Altera Jam Player User Guide

Table of Contents:

1. Overview

The Altera Jam Player is a software interpreter for the Jam Programming and Test Language. The Jam language is a programming language designed specifically to support programming logic and memory devices via the IEEE 1149.1 JTAG interface. The Jam language is an interpreted language, meaning that Jam program source code is executed directly by an interpreter program (such as the Altera Jam Player) running on a host computer, without first being compiled into binary executable code. As a Jam program is executed by the interpreter, signals may be produced on the IEEE 1149.1 JTAG interface to program a device under control of the Jam program.

The Altera Jam Player supports the Jam language as defined in the Jam Programming and Test Language Specification version 1.0. The language extensions which provide an interface to non-JTAG-compliant hardware (the VECTOR and VMAP instructions) are not supported by the Altera Jam Player.

The ‘C’ language source code for the Altera Jam Player is provided by Altera, and is available for use under license from Altera. (Please read the software distribution agreement that accompanies the source code.) Detailed instructions for installing this software on computer systems and embedded microprocessor systems are provided in Section 3, *Installing the Jam Player*.

A binary executable version of the Altera Jam Player is also provided. This executable file (**jam.exe**) works with computers running the Windows 95 or Windows NT 4.0 operating system, and is equipped with interface functions to support the Altera ByteBlaster parallel port download cable. Instructions for using **jam.exe** are provided in Section 4, *Executable Programs*.

The Jam language supports several different formats for data storage. Some of these data storage formats use a compressed representation to reduce the size of the Jam program. A software program called **jamdata.exe**, which converts data between these different data storage formats, is provided. The 'C' language source code for this program is also provided. Instructions for using **jamdata.exe** are also given in Section 4.

2. Features

2.1 Modes of Operation

The Altera Jam Player has the following operating modes:

- It can execute a Jam program.
- It can check the cyclic redundancy code (CRC) of a Jam program (without executing the program).
- It can extract information from the NOTE fields (without executing the program).

These functions are combined in a single software module because they share the basic functions that process Jam language statements. They are implemented as three separate exported functions in the Jam Player software interface.

2.2 Executing a Jam Program

To execute a Jam program, the Jam Player reads program statements from the Jam program file and executes the statements. All types of statements in the file are processed except NOTE statements (which are ignored) and CRC statements (which cause an error if they are processed). When an EXIT statement is encountered, the Jam program terminates. If an initialization list is provided, it is used to influence the initial values of data variables that can be declared inside the Jam program.

During execution of a Jam program, the DRSCAN, IRSCAN, WAIT, and STATE instructions cause the Jam Player to interact with the signals of the IEEE 1149.1 JTAG interface. If the Jam program contains any of these instruction types (which it must for any JTAG application), the Jam Player must be installed with the corresponding JTAG interface functions.

2.3 Checking the CRC

To check the integrity of the Jam program file, the actual CRC value of the file is compared to the expected CRC value stored in the CRC statement at the end of the file. The CRC statement and expected CRC value is (optionally) computed and added to the file at the time the file is generated. For "hand-edited" Jam programs, the CRC statement can be omitted. The CRC check has three possible outcomes: (1) successful match, (2) unsuccessful match, or (3) no CRC statement found.

2.4 Extracting NOTE Information

NOTE fields are stored in a Jam program as pairs of strings: the first of which is the key, and the second of which is the value. The key string must be 32 characters or less in length, first character alphabetic, subsequent characters alphanumeric or underscore, and is case-insensitive. The value string can contain any printable ASCII characters (that is, alphabetic, numeric, punctuation, and white-space characters) with the restriction that it must be enclosed in double quotation marks (") if it contains a colon (:), semicolon (;), or comment character (single quotation mark). The value string cannot contain double quotation mark characters. The value string is terminated by a semicolon. The software interface for extracting NOTE information supports the query of a NOTE value given the key, and extraction of all NOTE fields in the Jam program file.

