14. Using Jam STAPL for ISP via an Embedded Processor

MII51015-1.8

Introduction

Advances in programmable logic devices (PLDs) have enabled the innovative in-system programmability (ISP) feature. The Jam™ Standard Test and Programming Language (STAPL), JEDEC standard JESD-71, is compatible with all current PLDs that offer ISP via Joint Test Action Group (JTAG), providing a software-level, vendor-independent standard for in-system programming and configuration. Designers who use Jam STAPL to implement ISP enhance the quality, flexibility, and life-cycle of their end products. Regardless of the number of PLDs that must be programmed or configured, Jam STAPL simplifies in-field upgrades and revolutionizes the programming of PLDs.

This chapter describes MAX® II device programming support using Jam STAPL in embedded systems.

This chapter contains the following sections:

- "Embedded Systems" on page 14–1
- "Software Development" on page 14–4
- "Updating Devices Using Jam" on page 14–14

Embedded Systems

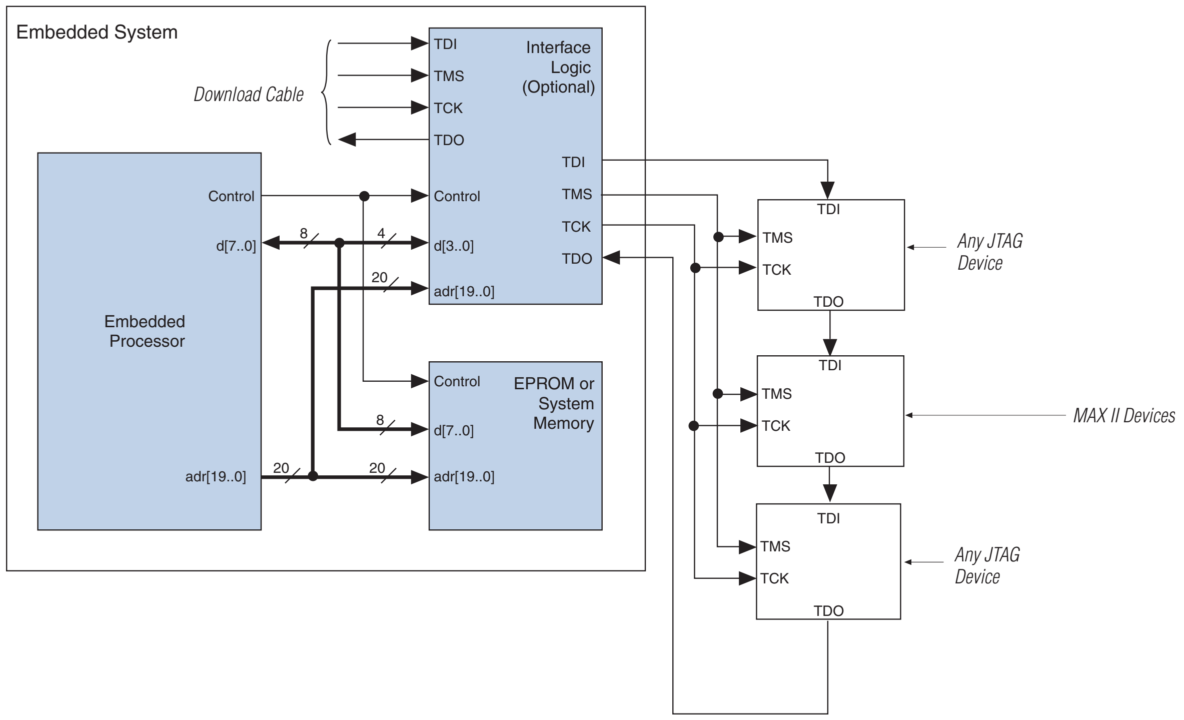
All embedded systems are made up of both hardware and software components. When designing an embedded system, the first step is to layout the printed circuit board (PCB). The second step is to develop the firmware that manages the board's functionality.

