

Introduction

Advances in programmable logic devices (PLDs) have enabled innovative in-system programmability (ISP) and in-circuit reconfigurability (ICR) features. The Jam™ Standard Test and Programming Language (STAPL), JEDEC standard JESD-71, is compatible with all current PLDs that offer ISP and ICR 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 and ICR 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 and configuration of PLDs.

This application note describes Altera's programming and configuration support using Jam STAPL in embedded systems.

Embedded Systems

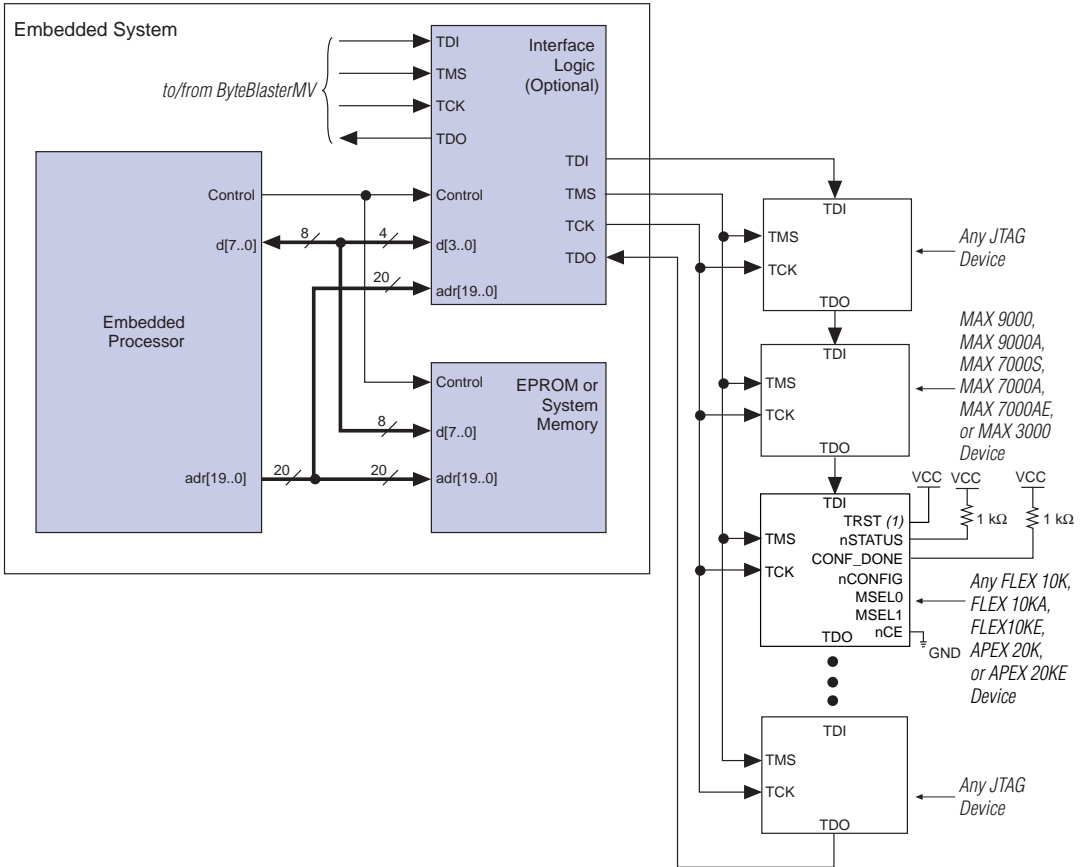
All embedded systems are made up of both hardware and software components. When designing an embedded system, the first step is to layout the printed circuit board (PCB). The second step is to develop the firmware that manages the board's functionality.

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 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 1. Embedded System Block Diagram



Both JTAG connection methods should include space for the MasterBlaster™ or ByteBlasterMV™ 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 save cost.

