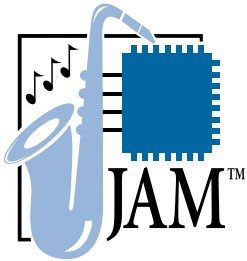


Introduction




In-system programming via an embedded processor, available in MAX® 9000 (including MAX 9000A), MAX 7000S, and MAX 7000A devices, enables easy design prototyping, streamlines production, and allows quick and efficient in-field upgrades. Devices that support in-system programmability (ISP) can be upgraded in the field easily by downloading new configurations using ROM, FLASH cards, modems, or other data links. Design changes can also be downloaded to a system in the field via an embedded processor. The embedded processor transfers programming data from a memory source to a device and allows easy design upgrades.

The Jam™ programming and test language, a new standard file format for ISP, is designed to support programming of any ISP-capable device that uses the IEEE Std. 1149.1 Joint Test Action Group (JTAG) interface. The Jam language is an interpreted language and is a freely licensable open standard. The Jam source code is executed directly by an interpreter program executed by an embedded processor, without being compiled into binary executable code (see “[Embedded Programming with the Jam Language](#)” on page 172). The Jam source code, or Jam File, contains the programming algorithm and data to upgrade one or more devices.

This application note describes how to use the Jam language to achieve the benefits of ISP via an embedded processor, including:

- Embedded system configuration and requirements
- Embedded programming with the Jam language

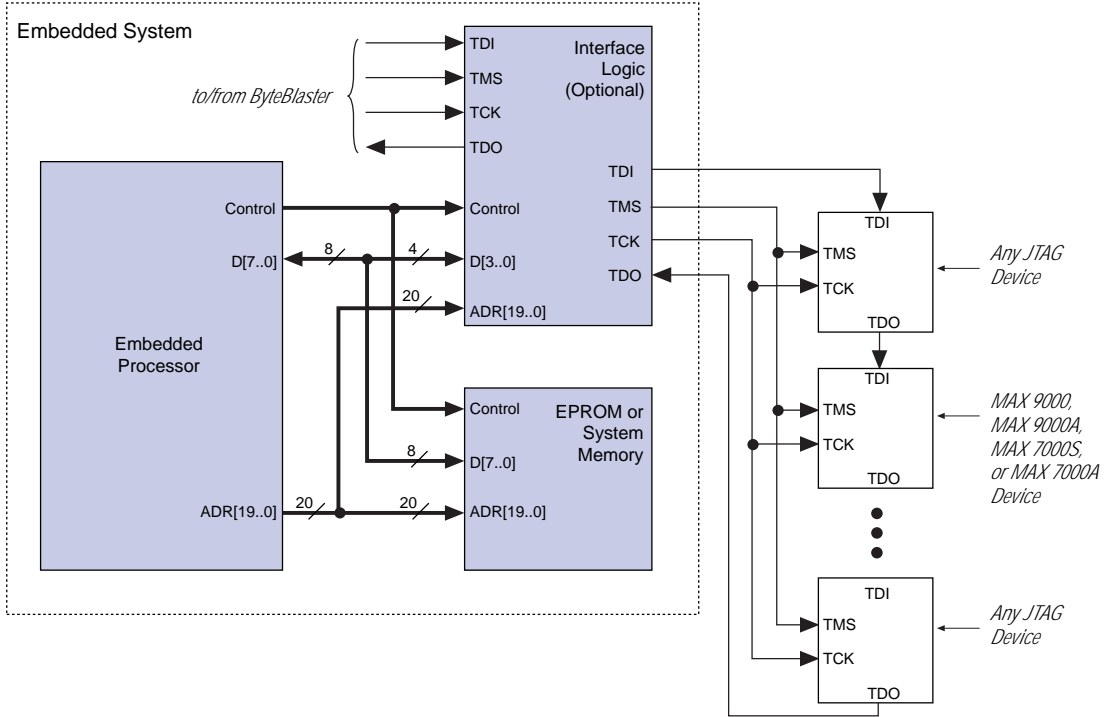
 This application note should be used with the *Jam Programming & Test Language Specification* in this handbook.

Embedded System Configuration & Requirements

To achieve the benefits of ISP, an embedded system must be able to program target devices using a small amount of system memory, and it must be flexible enough to adapt to a changing set of devices from multiple device vendors. The embedded system typically consists of an embedded processor, EPROM or system memory, and some interface logic. Programming data is stored in system memory (i.e., EPROM or FLASH memory).

During in-system programming, the embedded processor transfers programming data from system memory to the ISP-capable device(s). **Figure 1** shows a block diagram of an embedded system.