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 14–1 illustrates the second method, which is to connect the JTAG chain to an existing bus via 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 14–1. Embedded System Block Diagram



Both JTAG connection methods should include space for the MasterBlaster™, ByteBlaster™ II, or USB-Blaster™ header connection. The header is useful during prototyping because it allows designers to quickly verify or modify the PLD's contents. During production, the header can be removed to decrease cost.

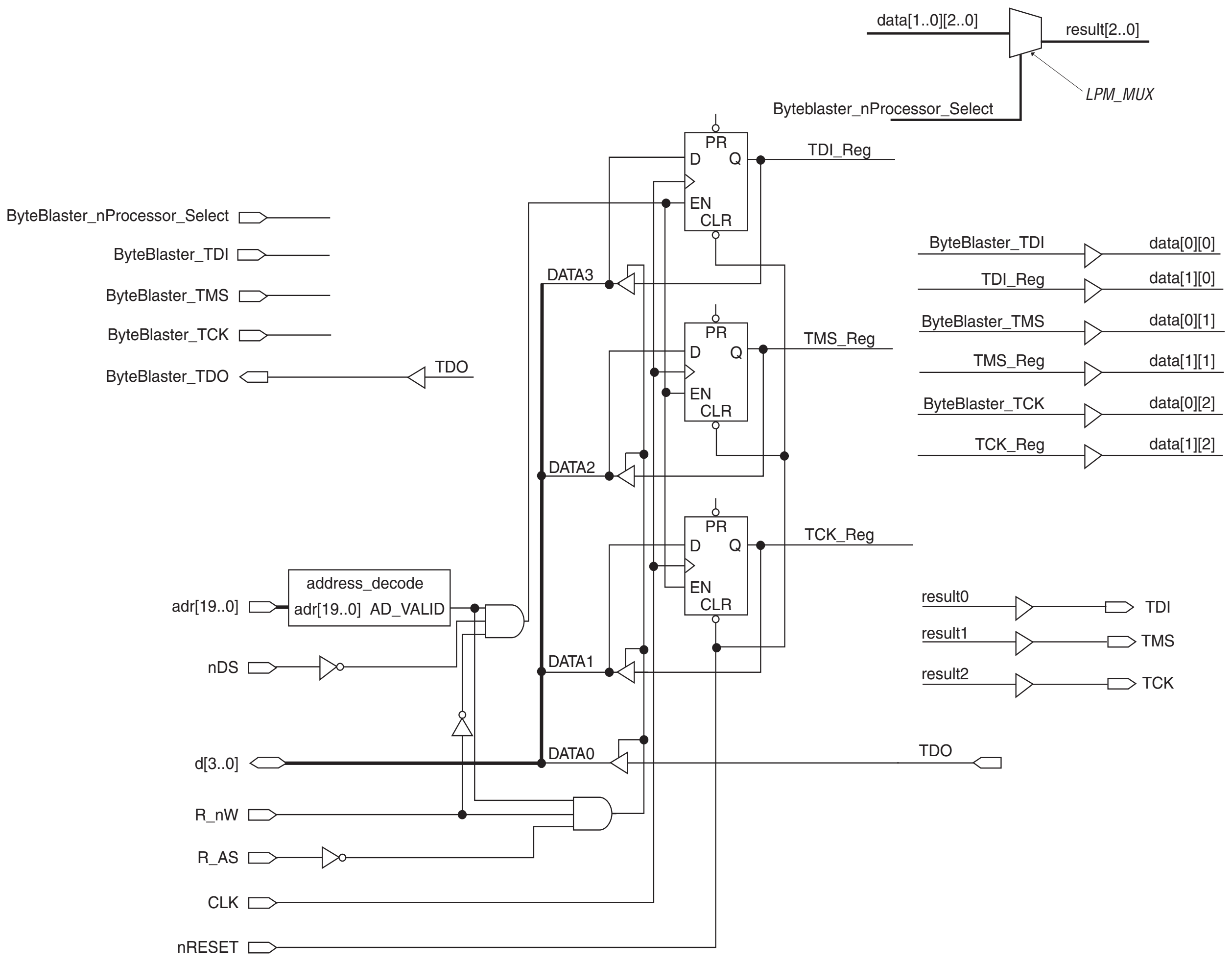
Example Interface PLD Design

Figure 14–2 shows an example design schematic of an interface PLD. A different design can be implemented; however, important points exemplified in this design are:

- `TMS`, `TCK`, and `TDI` should be synchronous outputs
- Multiplexer logic should be included to allow board access for the MasterBlaster, ByteBlaster II, or USB-Blaster download cable

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

Figure 14–2. Interface Logic Design Example



In Figure 14–2, the embedded processor asserts the JTAG chain's address, and the `R_nW` and `R_AS` signals can be set to tell 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 via 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 software to check the valid states of the registers in the interface PLD. In addition, multiplexer logic is included to permit a download cable to program the device chain. This capability is useful during the prototype phase of development, when programming must be verified.

Board Layout

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

- Treat the `TCK` signal trace as a clock tree

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

TCK Signal Trace Protection and Integrity

`TCK` 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 since this signal is software-generated and originates at a processor general-purpose I/O pin.

Pull-Down Resistors on TCK

`TCK` should be held low via 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 BST state, which may cause conflicts on the board. A typical resistor value is 1 kΩ.

JTAG Signal Traces

Short JTAG signal traces help eliminate noise and drive-strength issues. Special attention should be paid 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 `TDI` or `TDO`. Depending on the length and loading of the JTAG chain, some additional buffering may be required to ensure that the signals propagate to and from the processor with integrity.

External Resistors

You should add external resistors to output pins to pull outputs to a defined logic level during programming. Output pins will tri-state during programming. Also, on MAX® II devices, the pins will be pulled up by a weak internal resistor. Altera recommends that outputs driving sensitive input pins be tied to the appropriate level by an external resistor.

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 the *In-System Programmability Guidelines for MAX II Devices* chapter in the *MAX II Device Handbook*.

Software Development

Altera's embedded programming uses the Jam file output from the Quartus® II software tool with the standardized Jam Player software. Designing these tools requires minimal developer intervention because Jam files contain all of the data for programming MAX II devices. The bulk of development time is spent porting the Jam Player to the host embedded processor.

For more information about porting the Jam Byte-Code Player, see "Porting the Jam STAPL Byte-Code Player" on page 14–8.

Jam Files (.jam and .jbc)

Altera supports the following types of Jam files:

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

ASCII Text Files (.jam)

Altera supports two types of Jam files:

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

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)

