AN 586: Porting the Jam STAPL and Jam STAPL Byte-Code Players to an Embedded System

© August 2009 AN-586-1.0

The Jam™ Standard Test and Programming Language (STAPL) and Jam STAPL Byte-Code (JBC) Players are software that enable a processor to program or configure CPLD or FPGA devices with data based on the algorithms in a Jam file (**.jam**) or Jam Byte-Code file (**.jbc**).

Introduction

This application note provides information about the functions that you need to be aware of when porting the Jam STAPL and Jam STAPL Byte-Code Players to an embedded system.

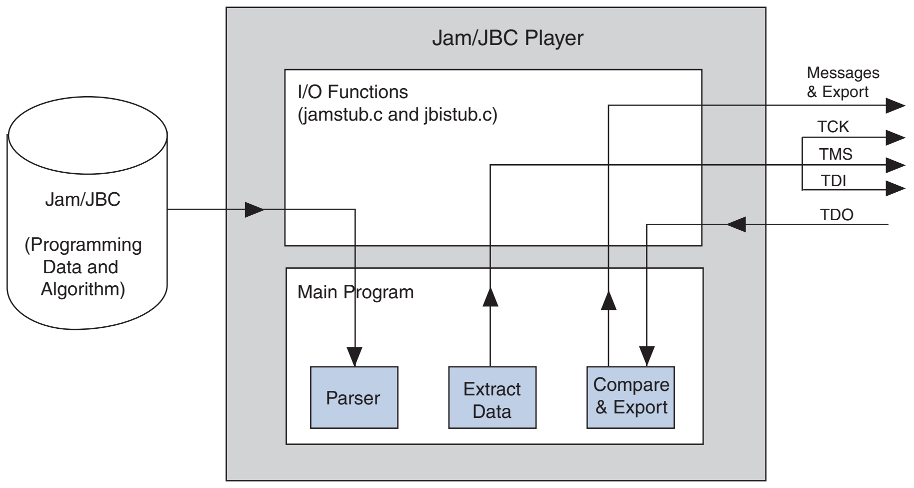
No examples are provided in this application note because the changes required for porting depend on the embedded system and operating system you use.

Overview for Porting the Jam STAPL or JBC Player

The Jam STAPL or JBC Player interprets and executes each Jam STAPL or JBC instruction in the **.jam** or **.jbc**. The main program performs all the basic functions of the Jam or JBC Player. Based on the targeted embedded system, you must modify the I/O functions of the Jam or JBC Player to customize your functions according to your embedded processor or operating system, for example, the Freescale™ V2 ColdFire Processor.

Figure 1 shows the functions that specify delay routines, operating system-specific functions, and routines for file I/O pins are contained in **jamstub.c** or **jbistub.c**.

Figure 1. Jam Player Source Code Structure *(Note 1)*



Note to Figure 1:

(1) `TCK`, `TMS`, `TDI`, and `TDO` are the JTAG I/O pins.

You can customize the I/O functions and compile the source code for any embedded system by editing **jamstub.c** or **jbistub.c**. The Jam or JBC Player is written in C programming language. To ensure maximum compatibility during compilation, Altera recommends porting to the Jam or JBC Player an embedded system supporting a C programming language-compatible compiler.

Jam and JBC Player source code only supports 32-bit processors. You need to port the Jam or JBC Player source code to support other type of processors.

CAUTION Altera recommends that you modify the Jam or JBC Player source code in **jamstub.c** and **jbistub.c** only.

Functions in jamstub.c and jbistub.c

This section lists the functions that require attention during porting. Table 1 shows the functions with their corresponding name in **jamstub.c** and **jbistub.c**. A brief explanation of each function follows the table.

Table 1. Functions in jamstub.c and jbistub.c

Functions	Functions in jamstub.c	Functions in jbistub.c
Main Function		
`main`	`main()`	`main()`
Delay Functions		
`get_tick_count`	`get_tick_count()`	`get_tick_count()`
`calibrate_delay`	`calibrate_delay()`	`calibrate_delay()`
`delay`	`jam_delay()`	`jbi_delay()`
Additional Functions		
`getc`	`jam_getc()`	—
`seek`	`jam_seek()`	—
`jtag_io`	`jam_jtag_io()`	`jbi_jtag_io()`
`message`	`jam_message()`	`jbi_message()`
`export_integer`	`jam_export_integer()`	`jbi_export_integer()`
`malloc`	`jam_malloc()`	`jbi_malloc()`
`initialize_jtag_hardware`	`initialize_jtag_hardware()`	`initialize_jtag_hardware()`
`close_jtag_hardware`	`close_jtag_hardware()`	`close_jtag_hardware()`
`read_byteblaster`	`read_byteblaster()`	`read_byteblaster()`
`write_byteblaster`	`write_byteblaster()`	`write_byteblaster()`

Main Function

main

This function is part of every C program and is the main building block for all C programs. In the Jam or JBC Player source code, the `main()` function contains the **.jam** or **.jbc** location, the initialization list, and the exit codes. In addition, the `jam_execute` or `jbi_execute` function (the main entry point to the Jam or JBC Player) is called by the `main()` function.