Example Interface PLD Design

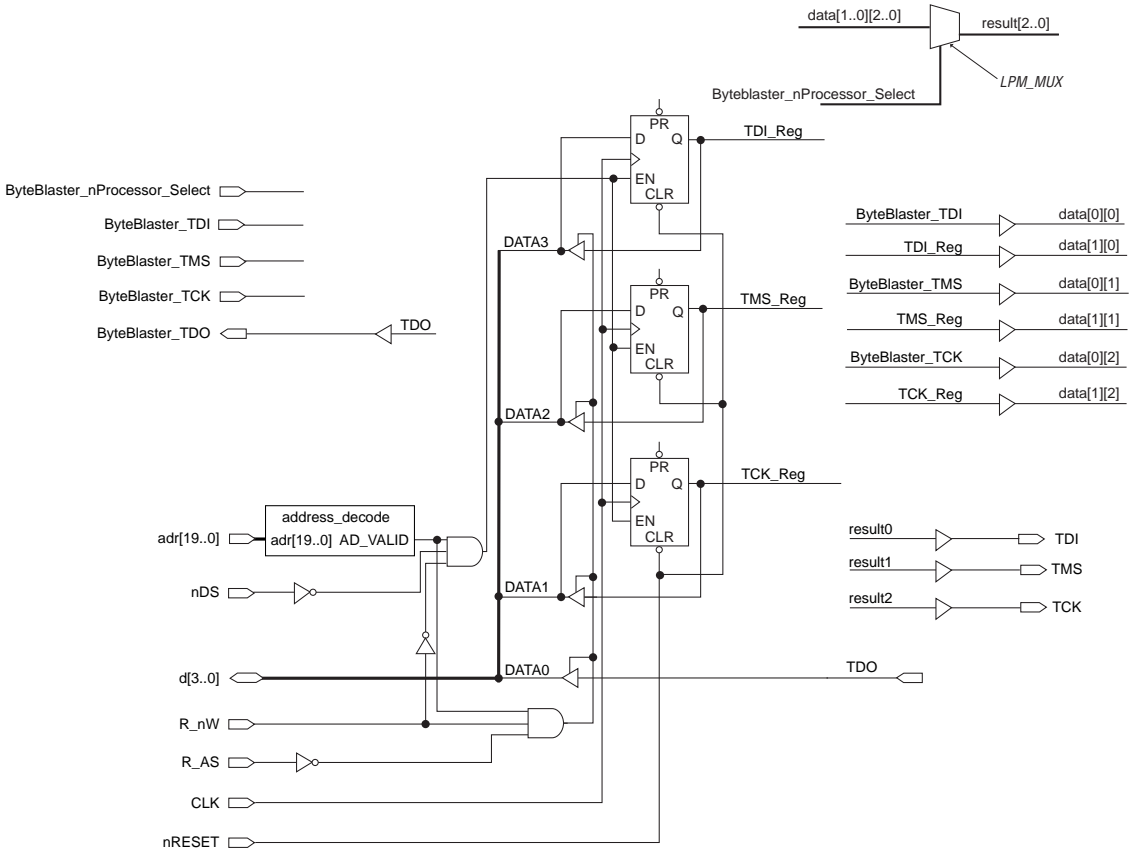
Figure 2 shows an example design schematic of an interface PLD. A different design can be implemented; however, important points exemplified in this design are:

- TMS, TCK, and TDI should be synchronous outputs
- Multiplexer logic should be included to allow board access for the MasterBlaster or ByteBlasterMV 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 on an embedded data bus.

Figure 2. Interface Logic Design Example



In [Figure 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 MasterBlaster or ByteBlasterMV download cable to program the device chain. This capability is useful during the prototype phase of development, when programming and configuration must be verified.



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 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 & 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 k Ω .

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 or configuration. Output pins will 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 (e.g., 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, see [Application Note 100 \(In-System Programmability Guidelines\)](#).

Software Development

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



For more information on porting the Jam Byte-Code Player, see ["Porting the Jam STAPL Byte-Code Player"](#) on page 9.