Figure 1. Embedded System Block Diagram



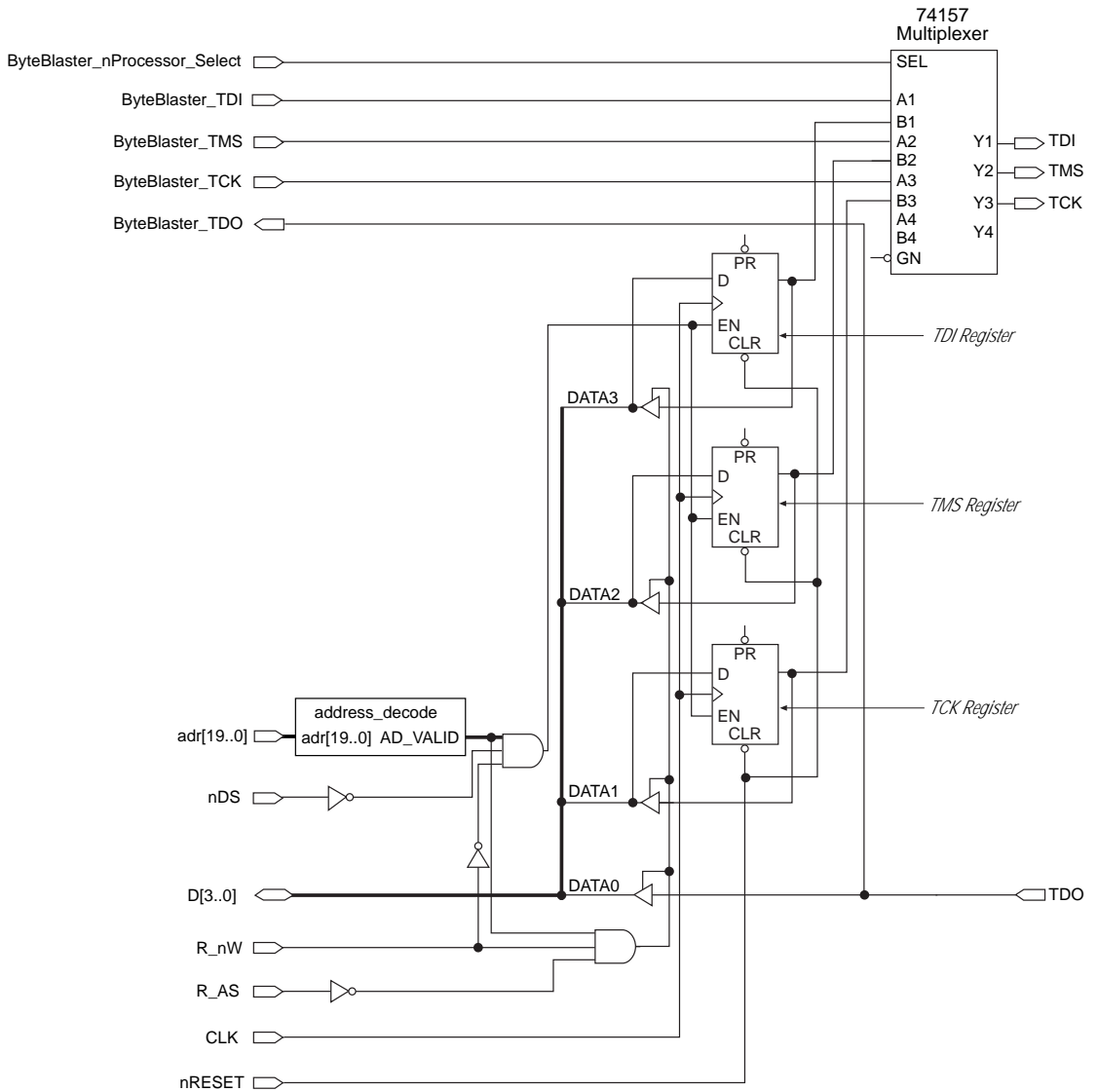
The embedded processor is connected to the EPROM or system memory and a programmable logic device (PLD) that stores the optional interface logic. The JTAG chain can connect directly to four of the embedded processor's data pins; however, adding the interface logic allows you to save these four ports, because it treats the JTAG chain as an address location on the existing bus. Additionally, you should install a 10-pin ByteBlaster™ header on the board to allow the MAX+PLUS® II software and ByteBlaster parallel port download cable to access and verify the JTAG chain.



For more information on the ByteBlaster parallel port download cable, see the [ByteBlaster Parallel Port Download Cable Data Sheet](#) in this handbook.

Figure 2 illustrates the embedded system's interface logic.

Figure 2. Interface Logic



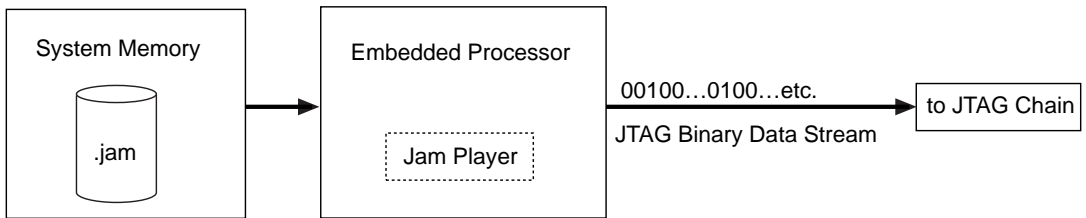
The interface logic is activated when it receives the proper address and control signals from the embedded processor. The registers then synchronize the timing of the TDI, TCK, and TMS signals and drive the output pins via a 74157 multiplexer. The multiplexer allows the ByteBlaster cable to access the JTAG chain for verification.

Embedded Programming with the Jam Language

Implementing the Jam language implementation has two parts: the Jam File (**.jam**) and the Jam Player. A Jam File is generated from the MAX+PLUS II development software and is stored in system memory. The Jam File contains all information required to program the ISP-capable device(s). The Jam Player runs on the embedded processor, interprets the information in the Jam File, and generates the binary data stream for device programming. Because upgrades are confined to the Jam File, the Jam Player can be used to program any vendor's device without requiring upgrades.

Figure 3 shows a block diagram of how in-system programming is achieved with the Jam language.

Figure 3. Block Diagram of ISP using Jam File & Jam Player



The Jam File (.jam)

Jam Files are compact and can be generated for any ISP-capable device that complies with the IEEE Std. 1149.1 (JTAG) specification.