2.5 Input and Output Interfaces

The following are Jam Player input sources:

- The Jam program file
- The initialization list (which supplies information used during execution of the Jam program file)
- Information that can be obtained from the IEEE 1149.1 JTAG hardware interface during execution of the Jam program file

The following are Jam Player outputs:

- Instruction, data, and control information to drive the JTAG hardware interface
- Optional text messages, which can be displayed on an output device (if available)
- Information exported using the EXPORT instruction
- An integer return code returned at the termination of processing

3. Installing the Jam Player

3.1 Software Interface

The Altera Jam Player communicates with other software using a ‘C’ language function-call interface. There are three “entry point” functions that can be called to invoke the Jam Player from outside the Jam Player software. In addition, there are eight “callback” functions that are called by the Jam Player. These functions provide access to external resources, such as the Jam program and the JTAG hardware interface. To install the Jam Player into a computer system or embedded microprocessor-based system, these callback functions must be customized as appropriate to provide access to the resources of that system.

3.2 Hardware Interface

The Jam Player controls the signals of the JTAG hardware exclusively through a callback function called `jam_jtag_io`, which is described in detail below. This function must be customized to permit access to the JTAG hardware interface in the host system. Similarly, the signals of the non-JTAG hardware interface are controlled through a callback function called `jam_vector_io`. The Altera Jam Player does not support the non-JTAG hardware interface.

3.3 Entry Point Functions

The three entry point functions corresponding to the three operating modes of the Jam Player are as follows:

- `jam_execute`
- `jam_check_crc`
- `jam_get_note`

3.3.1 `jam_execute` Entry Point Function

The `jam_execute` function executes a Jam program. The function prototype is as follows:

```
int jam_execute
(
    char **init_list,
    char *workspace,
    long size,
    long *error_line,
    int *exit_code
);
```

The `init_list` parameter is the address of a table of string pointers, each of which contains an initialization string. The table is terminated by a NULL pointer. Each initialization string is of the form "name=value". The name should be the name of an integer or Boolean variable in the Jam program, and the value is the numeric value to be assigned to that variable, which overrides the initialization value specified in the Jam program. If no initialization list is needed, then a NULL pointer may be used to signify an empty initialization list.

The `workspace` parameter is a pointer to a memory buffer, to be used for storage of variables and other information by the Jam Player.

The `size` parameter is the number of bytes available in this buffer. If the workspace buffer is too small, an error will occur.

The `error_line` parameter is a pointer to a long integer that receives the line number at which an error occurred, if any. (The occurrence of an error is indicated by the return code from the function.)

The `exit_code` parameter is a pointer to an integer that receives the value of the exit code specified in the EXIT statement which terminated the Jam program. This exit code is valid only if the Jam program processes sucessfully, as indicated by the return code from the function. An exit code of zero indicates that the Jam program had a successful result (for example, a device was programmed successfully).

The return code from `jam_execute` is an integer that indicates whether the Jam program completed successfully, or was terminated prematurely due to an error. A return value of zero indicates success, and non-zero values indicate error conditions. The error codes are shown in the Jam Player source code (file **jamexprt.h**).

3.3.2 `jam_check_crc` Entry Point Function

The `jam_check_crc` function checks the CRC of a Jam program, without executing the program. The function prototype is as follows:

```
JAM_RETURN_TYPE jam_check_crc
(
    unsigned short *expected_crc,
    unsigned short *actual_crc
);
```

The `expected_crc` parameter is a pointer to an unsigned short integer that receives the CRC value stored in the CRC statement at the end of the Jam program.

The `actual_crc` parameter is a pointer to an unsigned short integer that receives the CRC value calculated by processing the contents of the Jam program. These parameters permit the CRC values to be displayed in an error message, if desired.

The return code from the `jam_check_crc` function is an integer that indicates the result of the CRC check. Zero indicates success, while a non-zero value indicates that the CRC values do not match, or that no CRC statement was found in the Jam program.