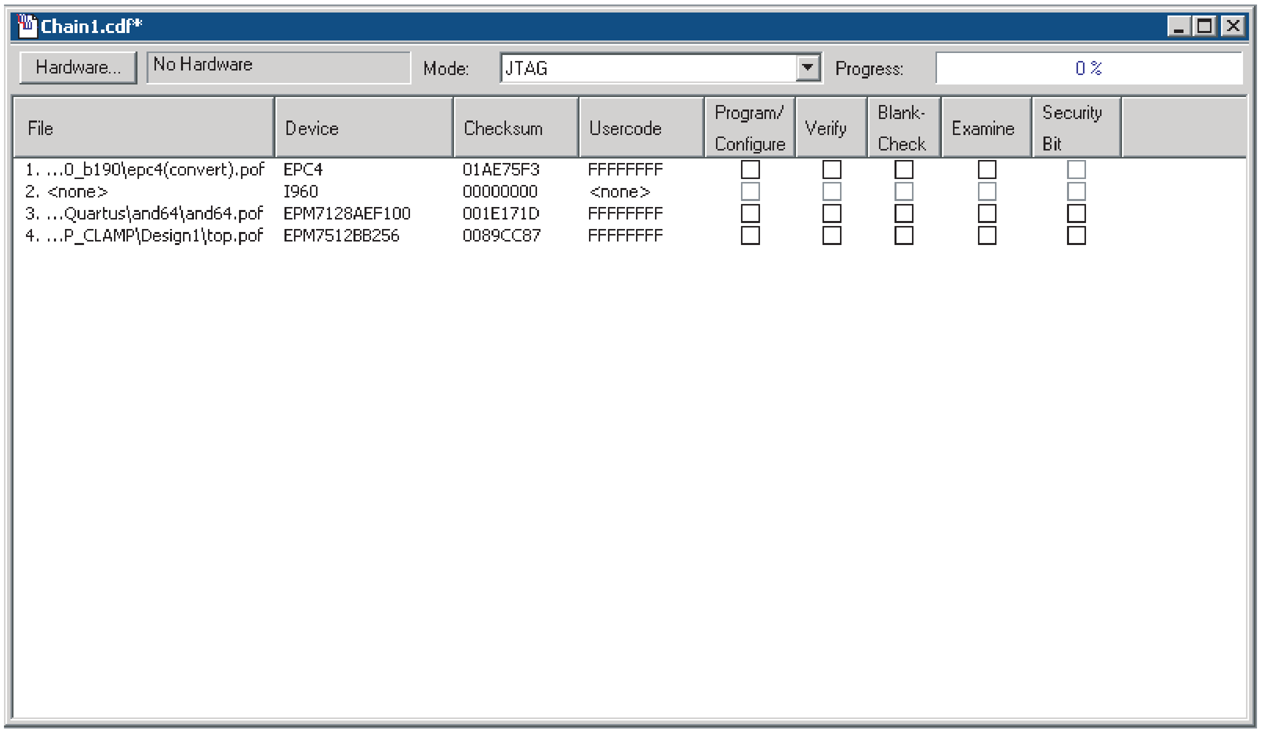
Altera recommends using Jam STAPL Byte-Code files in embedded applications because they use minimal memory.

Generating Jam Files

The Quartus II software can generate both Jam and JBC file types. In addition, Jam files can be compiled into JBC files via a stand-alone Jam Byte-Code compiler. The compiler produces a functionally equivalent JBC file.

Generating JBC files directly from the Quartus II software is simple. The software tool supports the programming and configuration of multiple devices from single or multiple JBC files. Figure 14–3 and Figure 14–4 show the dialog boxes that specify the device chain and JBC file generation in the Quartus II software.

Figure 14–3. Multi-Device JTAG Chain's Name and Sequence in Programmer Window in the Quartus II Software

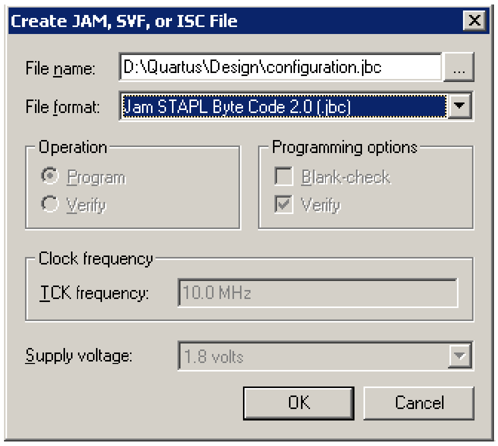


Chain1.cdf*

Hardware... No Hardware Mode: JTAG Progress: 0 %

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank- Check	Examine	Security Bit
1. ...0_b190\epc4(convert).pof	EPC4	01AE75F3	FFFFFFFF	☐	☐	☐	☐	☐
2. <none>	I960	00000000	<none>	☐	☐	☐	☐	☐
3. ...Quartus\and64\and64.pof	EPM7128AEF100	001E171D	FFFFFFFF	☐	☐	☐	☐	☐
4. ...P_CLAMP\Design1\top.pof	EPM7512BB256	0089CC87	FFFFFFFF	☐	☐	☐	☐	☐

Figure 14–4. Generating a JBC File for a Multi-Device JTAG Chain in the Quartus II Software



Create JAM, SVF, or ISC File

File name: D:\Quartus\Design\configuration.jbc

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

Operation: [● Program] ○ Verify

Programming options: ☐ Blank-check [✓ Verify]

Clock frequency: TCK frequency: 10.0 MHz

Supply voltage: 1.8 volts

OK Cancel

The following steps explain how to generate JBC files using the Quartus II software.

