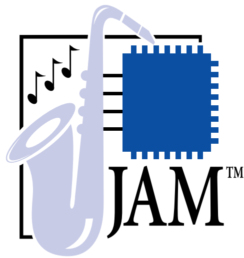
Using the Jam Language for ISP & ICR via an Embedded Processor

November 1998, ver. 3.01 — Application Note 88

Introduction

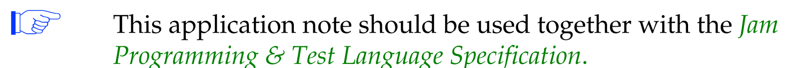


In-system programming and in-circuit configuration through an embedded processor—available in MAX® 9000, MAX 9000A, MAX 7000A, MAX 7000AE, MAX 7000S and FLEX® 10K devices—enables easy design prototyping, streamlines production, and allows quick and efficient in-field upgrades. Devices that support in-system programmability (ISP) and in-circuit reconfigurability (ICR) are easily upgraded in the field by downloading new configurations using ROM, FLASH cards, modems, or other data links. Design changes are 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. You can download the Jam Player from the Jam web site at **http://www.jamisp.com**. The Jam source code is executed directly by an interpreter program, without being compiled into binary executable code (see "Embedded Programming with the Jam Language" on page 4). The Jam source code, or Jam Byte-Code File (**.jbc**), 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 using an embedded processor, including:

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


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

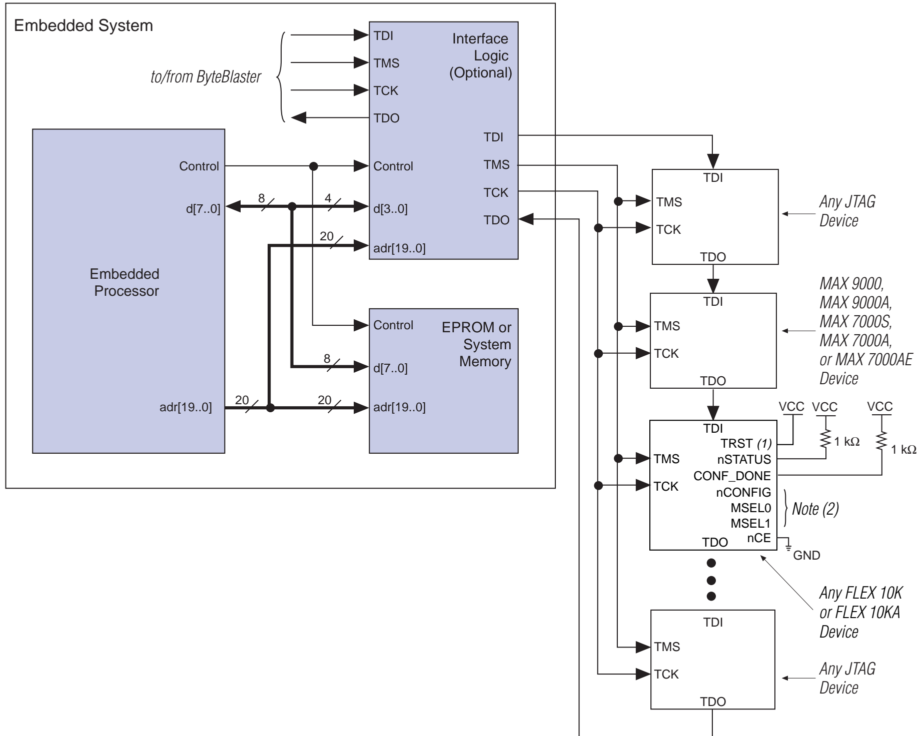
Embedded System Configuration & Requirements

To achieve the benefits of ISP or ICR, an embedded system must be able to program or configure 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).

A-AN-088-03.01

During in-system programming or in-circuit reconfiguration, the embedded processor transfers programming or configuration 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



Notes:

(1) Because FLEX 10KA devices in 144-pin thin quad flat pack (TQFP) packages do not have a `TRST` pin, you can ignore this connection.

(2) The `nCONFIG`, `MSEL0`, and `MSEL1` pins should be connected to support a FLEX configuration scheme. If only JTAG configuration is used, connect `nCONFIG` to V_{CC}, and connect `MSEL0` and `MSEL1` to ground.

The embedded processor is connected to an 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 may choose to install a 10-pin ByteBlaster™ header on the board to allow the MAX+PLUS® II software and ByteBlaster™ or ByteBlasterMV™ parallel port download cable to access and verify the JTAG chain.

For more information on the ByteBlaster or ByteBlasterMV parallel port download cables, see the *ByteBlaster Parallel Port Download Cable Data Sheet* or the *ByteBlasterMV Parallel Port Download Cable Data Sheet*.

Figure 2 illustrates the embedded system's interface logic.

Figure 2. Interface Logic *Note (1)*

Note:

(1) The `TDI`, `TMS`, and `TCK` signals fed back through `d[3..0]` are optional. These signals are used for diagnostic purposes only.