Jam Files (.jam & .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

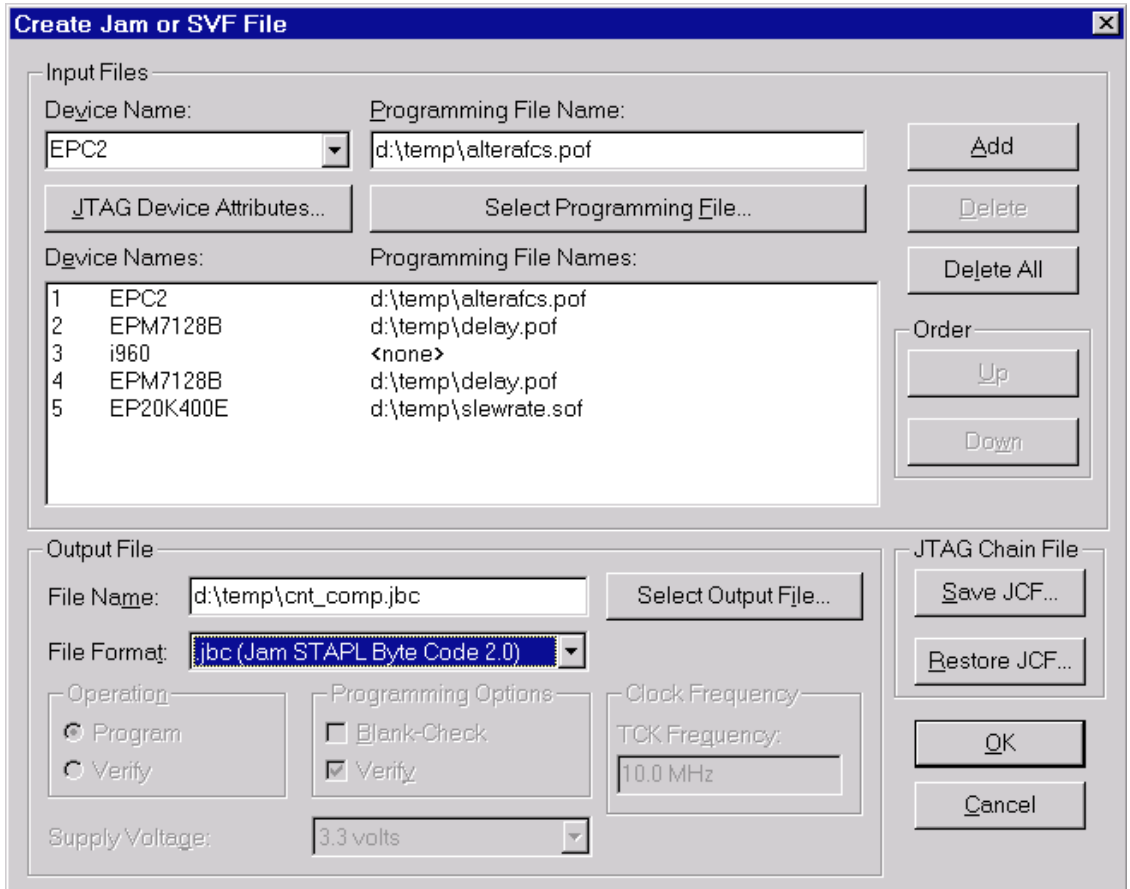
The MAX+PLUS II software can generate both Jam and JBC file types. In addition, Jam files can be compiled into JBC files via a stand-alone Jam Byte-Code compiler. The compiler produces a functionally equivalent JBC file.




The Jam Byte-Code compiler can be downloaded from the Jam web site at <http://www.jamisp.com>.

Generating JBC files directly from the MAX+PLUS II software is simple. The software tool supports the programming and configuration of multiple devices from single or multiple JBC files. [Figure 3](#) shows the dialog box 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



The following steps explain how to generate JBC files using the MAX+PLUS II software.

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

1. In the MAX+PLUS II Programmer, choose **Create Jam or SVF File** (File menu).
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. Specify a Jam STAPL Byte-Code File in the *File Format* drop-down list box.
4. 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.

Jam Players

Jam Players read the descriptive information in Jam files and translate them into data that programs or configures the target PLDs. Jam Players do not program or configure 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 application note apply to both player types; however, the following information focuses on the Jam STAPL Byte-Code Player.

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 on Altera's support for version 1.1 syntax, see [Application Note 88 \(Using the Jam Language for ISP & ICR via an Embedded Processor\)](#).

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. Some 8-bit processors are also supported by a specific subset of source code available on the Jam web site at <http://www.jamisp.com>.



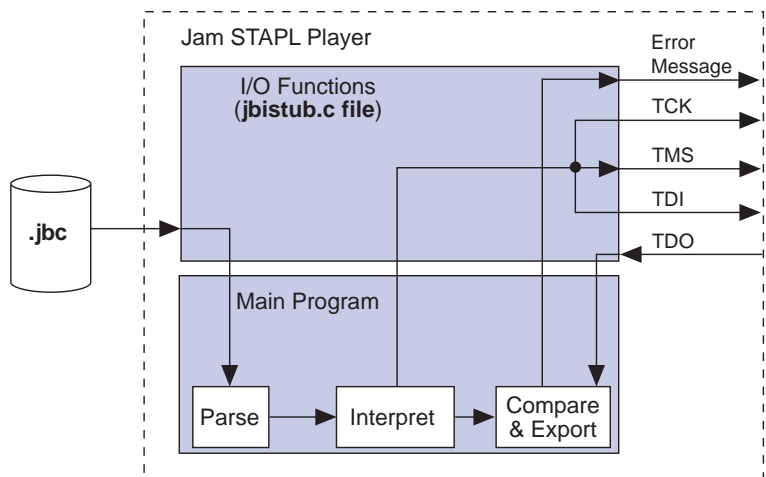
For more information about Altera's support for 8-bit processors, see [Application Note 111 \(Embedded Programming Using the 8051 & Jam Byte-Code\)](#).

