

This application note describes Altera's programming and configuration support using Jam™ Standard Test and Programming Language (STAPL) for in-system programming (ISP) using PCs or embedded processors.

Introduction

The Jam STAPL JEDEC standard JESD-71 is compatible with all current programmable logic devices (PLDs) that offer ISP using Joint Test Action Group (JTAG). The Jam STAPL provides a software-level, vendor-independent standard for in-system programming and configuration. To enhance the quality, flexibility, and life-cycle of their end products, designers use Jam STAPL to implement ISP. Regardless of the number of PLDs you must program or configure, Jam STAPL simplifies in-field upgrades and revolutionizes PLD programming and configuration.

Jam STAPL Players

The Jam STAPL Player is software that parses the descriptive information in Jam files and interprets it as data and algorithm that programs targeted PLDs. The Jam Player does not program a particular device architecture or vendor; it only reads and understands the syntax defined by the Jam file specification.

There are two types of Jam Players to accommodate the two types of Jam files that Altera supports:

- ASCII Jam STAPL Player for ASCII text files (**.jam**)
- Jam STAPL Byte-Code Player for Jam Byte-Code files (**.jbc**)

A new command-line executable, `quartus_jli`, is an alternative to the existing Jam STAPL Players to program and test Altera® devices with both **.jam** and **.jbc** using the command-line operation from the Quartus® II software. The `quartus_jli` command-line executable is available in the Quartus II software version 6.0 and later.

Differences Between the Jam STAPL Player and the `quartus_jli` Command-Line Executable

The Jam STAPL Player is an interpreter program that reads and executes a STAPL file. A single STAPL file can perform several functions, such as programming, configuring, verifying, erasing, and blank-checking a programmable device. The Jam STAPL Player has access to the IEEE 1149.1 signals that are used for all instructions based on the IEEE 1149.1 interface. In addition, the Jam STAPL Player is capable of processing the user-specified actions and procedures in a STAPL file.

The `quartus_jli` command-line executable has the same functionality as the Jam STAPL Player plus two additional capabilities. It provides command-line control of the Quartus II software from the UNIX or DOS prompt and supports all programming hardware available in the Quartus II software version 6.0 or later.

By default, the `quartus_jli` executable is located in the `bin` directory, which is created during the Quartus II software installation. The path to `quartus_jli` is:

```
<drive>:\<Quartus II system directory>\bin
```



You can download the Altera Jam STAPL Player from the Jam website at: [Altera Jam STAPL Software](#).

Table 1 lists the differences between the Jam STAPL Player and `quartus_jli` command-line executable in the Quartus II software.

Table 1. Differences Between the Jam STAPL Player and the `quartus_jli` Command-Line Executable

Features	Jam STAPL Player	<code>quartus_jli</code> Command-Line Executable
Download Cables	Supports only ByteBlaster™ II, ByteBlaster MV, and ByteBlaster parallel port download cables	All programming cables supported by the JTAG server such as USB-Blaster™, ByteBlaster II, ByteBlaster MV, ByteBlaster, MasterBlaster™, and EthernetBlaster
Porting of Source Code to Embedded Processor	Yes	No
Supported Platforms	<ul style="list-style-type: none"> ■ 16-bit and 32-bit embedded processors ■ 32-bit Windows ■ DOS ■ UNIX 	<ul style="list-style-type: none"> ■ 32-bit Windows ■ DOS ■ UNIX
Enable or disable procedure from the command-line syntax	<ul style="list-style-type: none"> ■ To enable the optional procedure, use the <code>-d<proc=1></code> option ■ To disable the recommended procedure, use the <code>-d<proc=0></code> option 	<ul style="list-style-type: none"> ■ To enable the optional procedure, use the <code>-e<proc></code> option ■ To disable the recommended procedure, use the <code>-d<proc></code> option

Jam STAPL Files

Altera supports two types of Jam files:

- JEDEC Jam STAPL format
- Jam version 1.1 (pre-JEDEC format)

This section describes both Jam file types.

ASCII Text Files (.jam)

The JEDEC Jam STAPL format uses the syntax specified by the JEDEC Standard JESD-71A specification.



Altera recommends using JEDEC Jam STAPL files for all new projects. In most cases, Jam files are used in tester environments.

Jam Byte-Code Files (.jbc)

JBC files are binary files that are compiled versions of Jam files. JBC files are compiled to a virtual processor architecture where the ASCII Jam commands are mapped to byte-code instructions compatible with the virtual processor. There are two types of JBC files:

- Jam STAPL Byte-Code (compiled version of JEDEC Jam STAPL file)
- Jam Byte-Code (compiled version of Jam version 1.1 file)



Altera recommends using Jam STAPL Byte-Code files in embedded applications because they use minimal memory.

Generating Jam Files

The MAX+PLUS® II and Quartus II software can generate both Jam and JBC file types. In addition, you can compile Jam files into JBC files using a stand-alone Jam Byte-Code compiler. The compiler produces a functionally equivalent JBC file.



You can download the Jam Byte-Code compiler from the Jam website at: [Altera Jam STAPL Software](#).

The MAX+PLUS II and Quartus II software tools support programming and configuration of multiple devices from single or multiple JBC files.

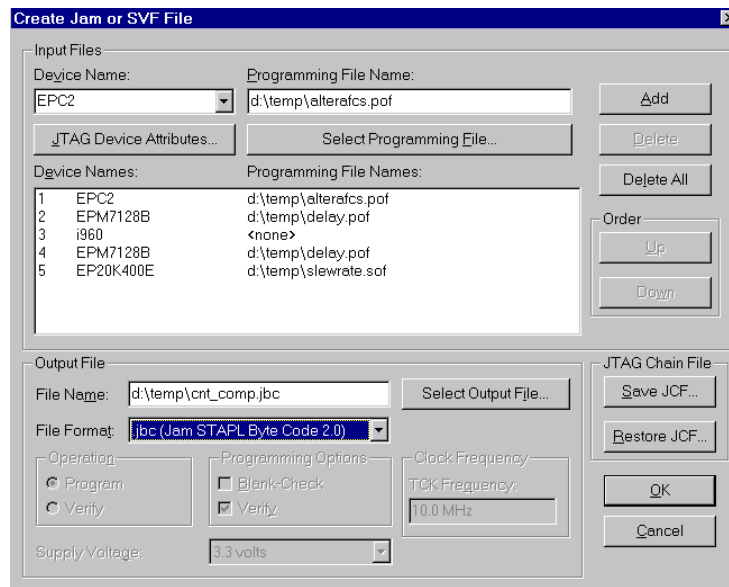
Figure 1 shows the dialog boxes that specify the multi-device JTAG device chain and sequence in the Quartus II software.