Generating a Jam File

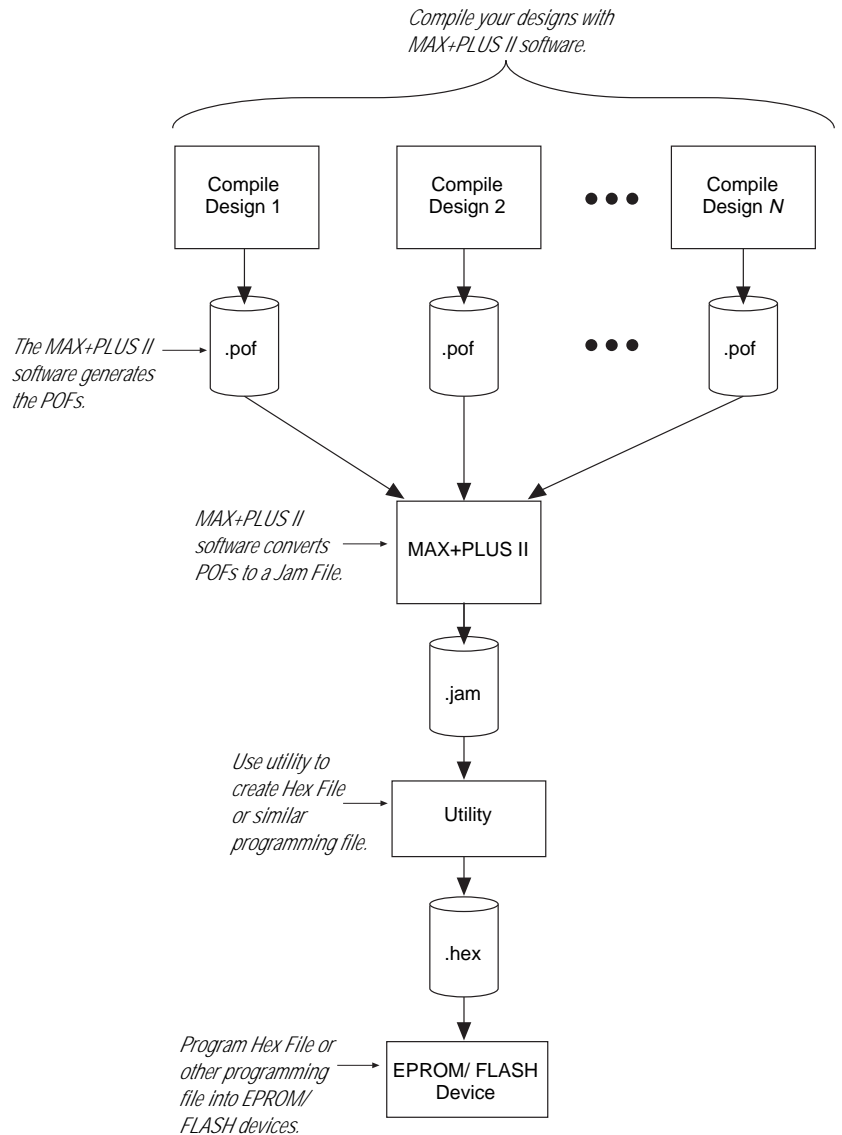
To program Altera devices using the Jam language, you must first create a Jam File with the MAX+PLUS II development software. You can use the MAX+PLUS II software to generate a Jam File from a Programmer Object File (**.pof**) using the **Create Jam/SVF File** command (File menu). Because Jam Files are generated from POFs, recompiling existing designs is not necessary. To store a Jam File in EPROM or FLASH memory, you must first convert it to a Hexadecimal (Intel-format) File (**.hex**) or a similar programming file. Embedded processor software packages or other utilities can automatically convert Jam Files for EPROM or FLASH programming. Likewise, some EPROM programmers support “raw binary” or “absolute binary” formats, which allow the Jam File to be read directly by the programmer without conversion.



For more information on how to create a Jam File, search for “Create Jam or SVF Files” in MAX+PLUS II Help.

Figure 4 describes how to generate a Jam File for in-system programming.

Figure 4. Generating a Jam File



Initialization Conventions

The MAX+PLUS II software generates Jam Files that use Jam conventions for initialization. This section describes special conventions that are supported by the Jam language; the conventions may vary for each device in a JTAG chain.

DO_PROGRAM

The DO_PROGRAM variable determines whether a device should be programmed. When DO_PROGRAM is set to 1, the Jam Player performs the silicon ID, bulk erase, and program functions for one or more ISP-capable devices. When programming more than one device in the same family, the MAX+PLUS II software will use a concurrent programming algorithm (i.e., programming data will shift through all devices of the same family at the same time.)

Targeted devices will tri-state I/O pins at the beginning of programming, and all I/O pins will leave the tri-state mode when the last device has finished programming. Both transitions happen simultaneously for all of the targeted devices in the JTAG chain.

DO_VERIFY

The DO_VERIFY variable tells the Jam File to verify the device. When DO_VERIFY is set to 1, the targeted devices are verified. The result of verification is indicated by the exit code of the Jam program.

DO_ERASE

The DO_ERASE variable causes the Jam File to perform a bulk erase (i.e., the device is completely erased). This process ensures the proper programming of each bit and the reliability of the device after multiple programming cycles.

DO_BLANKCHECK

The DO_BLANKCHECK variable ensures that the entire device has been properly bulk erased before programming. The DO_BLANKCHECK variable verifies that all data is erased.

READ_UESCODE

The READ_UESCODE variable is useful for tracking design revisions or the number of times a device has been programmed. When READ_USERCODE is set to 1, the user electronic signature (UES) is read from the device and reported.

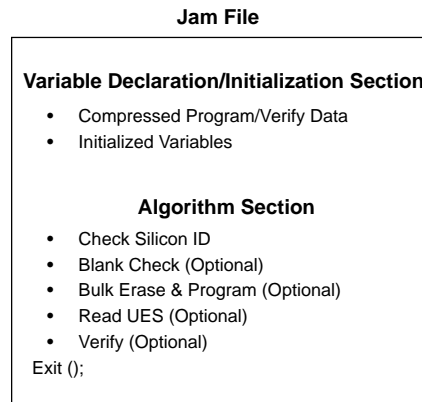


For more information on initialization conventions, see the [Jam Programming & Test Language Specification](#) in this handbook.

Jam File Structure

In an embedded system, a Jam File is placed in system memory that can be updated. A Jam File is structured to be compact; it has a Variable Declaration/Initialization Section and an Algorithm Section. [Figure 5](#) illustrates the Jam File structure.

Figure 5. Structure of the Jam File



The Variable Declaration and Initialization Section contains the declared variables that will be used in the Jam File. The variables can also be initialized to specific values. In the case of a BOOLEAN array, the variable can be initialized as a compressed data array, which is either the Run-Length Compression (RLC) or Advanced Compression Algorithm (ACA) formats. Variables of other types can be declared and initialized in this section; initialization of these variables is optional.



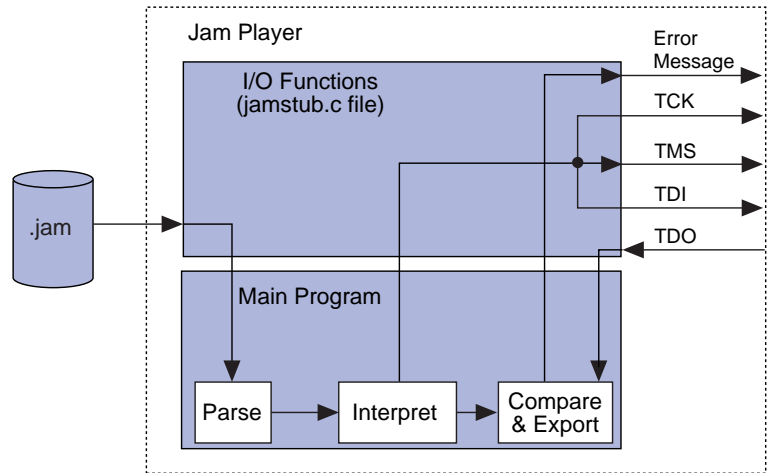
For more detail on the RLC or ACA data array formats, see the [Jam Programming & Test Language Specification](#).