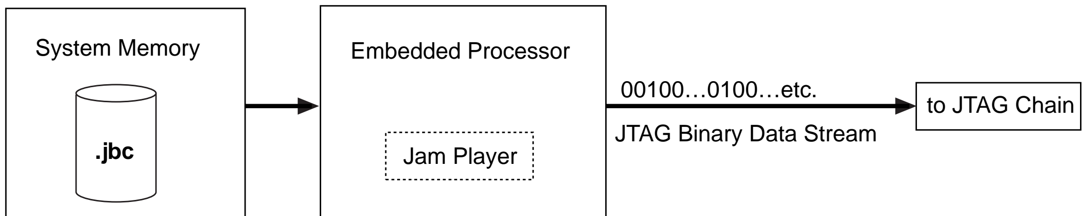
The interface logic activates when it receives the proper address and control signals from the embedded processor. The registers then synchronize the timing of the TDI, TCK, and TMS signals and drive the output pins through a 74157 multiplexer. The multiplexer allows the ByteBlaster or ByteBlasterMV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

Embedded Programming with the Jam Language

The Jam language has two parts: the Jam File and the Jam Player. A Jam File, which contains all the information to program ISP-capable devices, is generated from the MAX+PLUS II development software and is stored in system memory. 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 updates may only be needed by and are confined to the Jam File, the Jam Player requires no changes and is used to program any vendor's device.

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

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



The Jam File

Jam Files are compact files containing programming data and algorithm information needed to program any device through the IEEE Std. 1149.1 JTAG port. Altera supports two separate implementations of the Jam File: the Jam Byte-Code File (**.jbc**) and the ASCII Jam File (**.jam**). The JBC File is a binary file, while the Jam File is text only. Altera recommends using the JBC File for all new designs because it provides smaller file sizes and faster programming times. Altera will continue to support the ASCII Jam File format for backward compatibility.

Figure 4 describes how to generate a JBC File for in-system programming using the MAX+PLUS II software.

A similar flow is possible using development software from other vendors.

Figure 4. Generating a JBC File Using the MAX+PLUS II software

Compile your designs with the MAX+PLUS II software.

Compile Design 1
Compile Design 2
Compile Design *N*

The MAX+PLUS II software generates the Programmer Object File.

.pof
.pof
.pof

The MAX+PLUS II software converts the POFs into a JBC File.

MAX+PLUS II

.jbc

Use utility to create a Hex File or similar programming file.

Utility

.hex

Program Hex File or other programming file into EPROM/ FLASH devices.

EPROM/FLASH Device

Initialization Conventions

The MAX+PLUS II software generates JBC Files that use Jam conventions for initialization. This section describes special conventions that are supported by the Jam language.

DO_PROGRAM

The `DO_PROGRAM` variable determines whether a device should be programmed. By setting `DO_PROGRAM` to `1`, the Jam Player performs the silicon ID check, 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 uses a concurrent programming algorithm (i.e., programming data shifts through all devices of the same family at the same time).

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

DO_VERIFY

The `DO_VERIFY` variable tells the JBC File to verify the device. By setting `DO_VERIFY` 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 JBC File to completely erase the device. 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 is properly bulk erased before programming. The `DO_BLANKCHECK` variable verifies that all data is erased.

DO_SECURE

The `DO_SECURE` initialization variable causes the security bit to be programmed for those devices that have the security bit turned on in the corresponding POF. If the POF does not have the security bit turned on, initializing this variable to `1` has no effect.

DO_SECURE_ALL

The `DO_SECURE_ALL` initialization variable programs the security bit regardless of whether the security bit is turned on in the corresponding POF.

DO_READ_UES

The user electronic signature (UES) is useful for tracking design revisions. The `DO_READ_UES` variable tells the Jam Player to read the UES code out of the targeted device and report it. The Jam Player reads the UES code when the `DO_READ_UES` variable is initialized to `1` (i.e., `-dDO_READ UES=1`). The Jam Player returns the value of the UES code through the `jbi_export()` routine, using `printf`. A JTAG chain with two Altera devices supporting UES has the following output:

```
jbi -p378 -dDO_READ_UES=1 Altera.jbc
```

`FFFA` and `FFFB` are the 16-bit UES codes represented in hexadecimal format.

DO_CONFIGURE

The `DO_CONFIGURE` variable determines whether an SRAM-based device should be configured. Setting this variable to `1` when applying a JBC File that has been generated for a FLEX 10K device results in configuration.

For more information on initialization conventions, see the *Jam Programming & Test Language Specification.*

JBC File Structure

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

Figure 5. Structure of a JBC File

Variable Declaration/Initialization Section

- Compressed Program/Verify Data
- Initialized Variables

Algorithm Section

- Check Silicon ID
- Blank-Check (Optional)
- Bulk Erase & Program (Optional)
- Read UES (Optional)
- Verify (Optional)
- Exit Code

The Variable Declaration and Initialization Section contains the declared variables that are used in the JBC File. The variables can also be initialized to specific values. In a BOOLEAN array, the variable is initialized as a compressed data array, using the advanced compression algorithm (ACA) format. Variables of other types can be declared and initialized in this section; initialization of these variables is optional.