1. On the Tools menu, click **Programmer**.
2. Click **Add File** and select programming files for the respective devices.
3. On the File menu, point to **Create/Update** and click **Create JAM, SVF, or ISC File**. See Figure 14–4.
4. Specify a Jam STAPL Byte-Code File in the File format list.
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 will be bypassed.

Using Jam Files with the MAX II User Flash Memory Block

The Quartus II Programmer provides the option to individually target the entire device, logic array, or the user flash memory (UFM) block. As you can program the (UFM) section independently from the logic array, separate Jam STAPL and JBC options can be used in the command line to separately program UFM and configuration flash memory (CFM) blocks.

For more information, see "MAX II Jam/JBC Actions and Procedure Commands" on page 14–15.

Jam Players

Jam Players read the descriptive information in Jam files and translate them into data that programs the target PLDs. Jam Players do not program a particular device architecture or vendor; they only read and understand the syntax defined by the Jam file specification. In-field changes are confined to the Jam file, not the Jam Player. As a result, you do not need to modify the Jam Player source code for each in-field upgrade.

There are two types of Jam Players to accommodate the two types of Jam files: an ASCII Jam STAPL Player and a Jam STAPL Byte-Code Player. The general concepts within this chapter apply to both player types; however, the following information focuses on the Jam STAPL Byte-Code Player.

Jam Players can be used to program or write the MAX II configuration flash memory block and the UFM block separately since Jam STAPL and JBC files can be generated targeting only to either one or both sectors of the MAX II UFM block.

Jam Player Compatibility

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 Players are backward-compatible; they can play version 1.1 files and Jam STAPL files.

For more information about Altera's support for version 1.1 syntax, refer to *AN 122: Using Jam STAPL for ISP & ICR via an Embedded Processor*.

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.

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

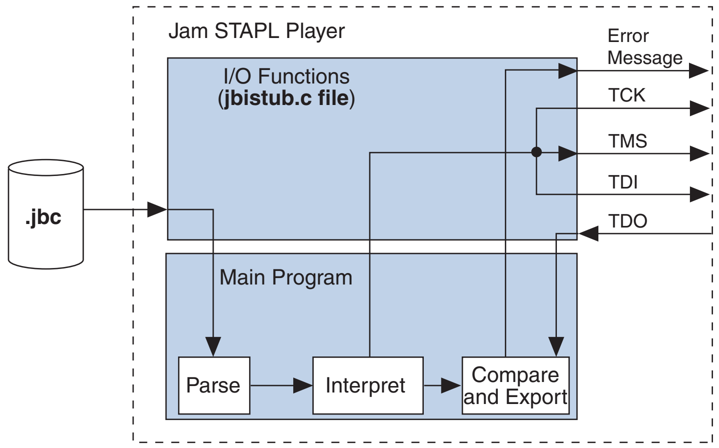
The 16-bit and 32-bit source code is divided into two categories:

- 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 14–5 illustrates 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 14–5. Jam STAPL Byte-Code Player Source Code Structure



Porting the Jam STAPL Byte-Code Player

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

To port the Jam Player, you need to customize several functions in the jbistub.c file, which are shown in Table 14–1.

Table 14–1. Functions Requiring Customization

Function	Description
`jbi_jtag_io()`	Interface 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

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

1. Set preprocessor statements to exclude extraneous code.
2. Map JTAG signals to hardware pins.
3. Handle text messages from `jbi_export()`.
4. Customize delay calibration.

Step 1: Set Preprocessor Statements to Exclude Extraneous Code

At the top of jbistub.c, 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 JTAG Signals to Hardware Pins

The `jbi_jtag_io()` function contains the code that sends and receives the binary programming data. Each of the four JTAG signals should be re-mapped to the embedded processor's pins. By default, the source code writes to the PC's parallel port. The `jbi_jtag_io()` signal maps the JTAG pins to the PC parallel port registers shown in Figure 14–6.

