****

Jam Player Version 2.12 README 9/2/99
Supporting STAPL v2.1

****

CONTENTS

A. DESCRIPTION

The Jam 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. Jam Player v2.12 supports the Standard Test and Programming file format - the standard format of the original Jam Language. The Jam Player parses and interprets information in Jam Files (.jam) to program and test programmable logic devices (PLDs), memories, and other devices in a JTAG chain. The construction of the Jam 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 File. The Jam Player version 2.12 is the fifth release of the Jam Player, and it supports Jam Files that comply with the Jam Standard Test and Programming Language Specification version 2.1. It is also backward-compatible with pre-standard Jam files using v1.1 syntax.

B. INCLUDED

The following tables provide the directory structure of the files in this release:

Directory	Filename	Description
\exe	\32-bit\jam.exe	Supports the BitBlaster serial ByteBlaster parallel, and Lattice ispDOWNLOAD cables for PCs running 32-bit Windows (Windows 95 and Windows NT)
	\16-bit\jam.exe	Supports the BitBlaster serial ByteBlaster parallel, and Lattice ispDOWNLOAD cables for PCs running 16-bit Windows or DOS
	jamdata.exe	32-bit Jam Data executable for Windows 95 and Windows NT 4.0. Use this program to develop Jam Files that use compressed data. (For developers only) Uses STAPL syntax.

Directory	Filename	Description

\source	jamarray.h	Source code for the Jam Player
	jamutil.h	
	jamsym.h	
	jamstack.h	
	jamjtag.h	
	jamheap.h	
	jamexprt.h	
	jamexp.h	
	jamexec.h	
	jamdefs.h	
	jamcomp.h	
	jamytab.h	
	jamcomp.c	
	jamsym.c	
	jamstub.c	
	jamstack.c	
	jamnote.c	
	jamjtag.c	
	jamheap.c	
	jamexp.c	
	jamexec.c	
	jamcrc.c	
	jamutil.c	
	jamarray.c	
\source\jamdata		
	jamdata.c	Source code for jamdata.exe

C. NEW IN VERSION 2.12

The updates in the Jam Player version 2.0 include:

* Adds support for the following constructs with STAPL files:
  - Use of literal ACA arrays
  - Use of reverse array index order
  - Literal array data containing white space

* Enhanced memory allocation for large, compressed data arrays.

D. PORTING THE JAM PLAYER

The Jam Player is designed to be easily ported to any processor-based hardware system. All platform-specific code should be placed in the jamstub.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 jamstub.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 jamstub.c file has been customized for a particular embedded application, but is not working, the equivalent DOS Jam 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 jamstub.c file targets the DOS operating system. To change the targeted platform, edit the following line in the jamstub.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 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 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 Player for embedded systems. To port the Jam Player, the following functions may need to be customized:

FUNCTION	DESCRIPTION
jam_getc()	The only function used to retrieve information from the Jam File (file I/O).
jam_seek()	Allows the Jam Player to move about in the Jam File (file I/O).
jam_jtag_io()	Interface to the IEEE 1149.1 JTAG signals, TDI, TMS, TCK, and TDO.
jam_message()	Prints information and error text to standard output, when available.
jam_export()	Passes information such as the User Electronic Signature (UES) back to the calling program.
jam_delay()	Implements the programming pulses or delays needed during execution.

Miscellaneous

jam_getc()
----
int jam_getc(void)

jam_getc() retrieves the next character in the Jam File. Each call to jam_getc() advances the current position in the file, so that successive calls to the function get sequential characters. This function is similar to the standard C function fgetc(). The function returns the character code that was read or a (-1) if none was available.

jam_seek()

```
int jam_seek(long offset)
```

jam_seek() sets the current position in the Jam File input stream. The function returns zero for success or a non-zero value if the request was out of range. This function is similar to the standard C function fseek(). The storage mechanism for the Jam File is a memory buffer. Alternatively, a file system can be used. In this case, this function would need to be customized to use the equivalent of the C language fopen() and fclose(),as well as store the file pointer.

jam_jtag_io()

```
int jam_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 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, jam_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, jam_jtag_io() returns the value of TDO read. Sample code is shown below:

```
int jam_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, jam_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, jam_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, 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.

jam_message()

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

When the Jam Player encounters a PRINT command within the Jam File, it processes the message text and passes it to jam_message(). The text is sent to stdio. If a standard output device is not available, jam_message() does nothing and returns. The Jam 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.

jam_export()

```
void jam_export(char *key, long value)
```

The jam_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.

jam_delay()

```
void jam_delay(long microseconds)
```

jam_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 Player is told how long to delay with the Jam File WAIT command. This function can be customized easily to measure the passage of time via a hardware-based timer. jam_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

jam_vector_map() and jam_vector_io()

The VMAP and VECTOR Jam commands are translated by these functions to assert signals to non-JTAG ports. If the Jam Player will only execute Jam files that interface only through the JTAG port, 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 STAPL Specification v2.1.

jam_malloc()

```
void *jam_malloc(unsigned int size)
```

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

jam_free()

```
void jam_free(void *ptr)
```

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

E. JAM PLAYER API

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

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

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 jamstub.c file (jam_execute() is defined in the jamexec.c file). Some processing is done in main() to check for valid data being passed to jam_execute(), and to set up some of the buffering required to store the Jam File.

The program parameter is a pointer to the memory location where the Jam File is stored (memory space previously malloc'd and assigned in main()). jam_execute() assigns this pointer to the global variable jam_program, which provides the rest of the Jam Player with access to the Jam File via the jam_getc() and jam_seek() functions.

program_size provides the number of bytes stored in the memory buffer occupied by the Jam 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 Player must do, including the space taken by the Jam File. Memory is only used in this way when the Jam Player is executed using the -m console option. If the -m option is not used, the Jam Player is free to allocate memory dynamically as it is needed. In this case, workspace points to NULL. jam_execute() assigns the workspace pointer to the global variable, jam_workspace, giving the rest of the Jam 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. jam_execute() assigns workspace_size to the global variable, jam_workspace_size.

action is the method of describing the function that is to be performed, as defined by STAPL. (i.e. PROGRAM, VERIFY, etc) The action pointer points to the string that tells the Player what function it is to use within the .jam file. See the STAPL specification for support action string names.

init_list is the address of a table of a string of pointers, each of which contains an initialization string. This method for specifying the function is pre-STAPL. 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 the 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 (default)	Program the device
DO_VERIFY	0	Do not verify the device
DO_VERIFY	1 (default)	Verify the device
DO_BLANKCHECK	0	Do not check the erased state of the device
DO_BLANKCHECK	1 (default)	Check the erased state of the device
READ_UESCODE	0 (default)	Do not read the JTAG UESCODE
READ_UESCODE	1	Read UESCODE and export it
DO_SECURE	0 (default)	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 Jam File.

If an error occurs during execution of the Jam File, error_line provides the line number of the Jam 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 Jam 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 E 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 Jam File that is older than the current device revision. Always use the latest version of MAX+PLUS II to generate the Jam File. For more support, see Section H.

jam_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 Jam File. These codes are defined in the jamexprt.h file.

F. SUPPORT

For additional support contact the vendor that is writing the STAPL or Jam files that are being used. If the issues is specific to the code contained within this release of the Jam Player, e-mail jam@altera.com.