3.3.3 `jam_get_note` Entry Point Function

The `jam_get_note` function extracts NOTE information from a Jam program, without executing the program. This function may be used either to search for a NOTE field with a given key string, or to retrieve all NOTE fields from the Jam program regardless of the key. The function prototype is as follows:

```
JAM_RETURN_TYPE jam_get_note
(
    long *offset,
    char *key,
    char *value,
    int length
);
```

To obtain the note value for a given key, the key string is provided for the `key` parameter, NULL is passed for the `offset` pointer, and a buffer of sufficient length is provided for the `value` parameter, with the `length` parameter set to the buffer length. This combination causes the entire Jam program file to be searched for a NOTE field with a matching key. If such a NOTE is found, the `value` string is copied to the `value` buffer (up to `length` bytes), and a zero return code is returned. A non-zero return code is returned if the NOTE key was not found.

To extract all NOTE fields, a buffer of at least 33 characters must be provided for the `key` parameter (32 key characters + NULL character). The `offset` parameter is a pointer to a long integer that stores the current position in the Jam program file, beginning with zero. The function can be called multiple times, each time yielding a NOTE `key`/`value` pair, until the function returns an unsuccessful (non-zero) return code. Each time the function is called, it saves the position of the end of the previous NOTE statement in the long integer pointed to by `offset`.

3.4 Callback Functions

The I/O interfaces of the Jam Player are encapsulated in a set of customizable callback functions. This approach makes the program portable to a wide variety of systems with different I/O interface requirements.

The interface for reading data from the Jam program file is encapsulated in two functions that must be customized for the target application. The two interface functions for Jam program input are `jam_getc` and `jam_seek`.

3.4.1 `jam_getc` Callback Function

The customizable function `jam_getc` gets a single character from the current position in the Jam program. The return value is the character code of the character that was read, or (-1) if no character is available — for example, if the end of the file was reached. Each call to `jam_getc()` advances the current position in the file, so that successive calls get sequential characters from the file. This behavior is similar to that of the standard ‘C’ function `fgetc()`. The function prototype is as follows:

```
int jam_getc(void);
```

3.4.2 `jam_seek` Callback Function

The customizable function `jam_seek` sets the current position in the Jam program file input stream. The function returns zero for success, or a non-zero value if the request offset was out of range. This behavior is similar to that of the standard ‘C’ function `fseek()`. The function prototype is as follows:

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

The storage mechanism for the Jam program can be a file system or a memory buffer. Note that if a file system is used, the equivalent of the ‘C’ language `fopen()` and `fclose()` functions, as well as storage of the file pointer, are not managed by the Jam Player and must be included in the customization code.

3.4.3 `jam_jatag_io` Callback Function

The JTAG hardware interface is implemented using a single customizable function. This function is as follows:

```
int jam_jtag_io(int tms_tdi);
```

This function provides access to the signals of the JTAG hardware interface. Each time this function is called, the logic levels of the JTAG `TMS` and `TDI` output signals are set to the requested values, the `TCK` clock signal is strobed (high, then low), then the `TDO` input signal from the JTAG hardware is sampled and returned in the return code. The `nTRST` signal is not used. The `tms_tdi` parameter contains three bits of information, indicating the state of the `TMS` signal and the `TDI` signal, and also whether the `TDO` output level should be read. (This parameter allows you to skip the reading of `TDO` if the value is not going to be used.) The least significant bit (mask 0x01) represents the `TMS` level, the next bit (mask 0x02) represents the `TDI` level, and the third bit (mask 0x04) indicates whether the `TDO` level should be read from the hardware: if set, the `TDO` value must be read from the hardware, otherwise it is not necessary to read the `TDO` value. The return code is zero if `TDO` was low, non-zero if `TDI` was high. If the actual `TDO` level was not read, zero should be returned.

3.4.4 `jam_message` Callback Function

If a console or teletype output device is available, it can be used to display the messages generated by PRINT statements. If no output device is available, PRINT statements will be ignored. The message function interface is implemented using the following function:

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

The Jam Player does not append a newline character at the end of the message string, so for some applications it may be appropriate to do so inside the `jam_message()` function. If the standard ‘C’ language console output functions are available, the function `puts()` can be used for text message output. If no message device is available, this function should simply do nothing, then return.