Figure 14–6. 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 14–6:

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

The mapping is highlighted 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 which inverts the `TDO` value is as follows:

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

If the target processor does not invert TDO, the code should look 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 would be as follows:

```
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, equivalent functions should be substituted.

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 the following assignment:

```
one_ms_delay = 1000
```

If this parameter is known, it should 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 while loop. This code is sampled over multiple tests and averaged to produce an accurate result upon which the delay can be based. 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 in MAX II devices should comply with the values given in the *DC and Switching Characteristics* chapter in the *MAX II Device Handbook*.

If the Jam STAPL Byte-Code Player does not operate within the timing specifications, the code should be optimized with the appropriate delays. Timing violations can occur if the processor is very powerful and can generate TCK at a rate faster than 18 MHz.

Other than the **jbistub.c file**, Altera strongly recommends keeping source code in other files unchanged from their default state. Altering the source code in these files will result 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 documents 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:

Equation 14–1.

$$\text{ROM}_{\text{Size}} = \text{JBC file size} + \text{Jam player size}$$

The JBC file size can be separated into two categories: the amount of memory required to store the programming data, and the space required for the programming algorithm. Use the following equation to estimate the JBC file size:

Equation 14–2.

$$\text{JBC file size} = \text{Alg} + \sum_{k=1}^{N} Data$$

Notes to Equation 14–2:

(1) Alg =Space used by algorithm.
(2) $Data$ =Space used by compressed programming data.
(3) k =Index representing device being targeted.
(4) N =Number of target devices in the chain.

This equation provides a JBC file size estimate that may vary by ±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 JBC file size (due to the data component) is linear.

Table 14–2 shows algorithm file size constants when targeting a single MAX II device.

Table 14–2. Algorithm File Size Constants Targeting a Single Altera Device Family

Device	Typical JBC File Algorithm Size (Kbytes)
MAX II	24.3

Table 14–3 shows data size constants for MAX II devices that support the Jam language for ISP.

Table 14–3. Data Constants

Device	Typical Jam STAPL Byte-Code Data Size (Kbytes)	
	Compressed	**Uncompressed** *(1)*
EPM240	12.4 *(2)*	12.4 *(2)*
EPM570	11.4	19.6
EPM1270	16.9	31.9
EPM2210	24.7	49.3

Notes to Table 14–3:

(1) For more information about how to generate JBC files with uncompressed programming data, contact Altera Applications.

(2) There is a minimum limit of 64K bits for compressed arrays with the JBC compiler. Programming data arrays smaller than 64K bits (8K bytes) are not compressed. The EPM240 programming data array is below the limit, which means the JBC files are always uncompressed. The reason for this limit is that 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.

After estimating the JBC file size, estimate the Jam Player size using the information in Table 14–4.

Table 14–4. 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 the following equation to estimate the maximum amount of DRAM required by the Jam Player:

Equation 14–3.

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

The JBC file size is determined by a single- or multi-device equation (see "Estimating ROM Usage" on page 14–11).

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, some RAM is used by the Player to uncompress the data and temporarily store it. The uncompressed data sizes are provided in Table 14–3. If an uncompressed JBC file is used, use the following equation:

Equation 14–4.

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

☞ 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 the **jbimain.c** file.

Estimating Memory Example

The following example uses a 16-bit Motorola 68000 processor to program an EPM7128AE device and an EPM7064AE device in an IEEE Std. 1149.1 JTAG chain via a JBC file that uses compressed data. To determine memory usage, first determine the amount of ROM required and 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 following multi-device equation to estimate the JBC file size. Because JBC files use compressed data, use the compressed data file size information, listed in Table 14–3, to determine Data size.where:

Equation 14–5.

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

Notes to Equation 14–5:

(1) Alg =21 Kbytes.

(2) $Data$ =EPM7064AE Data + EPM7128AE Data = 8 + 4 = 12 Kbytes.

Thus, the JBC file size equals 33 Kbytes.

2. Estimate the JBC Player size. This example uses a JBC Player size of 62 Kbytes because this 68000 is a 16-bit processor. Use the following equation to determine the amount of ROM needed:

Equation 14–6.

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

$$\text{ROM Size} = 95 \text{ Kbytes}$$

3. Estimate the RAM usage with the following equation:

Equation 14–7.

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

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 as follows:

- EPM7064AE = 8 Kbytes
- EPM7128AE = 12 Kbytes

Calculate the total DRAM usage as follows:

Equation 14–8.

$$\text{RAM Size} = 33 \text{ Kbytes} + (8 \text{ Kbytes} + 12 \text{ Kbytes}) = 53 \text{ Kbytes}$$

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 the 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.

The main entry point for execution of 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,
   init *exit_code
)
```