The Algorithm Section, written entirely in text, can be broken into subsections that contain the actual programming commands and programming code that performs other necessary functions (e.g., branching based on the results of verification, looping for multiple JTAG data register scans, or other administrative functions to track the targeted JTAG chain). The Algorithm Section contains the superset of functions (e.g., blank check, verify, etc.) that can be performed on the targeted device(s).

The Jam Player

The Jam Player is a C program that parses the Jam File, interprets each Jam instruction, and reads and writes data to and from the JTAG chain. The variables processed by the Jam Player depend on the initialization list variables present at the time of execution (see “[Executing the Jam Player](#)” on page 177 for more information). Because each application has unique requirements, the Jam Player source code can be modified easily. [Figure 6](#) illustrates the Jam Player source code structure.

Figure 6. Jam Player Source Code Structure



The main program performs all of the basic functions of the Jam Player without having to be modified. Only the I/O functions, which are contained in the `jamstub.c` file, need to be modified for your application. These functions include those that specify addresses to I/O pins, delay routines, operating system-specific functions, and routines for file I/O.

The Jam Player resides permanently in system memory, where it interprets the commands given in the Jam File and generates a binary data stream for device programming. This structure confines all upgrades to the Jam File, and it allows the Jam Player to adapt to any system architecture.

Customizing the Jam Player

The Jam Player has been structured to simplify customization based on platform requirements and applications. All file I/O and port configurations can be changed by simply editing the `jamstub.c` file. As an input, the `jamstub.c` file can retrieve data from the Jam File and/or read shifted data that comes out of TDO. As an output, the `jamstub.c` file can send processed JTAG data to the three JTAG pins: TDI, TMS, and TCK, send formatted error and information messages back to the calling program, and/or send status and information back to the calling program.

In addition, detailed information about porting the Jam Player is provided with the Jam Player source code. Contact Altera Applications at (800) 800-EPLD for more information.

Executing the Jam Player

The Jam Player provides the flexibility to specify which ISP functions will be performed by using flags that are passed to the Jam Player at the time of execution. Jam Player usage takes the following form:

```
jam[ -h ] [ -v ] [ -p<Hexadecimal parallel port address> ] [ -m<Memory size in bytes> ] -d<Initialization list> <Jam File>
```

Flags with brackets ([]) are optional. The Jam Player can process only one Jam File at a time. [Table 1](#) describes the function of each flag.

Flag	Definition	Function
-h (1)	Help	Reports the Jam Player version.
-v (1)	Verbose	Reports status and error messages with detailed real-time information.
-d	Initialize	Tells the Jam Player what functions to perform.
-p (1)	Port	Specifies the parallel port address where the Jam Player should send data.
-m (1)	Memory	Specifies the amount of memory the Jam Player can use.

Note:

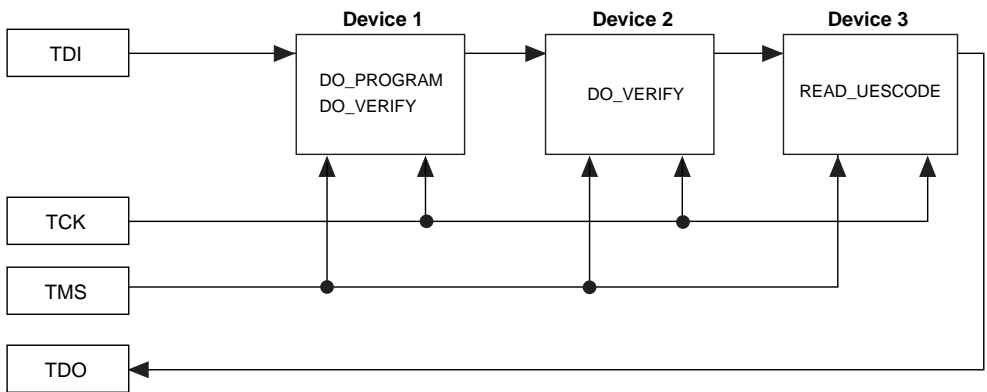
(1) This flag is optional.

When using the `-d` flag, certain variables from the initialization list are provided for initializing the Jam Player. [Table 2](#) describes the variable set used after the `-d` flag.

<i>Table 2. -d Flag Variable Names & Their Functions</i>		
Variable Name	Value	Function
DO_ERASE	0	Do not perform a bulk erase.
	1	Perform a bulk erase.
DO_BLANKCHECK	0	Do not check the erased state of the device.
	1	Check the erased state of the device.
DO_PROGRAM	0	Do not program the device.
	1	Program the device.
DO_VERIFY	0	Do not verify the device.
	1	Verify the device.
READ_UESCODE	0	Do not read the JTAG UES code.
	1	Read and report the UES code.
DO_SECURE	0	Do not set the security bit.
	1	Set the security bit.

Variables that are not initialized after a `-d` flag are set to 0. The order of variables in the initialization list is not important. Each variable set after the `-d` flag applies to the device(s) specified in the Jam File. [Figure 7](#) shows a JTAG chain with three devices that are each targeted for a specific function(s) by the Jam File.

Figure 7. JTAG Chain with Three Devices



In **Figure 7**, only the `DO_PROGRAM` and `DO_VERIFY` variables are initialized to 1 at the time the Jam Player is executed, device 1 is programmed and verified; device 2 is verified; and no operation is performed on device 3.

