```
******************************************************************************
            Jam STAPL Byte-Code Player Version 2.09 README 5/5/2000
******************************************************************************
```

CONTENTS

A. DESCRIPTION
B. INCLUDED IN THIS RELEASE
C. NEW IN VERSION 2.09
D. RUNNING THE PLAYER IN COMMAND-LINE MODE
E. PORTING THE JAM STAPL BYTE-CODE PLAYER
F. JAM STAPL BYTE-CODE PLAYER API
G. MEMORY USAGE
H. SUPPORT

A. DESCRIPTION

The Jam STAPL Byte-Code Player is a software driver that allows test and programming algorithms for IEEE 1149.1 Joint Test Action Group (JTAG)-compliant devices to be asserted via the JTAG port. The Jam STAPL Byte-Code Player reads and decodes information in Jam STAPL Byte-Code Files (.jbc) to program and test programmable logic devices (PLDs), memories, and other devices in a JTAG chain. The Jam STAPL Byte-Code Player complies with STAPL (Standard Test and Programming Language) Specification JESD-71. The construction of the Player permits fast programming times, small programming files, and easy in-field upgrades. Upgrades are simplified, because all programming/test algorithms and data are confined to the Jam STAPL Byte-Code File. Version 2.09 supports Jam STAPL Byte-Code Files (.jbc) that have been compiled using Jam STAPL Byte-Code Compiler v2.09. The Player is also able to read and "play"older Jam Byte-Code files based on Jam v1.1 syntax.

The .jbc File is a binary version of the ASCII Jam File (.jam). The Jam STAPL Byte-Code format consists, among other things, of a "byte code" representation of Jam commands, as they are defined in STAPL Specification JESD-71. This means that the .jbc File is simply a different implementation of the .jam file. This binary implementation results in smaller file sizes and shorter programming times.

This document should be used together with AN 122 (Using STAPL for ISP & ICR via an Embedded Processor).

B. INCLUDED IN THIS RELEASE

The following tables provide the directory structure of the files on this CD-ROM:

Directory	Filename	Description
\exe	\16-bit-DOS\jbi.exe	Supports the BitBlaster serial, ByteBlaster parallel, Xilinx Parallel Download Cable III, and Lattice ispDOWNLOAD cables for PCs running 16-bit DOS platforms.
	\Win9598-WinNT\jbi.exe	Supports the BitBlaster serial ByteBlaster parallel, Xilinx

```
                                Parallel Download Cable III, and
                                Lattice ispDOWNLOAD cables for
                                PCs running 32-bit Windows
                                (Windows 95 and Windows NT)


Directory   Filename                Description
---------   --------                ----------------------------------------
\code       jbicomp.h               Source code for the Jam STAPL Byte-Code
Player
            jbiexprt.h
            jbijtag.h
            jbicomp.c
            jbijtag.c
            jbimain.c
            jbistub.c

Directory   Filename                Description
---------   --------                ----------------------------------------
\make       \microsoft\makefile.mak Make file compatible with Microsoft Visual
                                    C++ compiler v5.0. Builds a 32-bit windows
                                    console executable.
            \borland\make32.bat     Batch file that compiles source code for
                                    32-bit Borland compiler. Uses jbi32b.rsp.
            \borland\jbi31b.rsp     Linker script file containing 32-bit
Borland linker commands and flags.
            \borland\make16.bat     Batch file that compiles source code for
                                    16-bit Borland compiler. Uses jbi16b.rsp.
            \borland\jbi16b.rsp     Linker script file containing 16-bit
Borland linker commands and flags.
```

C. NEW IN VERSION 2.09

Updates in the Jam STAPL Byte-Code Player version 2.09 include:

* Added support for Xilinx Parallel Cable III.
* Backward compatibility with existing Jam v1.1 (pre-STAPL) .jbc files.
* Updated to report all exit codes defined by JESD-71.
* Enhanced error reporting in command-line mode.