The code within main(), in **jbistub.c**, determines the variables that will be passed to `jbi_execute()`. In most cases, this code is not applicable to an embedded environment; therefore, this code can be removed and the `jbi_execute()` routine can be set up for the embedded environment. Table 14–5 describes each parameter.

Table 14–5. Parameters *(Note 1)* (Part 1 of 2)

Parameter	Status	Description
`program`	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 `jbi_execute()`.
`program_size`	Mandatory	Amount of memory (in bytes) that the JBC file occupies.
`workspace`	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 pre-defined memory space. This memory should be allocated before calling `jbi_execute()`. If 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.

Table 14–5. Parameters *(Note 1)* (Part 2 of 2)

Parameter	Status	Description
`action`	Mandatory	A pointer to a string (text that directs the Player). Example actions are PROGRAM or VERIFY. In most cases, this parameter will be set to the string PROGRAM. The Player is not case-sensitive, so the text can be either uppercase or lowercase. The Player supports all actions defined in the *Jam Standard Test and Programming Language Specification*. See Table 14–6. Note that the string must be null terminated.
`init_list`	Optional	An array of pointers to strings. This parameter is used when applying Jam version 1.1 files. *(2)*
`error_line`	—	A pointer to a long integer. If an error is encountered during execution, the Player will record the line of the JBC file 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 file. If this kind of error is encountered, the supporting vendor should be contacted with a detailed description of the circumstances in which the exit code was encountered.

Notes to Table 14–5:

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

(2) For more information, refer to *AN 122: Using Jam STAPL for ISP & ICR via an Embedded Processor.*

MAX II Jam/JBC Actions and Procedure Commands

The Jam/JBC supported action commands for MAX II devices are listed in Table 14–6, including their definitions. The optional procedures that you can execute with each action are listed along with their definitions in Table 14–7.

Table 14–6. MAX II Jam/JBC Actions (Part 1 of 2)

Jam/JBC Action	Description	Optional Procedures (Off by Default)

`PROGRAM`	Programs the device. You can optionally program CFM and UFM separately.	`DO_BYPASS_CFM`
		`DO_BYPASS_UFM`
		`DO_SECURE`
		`DO_REAL_TIME_ISP`
		`DO_READ_USERCODE`

`BLANKCHECK`	Blank checks the entire device. You can optionally blank check CFM and UFM separately.	`DO_BYPASS_CFM`
		`DO_BYPASS_UFM`
		`DO_REAL_TIME_ISP`

`VERIFY`	Verifies the entire device against the programming data in the Jam file. You can optionally verify CFM and UFM separately.	`DO_BYPASS_CFM`
		`DO_BYPASS_UFM`
		`DO_REAL_TIME_ISP`
		`DO_READ_USERCODE`

Table 14–6. MAX II Jam/JBC Actions (Part 2 of 2)

Jam/JBC Action	Description	Optional Procedures (Off by Default)

`ERASE`	Erases the programming content of the device. You can optionally erase CFM and UFM separately.	`DO_BYPASS_CFM`
		`DO_BYPASS_UFM`
		`DO_REAL_TIME_ISP`