Figure 1. Multi-Device JTAG Chain's Name and Sequence in Programmer Window in the Quartus II Software

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank- Check	Examine	Security Bit
1. ..._0_b190\epc4(convert).pof	EPC4	01AE75F3	FFFFFFFF	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
2. <none>	I960	00000000	<none>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3. ...Quartus\and64\and64.pof	EPM7128AEF100	001E171D	FFFFFFFF	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4. ...P_CLAMP\Design1\top.pof	EPM7512B8256	0089CC87	FFFFFFFF	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 2 shows the dialog boxes that specify JBC file generation for a multi-device JTAG chain in the Quartus II software.

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



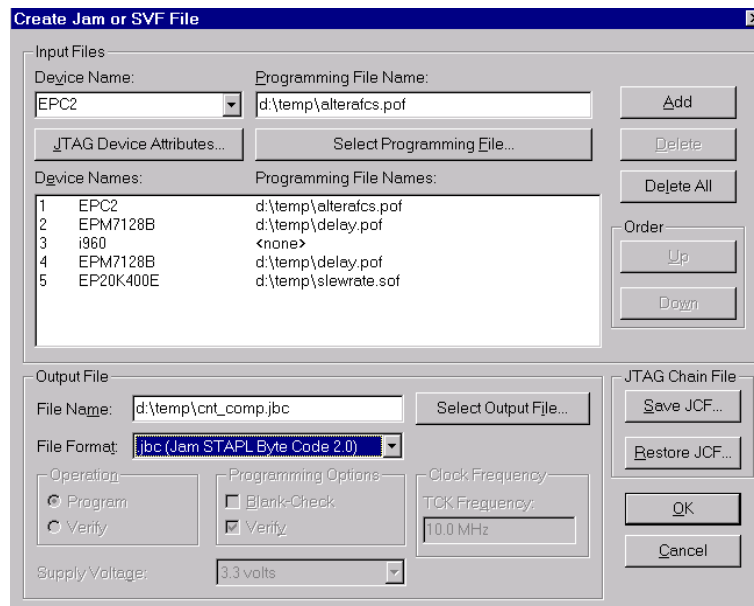
To generate JBC files using the Quartus II software, follow these steps:

1. On the Tools menu, click **Programmer**.
2. Click **Add File** and select the programming files for the respective devices.
3. On the File menu, point to **Create/Update** and click **Create Jam, SVF, or ISC File**.
4. From the **File Format** list, select a Jam STAPL Byte-Code file (Figure 2).
5. Click **OK**.

You can include both 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 3 shows the dialog boxes that specifies which JBC files are generated by the MAX+PLUS II software.

Figure 3. Generating a JBC File for a Multi-Device JTAG Chain in the MAX+PLUS II Software



To generate JBC files using the MAX+PLUS II software, follow these steps:

 To generate JBC files for APEX™ devices, follow the same procedure using SRAM Object Files (.sof)-generated by the Quartus II software.

1. In the MAX+PLUS II Programmer, on the File menu, click **Create Jam or SVF File**.
2. In the **Create Jam or SVF File** dialog box, specify the name and sequence of devices in the JTAG chain as well as the programming file associated with each device.
3. From the **File Format** drop-down list, select a Jam STAPL Byte-Code file.
4. Click **OK**.

List of Supported Jam and JBC Actions and Procedures

A STAPL file consists of two types of statements, “action” and “procedure”. An action statement contains a sequence of steps required to implement a complete operation. A procedure statement is one of the steps contained in an action statement.


 While an action statement can contain one or more procedure statements, it is also possible to have an action statement with no procedure statements. When an action statement contains one or more procedure statements, the action statements are called in a specified order to complete the associated operation. Some procedure statements in an action statement may be identified as “recommended” or “optional”, meaning that they may be included or excluded from the execution of an action statement as you desire.

Table 2 shows the supported action statement commands and the optional procedures that you can execute with each action for different Altera device families.

Table 2. Jam/JBC Actions and Procedures for Altera Devices (Part 1 of 2)

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	—
MAX II	Program	do_blank_check do_secure do_disable_isp_clamp do_bypass_cfm do_bypass_ufm do_real_time_isp do_read_usercode
	Blankcheck	do_disable_isp_clamp do_bypass_cfm do_bypass_ufm do_real_time_isp
	Verify	do_disable_isp_clamp do_bypass_cfm do_bypass_ufm do_real_time_isp do_read_usercode
	Erase	do_disable_isp_clamp do_bypass_cfm do_bypass_ufm do_real_time_isp
	Read_usercode	—
Stratix® Stratix II Cyclone® Cyclone II	Configure	do_read_usercode do_halt_on_chip_cc do_ignore_idcode_errors
	Read_usercode	—

Table 2. Jam/JBC Actions and Procedures for Altera Devices (Part 2 of 2)

Devices	Jam/JBC Action	Optional Procedures (Off by Default)
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
	Blankcheck	—
	Verify	—
	Erase	—
	Read_usercode	—

Table 3 provides description of each action and procedure.

Table 3. Definitions of Jam and JBC Action and Procedure Statements (Part 1 of 2)

Action/Procedure	Descriptions
Action	
Program	Programs the device.
Blankcheck	Checks the erased state of the device.
Verify	Verifies the entire device against the programming data in the Jam file.
Erase	Performs a bulk erase of the device.
Read_usercode	Returns the JTAG USERCODE register information from the device.
Configure	Configures the device.
Init_configuration	Specifies that the configuration device would configure the attached devices immediately after programming.
Procedure	
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.

Table 3. Definitions of Jam and JBC Action and Procedure Statements (Part 2 of 2)

Action/Procedure	Descriptions
do_bypass_cfm	When enabled, the procedure performs the specified action on the user flash memory only.
do_bypass_ufm	When enabled, the procedure performs the specified action on the CFM only.
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 FPGA 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 CFI flash memory that is attached to the parallel flash loader (PFL) of the MAX II device.

Exit Codes

Exit codes are the integer values that are used to indicate the result of execution of a STAPL file. An exit code value of zero indicates success, while a non-zero value indicates failure and identifies the general type of failure that occurred. An example of a successful execution with an exit code value of zero is shown in [Figure 4 on page 9](#).

Both Jam Player and the quartus_jli command-line executable can return any one of the exit codes in [Table 4](#), as defined in the Jam STAPL Specification (JESD-71).