3.4.5 `jam_delay` Callback Function

The following customizable function can be used to make accurate delays in real time:

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

This function can use a software delay loop that is calibrated for the speed of the particular target system, or it can make reference to a hardware timer device in the target system to measure the passage of time. For successful device programming, this function must perform accurately over the range of one millisecond (1000 microseconds) to one second (1,000,000 microseconds), with a tolerance of zero in the negative direction (i.e., the function must not return in less than the time specified), and up to 100% in the positive direction (i.e., the function can consume up to twice the requested time). Of course, any error in the positive direction will increase the total programming time for a device programming application.

3.4.6 `jam_export` Callback Function

The `jam_export` function is a customizable function that transmits information from the Jam Player to the calling program in the form of a text string and an associated integer value. The text string is called the `key` string, and it determines the significance and interpretation of the integer value. The syntax is as follows:

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

The `jam_export` function should ignore any invocation that does not carry a recognized `key` string.

3.4.7 `jam_vector_map` & `jam_vector_io` Callback Functions

Two customizable functions are provided to support non-JTAG hardware access. These functions correspond to the optional VMAP and VECTOR instructions in the Jam language.

```
int jam_vector_map
(
      int signal_count,
      char **signals
);

int jam_vector_io
(
      int signal_count,
      long *dir_vect,
      long *data_vect,
      long *capture_vect
);
```

The `jam_vector_map` function is used to establish a mapping between signal names and vector bit positions for non-JTAG signals. The order of the signal names in the `signals` array determines the order of bits in the vectors processed by the `jam_vector_io` function. The `jam_vector_map` function should match these signal names to the names of non-JTAG signals available in the host system, and store this mapping.

The `jam_vector_io` function applies a vector to the non-JTAG signals in the current vector mapping (the signals identified in the most recent call to `jam_vector_map`). The number of signals (`signal_count`) must be identical to the signal count specified in the most recent call to `jam_vector_map`. When `jam_vector_io` is called, the direction and data of non-JTAG signals is updated as follows:

If the corresponding bit of `dir_vect` is zero, then that signal will be tri-stated (high-impedance).
If the bit is one, then that signal will be made to drive the logic level specified in the corresponding bit of `data_vect`.

If the pointer to `capture_vect` is a NULL pointer, then no information is read back from the non-JTAG signals.

If the pointer to `capture_vect` is not NULL, then for each signal whose direction bit is zero, the current logic level of that signal is sensed and stored in the corresponding bit of `capture_vect`.

All three vector pointers are pointers to arrays of long integers. The bit positions in these vectors are defined as follows: bit zero is the least significant bit of the first long integer addressed by the pointer; bit 31 is the most significant bit of the first long integer; bit 32 is the least significant bit of the second long integer; and so on.

3.5 Memory Usage

The Jam Player uses memory in a simple and predictable way, in order to be compatible with embedded systems that lack a memory-allocation service. The memory requirements of the Jam Player are as follows:

- Jam program variables (scalar and array variables)
- Jam program stack (used for CALL, FOR and PUSH statements)
- Jam program symbol table (labels and variable names)
- Temporary data used by the Jam Player

The host system must provide the Jam Player with a single memory buffer of known size. The Jam Player automatically divides that memory intelligently to satisfy the demands listed above, failing if the size of the memory buffer is insufficient.

The Jam Player will also use some space on the system stack. This system stack usage should be small (a few Kbytes at the most), and is a necessary consequence of implementing the Jam Player in a high-level language (‘C’).

4. Executable Programs

4.1 The Altera Jam Player (jam.exe)

The PC executable program **jam.exe** is a version of the Altera Jam Player, equipped with appropriate interface code for reading a Jam program from a file, and controlling a JTAG hardware interface using the Altera ByteBlaster parallel port download cable. The program can also read the NOTE fields and check the CRC of a Jam program. The program is built as a WIN32 console mode application, meaning that it must be invoked from a command prompt in Windows 95 or Windows NT 4.0. The syntax of **jam.exe** is as follows:

```
jam [-h] [-v] [-m<bytes>] [-p<port>] <filename>
```

Options:

-d : define an initialization value for a variable
-h : help message
-m : memory workspace size (in bytes)
-p : specify LPT number for ByteBlaster
-v : verbose (display NOTEs, CRC, execution time)

Example:

```
jam -v -p2 -d"DO_PROGRAM=1" -d"DO_SECURE=1" program.jam
```

This command executes the Jam program **program.jam**, initializing the variables DO_PROGRAM and DO_SECURE to 1. This command will cause the Jam program to program and secure a device. Also, because the verbose option is selected, all NOTE strings will be displayed, along with the CRC value and the elapsed time for the operation.