D. RUNNING THE JAM STAPL BYTE-CODE PLAYER IN COMMAND-LINE MODE

If the Player is going to be run on a PC or a workstation, the following commands can be
used to execute programming or other tasks:

```
Usage:  jbi [-h] [-v] [-d<var=val> OR -a<action>] [-p<port>] [-s<port>]
<filename>

            -a : action name
        -h : help message
        -v : verbose messages
        -d : initialize variable to specified value
        -p : parallel port number or address (for ByteBlaster)
```

```
        -s : serial port name (for BitBlaster)
        -c : alternative download cable compatibility. -cl or -cx

Command line text is not case-sensitive.

Use the -a flag when applying Jam STAPL Byte-Code files. Use the -d flag when
applying Jam v1.1
Byte-Code files.

Valid action names, as specified by JEDEC Standard JESD-71 are:

Action Name             Description
-----------             -----------
CHECKCHAIN              Verify the continuity of the IEEE 1149.1 JTAG scan
chain
READ_IDCODE             Read the IEEE 1149.1 IDCODE and EXPORT it (print it)
READ_USERCODE           Read the IEEE 1149.1 USERCODE and EXPORT it (print it)
READ_UES                Read the IEEE 1149.1 UESCODE and EXPORT it (print it)
ERASE                   Perform a bulk erase of the device(s)
BLANKCHECK              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

Valid initialization variables and values for the -d flag are:

Initialization String   Value       Action
---------------------   -----       ------
DO_PROGRAM              0           Do not program the device
DO_PROGRAM              1           Program the device
DO_VERIFY               0           Do not verify the device
DO_VERIFY               1           Verify the device
DO_BLANKCHECK           0           Do not check the erased state of the
device
DO_BLANKCHECK           1           Check the erased state of the device
READ_UESCODE            0           Do not read the JTAG UESCODE
READ_UESCODE            1           Read UESCODE and export it
DO_SECURE               0           Do not set the security bit
DO_SECURE               1           Set the security bit


E. PORTING THE JAM STAPL BYTE-CODE PLAYER
-----------------------------------------
The Jam STAPL Byte-Code Player is designed to be easily ported to any
processor-based hardware system. All platform-specific code is placed in the
jbistub.c and jbimain.c files. Routines that perform any interaction with the
outside world are confined to the jbistub.c source
```

file. Preprocessor statements encase operating system-specific code and code pertaining to specific hardware. All changes to the source code for porting are mostly confined to the jbistub.c file and in some cases porting the Jam Player is as simple as changing a single #define statement. This process also makes debugging simple. For example, ifthe jbistub.c file has been customized for a particular embedded application, but is not working, the equivalent DOS Jam STAPL Byte-Code Player and a download cable can be used to check the hardware continuity and provide a "known good" starting point from which to attack the problem.

The jbistub.c and jbimain.c files in this release target the DOS operating system, by default. To change the targeted platform, edit the following line in the jbistub.c and jbimain.c files:

```
#define PORT DOS
```

The preprocessor statement takes the form:

```
#define PORT [PLATFORM]
```

Change the [PLATFORM] field to one of the supported platforms: EMBEDDED, DOS, WINDOWS, or UNIX. The following table explains how to port the Jam STAPL Byte-Code Player for each of the supported platforms:

PLATFORM	COMPILER	ACTIONS
EMBEDDED	16 or 32-bit	Change #define and see EMBEDDED PLATFORM below
DOS	16-bit	Change #define and compile
WINDOWS	32-bit	Change #define and compile
UNIX	32-bit	Change #define and compile

The source code supplied in this release is ANSI C source. In cases where a different download cable or other hardware is used, the DOS, WINDOWS, and UNIX platforms will require additional code customization, which is described below.

EMBEDDED PLATFORM

Because there are many different kinds of embedded systems, each with different hardware and software requirements, some additional customization must be done to port the Jam STAPL Byte-Code Player for embedded systems. To port the Player, the following functions may need to be customized:

FUNCTION	DESCRIPTION
jbi_jtag_io()	Interface to the IEEE 1149.1 JTAG signals, TDI, TMS, TCK, and TDO.
jbi_message()	Prints information and error text to standard output, when available.
jbi_export()	Passes information such as the User Electronic Signature (UES) back to the calling program.
jbi_delay()	Implements the programming pulses or delays needed during execution.

Miscellaneous
jbi_vector_map() Processes signal-to-pin map for non-IEEE 1149.1 JTAG signals.
jbi_vector_io() Asserts non-IEEE 1149.1 JTAG signals as defined in the VECTOR MAP.

jbi_jtag_io()
----
int jbi_jtag_io(int tms, int tdi, int read_tdo)

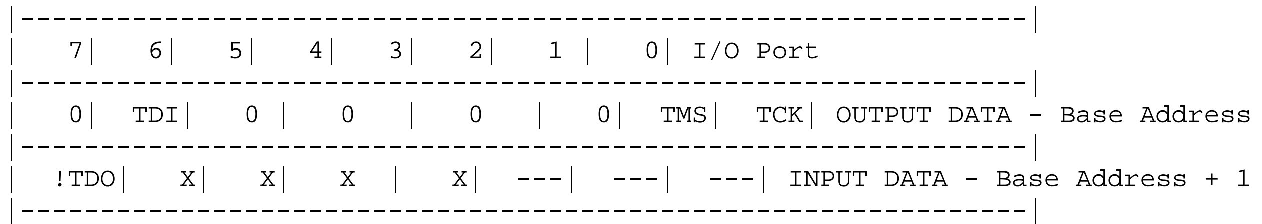
This function provides exclusive access to the IEEE 1149.1 JTAG signals. You must always customize this function to write to the proper hardware port.