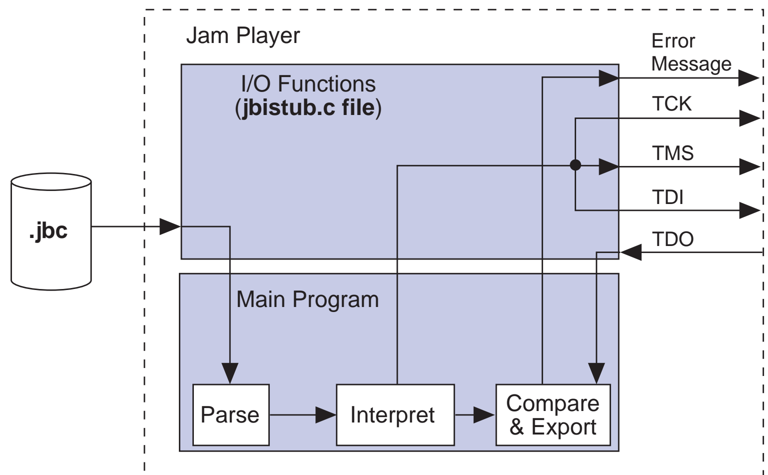
For more detail on the ACA data array format, see the *Jam Programming & Test Language Specification*.

The Algorithm Section contains 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 and verify) that are performed on the targeted device(s).

The Jam Player

The Jam Player is a C program that parses the JBC File, interprets each Jam instruction, and reads and writes data to and from the JTAG chain. The source code is written for compilation using 16- or 32-bit processors. 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 10 for more information). Because each application has unique requirements, the Jam Player source code was designed to be easily modified. 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 modification. Only the I/O functions, which are contained in the **jbistub.c** file, need to be modified for a given 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 JBC File and generates a binary data stream for device programming. This structure confines all upgrades to the JBC File, and allows the Jam Player to adapt to any system architecture.

Two types of Jam Players are available, for use with ASCII Jam Files and JBC Files. Table 1 documents the compatibility of the Jam Player with the various file options.

Table 1. Jam Player Compatibility

Jam File Type	Jam Player Type	
	ASCII (jam.exe)	Byte Code (jbi.exe)
ASCII	✓	
Byte Code		✓

JBC Files are applied using the Jam Byte-Code Player, while ASCII Jam Files are applied with the ASCII Jam Player. This separation of functionality reduces the file size of the Jam Player binaries. Altera recommends using JBC Files in all cases except where existing projects require the use of ASCII Jam Files.

Customizing the Jam Player

The Jam Player is structured to simplify customization based on platform requirements and applications. All file I/O and port configurations are changed by editing the **jbistub.c** file. As an input, the **jbistub.c** file retrieves data from the JBC File and/or reads shifted data from `TDO`. As an output, the **jbistub.c** file sends processed JTAG data to the three JTAG pins: `TDI`, `TMS`, and `TCK`, sends formatted error and information messages back to the calling program, and/or sends status and information back to the calling program.

The readme file included with Jam Player source code provides detailed information about porting the Jam Player. 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 are performed. At the time of execution, command-line options are passed to the Jam Player. Jam Player usage takes the following form:

`jam` [`–h`] [`–v`] [`–p`<*Hexadecimal parallel port address*>]
`–d`<*Initialization variable*> [`–d` <*Initialization variable*>] <*Jam File name*>

Command-line options in brackets ([]) are optional. The Jam Player processes only one JBC File at a time. Table 2 describes the function of each command-line option.

Table 2. Jam Player Command-Line Options

Command-Line Option	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 which functions to perform.
`-p`, *(1)*	Port	Specifies the parallel port address where the Jam Player should send data.
`-s`, *(1)*	Port	Specifies the serial port address where the Jam Player should send data.
`-1`, *(1)*	Cable	Use this ISP download cable. The default uses the ByteBlaster parallel port download cable.

Note:
(1) This command-line option is optional.

When using the `–d` command-line option, certain variables from the initialization list are provided for initializing the Jam Player. Table 3 describes the variables that can be used after the `–d` command-line option.

Table 3. -d Command-Line Option Variable Names & Their Functions

Variable Name	Value	Function
`DO_ERASE`	0	Do not erase the device.
	1	Erase the device.
`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.
`DO_SECURE`	0	Do not set the security bit.
	1	If the corresponding POF sets the security bit, set the security bit.
`DO_SECURE_ALL`	0	Do not set the security bit.
	1	Set the security bit, overriding the POF setting.
`DO_READ_UES`	0	Do not read the UES code.
	1	Read and report the UES code.
`DO_CONFIGURE`	0	Do not configure the device.
	1	Configure the device.