The following text typed at the command line causes the Jam Player to report error and information messages, program and verify the device(s), limit memory to 1 Mbyte, and write/read data through the parallel port at base address 0x378 for the Jam File **chiptrip.jam**:

```
jam -v -p378 -m1000000 -dDO_PROGRAM=1 -dDO_VERIFY=1  
chiptrip.jam
```

Jam Player Memory Usage

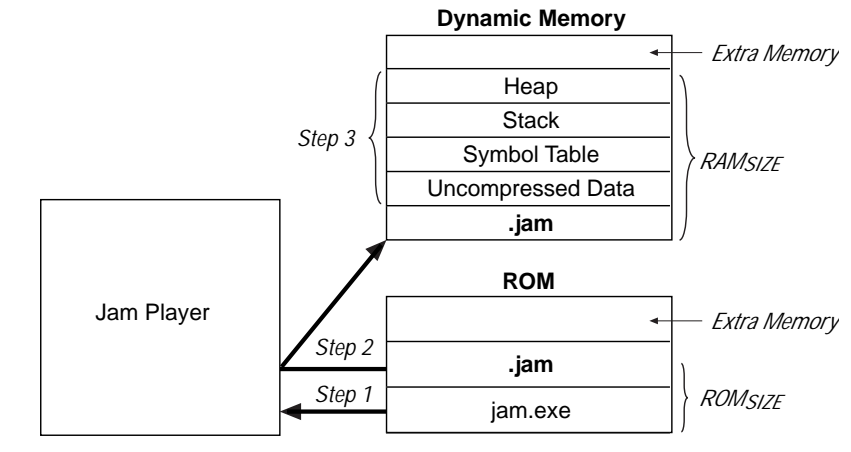
The Jam Player uses memory in a predictable manner, which simplifies in-field upgrades by confining updates to the Jam File. The Jam Player memory uses ROM and dynamic memory (RAM). ROM is used to store the Jam Player binary and the Jam File. Dynamic memory is used when the Jam Player is called.

The Jam Player uses memory in the following steps:

1. The Jam Player is called by the controlling software.
2. The Jam Player reads the Jam File into dynamic memory.
3. The Jam Player inflates the compressed data and initializes memory for the symbol table and stack.

Figure 8 shows how the Jam Player uses memory.

Figure 8. Jam Player Memory



When the Jam Player is called, it reads the entire Jam File into a buffer, and inflates all compressed programming data contained within the Jam File. Next, it initializes the symbol table, stack, and heap. The symbol table stores variable and label names declared in the Jam File. The stack stores the information used for executing FOR loops, CALL statements, and PUSH statements. The heap is temporary memory for evaluating arithmetic expressions and stores padding data. Once the symbol table, stack, and heap are initialized, the Jam Player is ready to parse and execute the Jam File. While the Jam Player processes the Jam File, the stack and heap expands and shrinks as commands are encountered. During this process, the amount of memory used by the Jam File, the uncompressed data, and the symbol table remains constant.

Estimating ROM Usage

The following equation is used to estimate the maximum amount of ROM required to store the Jam Player and Jam File:

$$ROM_{SIZE} = \text{Jam Player size} + \text{Jam File size}$$

The Jam File size can be broken into two categories: the amount of memory required to store just the programming data, and the space required for the programming algorithm. To describe the Jam File size as a function of the number of targeted devices, use the following equation:

$$\text{Jam File size} = \text{Alg} + \sum (\text{for } N = 1 \text{ to } k) \text{ Data}$$

Where:

Alg = Space used by algorithm
 Data = Space used by compressed programming data
 N = Index representing family type(s) being targeted
 k = Number of targeted devices in the chain

This equation provides a Jam File size estimate that may vary by $\pm 10\%$, based on device utilization. When device utilization is low, Jam File sizes tend to be smaller. Compressed algorithms will likely find repetitive data for devices with lower logic utilization.

The equation also states that the algorithm size will stay constant for a device family. However, the programming data will grow slightly as more devices are targeted. For a device family, the increase in Jam File size (due to the data component) will be linear.

Tables 3 and 4 show algorithm and data constants for certain Altera devices.

<i>Table 3. Algorithm Constants</i>	
Device Family	Typical Algorithm Size (Kbytes)
MAX 7000S and MAX 7000A	30
MAX 9000, MAX 7000A, and MAX 7000S	39
MAX 9000	26
FLEX 10K, MAX 7000A, and MAX 7000S (1)	19
FLEX 10K, MAX 9000, MAX 7000A, and MAX 7000S (1)	39
FLEX 10K	4

Note:

- (1) When configuring these FLEX 10K devices and programming these MAX 9000, MAX 7000A, or MAX 7000S devices, the algorithm adds negligible memory usage.

Device	Typical Data Size per Device (Kbytes)
EPM7032S, EPM7032A	4
EPM7064S, EPM7064A	7
EPM7128S, EPM7128A	10
EPM7160S, EPM7160A	14
EPM7192S, EPM7192A	19
EPM7256S, EPM7256A	21
EPM9320, EPM9320A	26
EPM9480, EPM9480A	39
EPM9560, EPM9560A	40
EPF10K10	16
EPF10K20	31
EPF10K30	36
EPF10K40	40
EPF10K50	49
EPF10K70	68
EPF10K100	98

Once the Jam File is estimated, the Jam Player size can be estimated using [Table 5](#).

Processor		Typical Size (Kbytes)
16-bit	Pentium/486 (16-bit DOS) using the ByteBlaster™ parallel port download cable	105
32-bit	Pentium/486 (16-bit DOS) using the BitBlaster™ serial download cable	115

Estimating Dynamic Memory Usage

Use the following equation to estimate the maximum amount of dynamic memory (RAM) required by the Jam Player:

$$\text{RAM}_{\text{SIZE}} = \text{Jam File size} + \sum_{N=1}^k \text{ACA variable } N + \text{Symbol table size}$$

The Jam File size is determined by a single- or multiple- device equation (see “[Estimating ROM Usage](#)” on page 180).

The ACA variable is the size of the N th compressed array when inflated, where k is the total number of ACA compressed arrays within the Jam File. To determine the ACA variable size, look in the Jam File’s Variable Declaration/Initialization section. The size of each array is stated within brackets of the Variable Declaration statement. For example:

```
BOOLEAN A21[104320] = ACA mB300u...
```

In this example, the ACA variable will be 104,320 bits long when inflated.

