****

Jam Byte-Code Player Version 1.0 README 8/31/98

****

CONTENTS

A. DESCRIPTION

The Jam Byte-Code Player is a software driver that allows test and programming algorithms for IEEE Std. 1149.1 Joint Test Action Group (JTAG)-compliant devices to be asserted via the JTAG port. The Jam Byte-Code Player reads and decodes information in Jam Byte-Code Files (.jbc) to program and test programmable logic devices (PLDs), memories, and other devices in a JTAG chain. The construction of the Jam Byte-Code 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 Byte-Code File. Version 1.0 is the first release of the Jam Byte-Code Player, and it supports Jam Byte-Code Files that have been compiled using the Jam Byte-Code Compiler v1.0.

The JBC File is a binary version of the traditional Jam File (.jam). The Jam Byte-Code format consists, among other things, of a "byte code" representation of Jam commands, as they are defined in the Jam Specification version 1.1. 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 88 (Using the Jam Language for ISP via an Embedded Processor), which is included on this CD-ROM.

B. INCLUDED ON THIS CD-ROM

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, and Lattice ispDOWNLOAD cables for PCs running 16-bit DOS platforms.
	\Win95-WinNT\jbi.exe	Supports the BitBlaster serial ByteBlaster parallel, 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 Byte-Code Player
	jbiexprt.h	
	jbijtag.h	
	jbicomp.c	
	jbijtag.c	
	jbimain.c	
	jbistub.c	

C. PORTING THE JAM BYTE-CODE PLAYER

The Jam 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 file. Routines that perform any interaction with the outside world are confined to this source file. Preprocessor statements encase operating system-specific code and code pertaining to specific hardware. All changes to the source code for porting are then 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, if the jbistub.c file has been customized for a particular embedded application, but is not working, the equivalent DOS Jam 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 file on this CD-ROM targets the DOS operating system. To change the targeted platform, edit the following line in the jbistub.c file:

```
#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 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 on this CD-ROM 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 Byte-Code Player for embedded systems. To port the Jam Player, the following functions may need to be customized:

FUNCTION	DESCRIPTION
jbi_jtag_io()	Interface to the IEEE Std. 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 Std. 1149.1 JTAG signals. You must always customize this function to write to the proper hardware port.

The code on this CD-ROM 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 file does 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 passed the 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 will 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, 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 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 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 UES 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 and memories, and configuring SRAM-based devices. These delays are implemented using software loops calibrated to the speed of the targeted embedded processor. The Jam 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. (In Altera devices, it does not matter how much greater the actual delay is over the specified delay.)

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 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 Jam Specification version 1.1.

jbi_malloc()

void *jam_malloc(unsigned int size)

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

D. JAM BYTE-CODE PLAYER API

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

```
JAM_RETURN_TYPE jbi_execute
(
	PROGRAM_PTR *program,
	long program_size,
	char *workspace,
	long workspace_size,
	char **init_list,
	long *error_line,
	int *exit_code
)
```

This routine receives 5 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 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 Byte-Code Player must do, including the space taken by the JBC File. Memory is only used in this way when the Jam 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 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.

init_list is the address of a table of a string of pointers, each of which contains an initialization string. The table is terminated by a NULL pointer. Each initialization string is of the form "string=value". The following list provides some strings defined in Jam Specification version 1.1, along with their corresponding actions:

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

If an initialization list is not needed, a NULL pointer can be used to signify an empty initialization list. This would be the case if the action is always the same and if the action(s) are already defined by default in the JBC File. Note that when programming Altera devices, these initialization variables default to 0. This means that some initialization variables must be passed to jbi_execute, otherwise the Jam Byte-Code Player will do nothing.

If an error occurs during execution of the JBC File, error_line provides the line number of the JBC File where the error occurred. 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. The following conventions are defined by the Jam Specification version 1.1:

exit_code	Description
0	Success
1	Illegal flags specified in initialization list
2	Unrecognized device ID
3	Device version is not supported
4	Programming failure
5	Blank-check failure
6	Verify failure
7	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 ID" error will be issued. In this case, first take the steps outlined in Section C 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 F.

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 jbiexprt.h file.

F. MEMORY USAGE

Memory usage is documented in detail in AN 88 (Using the Jam Language for ISP via an Embedded Processor), which is included on this CD-ROM.

H. SUPPORT

For additional support, contact Altera Applications by phone at (800) 800-EPLD or by e-mail at ISPembed@altera.com. For 24-hour support, visit the Atlas Solutions Database on Altera's web site (http://www.altera.com). Bugs or suggested enhancements can also be communicated via these channels.