The initialization variables apply to all Altera devices that are targeted in the JTAG chain. When the JBC File is generated, targeted devices are assigned a configuration file. To get to the dialog box that is used to generated JBC Files, perform the following steps:

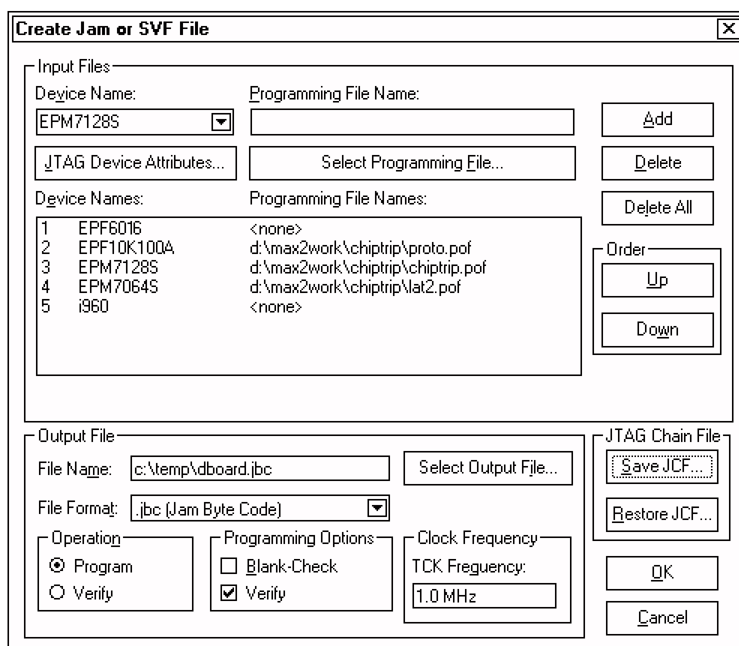
1. Run the MAX+PLUS II software.

2. Choose **Programmer** (MAX+PLUS II menu).

3. Choose **Create Jam or SVF File** (File menu).

Figure 7 shows the dialog box that specifies which JBC Files are generated by the MAX+PLUS II software.

Figure 7. Generating a JBC File for a Multi-Device JTAG Chain



The devices showing <*none*> next to them are bypassed during configuration. Initialization variables that are passed to the Jam Player are applied to each device that has a programming file name next to it. For example, the following command configures the EPF10K100A device and concurrently programs and verifies the EPM7064S and EPM7128S devices, while bypassing the EPF6016 device and the i960 processor:

```
jbi -v -p378 -dDO_CONFIGURE=1 -dDO_PROGRAM=1 -dDO_VERIFY=1
    dboard.jbc
```

Jam Player Memory Usage

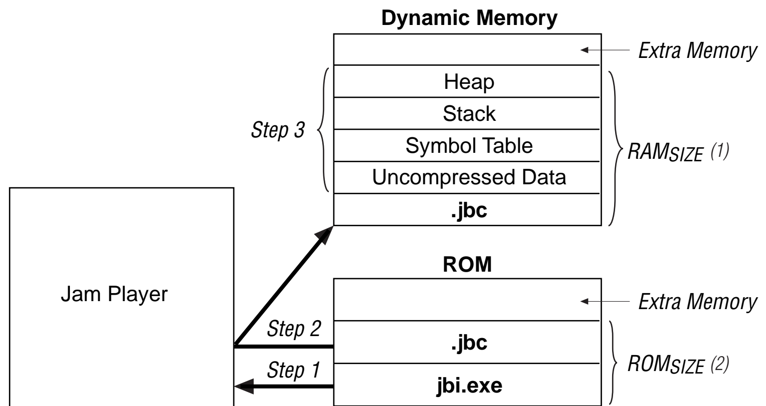
The Jam Player binary code and the JBC File are stored in non-volatile system memory. Once the Jam Player is called, it uses dynamic RAM (DRAM) to perform all of the tasks for device programming. Field upgrades are simplified by confining updates to the Jam File.

The Jam Player uses memory as follows:

1. The controlling software calls the Jam Player.
2. The Jam Player reads the JBC File into DRAM.
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



Notes:
(1) The RAM_{SIZE} is the maximum amount of DRAM required by the Jam Player.
(2) ROM_{SIZE} is the maximum amount of ROM required to store the Jam Player and JBC File.

When the Jam Player is called, it reads the entire JBC File into a buffer, and decompresses programming data, contained within the JBC File. In some cases, a JBC File can be generated such that the Jam Player does not need to decompress any data.

Next, the Jam Player initializes the symbol table, stack, and heap. The symbol table stores variable and label names declared in the JBC File. The stack is used for executing `FOR` loops, `CALL` statements, and `PUSH` statements. The heap is temporary memory for evaluating arithmetic expressions and storing padding data.

