutine. In this case, the subroutine is given actions and a file name, and then it performs its function.

In some cases, you can perform in-field upgrades depending on whether the current device design is up-to-date. The JTAG USERCODE value is often used as an electronic "stamp" that indicates the device design revision. If the USERCODE is set to an older value, the embedded firmware updates the device.

The following *pseudocode* shows how you can call the Jam Byte-Code Player multiple times to update the target Altera device:

```
result = jbi_execute(jbc_file_pointer, jbc_file_size, 0, 0,\
"READ_USERCODE", 0, error_line, exit_code);
```

The Jam STAPL Byte-Code Player reads the JTAG USERCODE and exports it using the `jbi_export()` routine. The code then branches based on the result.

With Jam STAPL Byte-Code software support, updates to the supported Altera devices are as easy as adding a few lines of code.

Example 7: Switch Statement

You can use a switch statement, as shown in this example, to determine which device needs to be updated and which design revision you must use.

```
switch (USERCODE)
{
    case "0001":        /*Rev 1 is old - update to new Rev*/
        result = jbi_execute (rev3_file, file_size_3, 0, 0,\
        "PROGRAM", 0, error_line, exit_code);
    case "0002":        /*Rev 2 is old - update to new Rev*/
        result = jbi_excecute(rev3_file, file_size_3, 0, 0,\
        "PROGRAM", 0, error_line, exit_code);
    case "0003":
        ;               /*Do nothing - this is the current Rev*/
    default:            /*Issue warning and update to current Rev*/
        Warning - unexpected design revision;
```

```
                    /*Program device with newest Rev anyway*/
            result = jbi_execute(rev3_file, file_size_3, 0, 0,\
            "PROGRAM", 0, error_line, exit_code);
    }
```

Document Revision History

Date	Version	Changes
April 2017	2017.04.10	• Added optional .jam procedures in *Supported .jam or .jbc Actions and Optional Procedures for Each Action in Altera Devices*. • Added .jam procedure statement and description in *Definitions of .jam Procedure Statements*.
December 2016	2016.12.09	Updated the typical jam STAPL byte-code data size for MAX V and MAX 10 devices in Data Constants for Altera Devices Supporting the Jam Language (for ISP) table.
September 2014	2014.09.22	• Added information for MAX 10 devices. • Added the "do_epcs_unprotect" optional `.jam` procedure for serial configuration devices to disable EPCS protection mode. • Restructured and rewrote several sections for clarity and style update. • Updated template.
December 2010	5.0	• Changed chapter and topic titles ("Differences Between the Jam STAPL Players and quartus_jli" on page 2, "ASCII Text Files" on page 3, "Byte-Code Files" on page 3, "Generating Jam STAPL Files" on page 3, "Using the quartus_jli Command-Line Executable" on page 10, and "Embedded Jam STAPL Players" on page 16). • Updated all screenshots. • Updated several table and figure titles (minor text changes). • Added information for MAX V devices. • Corrected text errors in Figure 9. • Updated codes in "Step 2: Map the JTAG Signals to the Hardware Pins" and "Updating Devices Using Jam". • Updated equations for clarity. Involves changes in equations numbering throughout the document. • Corrected minor error in "Notes to Table 9:" on page 23. • Removed "Conclusion" chapter. • Major text edits throughout the document.
July 2010	4.0	Technical publication edits. Updated screen shots.
July 2009	3.0	Technical publication edits only. No technical content changes.

Date	Version	Changes
August 2008	2.1	- Added new paragraph: "Updating Devices Using Jam". - Updated Table 3. - Updated Table 1.
November 2007	2.0	- Updated "Introduction". - Added new sections: "Jam STAPL Players", "Jam STAPL Files", "Using the Jam STAPL for ISP via an Embedded Processor", "Embedded Jam Players", and "Updating Devices Using Jam".
December 2006	1.1	- Changed chapter title. - Updated "Introduction" section. - Updated "Differences Between Jam STAPL Player and quartus_jli Command-Line Executable". - Updated Figure 6, Figure 7, and Figure 8.