The Symbol table size is determined by the following equation:

Symbol table size = 48 bytes \times JAM_C_MAX_SYMBOL_COUNT

48 bytes is the size of a variable or label name. JAM_C_MAX_SYMBOL_COUNT is defined in the **jamdefs.h** file, and the default value is 1,021. However, most Jam Files will use a maximum of 400 variable and label names. To conserve memory, you should place the default value around 400.



The memory requirement for the stack and heap are negligible, with respect to the total amount of memory used by the Jam Player. The maximum depth of the stack is set by the JAMC_MAX_NESTING_DEPTH constant in the **jamdefs.h** file.

Estimating Memory: Example

The following example uses a Motorola 68K processor to program an EPM7128S and EPM7064S device in an IEEE 1149.1 (JTAG) chain. To determine memory usage, you must first determine the amount of ROM required and then estimate the RAM usage. You can use the following steps to calculate the amount of dynamic memory (RAM) required by the Jam Player.

1. Determine the Jam File size. The Jam File size is estimated by using the multi-device equation:

$$\text{Jam File size} = \text{Alg} + \sum (\text{for } N = 1 \text{ to } k) \text{ Data}$$

Where:

$$\text{Alg} = 19 \text{ Kbytes}$$

$$\text{Data} = \text{EPM7064S Data} + \text{EPM7128S Data} = 7 + 10 = 17 \text{ Kbytes}$$

Thus, the Jam File size equals 36 Kbytes.

2. Estimate the Jam Player size. In this example, 115 Kbytes will be used for the binary size estimation. Use the following equation to determine the amount of ROM needed:

$$\text{ROM}_{\text{SIZE}} + \text{Jam File size} + \text{Jam Player size}$$

For this example, the $\text{ROM}_{\text{SIZE}} = 151 \text{ Kbytes}$

3. Next, you should estimate the RAM usage. First, determine the amount of memory needed to inflate the compressed data. The ACA variables are (open the Jam File to find the compressed arrays):

```
BOOLEAN A21[150120] = ACA Db400u...
```

```
BOOLEAN A22[97640] = ACA j_200u...
```

Inflating the compressed data will use the following amount of RAM:

$$\frac{(150,120 + 97,640) \text{ bits}}{8 \frac{\text{bits}}{\text{byte}}} = 30 \text{ Kbytes}$$

4. With `JAMC_MAX_SYMBOL_COUNT` defined as 400, the Symbol table size can be calculated as follows:

$$48 \text{ bytes} \times 400 = 19 \text{ Kbytes}$$

5. Therefore, the total dynamic memory usage is calculated as follows:

$$\text{RAM}_{\text{SIZE}} = 36 \text{ Kbytes} + 30 \text{ Kbytes} + 19 \text{ Kbytes} = 85 \text{ Kbytes}$$

As shown in this example, Altera's Jam implementation will utilize more RAM than ROM, which is desirable because RAM is cheaper. Although in-system programming can be implemented with smaller memory utilization, the trade-off is easy to upgrade. Likewise, the overhead associated with easy upgrade becomes a lesser factor as larger number of devices are programmed. In most applications, easy upgrades outweigh the memory costs.

Low-Level Jam Player Operation

The Jam Player provides a low-level interface for manipulating the IEEE 1149.1 (JTAG) Test Access Port (TAP) state machine. The TAP controller is a 16-state state machine that is clocked on the rising edge of TCK , and uses the TMS pin to control JTAG operation in a device. [Figure 9](#) shows the flow of a IEEE 1149.1 (JTAG) TAP controller state machine.

Figure 9. JTAG TAP Controller State Machine

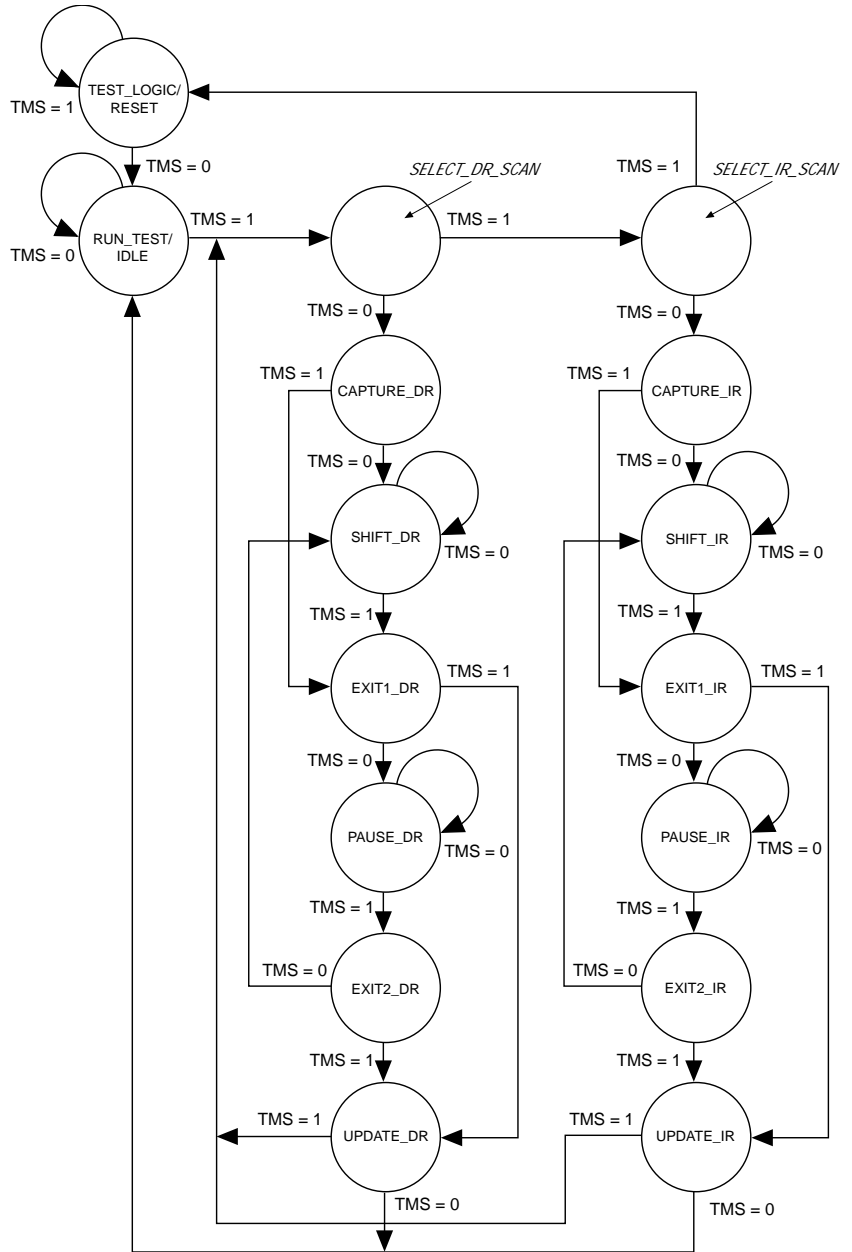
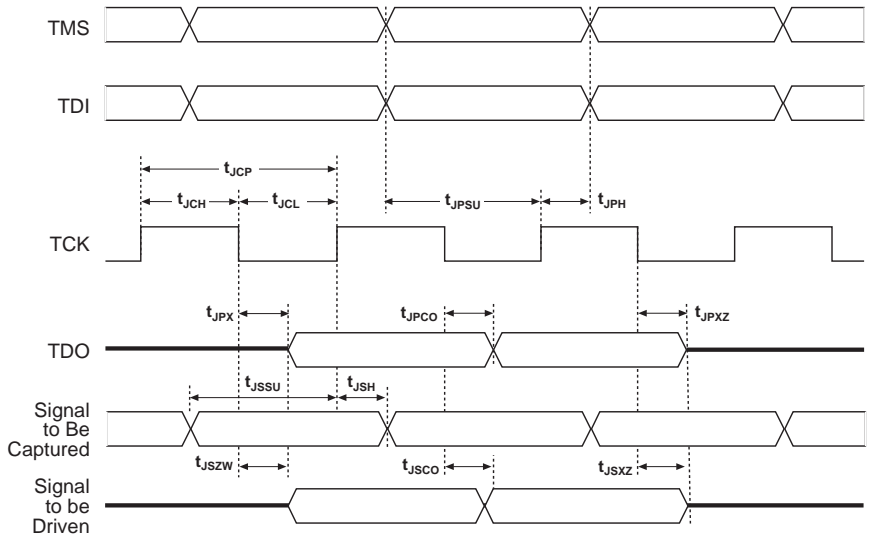