Once the symbol table, stack, and heap are initialized, the Jam Player is ready to parse and execute the JBC File. While the Jam Player processes the JBC File, the stack and heap expand and shrink as commands are encountered. During this process, the amount of memory used by the JBC File, the uncompressed data, and the symbol table remains constant.

Estimating ROM Usage

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

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

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:

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

where:

Alg	=	Space used by algorithm
Data	=	Space used by compressed programming data
k	=	Index representing family type(s) being targeted
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 and compression algorithms are 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 for all possible combinations of Altera device families that support the Jam language.

Table 4. Algorithm File Size Constants for Altera Device Families

Device Family	Typical Algorithm Size (Kbytes)	
	ASCII Jam File	JBC File
MAX 7000S, MAX 7000A	22	18
MAX 9000	26	21
MAX 9000, MAX 7000S, MAX 7000A	42	35
FLEX 10K, MAX 7000S, MAX 7000A	42	35
FLEX 10K, MAX 9000, MAX 7000A, MAX 7000S, *Note (1)*	42	35
FLEX 10K	6	4
MAX 7000AE	*Note (2)*	*Note (2)*

Note:

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

(2) For information on MAX 7000AE devices, contact Altera Applications at (800) 800-EPLD.

Table 5 shows data constants for all possible combinations of Altera devices that support the Jam language for ISP.

Table 5. Data Constants *Note (1)*

Device	Typical Data Size (Kbytes)		
	Compressed		Uncompressed
	ASCII Jam	JBC	JBC, *Note (2)*
EPM7032S	1	4	4
EPM7064S	3	9	9
EPM7128S, EPM7128A	6	5	20
EPM7160S	9	6	27
EPM7192S	10	7	34
EPM7256S, EPM7256A	14	10	49
EPM9320, EPM9320A	20	15	59
EPM9400	26	19	70
EPM9480	25	18	73
EPM9560, EPM9560A	27	20	96
EPF10K10, EPF10K10A	11	8	14
EPF10K20	23	17	28
EPF10K30, EPF10K30A,EPF10K30E	39	28	46
EPF10K40	51	37	61
EPF10K50, EPF10K50E, EPF10K50V	60	44	76
EPF10K70	95	69	109
EPF10K100, EPF10K100A, EPF10K100B, EPF10K100E	130	95	146
EPF10K130E, EPF10K130V	177	128	194
EPF10K200E	196	143	322
EPF10K250A, EPF10K250E	245	179	403

Notes:
(1) For information on MAX 7000AE devices, contact Altera Applications at (800) 800-EPLD.
(2) 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 6.

Table 6. Jam Player Binary Sizes

Processor		Typical Size (Kbytes)	
Size	**Description**	**ASCII Jam Player**	**JBC Player**
16-bit	Pentium/486 using the BitBlaster™, ByteBlaster, or ByteBlasterMV parallel port download cables	105	62
32-bit	Pentium/486 using the BitBlaster™, ByteBlaster, or ByteBlasterMV serial download cables	115	68

Estimating Dynamic Memory Usage

Use the following equation to estimate the maximum amount of DRAM required by the Jam Player:

$$\mathrm{RAM}_{\mathrm{SIZE}} = \text{JBC File Size} + \sum_{k=1}^{N} \text{ACA variable } k$$

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

The ACA variable is the size of the kth compressed array when inflated, where N is the total number of ACA compressed arrays within the JBC File. To determine the ACA variable size, look in the ASCII Jam File's Variable Declaration/Initialization section. The size of each array (in bits) is stated within brackets of the Variable Declaration statement. For example:

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

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

The memory requirements for the stack and heap are negligible, with respect to the total amount of memory used by the Jam 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 Motorola 68000 processor to program an EPM7128S and EPM7064S device in an IEEE Std. 1149.1 JTAG chain via a JBC File. 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 multi-device equation to estimate the JBC File size:

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

where:

Alg = 18 Kbytes
$Data$ = EPM7064S $Data$ + EPM7128S $Data$ = 9 + 5 = 14 Kbytes

Thus, the JBC File size equals 32 Kbytes.

2. Estimate the Jam Byte-Code Player size. This example uses 62 Kbytes for the binary size estimation. Use the following equation to determine the amount of ROM needed:

ROM_{SIZE} = JBC File Size + Jam Player Size

ROM_{SIZE} = 94 Kbytes.

3. Estimate the RAM usage with the following equation:

$$RAM_{SIZE} = 32\text{ Kbytes} + \sum_{k=1}^{N} \text{ACA variable } k$$

The ACA variables are as follows (open the ASCII Jam File to find the compressed arrays):

```
BOOLEAN A21[150120] = ACA Db400u...
BOOLEAN A22[97640] = ACA j_200u...
```

Inflating the compressed data uses the following amount of RAM:

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

Calculate the total DRAM usage as follows:

RAM_{SIZE} = 32 Kbytes + 30 Kbytes = 62 Kbytes

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

Jam Player Operation

The Jam Player provides an interface for manipulating the IEEE Std. 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 an IEEE Std. 1149.1 TAP controller state machine.