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 4 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 4. 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 1](#).

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 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 5](#).

Figure 5. 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:

(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 (e.g., 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, see [Application Note 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 for MAX[®], FLEX[®], and APEX devices should comply with the values given in [Tables 2](#) and [3](#).

Table 2. IEEE Std. 1149.1 Timing Parameters

Symbol	Parameter	MAX 9000		MAX 7000A		MAX 7000AE		MAX 7000S		Unit
		Min	Max	Min	Max	Min	Max	Min	Max	
t_{JCP}	TCK clock period	100		100		100		100		ns
t_{JCH}	TCK clock high time	50		50		50		50		ns
t_{JCL}	TCK clock low time	50		50		50		50		ns
t_{JPSU}	JTAG port setup time	20		20		20		20		ns
t_{JPH}	JTAG port hold time	45		45		45		45		ns
t_{JPCO}	JTAG port clock to output		25		25		25		25	ns
t_{JPZX}	JTAG port high-impedance to valid output		25		25		25		25	ns
t_{JPXZ}	JTAG port valid output to high-impedance		25		25		25		25	ns
t_{JSSU}	Capture register setup time	20		20		20		20		ns
t_{JSH}	Capture register hold time	45		45		45		45		ns
t_{JSCO}	Update register clock to output		25		25		25		25	ns
t_{JSZX}	Update register high-impedance to valid output		25		25		25		25	ns
t_{JSXZ}	Update register valid output to high-impedance		25		25		25		25	ns

Table 3. IEEE Std. 1149.1 Timing Parameters

Symbol	Parameter	MAX 3000A		MAX 7000B		APEX 20K		FLEX 10K		EPC2		Unit
		Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	
t_{JCP}	TCK clock period	100		100		100		100		100		ns
t_{JCH}	TCK clock high time	50		50		50		50		50		ns
t_{JCL}	TCK clock low time	50		50		50		50		50		ns
t_{JPSU}	JTAG port setup time	20		20		20		20		20		ns
t_{JPH}	JTAG port hold time	45		45		45		45		45		ns
t_{JPCO}	JTAG port clock to output		25		25		25		25		25	ns
t_{JPZX}	JTAG port high-impedance to valid output		25		25		25		25		25	ns
t_{JPXZ}	JTAG port valid output to high-impedance		25		25		25		25		25	ns
t_{JSSU}	Capture register setup time	20		20		20		20		20		ns
t_{JSH}	Capture register hold time	45		45		45		45		45		ns
t_{JSCO}	Update register clock to output		25		25		25		25		25	ns
t_{JSZX}	Update register high-impedance to valid output		25		25		25		25		25	ns
t_{JSXZ}	Update register valid output to high-impedance		25		25		25		25		25	ns

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 occur only if the processor is very powerful and can generate TCK at a rate faster than 10 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:

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

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

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

where:

- Alg = Space used by algorithm
- $Data$ = Space used by compressed programming data
- k = Index representing 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 JBC file size (due to the data component) is linear.

Table 4 shows algorithm file size constants when targeting a single device family, and Table 5 shows algorithm file size constants for all possible combinations of Altera device families that support the Jam language.

Device	Typical JBC File Algorithm Size (Kbytes)
APEX 20K	14
APEX 20KE	14
FLEX 10K	15
FLEX 10KE	15
FLEX 10KA	15
FLEX 10KB	15
EPC2	19
MAX 7000AE	21
MAX 7000	21
MAX 3000A	21
MAX 9000	21
MAX 7000S	25
MAX 7000A	25
MAX 7000B	17

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:

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

Table 6 shows data size constants for Altera devices that support the Jam language for ISP.

Table 6. Data Constants (Part 1 of 2)		
Device	Typical Jam STAPL Byte-Code Data Size (Kbytes)	
	Compressed	Uncompressed ⁽¹⁾
EPM7032S	8	8
EPM7032AE	6	6
EPM7064S	13	13
EPM7064AE	8	8
EPM7128S, EPM7128A	5	24
EPM7128AE	4	12
EPM7128B	4	12
EPM7160S	10	28
EPM7192S	11	35
EPM7256S, EPM7256A	15	51
EPM7256AE	11	18
EPM7512AE	18	37
EPM9320, EPM9320A	21	57
EPM9400	21	71
EPM9480	22	85
EPM9560, EPM9560A	23	98
EPF10K10, EPF10K10A	12	15
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

Device	Typical Jam STAPL Byte-Code Data Size (Kbytes)	
	Compressed	Uncompressed (1)
EP20K100	128	244
EP20K200	249	475
EP20K400	619	1,180
EPC2	136	212

Note:

- (1) For more information on how to generate JBC files with uncompressed programming data, contact Altera Applications at (800) 800-EPLD.