`READ_USERCODE`	Returns the JTAG USERCODE register information from the device. READ_USERCODE can be set to a specific value in the programming file in the Quartus II software by using the Assignments menu -> Device -> Device and Pin options -> General tab, which has a USERCODE data entry.	—

Table 14–7. MAX II Jam/JBC Optional Procedure Definitions

Procedure	Description
`DO_BYPASS_CFM`	When set =1, DO_BYPASS_CFM bypasses the CFM and performs the specified action on the UFM only. When set =0, this option is ignored (default).
`DO_BYPASS_UFM`	When set =1, DO_BYPASS_UFM bypasses the UFM and performs the specified action on the CFM only. When set =0, this option is ignored (default).
`DO_BLANKCHECK`	When set =1, the device, CFM, or UFM is blank checked. When set =0, this option is ignored (default).
`DO_SECURE`	When set =1, the device's security bit is set. The security bit only affects the CFM data. The UFM cannot be protected. When set =0, this option is ignored (default).
`DO_REAL_TIME_ISP`	When set =1, the real-time ISP feature is enabled for the ISP action being executed. When set =0, the device uses normal ISP mode for any operations.
`DO_READ_USERCODE`	When set =1, the player returns the JTAG USERCODE register information from the device.

Executing the Jam file from a command prompt requires that an action is specified using the -a option, as shown in the following example:

```
jam -aPROGRAM <filename>
```

This command programs the entire MAX II device with the Jam file specified in the filename.

You can execute the optional procedures with its associated actions by using the -d option, as shown in the following example:

```
jam -aPROGRAM -dDO_BYPASS_UFM=1
        -dDO_REAL_TIME_ISP=1 <filename>
```

This command programs the MAX II CFM block only with real-time ISP enabled (i.e., the device remains in user mode during the entire process).

The JBC player uses the same format except for the executable name.

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 one of the following exit codes in Table 14–8, as defined in the *Jam Standard Test and Programming Language Specification*.

Table 14–8. Exit Codes

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

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, in-field upgrades can be performed 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 illustrates how the Jam Byte-Code Player could 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 will now read the JTAG USERCODE and export it using the `jbi_export()` routine. The code can then branch based upon the result.

The following shows example code for 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);
}
```

A switch statement can be used to determine which device needs to be updated and which design revision should be used. With Jam STAPL Byte-Code software support, PLD updates become as easy as adding a few lines of code.

Conclusion

Using Jam STAPL provides an simple way to benefit from ISP. Jam meets all of 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. Executing the Jam Player is straightforward, as is the calculation of resources that will be used. For the most recent updates and information, visit the Jam website at: www.altera.com/jamisp.

Referenced Documents

This chapter references the following documents:

- AN 39: IEEE 1149.1 (JTAG) Boudary-Scan Testing in Altera Devices
- AN 111: Embedded Programming Using the 8051 & Jam Byte-Code
- AN 122: Using Jam STAPL for ISP & ICR via an Embedded Processor
- *DC and Switching Characteristics* chapter in the *MAX II Device Handbook*
- *In-System Programmability Guidelines for MAX II Devices* chapter in the *MAX II Device Handbook*

Document Revision History

Table 14–9 shows the revision history for this chapter.

Table 14–9. Document Revision History

Date and Revision	Changes Made	Summary of Changes
October 2008, version 1.8	■ Updated New Document Format.	—
December 2007, version 1.7	■ Added "Referenced Documents" section.	—
December 2006, version 1.6	■ Added document revision history.	—
August 2006, version 1.5	■ Updated "Embedded Systems" section.	—
August 2005, version 1.4	■ Updated Tables 14-2 and 14-3.	—
June 2005, version 1.3	■ Removed Table 14-6 from v1.2. ■ Added a new section "MAX II Jam/JBC Actions and Procedure Commands".	—
January 2005, version 1.2	■ Previously published as Chapter 15. No changes to content.	—
December 2004, version 1.1	■ Changed document reference from AN 88 to AN 122.	—