Figure 9. JTAG TAP Controller State Machine

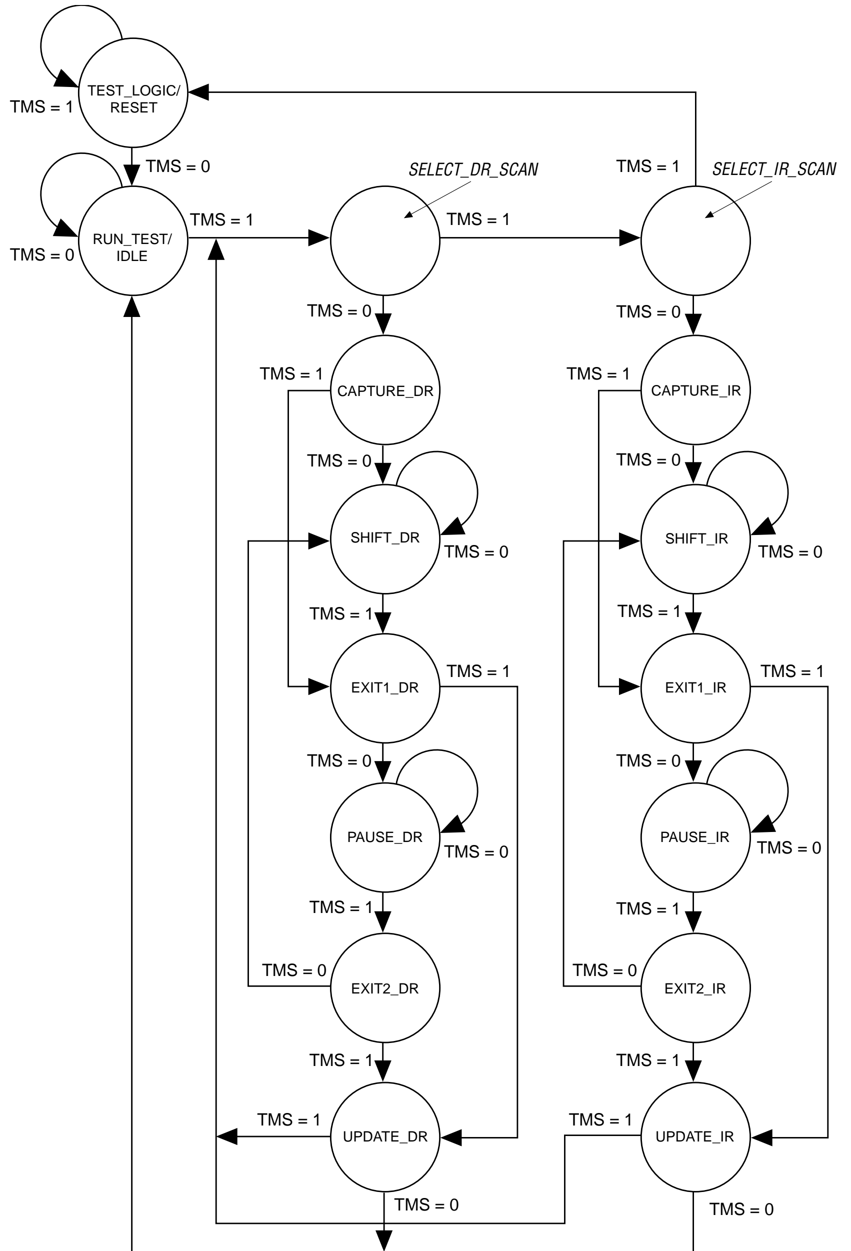


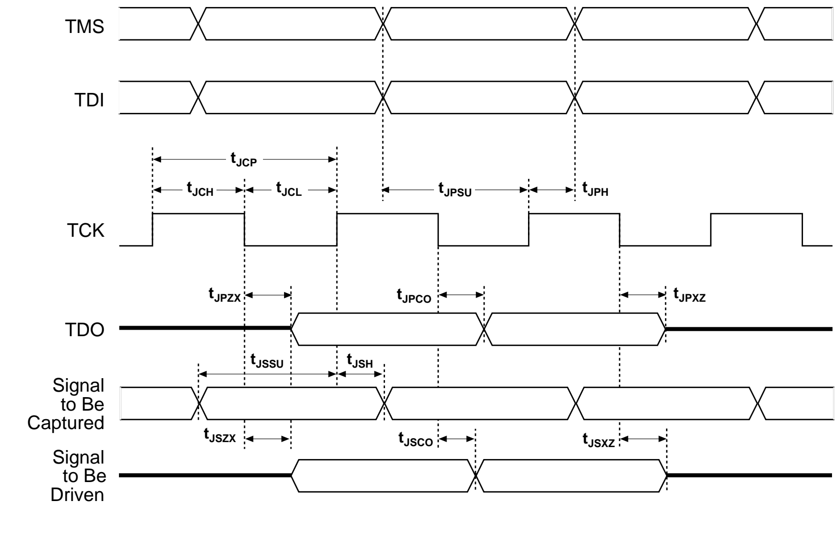
Table 7 shows the TAP state machine timing specifications. These timing parameters are the same as those specified in the IEEE Std. 1149.1 specification.