Table 4. Exit Codes (Part 1 of 2)

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

Table 4. Exit Codes (Part 2 of 2)

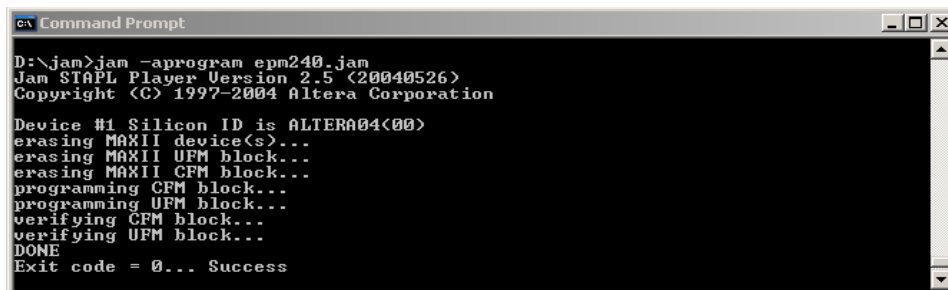
Exit Code	Description
16	Exiting ISP failure
17	Performing system test failure

Using the Jam STAPL Player

Commands for executing the Jam STAPL Player or performing other tasks are not case-sensitive and you can write the option flags in any sequence. An action is specified using the `-a` option, as shown:

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

This command programs the entire device using the specified Jam file (Figure 4).

Figure 4. Programming an EPM240 Device Using the Jam STAPL Player


```

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

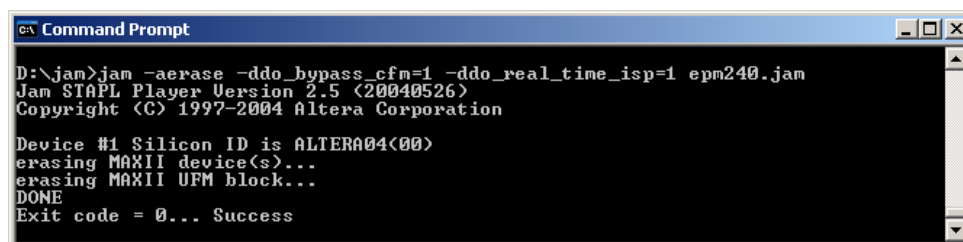
Device #1 Silicon ID is ALTERA04(00)
erasing MAXII device(s)...
erasing MAXII UFM block...
erasing MAXII CFM block...
programming CFM block...
programming UFM block...
verifying CFM block...
verifying UFM block...
DONE
Exit code = 0... Success

```

You can execute the optional procedures associated with each action using the `-d` option, as shown:

```
jam -aerase -ddo_bypass_cfm=1 -ddo_real_time_isp=1 <file name>.jam
```

This command erases the MAX II UFM block only if you enabled the real-time ISP (Figure 5).

Figure 5. Erasing Only the UFM Block of the EPM240 with the Real-Time ISP Feature Enabled


```

C:\> Command Prompt
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

```



The Jam STAPL Byte-Code Player uses the same command to run `.jbc` except for the executable name.

Using the Command-Line Executable in the Quartus II Software

The command-line executable `quartus_jli` supports all Altera download cables; for example, ByteBlaster, ByteBlasterMV, ByteBlaster II, USB-Blaster, MasterBlaster, and Ethernet Blaster. The command-line text is not case-sensitive and you can write the option in any sequence.

Table 5 shows the supported command-line executable options.

Table 5. Command-line Executable Options for Command-Line Executable `quartus_jli`

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 file 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.

The following examples show the syntax to run the `quartus_jli` command-line executable at the command prompt:

```
quartus_jli -n
```


To display the list of available download cables on a machine, use the following syntax (Figure 6). For more information about download cables, refer to Table 1 on page 2.

Figure 6. Display of the Available Download Cables

```

c:\altera\quartus60_h176\bin>quartus_jli -n
Info: *****
Info: Running Quartus II Jam Tools
Info: Version 6.0 Build 176 04/19/2006 SJ Full Version
Info: Copyright (C) 1991-2006 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 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: Thu Oct 12 13:59:17 2006
Info: Command: quartus_jli -n
1) ByteBlaster [LPT1]
2) USB-Blaster [USB-0]
Info: Quartus II Jam Tools was successful. 0 errors, 0 warnings
Info: Processing ended: Thu Oct 12 13:59:18 2006
Info: Elapsed time: 00:00:01

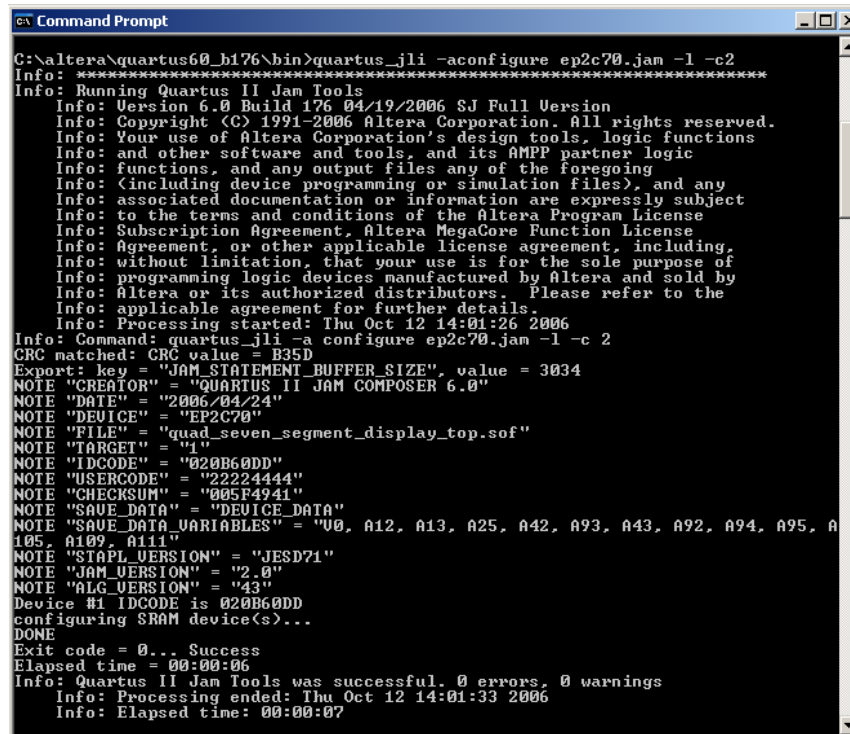
```

 In Figure 6, numbers 1) and 2) refer to < cable index >.