The code in this release supports a serial mode specific to the Altera BitBlaster download cable. If a serial interface is required, this code can be customized for that purpose. However, this customization would require some additional processing external to the embedded processor to turn the serial data stream into valid JTAG vectors. This readme filedoes not discuss customization of serial mode. Contact Altera Applications at (800) 800-EPLD for more information.

In most cases a parallel byte mode is used. When in byte mode, jbi_jtag_io() is passedthe values of TMS and TDI. Likewise, the variable read_tdo tells the function whether reading TDO is required. (Because TCK is a clock and is always written, it is written implicitly within the function.) If requested, jbi_jtag_io() returns the value of TDO read. Sample code is shown below:

```
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 ? 0x02 : 0));

	write_byteblaster(0, data);

	if (read_tdo)
	{
		tdo = (read_byteblaster(1) & 0x80) ? 0 : 1;
	}

	write_byteblaster(0, data | 0x01);

	write_byteblaster(0, data);

	return (tdo);
}
```

The code, as shown above, is configured to read/write to a PC parallel port. initialize_jtag_hardware() sets the control register of the port for byte mode. As shown above, jbi_jtag_io() reads and writes to the port as follows:



```
|---------------------------------------------------------------|
|   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
|---------------------------------------------------------------|
```

The PC parallel port inverts the actual value of TDO. Thus, jbi_jtag_io() inverts it again to retrieve the original data. Inverted:

```
            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 simply use the left shift (<<) or right shift (>>) operators. For example, if TMS and TDI are at ports 2 and 3, respectively, then the code would be as shown below:

```
            data = (((tdi ? 0x40 : 0)>>3) | ((tms ? 0x02 : 0)<<1));
```

The same process applies to TCK and TDO.

read_byteblaster() and write_byteblaster() use the inp() and outp() <conio.h> functions, respectively, to read and write to the port. If these functions are not available, equivalent functions should be substituted.

```
jbi_message()
--------------
void jam_message(char *message_text)
```

When the Jam STAPL Byte-Code Player encounters a PRINT command within the .jbc File, it processes the message text and passes it to jbi_message(). The text is sent to stdio. If a standard output device is not available, jbi_message() does nothing and returns. The Jam STAPL Byte-Code Player does not append a newline character to the end of the text message. This function should append a newline character for those systems that require one.

```
jbi_export()
------------
void jbi_export(char *key, long value)
```

The jbi_export() function sends information to the calling program in the form of a text string and associated integer value. The text string is called the key string and it determines the significance and interpretation of the integer value. An example use of this function would be to report the device USERCODE back to the calling program.

jbi_delay()
----
void jbi_delay(long microseconds)

jbi_delay() is used to implement programming pulse widths necessary for programming PLDs, memories, and configuring SRAM-based devices. These delays are implemented using software loops calibrated to the speed of the targeted embedded processor. The Jam STAPL Byte-Code Player is told how long to delay with the .jbc File WAIT command. This function can be customized easily to measure the passage of time via a hardware-based timer. jbi_delay() must perform accurately over the range of one millisecond to one second. The function can take more time than is specified, but cannot return in less time. To minimize the time to execute the Jam statements, it is generally recommended to calibrate the delay as accurately as possible.

Miscellaneous Functions
----
jbi_vector_map() and jbi_vector_io()

The VMAP and VECTOR Jam commands are translated by these functions to assert signals to non-JTAG ports. Altera .jbc Files do not use these commands. If the Jam STAPL Byte-Code Player will be used only to program Altera devices, these routines can be removed. In the event that the Jam Player does encounter the VMAP and VECTOR commands, it will process the information so that non-JTAG signals can be written and read as defined by JEDEC Specification JESD-71.

jbi_malloc()

void *jam_malloc(unsigned int size)

During execution, the Jam STAPL Byte-Code Player will allocate memory to perform its tasks. When it allocates memory, it calls the jbi_malloc() function. If malloc() is not available to the embedded system it must be replaced with an equivalent function.

jbi_free()

void jbi_free(void *ptr)

This function is called when the Jam STAPL Byte-Code Player frees memory. If free() is not available to the embedded system, it must be replaced with an equivalent function.

F. JAM STAPL Byte-Code Player API

The main entry point for the Jam Player is the jbi_execute function:

```
JAM_RETURN_TYPE jbi_execute
(
            PROGRAM_PTR program,
            long program_size,
            char *workspace,
            long workspace_size,
            char *action,
```

```
            char **init_list,
            long *error_line,
            int *exit_code,
            int *format_version
)
```

This routine recieves 6 parameters, passes back 2 parameters, and returns a status code (of JAM_RETURN_TYPE). This function is called once in main(), which is coded in the jbistub.c file (jbi_execute() is defined in the jbimain.c file). Some processing is done in main() to check for valid data being passed to jbi_execute(), and to set up some of the buffering required to store the .jbc File.

The program parameter is a pointer to the memory location where the .jbc File is stored (memory space previously malloc'd and assigned in main()). jbi_execute() assigns this pointer to the global variable jbi_program, which provides the rest of the Jam STAPL Byte-Code Player with access to the .jbc File via the GET_BYTE, GET_WORD, and GET_DWORD macros.

program_size provides the number of bytes stored in the memory buffer occupied by the .jbc File.

workspace points to memory previously allocated in main(). This space is the sum of all memory reserved for all of the processing that the Jam STAPL Byte-Code Player must do, including the space taken by the .jbc File. Memory is only used in this way when the Jam STAPL Byte-Code Player is executed using the -m console option. If the -m option is not used, the Jam Byte Code Player is free to allocate memory dynamically as it is needed. In this case, workspace points to NULL. jbi_execute() assigns the workspace pointer to the global variable, jbi_workspace, giving the rest of the Jam STAPL Byte-Code Player access to this block of memory.

workspace_size provides the size of the workspace in bytes. If the workspace pointer points to NULL this parameter is ignored. jbi_execute() assigns workspace_size to the global variable, jbi_workspace_size.

action is the way the Player is told what function should be performed, as defined by STAPL. (i.e. PROGRAM, READ_USERCODE, etc) The action pointer points to the string that tells the Player what functions to execute within the .jbc file. Each action can contain "recommended" and "optional" sub-actions. "Recommended" sub-actions are those that will be executed by default, while "optional" sub-actions will be skipped. For example, passing "PROGRAM\0" will result in the following steps for an Altera .jbc file:

- ERASE (recommended)
- BLANKCHECK (optional)
- PROGRAM (recommended)
- VERIFY (recommended)

So, by simply passing "PROGRAM\0" the device will be programmed and verified. This is the action Altera recommends using with it's .jbc files. If you want to add the BLANKCHECK step you must pass "DO_BLANKCHECK=1\0" via the init_list pointer. See Section D for other valid action strings. Note that the action string must be NULL terminated.

init_list is a parameter that is used when applying pre-JEDEC, Jam v1.1 .jbc files, or when overriding optional sub-actions, as in the example above. While older pre-JEDEC .jbc files can be played, it is strongly recommended that STAPL-based .jbc files be used. When using STAPL-based .jbc files, init_list should point to NULL. If an older .jbc file must be used, see AN 88 for more details on the parameters that init_list can point to.

If an error occurs during execution of the .jbc File, error_line provides the line number of the .jbc File where the error occured. This error is associated with the function of the device, as opposed to a syntax or software error in the .jbc File.

exit_code provides general information about the nature of an error associated with a malfunction of the device or a functional error:

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
7	Device version is not supported
8	Erase failure
9	Device is not blank
10	Device programming failure
11	Device 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

These codes are intended to provide general information about the nature of the failure. Additional analysis would need to be done to determine the root cause of any one of these errors. In most cases, if there is any device-related problem or hardware continuity problem, the "Unrecognized device" error will be issued. In this case, first take the steps outlined in Section D for debugging the Jam Player. If debugging is unsuccessful, contact Altera for support.

If the "Device version is not supported" error is issued, it is most likely due to a .jbc File that is older than the current device revision. Always use the latest version of MAX+PLUS II to generate the .jbc File. For more support, see Section G.

jbi_execute() returns with a code indicating the success or failure of the execution. This code is confined to errors associated with the syntax and structural accuracy of the .jbc File. These codes are defined in the jbistub.c file, where the array variable "error_text[]".

format_version should be set equal to "2" when calling jbi_execute. This means that the Player will expect a STAPL-based .jbc file.

G. MEMORY USAGE

Memory usage is documented in detail in AN 122 (Using the Jam Language for ISP via an Embedded Processor).

H. SUPPORT

For additional support, e-mail jam@altera.com. Bugs or suggested enhancements can also be communicated via this channel.