Table 7. IEEE Std. 1149.1 Timing Parameters

Symbol	Parameter	MAX 9000		MAX 7000A		MAX 7000AE		MAX 7000S		FLEX 10K		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

Figure 10 illustrates waveforms that correspond to each timing parameter. By using these timing parameters, you can ensure proper Jam Player operation for any system.

Figure 10. JTAG Waveforms



While the Jam Player provides a driver that manipulates the TAP controller, the JBC File provides the high-level intelligence needed to program a given device. All Jam instructions that send JTAG data to the device involves 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 that 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 JBC File, until all of the target 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 JBC 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. Jam Player Flow Diagram (Part 1 of 2)

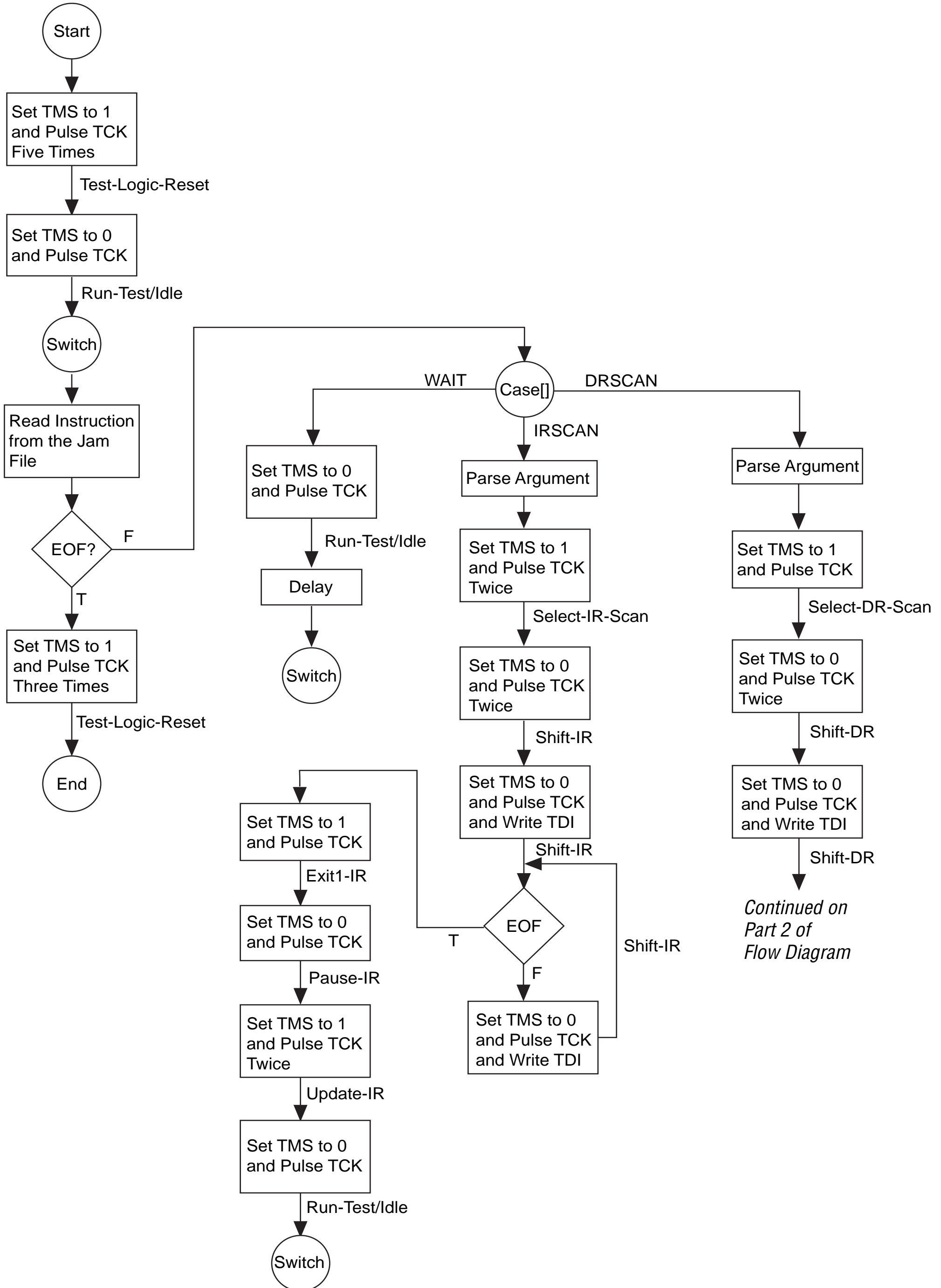
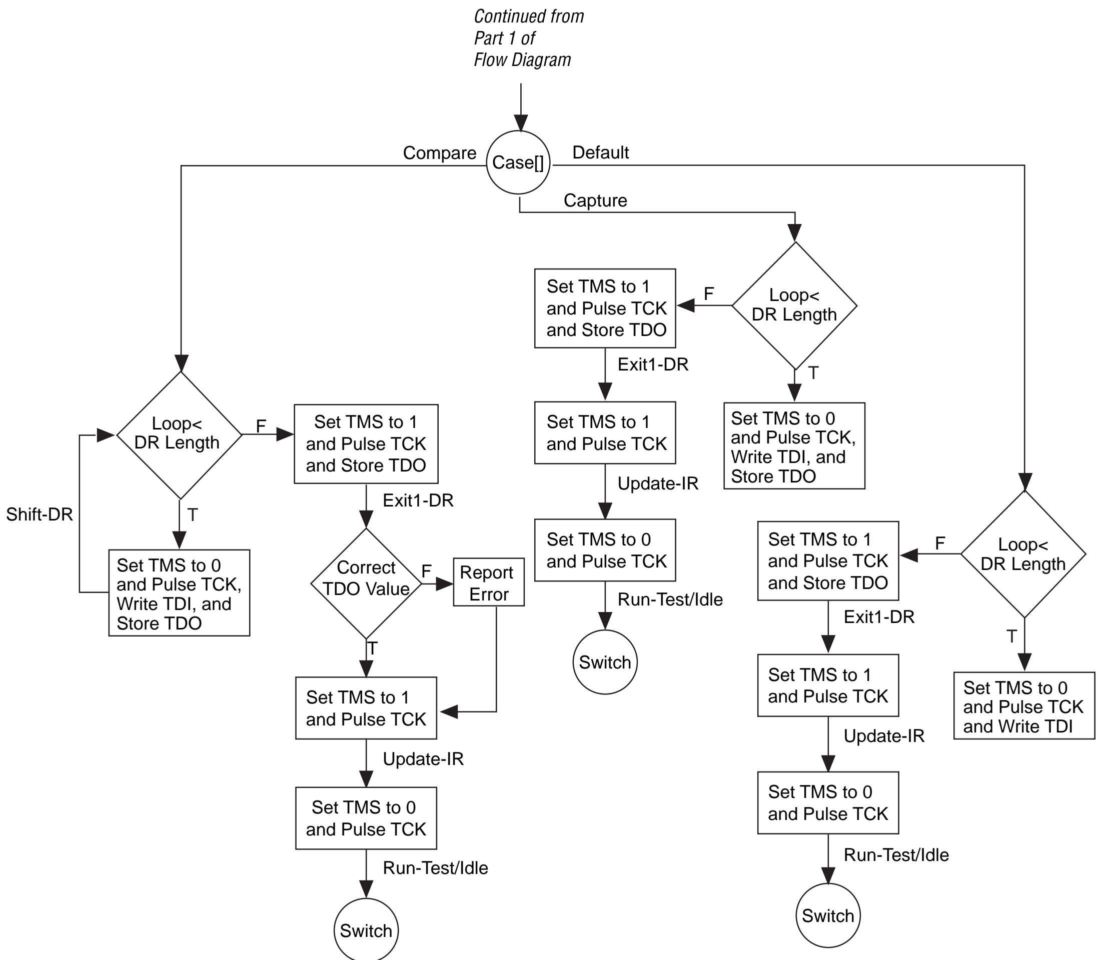