Table 6 shows the TAP state machine timing specifications. These timing parameters are the same as those specified in the IEEE 1149.1 (JTAG) specification.

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

Figure 10 illustrates waveforms that correspond to each timing parameter. The system designer should ensure proper Jam Player operation for any system, using these timing parameters.

Figure 10. JTAG Waveforms



While the Jam Player provides a low-level driver that manipulates the TAP controller, the Jam File provides the high-level intelligence needed to program a given device. All Jam instructions that force JTAG data to the device will involve moving the TAP controller through either the data register leg of the state machine or the instruction register leg. For example, loading a JTAG instruction involves moving the TAP controller to the `SHIFT_IR` state and shifting the instruction into the instruction register via the TDI pin. Next, the TAP controller is moved to the `RUN_TEST/IDLE` state where a delay is implemented to allow the instruction time to be latched. This process is identical for data register scans, except the data register leg of the state machine is traversed.

The high-level Jam instructions are the `DRSCAN` instruction for scanning the JTAG data register, the `IRSCAN` instruction for scanning the instruction register, and the `WAIT` command that causes the state machine to sit idle for a specified period of time. Each leg of the TAP controller is scanned repeatedly, according to instructions in the Jam File, until all of the targeted devices are programmed.



For more information on Jam instructions, see the [Jam Programming & Test Language Specification](#).

Figure 11 illustrates the functional behavior of the Jam Player when it parses the Jam File. Upon encountering a DRSCAN, IRSCAN, or WAIT instruction, the Jam Player generates the proper data on TCK, TMS, and TDI to complete the instruction. The flow diagram shows branches for the DRSCAN, IRSCAN, and WAIT instructions. Although the Jam Player supports other instructions, they are omitted from the flow diagram for simplicity.

Figure 11. Low-Level Jam Player Flow Diagram (Part 1 of 2)