To display the header file information in a Jam file when executing an action statement, use the following syntax (refer to [Figure 7](#)):

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

Figure 7. Header File Information of a Jam File when Executing an Action Statement



```

C:\altera\quartus60_b176\bin>quartus_jli -aconfigure ep2c70.jam -l -c2
Info: *****
Info: Running Quartus II Jam Tools
Info: Version 6.0 Build 176 04/19/2006 SJ Full Version
Info: Copyright (C) 1991-2006 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 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: Thu Oct 12 14:01:26 2006
Info: Command: quartus_jli -a configure ep2c70.jam -l -c 2
CRC matched: CRC value = B35D
Export: key = "JAM_STATEMENT_BUFFER_SIZE", value = 3034
NOTE "CREATOR" = "QUARTUS II JAM COMPOSER 6.0"
NOTE "DATE" = "2006/04/24"
NOTE "DEVICE" = "EP2C70"
NOTE "FILE" = "quad_seven_segment_display_top.sof"
NOTE "TARGET" = "1"
NOTE "IDCODE" = "020B60DD"
NOTE "USERCODE" = "22224444"
NOTE "CHECKSUM" = "005F4941"
NOTE "SAVE_DATA" = "DEVICE_DATA"
NOTE "SAVE_DATA_VARIABLES" = "U0, 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" = "43"
Device #1 IDCODE is 020B60DD
configuring SRAM device(s)...
DONE
Exit code = 0... Success
Elapsed time = 00:00:06
Info: Quartus II Jam Tools was successful. 0 errors, 0 warnings
Info: Processing ended: Thu Oct 12 14:01:33 2006
Info: Elapsed time: 00:00:07

```

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

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

To enable or disable any procedure associated with an action statement, use:

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

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

For more information about a specific option, use:

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

To configure and return the JTAG USERCODE of an FPGA device using the second download cable on the machine using the specified Jam file (refer to [Figure 8](#)), use:

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

Figure 8. Configuring and Reading the JTAG USERCODE of the EP2C70 Device Using the USB Blaster Cable

```

C:\altera\quartus60_b176\bin>quartus_jli -aconfigure -edo_read_usercode -c2 ep2c70.jam
Info: *****
Info: Running Quartus II Jam Tools
Info: Version 6.0 Build 176 04/19/2006 SJ Full Version
Info: Copyright (C) 1991-2006 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 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: Thu Oct 12 13:55:12 2006
Info: Command: quartus_jli -a configure -e do_read_usercode -c 2 ep2c70.jam
Device #1 IDCODE is 020B60DD
configuring SRAM device(s)...
Device #1 USERCODE code is 22224444
DONE
Exit code = 0... Success
Info: Quartus II Jam Tools was successful. 0 errors, 0 warnings
Info: Processing ended: Thu Oct 12 13:55:29 2006
Info: Elapsed time: 00:00:17

```

 The syntax to run `.jbc` using the command-line executable is similar to running `.jam`.

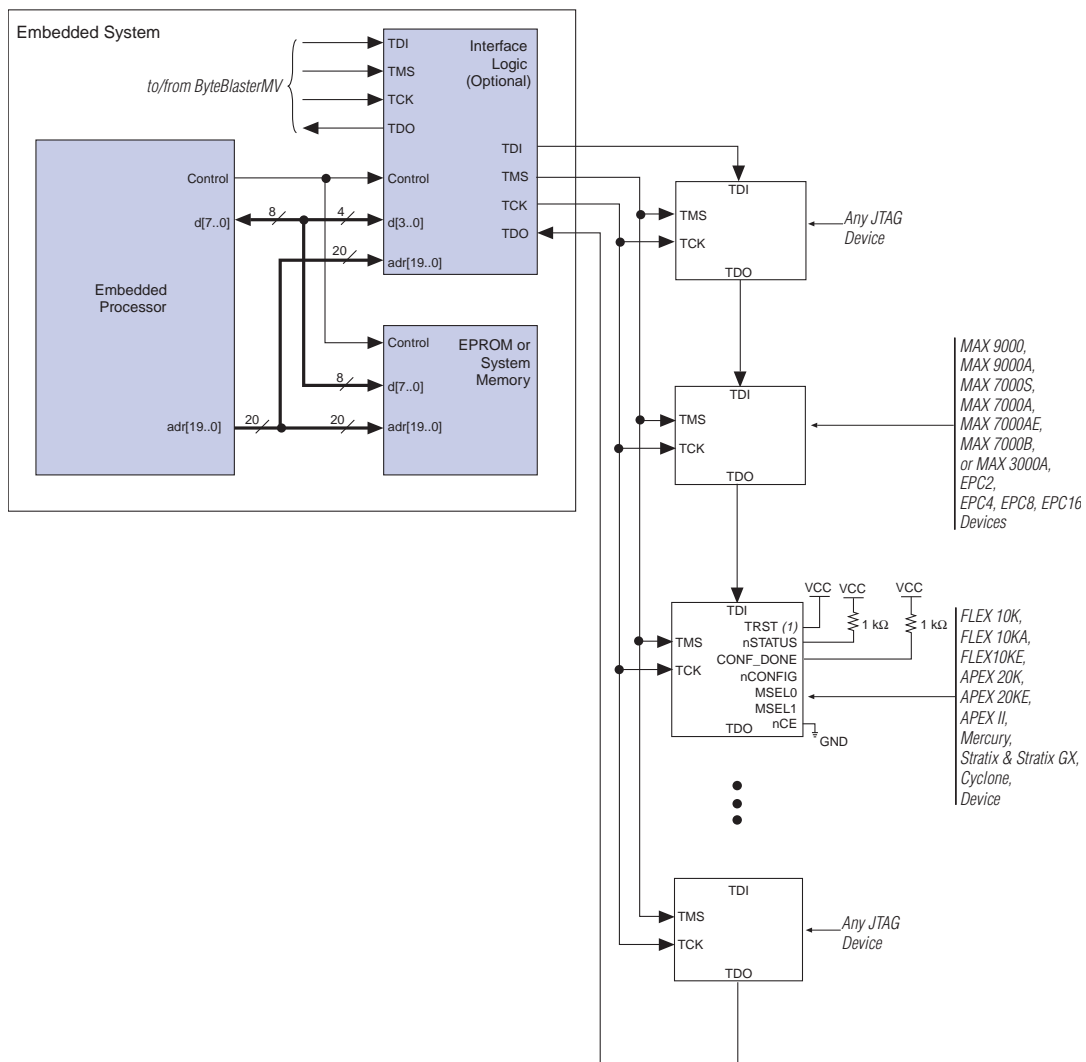
Using the Jam STAPL for ISP with an Embedded Processor

Embedded systems comprise both hardware and software components. When designing an embedded system, the first step is to lay out the PCB. The second step is to develop the firmware that manages the board's functionality.