After estimating the JBC file size, estimate the Jam Player size using the information in [Table 7](#).

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:

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

The JBC file size is determined by a single- or multi-device equation (see [“Estimating ROM Usage” on page 15](#)).

The amount of RAM used by the Jam Player will be 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 will be used by the Player to uncompress the data and temporarily store it. The uncompressed data sizes are provided in [Tables 5 and 6](#). If an uncompressed JBC file is used, use the following equation:

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



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 `jbmain.c` file.

Estimating Memory Example

The following example uses a 16-bit Motorola 68000 processor to program an EPM7128AE and 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 [Tables 5 and 6](#), to determine *Data* size.

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

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.

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:

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

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

3. Estimate the RAM usage with the following equation:

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

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:

$$\text{EPM7064AE} = 8 \text{ Kbytes}$$

$$\text{EPM7128AE} = 12 \text{ Kbytes}$$

Calculate the total DRAM usage as follows:

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

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 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,
    int *exit_code
)
```

The code within `main()`, in `jbistub.c`, determines the variables that will be passed to `jbi_execute()`. In most cases, this code will not be 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 8](#) describes each parameter, and [Table 9](#) describes each action name.

Updating Devices Using Jam

Table 8. Parameters *Note (1)*

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 <code>jbi_execute()</code> .
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 <code>jbi_execute()</code> . 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 <code>workspace</code> points.
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 upper or lower case. The Player supports all actions defined in the <i>Jam Standard Test and Programming Language Specification</i> . See Table 9 . 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:

- (1) Mandatory parameters must be passed for the Player to run.
- (2) For more information, refer to [Application Note 88 \(Using the Jam Language for ISP & ICR via an Embedded Processor\)](#).

Table 9. Supported Actions <i>Note (1)</i>	
Action Name	Description
CHECKCHAIN	Verify the continuity of the IEEE Std. 1149.1 scan chain
READ_IDCODE	Read the IEEE 1149.1 IDCODE and EXPORT it
READ_USERCODE	Read the IEEE 1149.1 USERCODE and EXPORT it
READ_UES	Read the UESCODE and EXPORT it
ERASE	Perform a bulk erase of the device(s)
BLANK_CHECK	Check the erased state of the device(s)
PROGRAM	Program the device
VERIFY	Verify the programming data of the device(s)
READ	Read the programming data of the device(s)
CHECKSUM	Calculate one fuse checksum of the programming data of the device(s)
SECURE	Set the security bit of the device(s)
QUERY_SECURITY	Check whether the security bit is set
TEST	Perform a test. This test can include tests such as boundary scan, internal, vector, and built-in self tests.

Note:

- (1) Not all actions are applicable to every Altera device. See the appropriate device family data sheet to learn which functions apply.

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 10](#), as defined in the *Jam Standard Test and Programming Language Specification*.

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.

Figure 6 shows an example code for using the Jam Player.

Figure 6. Jam Player Code Example

```

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_execute(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 a simple way to benefit from ISP and ICR. 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 web site at: <http://www.jamisp.com>.



101 Innovation Drive
 San Jose, CA 95134
 (408) 544-7000
<http://www.altera.com>
Applications Hotline:
 (800) 800-EPLD
Customer Marketing:
 (408) 544-7104
Literature Services:
 (888) 3-ALTERA
lit_req@altera.com

Altera, APEX, ByteBlaster, ByteBlasterMV, EPF10K10, EPF10K10A, EPF10K20, EPF10K30, EPF10K30A, EPF10K30E, EPF10K40, EPF10K50, EPF1050E, EPF10K50V, EPF10K70, EPF10K100, EPF10K100A, EPF10K100B, EPF10K100E, EPF10K130E, EPF10K130V, EPF10K200E, EPF10K250A, EPF10K250E, EPM7032S, EPM7064AE, EPM7064S, EPM7128A, EPM7128AE, EPM7128S, EPM7160S, EPM7192S, EPM7256A, EPM7256S, EPM9400, EPM9480, EPM9560, EPM9560A, FLEX 10K, FLEX 10KA, Jam, MasterBlaster, MAX, MAX+PLUS, Quartus, MAX 7000A, MAX 7000AE, MAX 7000S, MAX 9000, and MAX 9000A are trademarks and/or service marks of Altera Corporation in the United States and other countries. Altera acknowledges the trademarks of other organizations for their respective products or services mentioned in this document. 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.

Copyright © 2000 Altera Corporation. All rights reserved.



I.S. EN ISO 9001