4.2 The Jam Data Conversion Utility (jamdata.exe)

The PC executable **jamdata.exe** converts data from a binary file to any of the four representations supported for Boolean arrays in the Jam language. It can also convert data in any of those formats back to a binary file. It is a WIN32 console mode application. The syntax is as follows:

```
jamdata [-h] [-d] [-e] -f<format> <input-file> [<output-file>]
```

Options:

-d : decode one of the Jam formats into a binary file
-e : encode a binary file into one of the Jam formats
-f : format (BIN, HEX, RLC, or ACA)
-h : help message

Examples:

```
jamdata -fACA -e data.bin data.aca
jamdata -fACA -d data.aca data.new
```

The first command encodes the information in the binary file **data.bin** using the ACA compression algorithm, and stores the resulting ACA data in the text file **data.aca**. The second command decodes the file **data.aca**, using the ACA decompression algorithm, and stores the resulting data in the binary file **data.new**. The file **data.new** should be identical to the original file **data.bin**. Try it!

5. Creating Jam Programs

5.1 General Guidelines

The Jam Programming and Test Language is a flexible language, capable of supporting a wide range of applications. Because the application of interest to Altera is the programming of logic devices using the JTAG interface, the following guidelines apply mostly to Jam Programs that are used for device programming.

- Jam programs that program a device should have NOTE fields specifying the DEVICE to be programmed and the CREATOR of the Jam program (usually the CREATOR is the name and version of the software that generated the file).
- Jam programs should have a CRC. The CRC is important to ensure the integrity of the file.
- Jam programs that program a device should use the variables DO_BLANKCHECK, DO_PROGRAM, DO_VERIFY, and DO_SECURE in the manner described in the Jam Language Specification. This methodology permits the Jam Player to utilize these different functions in every Jam program.
- Jam programs should always terminate with an EXIT code of zero for success, and non-zero for failure. These EXIT codes are the only way that the host system can determine the outcome of the operation performed by the Jam program. (If the Jam program terminates prematurely because of a run-time error, that fact will be indicated by the return code from the Jam Player entry point function `jam_execute`. That return code is a different code and should not be confused with the EXIT code from the Jam program.)

5.2 Optimizing for Speed

To speed up the execution of a Jam program, try to reduce the number of Jam statements which must be processed. Speed increases can be achieved by avoiding manipulations of array elements; where possible, you should arrange array elements in a convenient order in an initialized array before execution begins. Commonly used array patterns may be stored in initialized arrays rather than calculated “on the fly” in Jam code. On the other hand, if a complex arithmetic calculation must be performed on Boolean array variables, it could be faster to convert the quantities into integers and use the built-in arithmetic functions of the Jam Player, rather than perform bit manipulations on the arrays.

Using literal values for constant quantities, rather than named variables, eliminates the need for the Jam Player to look up the value in the symbol table, thus saving execution time. This technique can be used for integer and Boolean quantities, and also for constant Boolean array patterns. Of course, this technique could make the code harder to understand. If you expect the Jam program to be readable by human beings, use this technique in moderation, and add comments as appropriate to explain what the literal quantities represent.

5.3 Reducing File Size

Most Jam programs for device programming will have one or more large Boolean arrays to store the programming data. Using one of the compressed data formats (either RLC or ACA) can dramatically reduce the file size of such programs. The Jam data conversion utility, **jamdata.exe**, converts data into these formats. This program can be used to convert data into a form to be included in a Jam program, or the conversion can be integrated into other software which generates Jam program files. The ‘C’ language source code for the **jamdata.exe** program is provided by Altera.