By default, the initialization list, action, and file location is set to NULL. The file name and the initialization list are read from the input stream with a terminal program to give instructions to the Jam or JBC Player using the command prompt. You must customize this section if you do not have a user interface for your embedded processor. For a description of the initialization list and action, see "Initialization List and Action".

Initialization List and Action

The following sections explain the initialization list and action of the `main()` function.

Initialization List

The initialization list, `init_list`, is the address of a string of pointers, each containing an initialization string. Each initialization string is in the "string=value" form. An initialization list provides instructions to the Jam or JBC Player as to which initialization string to perform. Table 2 lists the strings defined in the Jam Specification version 1.1.

Table 2. Strings Defined in the Jam Specification Version 1.1

Initialization String	Value	Description
DO_PROGRAM	0	Do not program the device.
	1 (default)	Program the device.
DO_VERIFY	0	Do not verify the device.
	1 (default)	Verify the device
DO_BLANKCHECK	0	Do not check the erased state of the device.
	1 (default)	Check the erased state of the device.
READ_USERCODE	0 (default)	Do not read the JTAG USERCODE.
	1	Read USERCODE and export it
DO_SECURE	0 (default)	Do not set the security bit
	1	Set the security bit

You must pass the initialization list in the correct manner. If an invalid initialization list or no initialization list is passed, the Jam or JBC Player only performs a syntax check on the **.jam** or **.jbc**. If the syntax check passes, the Jam or JBC Player issues a successful exit code without performing any function. Example 1 shows how to define the code to set up `init_list` to instruct the Jam or JBC Player to perform a program and verify operation.

Example 1.

```
Char CONSTANT_AREA init_list[] [] ="DO_PROGRAM=1",
"DO_VERIFY=1"
```

Action

`action` specifies the action that the Jam or JBC Player performs. By default, the `action` is set to NULL. If an initialization list is not required, you can use a NULL pointer to signify an empty initialization list. This is only applicable if the action is already defined in the Jam or JBC Player. Table 3 lists the actions available in Jam and JBC Players.

Table 3. Actions Available in the Jam and JBC Players

Action	Description
PROGRAM	Program the device
VERIFY	Verify the device
BLANKCHECK	Check the erased state of the device
READ_USERCODE	Read USERCODE and export it

Delay Functions

There are three inter-related delay functions in **jamstub.c** and **jbistub.c**: `delay()`, `calibrate_delay()`, and `get_tick_count()`. The `get_tick_count()` function obtains the system tick count value and returns the value to the `calibrate_delay()` function. The `calibrate_delay()` function then uses the system tick count to determine the loops required for a one-millisecond delay. This information is then used by the `delay()` function to execute the delay required for the `WAIT` command.

get_tick_count

This function is called by the `calibrate_delay()` function to obtain the system tick count in milliseconds. By default, the source code is tailored for the operating system listed below:

- WINDOWS—`GetTickCount()` function
- UNIX—`clock()` system function

You must customize this function accordingly if the operating system for your embedded processor does not use any of the functions listed for Windows or UNIX.

calibrate_delay

This function determines how many loops are required for a one-millisecond delay. By default, the source code includes the calculation for the Windows operating system.

You need to customize this function if your embedded processor’s operating system is not Windows.

delay

This function implements 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 processor. For example, pulses of varying widths are used to program the internal EEPROM cells of Altera's MAX® CPLDs. The Jam or JBC Player uses the `delay()` function to implement these pulse widths. The `WAIT` command in the **.jam** or **.jbc** specifies the delay required.

You must customize this function based on the speed of the processor and the time the processor takes to execute a single loop. To minimize the time to execute the Jam or JBC STAPL statements, Altera recommends you calibrate the delay, as accurately as possible, over the range of one millisecond to one second.

Additional Functions

getc

This function retrieves the characters in a **.jam**. Each call to the `getc()` function advances the current position of the pointer in the file. Successive calls of the function are needed to get a string of characters. If the successive call reaches the end of the file, the end-of-file indicator is set and the `getc()` function returns EOF. If a read error occurs, the error indicator is set and `getc()` returns EOF. This function is similar to the standard `fgetc()` C function. The function returns the character code that was read, or a (-1) if none was available.