Figure 11. Jam Player Flow Diagram (Part 2 of 2)



Conclusion

To achieve the benefits of in-system programming and ICR through 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 ISP and ICR through 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 reduce device inventories by eliminating the need to stock and track programmed devices.

Revision History

The information contained in *Application Note 88 (Using the Jam Language for ISP & ICR via an Embedded Processor)* version 3.01 supersedes information published in previous versions.

Application Note 88 (Using the Jam Language for ISP & ICR via an Embedded Processor) version 3.01 contains the following changes:

- Added a note clarifying the operation of the `TDI`, `TMS`, and `TCK` signals to Figure 2.
- Made minor textual and style changes throughout the document.

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, Jam, MAX, MAX 9000, MAX 9000A, MAX 7000A, MAX 7000AE, MAX 7000S, MAX, MAX+PLUS, MAX+PLUS II, FLEX, FLEX 10K, FLEX 10KA, EPM7032S, EPM7032A, EPM7064S, EPM7064A, EPM7128S, EPM7128A, EPM7160S, EPM7192S, EPM7256S, EPM7256A, EPM9320, EPM9320A, EPM9400, EPM9480, EPM9560, EPM9560A, EPF10K10, EPF10K20, EPF10K30, EPF10K40, EPF10K50, EPF10K70, EPF10K100, EPF10K130A, EPF10K250A, BitBlaster, ByteBlaster, and ByteBlasterMV 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.

Printed on Recycled Paper.