Connecting the JTAG Chain to the Embedded Processor

There are two ways to connect the JTAG chain to the embedded processor. The most straightforward method is to connect the embedded processor directly to the JTAG chain. In this method, four of the processor pins are dedicated to the JTAG interface, thereby saving board space but reducing the number of available embedded processor pins.

[Figure 9](#) shows the second method, which is to connect the JTAG chain to an existing bus using an interface PLD. In this method, the JTAG chain becomes an address on the existing bus. The processor then reads from, or writes to, the address representing the JTAG chain.

Figure 9. Connecting the JTAG Chain to the Embedded System

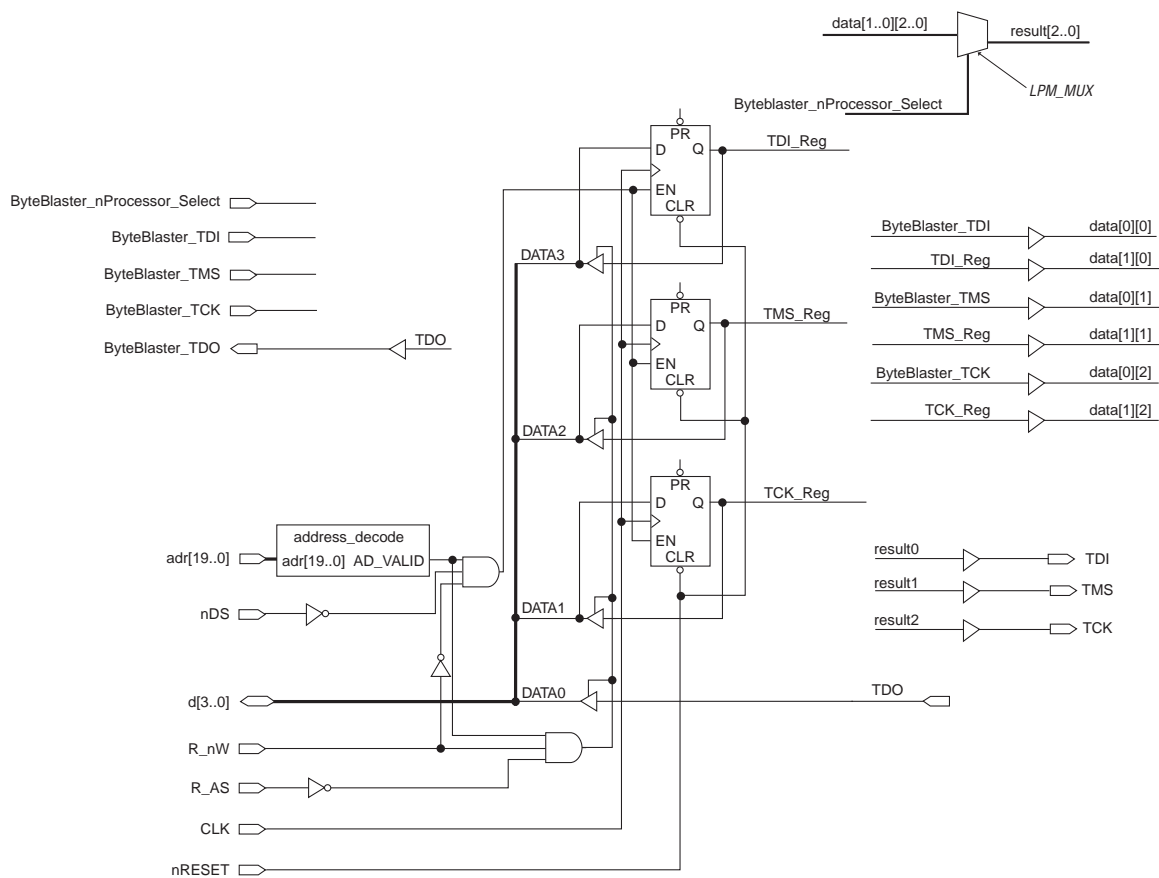
Both JTAG connection methods must include space for the MasterBlaster or ByteBlasterMV header connection. The header is useful during prototyping because it allows you to quickly verify or modify the PLD's contents. During production, you can remove the header to save cost.

Example Interface PLD Design

Figure 10 shows an example design schematic of an interface PLD. Note that if you use this example:

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

This design example is for reference only. All of the inputs except data[3..0] are optional and are included only to show how an interface PLD can act as an address on an embedded data bus.

Figure 10. Interface Logic Design Example

In [Figure 10](#), the embedded processor asserts the JTAG chain's address. You can set the `R_nW` and `R_AS` signals to notify the interface PLD when the processor wants to access the chain.

A write involves connecting the data path `data[3..0]` to the JTAG outputs of the PLD using the three D registers that are clocked by the system clock (`CLK`). This clock can be the same clock that the processor uses. Likewise, a read involves enabling the tri-state buffers and letting the `TDO` signal flow back to the processor.

The 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, allowing the software to check the valid states of the registers in the interface PLD. In addition, multiplexer logic is included 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 must verify the programming and configuration.



This interface PLD design is available as a MAX+PLUS II Graphic Design File (.gdf) on the Altera FTP site at <ftp://ftp.altera.com/pub/misc/intpld.zip>.

Board Layout

The following elements are important when laying out a board that programs or configures using the IEEE Std. 1149.1 JTAG chain:

- Treat the TCK signal trace as a clock tree
- Use a pull-down resistor on the TCK signal
- Make the JTAG signal traces as short as possible
- Add external resistors to pull the outputs to a defined logic level

The TCK Signal Trace Protection and Integrity

The TCK signal is the clock for the entire JTAG chain of devices. These devices are edge-triggered on the TCK signal, so it is imperative that TCK is protected from high-frequency noise and has good signal integrity. Ensure that the signal meets the rise time (t_R) and fall time (t_F) parameters in the appropriate device family data sheet. The signal may also need termination to prevent overshoot, undershoot, or ringing. This step is often overlooked because this signal is software-generated and originates at a processor general-purpose I/O pin.

Pull-Down Resistors on the TCK Signal

The TCK signal must be held 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 a JTAG boundary scan test (BST) state, which may cause conflicts on the board. A typical resistor value is 1 k Ω .

JTAG Signal Traces

Short JTAG signal traces help eliminate noise and drive-strength issues. Pay special attention to the TCK and TMS pins. Because TCK and TMS are connected to every device in the JTAG chain, these traces will see higher loading than the TDI or TDO signals. Depending on the length and loading of the JTAG chain, additional buffering may be required to ensure that the signals propagate to and from the processor with integrity.