By default, the source code has taken care of the algorithm to retrieve the characters in a **.jam** for the Windows operating system. If the operating system for your processor uses a different algorithm, you need to customize this function.

seek

This function sets the current file position pointer in a **.jam** input stream based on the specified offset. The function returns a zero if the offset is within the file length, otherwise a non-zero value is returned. This function is similar to the standard C function `fseek()`.

In the source code, the storage mechanism for a **.jam** is a memory buffer. Alternatively, you can use a file system as the storage machinism. In this case, you must customize the function to use the equivalent of the C language `fopen()` and `fclose()` functions, as well as to store the file pointer.

jtag_io

This function provides access to the IEEE 1149.1 JTAG signals `TDI`, `TMS`, `TCK`, and `TDO`. The `jtag_io()` function contains the code that sends and receives the binary programming data. You must re-map each of the four JTAG signals to the embedded processor's pins. By default, the source code writes to the PC's parallel port.

In the current source code, the PC parallel port inverts the actual value of `TDO`. The `jtag_io()` source code inverts the `TDO` value again to retrieve the original data.

```
tdo=(read_byteblaster(1)&0x80)?0:1;
```

If the target processor does not invert `TDO`, the code is:

```
tdo=(read_byteblaster(1)&0x80)?1:0;
```

To map the signals to the correct addresses, use the left shift (<<) or right shift (>>) operators. For example, if `TDI` and `TMS` are at the third and second port respectively, the code is:

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

Apply the same process to `TCK` and `TDO`.

If you are not using the PC parallel port, you must customize this function to write to the proper hardware port.

message

When the Jam or JBI Player encounters a `PRINT` command within **.jam** or .**jbc**, the Player processes the text message and passes the result to the `message()` function. If a standard output device is not available, the `message()` function does nothing.

The Player does not append a newline character to the end of the text message. If your embedded system requires that you append a new line, you must modify the `message()` function print information and error messages of this function to standard output. If you do not use this function, you can either remove this routine or comment out the call to the `puts()` function located in the `message()` function.

export_integer

The `export_integer()` and `export_boolean_array()` functions return information from the Jam or JBC Player to the calling program. The most common use of this routine is to transfer the user electronic signature (UES) instruction code from a device back to the program that calls the Jam or JBC Player. These functions send text messages to `stdio`, using the `printf()` function. The Jam or JBC Player uses the `export_integer()` and `export_boolean_array()` functions to pass information (for example, the UES instruction code or USERCODE of the device) to the operating system or software that calls the Jam or JBC Player.

By default, the Jam or JBC Player prints the value using the `printf` command. You will need to modify these functions if the `printf` command is not available or if there is no device available to `stdout`. You can redirect the information to a file or a storage device, or pass the information back as a variable to the program that calls the Jam or JBC Player.

malloc

This function allocates the required memory whenever this function is called. During program execution, the Jam or JBC Player must allocate memory to perform the tasks. When the Jam or JBC Player allocates memory, the `malloc()` function is called. For example, if the program is to write the Jam or JBC file into the memory, the Jam or JBC Player uses this function to allocate the required memory to put the **.jam** or **.jbc**. The Jam or JBC file size depends on which, and how many devices are targeted for programming. You must evaluate each design to select a suitable memory resource.

In some cases, the `malloc()` function is not supported by the embedded system. If this is the case, you must replace this function with an equivalent function.

For more information about how to estimate the ROM and RAM memory usage, refer to the "Jam STAPL Byte-Code Player Memory Usage" section of *AN 425: Using Command-Line Jam STAPL Solution for Device Programming*.

initialize_jtag_hardware

This function initializes your hardware I/Os so that the player is able to write to the JTAG port. By default, the Jam or JBC Player source code contains the routine to initialize the hardware I/Os for the Windows operating system.

You must customize this function to initialize your hardware I/Os based on your operating systems and hardware requirement.

close_jtag_hardware

This function closes (or inactivates) your hardware I/Os so that the player does not write to the JTAG port. By default, the Jam or JBC Player source code contains the routine to close the communication port for the Windows operating system.

You must customize this function to close your hardware I/Os for other operating systems.

read_byteblaster

This function reads data through the ByteBlaster™ II download cable. The `read_byteblaster` function uses the `inp()` function from the `conio.h` library to read from the parallel port. This function is customized for the Windows system only.

You must customize this function with the equivalent function in your embedded processor that performs the read operation through the ByteBlaster II download cable.

write_byteblaster

This function writes data through the ByteBlaster II download cable. The `write_byteblaster` function uses the `outp()` function from the `conio.h` library to write to the parallel port. This function is customized for the Windows operating system only.

You must customize this function with the equivalent function in your embedded processor that performs

the write operation through the ByteBlaster II download cable.

Document Revision History

Table 4 shows the revision history for this application note.

Table 4. Revision History

Date and Revision	Changes Made	Summary of Changes
August 2009	Initial release.	—