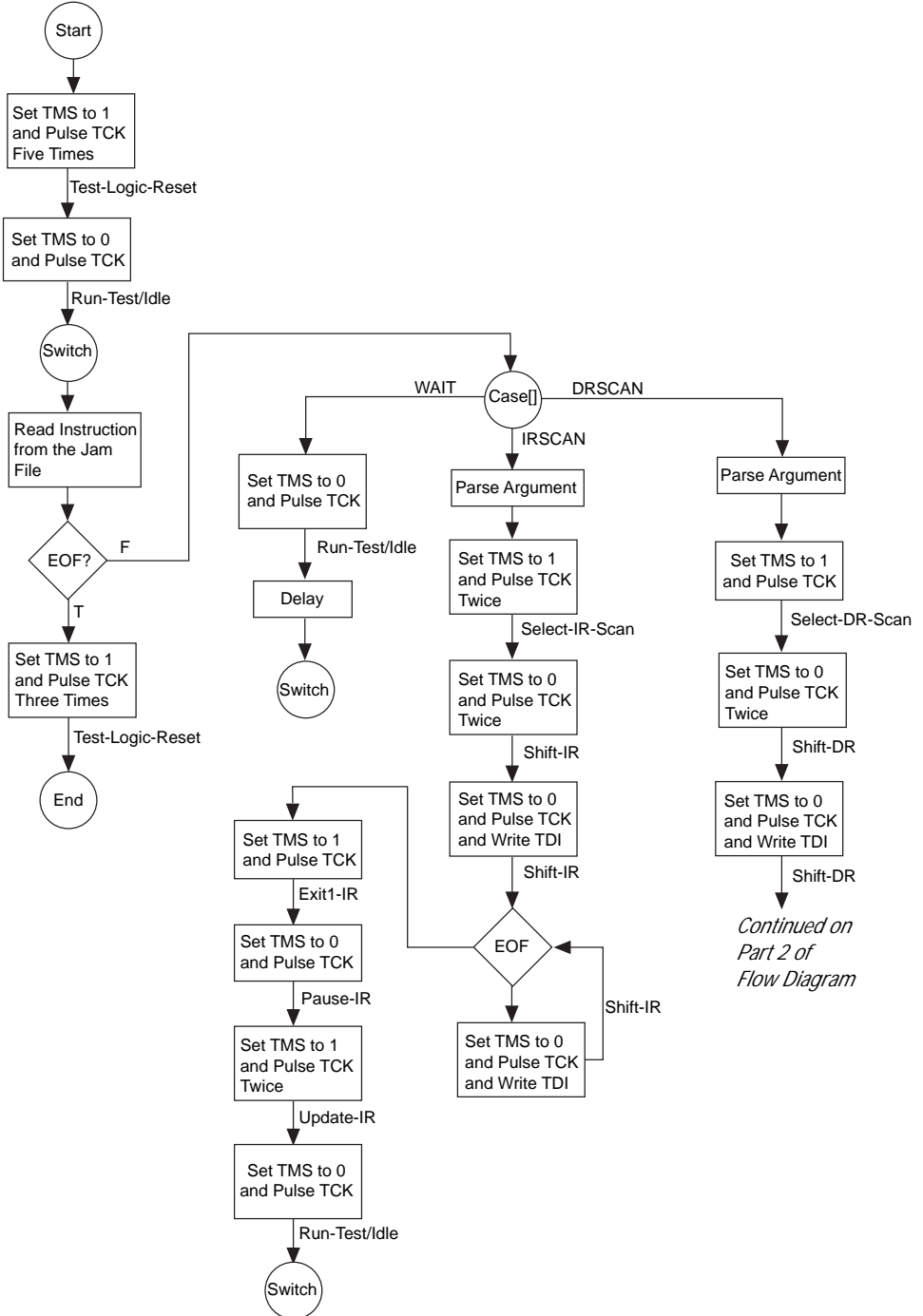
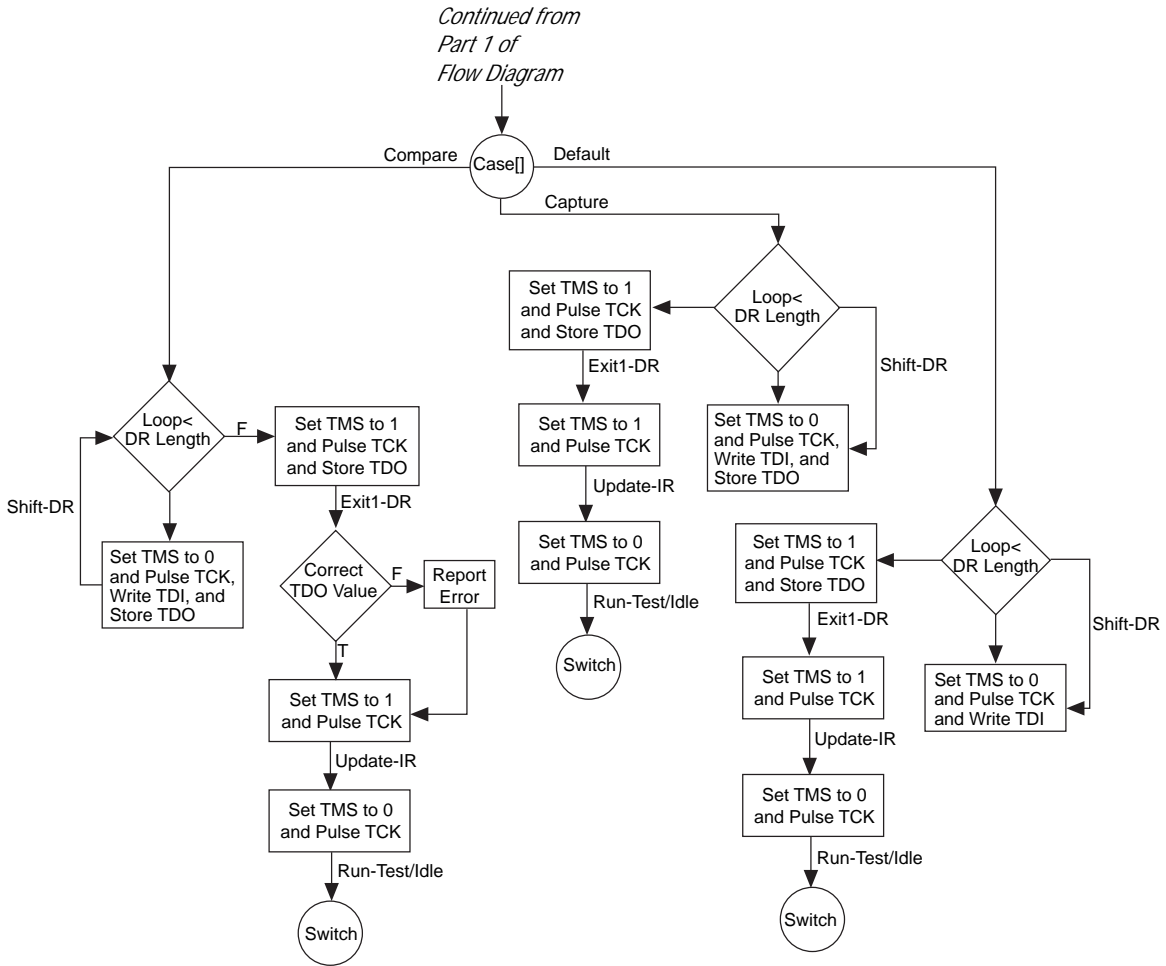


Figure 11. Low-Level Jam Player Flow Diagram (Part 2 of 2)



Conclusion

To achieve the benefits of in-system programming via an embedded processor, the Jam Programming and Test Language successfully meets necessary system requirements such as small file sizes, ease of use, and platform independence. Using the Jam language for in-system programming via an embedded processor supports in-field upgrades, easy design prototyping, and fast production. These benefits lengthen the life and enhance the quality and flexibility of the end products, and they can also reduce device inventories by eliminating the need to stock and track programmed devices.

Copyright © 1995, 1996, 1997, 1998 Altera Corporation, 101 Innovation Drive, San Jose, CA 95134, USA, all rights reserved.

By accessing this information, you agree to be bound by the terms of Altera's Legal Notice.