External Resistors

During programming or configuration, you must add external resistors to the output pins to pull the outputs to a defined logic level. Output pins tri-state during programming or configuration. Also, on MAX 7000, FLEX 10K®, APEX™ 20K, and all configuration devices, the pins will be pulled up by a weak internal resistor (for example, 50 k Ω). However, not all Altera devices have weak pull-up resistors during in-system programming or in-circuit reconfiguration. Refer to the appropriate device family data sheet to learn which devices have weak pull-up resistors.



Altera recommends that outputs driving sensitive input pins be tied to the appropriate level by an external resistor, on the order of 1 k Ω

Each preceding board layout element may require further analysis, especially signal integrity. In some cases, you may need to analyze the loading and layout of the JTAG chain to determine whether to use discrete buffers or a termination technique.

For more information, refer to *AN 100: In-System Programmability Guidelines*.

Embedded Jam Players

The embedded Jam Player is able to read Jam files that conform to the standard JEDEC file format. The embedded Jam Player is compatible with legacy Jam files that use version 1.1 syntax. Both the ASCII Jam STAPL Player and the Jam STAPL Byte-Code Player are backward compatible; they can play version 1.1 files and Jam STAPL files. The following information describes porting the Jam STAPL Byte-Code Player.

The Jam STAPL Byte-Code Player

The Jam STAPL Byte-Code Player is coded in the C programming language for 16- and 32-bit processors. Some 8-bit processors are also supported by a specific subset of source code available on the Jam website at: [Altera Jam STAPL Software](#).

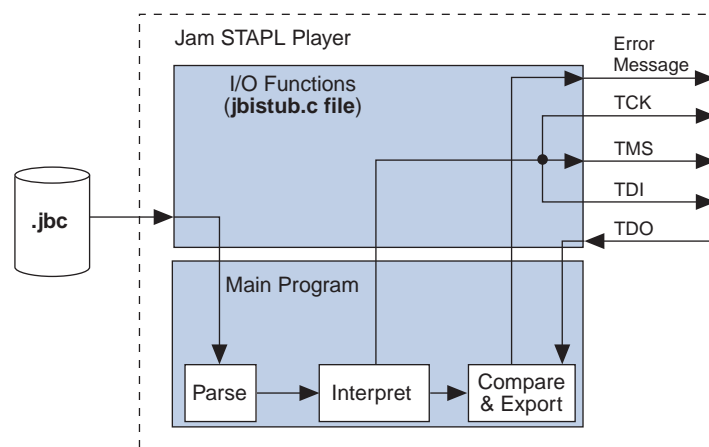
For more information about Altera's support for 8-bit processors, refer to *AN 111: Embedded Programming Using the 8051 & Jam Byte-Code*.

The 16- and 32-bit source code is divided into two categories:

- Platform-specific code that handles I/O functions and applies to specific hardware (`jbistub.c`)
- Generic code that performs the Player's internal functions (all other C files)

Figure 11 shows the organization of the source code files by function. Keeping the platform-specific code inside the `jbistub.c` file simplifies the process of porting the Jam STAPL Byte-Code Player to a particular processor.

Figure 11. Jam STAPL Byte-Code Player Source Code Structure



Porting the Jam STAPL Byte-Code Player

The default configuration of `jbistub.c` includes code for DOS, 32-bit Windows, and UNIX so that the source code is compiled and evaluated for the correct functionality and debugging of these operating systems. For the embedded environment, this code is removed using a single preprocessor `#define` statement. In addition, porting the code involves making minor changes to specific parts of the code in `jbistub.c`.

To port the Jam Player, you need to customize several functions in `jbistub.c`, which are shown in [Table 6](#).

Table 6. Functions Requiring Customization

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

To ensure that you have customized all the necessary code, follow these steps:

1. Set the preprocessor statements to exclude extraneous code. Refer to [“Step 1: Set the Preprocessor Statements to Exclude Extraneous Code”](#) on page 17.
2. Map the JTAG signals to the hardware pins. Refer to [“Step 2: Map the JTAG Signals to the Hardware Pins”](#) on page 17.
3. Handle text messages from `jbi_export()`. Refer to [“Step 3: Handle Text Messages from jbi_export\(\)”](#) on page 19.
4. Customize delay calibration. Refer to [“Step 4: Customize Delay Calibration”](#) on page 19.

Step 1: Set the Preprocessor Statements to Exclude Extraneous Code

At the top of `jbistub.c`, use the following code to change the default `PORT` parameter to `EMBEDDED` to eliminate all DOS, Windows, and UNIX source code and included libraries.

```
#define PORT EMBEDDED
```

Step 2: Map the JTAG Signals to the Hardware Pins

The `jbi_jtag_io()` function contains the code that sends and receives the binary programming data. Each of the four JTAG signals must be remapped to the embedded processor's pins. By default, the source code writes to the PC's parallel port. [Figure 12](#) shows how the `jbi_jtag_io()` signal maps the JTAG pins to the PC parallel port registers.

Figure 12. Default PC Parallel Port Signal Mapping (Note 1)

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

Note to Figure 12:

(1) The PC parallel port hardware inverts the most significant bit, TDO.

The mapping is shown in the following `jbi_jtag_io()` source code:

```
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);
}
```

In the previous code, the PC parallel port inverts the actual value of TDO. The `jbi_jtag_io()` source code inverts it again to retrieve the original data. The line that inverts the TDO value is:

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

If the target processor does not invert TDO, the code looks like:

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

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

```
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 the equivalent functions.

Step 3: Handle Text Messages from `jbi_export()`

The `jbi_export()` function sends text messages to `stdio`, using the `printf()` function. 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 Player. The function passes text (in the form of a string) and numbers (in the form of a decimal integer).



For definitions of these terms, refer to *AN 39: IEEE 1149.1 (JTAG) Boundary-Scan Testing in Altera Devices*.

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

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 hard-coded as 1,000 loops per millisecond and represented as:

```
one_ms_delay = 1000
```

If this parameter is known, it must be adjusted accordingly. If it is not known, you can use code similar to that for Windows and DOS platforms. Code is included for these platforms that count the number of clock cycles that run in the time it takes to execute a single loop. This code is sampled over multiple tests and averaged to produce an accurate delay result upon. The advantage to this approach is that calibration can vary based on the speed of the host processor.

Once 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 MAX II and MAX devices must comply with the values given in the respective device family data sheet for the JTAG timing parameters and values.

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 if the processor is very powerful and can generate TCK at a rate faster than 10 MHz.



Other than `jbistub.c`, Altera strongly recommends keeping the source code in other files at their default state. Altering the source code in these files results in unpredictable Jam Player operation.

Jam STAPL Byte-Code Player Memory Usage

The Jam STAPL Byte-Code Player uses memory in a predictable manner. This section describes how to estimate both ROM and RAM memory usage.

Estimating ROM Usage

Use the following equation to estimate the maximum amount of ROM required to store the Jam Player and JBC file:

$$\text{ROM Size} = \text{JBC File Size} + \text{Jam Player Size}$$

The JBC file size can be separated into two categories:

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

Use [Equation 1](#) to estimate the JBC file size.

Equation 1.

$$\text{JBC File Size} = Alg + \sum_{k=1}^N \text{Data}$$

where:

Alg = Space used by algorithm

Data = Space used by the compressed programming data

k = Index representing the device being targeted

N = Number of target devices in the chain

This equation provides a JBC file size estimate that may vary by $\pm 10\%$, depending on device utilization. When device utilization is low, JBC file sizes tend to be smaller because the compression algorithm used to minimize file size is more likely to find repetitive data.

The 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 file size (due to the data component) is linear.

[Table 7](#) shows algorithm file size constants when targeting a single device family.

Table 7. Algorithm File Size Constants Targeting a Single Altera Device Family (Part 1 of 2)

Device	Typical JBC File Algorithm Size (KBytes)
Stratix	15
Stratix GX	15
Cyclone	15
APEX II	15
Mercury	15
APEX 20K	14
APEX 20KE	15
FLEX 10K	15
FLEX 10KE	15
FLEX 10KA	15
FLEX 10KB	15
EPC 16	24

Table 7. Algorithm File Size Constants Targeting a Single Altera Device Family (Part 2 of 2)

Device	Typical JBC File Algorithm Size (KBytes)
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

Table 8 shows algorithm file size constants for possible combinations of Altera device families that support the Jam language.

Table 8. Algorithm File Size Constants Targeting Multiple Altera Device Families

Devices	Typical JBC File Algorithm Size (KBytes)
FLEX 10K, MAX 7000A, MAX 7000S, MAX 7000AE (1)	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

Note to Table 8:

(1) When configuring FLEX or APEX devices and programming MAX devices, the FLEX or APEX algorithm adds negligible memory.

Table 9 shows the data size constants for Altera devices that support the Jam language for ISP.

Table 9. Data Constants for Altera Devices Supporting the Jam Language (for ISP) (Note 2), (3), (4), (5), (6) (Part 1 of 3)

Device	Typical Jam STAPL Byte-Code Data Size (KBytes)	
	Compressed	Uncompressed (1)
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

Table 9. Data Constants for Altera Devices Supporting the Jam Language (for ISP) *(Note 2), (3), (4), (5), (6)* (Part 2 of 3)

Device	Typical Jam STAPL Byte-Code Data Size (KBytes)	
	Compressed	Uncompressed <i>(1)</i>
EP1C12	100	294
EP1C20	162	449
EPC4 <i>(2), (5)</i>	242	370
EPC8 <i>(2), (5)</i>	242	370
EPC8 <i>(3), (5)</i>	547	822
EPC16 <i>(2), (5)</i>	242	370
EPC16 <i>(4), (5)</i>	827	1344
EP1SGX25	243	992
EP1SGX40	397	1561
EP1M120	30	167
EP1M350	76	553
EP20K30E	14	48
EP20K60E	22	85
EP20K100E	32	130
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

Table 9. Data Constants for Altera Devices Supporting the Jam Language (for ISP) (Note 2), (3), (4), (5), (6) (Part 3 of 3)

Device	Typical Jam STAPL Byte-Code Data Size (KBytes)	
	Compressed	Uncompressed (1)
EPM9320, EPM9320A	21	57
EPM9400	21	71
EPM9480	22	85
EPM9560, EPM9560A	23	98
EPF10K10, EPF10K10A	12	15
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 (6)	12.4 (6)
EPM570	11.4	19.6
EPM1270	16.9	31.9
EPM2210	24.7	49.3

Notes to Table 9:

- (1) For more information about how to generate JBC files with uncompressed programming data, contact Altera Applications Technical Support at: www.altera.com/mysupport.
- (2) The programming file targets one EP1S10 device.
- (3) The programming file targets one EP1S25 device.
- (4) The programming file targets one EP1S40 device.
- (5) The enhanced configuration device (EPC) data sizes use a compressed programmer object file (.pof) file.
- (6) There is a minimum limit of 64 Kbits for compressed arrays with the JBC compiler. Programming data arrays smaller than 64 Kbits (8 KBytes) are not compressed. The EPM240 programming data array is below the limit, which means the JBC files are always uncompressed. A memory buffer is needed for decompression, and for small embedded systems, it is more efficient to use small uncompressed arrays directly rather than to uncompress the arrays.

Using the information in [Table 10](#), estimate the Jam Player size.

Table 10. Jam STAPL Byte-Code Player Binary Sizes

Build	Description	Size (KBytes)
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

Use [Equation 2](#) to estimate the maximum amount of DRAM required by the Jam Player.

Equation 2.

$$\text{RAM Size} = \text{JBC File Size} + \sum_{k=1}^N \text{Data (Uncompressed Data Size)}_k$$

The JBC file size is determined by a single- or multi-device equation (refer to [“Estimating ROM Usage” on page 20](#)).

The amount of RAM used by the Jam Player is the size of the JBC file plus the sum of the data required for each device that is targeted. If the JBC file is generated using compressed data, then some RAM is used by the Player to uncompress the data and temporarily store it. The uncompressed data sizes are shown in [Table 9 on page 21](#). If an uncompressed JBC file is used, use [Equation 3](#).

Equation 3.

$$\text{RAM Size} = \text{JBC File Size}$$



The memory requirements for the stack and heap are negligible with respect to 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`.

Estimating Memory Example

The following example uses a 16-bit Motorola 68000 processor to program EPM7128AE and EPM7064AE devices in an IEEE Std. 1149.1 JTAG chain using a JBC file that uses compressed data. To determine memory usage, first determine the amount of ROM required, then estimate the RAM usage. Use the following steps to calculate the amount of DRAM required by the Jam Byte-Code Player:

1. Determine the JBC file size. Use the multi-device [Equation 4](#) to estimate the JBC file size. Because JBC files use compressed data, use the compressed data file size information listed in [Table 9 on page 21](#), to determine the data size.

Equation 4.

$$\text{JBC File Size} = Alg + \sum_{k=1}^N \text{Data}$$

where:

$$Alg = 21 \text{ KBytes}$$

$$\text{Data} = \text{EPM7064AE Data} + \text{EPM7128AE Data} = 8 + 4 = 12 \text{ KBytes}$$

Thus, the JBC file size equals 33 KBytes.

- Estimate the JBC Player size. This example uses a JBC Player size of 62 KBytes because the 68000 processor is a 16-bit processor. Use [Equation 5](#) to determine the amount of ROM needed.

Equation 5.

$$\text{ROM Size} = \text{JBC File Size} + \text{Jam Player Size}$$

$$\text{ROM Size} = 95 \text{ KBytes.}$$

- Estimate the RAM usage with [Equation 6](#).

Equation 6.

$$\text{RAM Size} = 33 \text{ KBytes} + \sum_{k=1}^N \text{Data (Uncompressed Data Size)}_k$$

Because the JBC file uses compressed data, the uncompressed data size for each device must be summed to find the total amount of RAM used. The uncompressed data size constants are:

- EPM7064AE = 8 KBytes
- EPM7128AE = 12 KBytes

Calculate the total DRAM usage:

- RAM Size = 33 KBytes + (8 KBytes + 12 KBytes) = 53 KBytes

In general, Jam files use more RAM than ROM, which is desirable because RAM is cheaper and the overhead associated with easy upgrades becomes less of a factor as a larger number of devices are programmed. In most applications, easy upgrades outweigh memory costs.

Updating Devices Using Jam

Updating a device in the field means downloading a new JBC file and running the Jam STAPL Byte-Code Player with what, in most cases, is the “program” action statement.

The main entry point for the 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.\

```

JBI_RETURN_TYPE jbi_execute
(
    PROGRAM_PTR program
    long program_size,
    char *workspace,
    long workspace_size,
    *action,
    char **init_list,
    long *error_line,
    int *exit_code
)

```

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 you can set up the `jbi_execute()` routine for the embedded environment.

Prior to calling the `jbi_execute` function, you need to construct `init_list` with the correct arguments that correspond to the valid actions in `.jbc`, as specified in the JEDEC standard JESD-71 specification. The `init_list` is a null terminated array of pointers to strings.

An initialization list tells the Jam Player the types of functions to perform—for example, program and verify—and is passed to `jbi_execute()`. The initialization list must be passed in the correct manner. If an invalid initialization list or no initialization list is passed, the Jam Player simply checks `.jbc`. If the syntax check passes, the Jam Player issues a successful exit code without performing the program function.

For example, you can use the following code to set up `init_list` to instruct the Jam Player to perform a program and verify operation:

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

This code declares the `init_list` variable while setting it equal to the appropriate parameters. `CONSTANT_AREA` is the identifier that instructs the compiler to store `init_list` in the program memory. The default code sets `init_list` differently and is used with a terminal program to give instructions to the Jam Player using a command prompt.

Once the Jam Player has completed a task, the Player returns a status code of type `JBI_RETURN_TYPE` or integer. This value indicates whether the action was successful (returns "0"). `jbi_execute()` can return any of the exit codes in [Table 4 on page 8](#), as defined in the Jam Standard Test and Programming Language Specification.

[Table 11](#) describes each parameter in the `jbi_execute()` routine.

Table 11. Parameters in the `jbi_execute()` Routine (*Note 1*)

Parameter	Status	Description
<code>program</code>	Mandatory	A pointer to the JBC file. For most embedded systems, setting up this parameter is as easy as assigning an address to the pointer before calling <code>jbi_execute()</code> .
<code>program_size</code>	Mandatory	Amount of memory (in bytes) that the JBC file occupies.
<code>workspace</code>	Optional	A pointer to dynamic memory that can be used by the JBC Player to perform its necessary functions. The purpose of this parameter is to restrict Player memory usage to a predefined memory space. This memory must be allocated before calling <code>jbi_execute()</code> . 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.
<code>workspace_size</code>	Optional	A scalar representing the amount of memory (in bytes) to which <code>workspace</code> points.
<code>action</code>	Mandatory	A pointer to a string (text that directs the Player). Example actions are <code>PROGRAM</code> or <code>VERIFY</code> . In most cases, this parameter is set to the string <code>PROGRAM</code> . The Player is not case-sensitive, so the text can be either upper or lower case. The Player supports all actions defined in the Jam Standard Test and Programming Language Specification. See Table 10 on page 24 . Note that the string must be null terminated.
<code>init_list</code>	Optional	An array of pointers to strings. This parameter is used when applying Jam version 1.1 files, or when overriding optional sub-actions. Altera recommends that you use the STAPL-based <code>.jbc</code> with <code>init_list</code> . When using STAPL-based <code>.jbc</code> files, <code>init_list</code> must be a null terminated array of pointers to strings.
<code>error_line</code>	—	A pointer to a long integer. If an error is encountered during execution, the Player records the line of the JBC file where the error occurred.
<code>exit_code</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 file. 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.

Note to Table 11:

- (1) Mandatory parameters must be passed for the Player to run.

Running the Jam STAPL Byte-Code Player

Calling the Jam STAPL Byte-Code Player is like calling any other sub-routine. In this case, the sub-routine 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 is often used as an electronic “stamp” that indicates the PLD design revision. If the USERCODE is set to an older value, the embedded firmware updates the device. The following pseudocode shows how the Jam Byte-Code Player can be called multiple times to update the target PLD:

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

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

The following shows an example code for using the Jam Player.

```
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);
}
```

You can use a switch statement to determine which device needs to be updated and which design revision you must use. With Jam STAPL Byte-Code software support, PLD updates are as easy as adding a few lines of code.

Conclusion

Using Jam STAPL provides a simple way to benefit from ISP. This application note provides you with a guideline for the Jam STAPL Player and the `quartus_jli` command-line executable. Jam meets all the necessary embedded system requirements, such as small file sizes, ease of use, and platform independence. In-field upgrades are simplified by confining updates to the Jam STAPL Byte-Code file.

Referenced Documents

- [AN 39: IEEE 1149.1 \(JTAG\) Boundary-Scan Testing in Altera Devices](#)
- [AN 100: In-System Programmability Guidelines](#)
- [AN 111: Embedded Programming Using the 8051 & Jam Byte-Code](#)

Document Revision History

Table 12 shows the revision history for this application note.

Table 12. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
July 2009, v3.0	<ul style="list-style-type: none"> ■ Technical publication edits only. No technical content changes. 	—
August 2008 v2.1	<ul style="list-style-type: none"> ■ Added new paragraph: “Updating Devices Using Jam”. ■ Updated Table 3. ■ Updated Table 1. 	—
November 2007 v2.0	<ul style="list-style-type: none"> ■ 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 v1.1	<ul style="list-style-type: none"> ■ 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. 	—



101 Innovation Drive
San Jose, CA 95134
www.altera.com
Technical Support
www.altera.com/support

Copyright © 2009 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.



I.S. EN ISO 9001