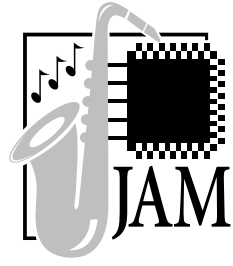
Jam Programming & Test Language Specification

Preliminary Information



Version 0.9
July 8, 1997

Developed by:
Altera Corporation

Contents

Jam Programming & Test Language Specification

Introduction

The Jam™ programming and test language is designed to support the programming of programmable logic and memory devices through the IEEE 1149.1 JTAG interface. The Jam language is an interpreted language, meaning that the Jam program source code is executed directly by an interpreter program running on a host computer, without first being compiled into binary executable code. As a Jam program is executed by the interpreter, signals are produced on the IEEE 1149.1 JTAG interface, as described in the Jam program. The Jam language supports any JTAG 1149.1-compliant programmable logic device. Both the source code and interpreter are intended to be easily portable to any system with minimal effort and resource usage.

Language Overview

The Jam language is a line-oriented language consisting of a sequence of program statements. Jam statements consist of labels (optional), instructions, and arguments, and terminate with a semicolon (;). Arguments may be literal constants, variables, or expressions resulting in the desired data type (i.e., Boolean or integer). Each statement usually occupies one line of the Jam program file, but this is not required. Line breaks are not significant to the Jam language syntax, except for terminating comments. A REM instruction or a single quotation mark (') can be used to signify a comment, which is ignored by the interpreter.

Program Flow

The program flow in the Jam language is controlled using simple GOTO, CALL/RETURN, and FOR/NEXT structures. The GOTO and CALL instructions refer to labels, which are symbolic names for program statements located elsewhere in the Jam program. The language itself enforces almost no constraints on the organizational structure or control flow of a program.

Data Management

All variables in the Jam language must be declared before they are used, and they always have global scope (i.e., they are available to all statements encountered after the declaration statement). Jam programs have variables of two types: integer and Boolean. Integers are 32-bit signed numbers. Boolean variables can be considered to be single-bit unsigned integers, although they cannot be used interchangeably with integer variables. One-dimensional Boolean or integer arrays can be declared. These arrays are indexed to give access to a single element or a range of elements inside the array. Multi-dimensional arrays are not supported.

The Jam language does not support string variables. However, string constants and string representations of integer values can be used to form text output messages. A complete set of arithmetic, logical, and relational operators is available for integers, and a complete set of logical operators is provided for Boolean expressions. No operators are provided to work directly on integer arrays or Boolean arrays. For strings, concatenation is available to permit the construction of simple messages.

The initialization value of scalar integer or Boolean variables can be set at run-time using an “initialization list”, a list of variable names and values supplied to the Jam interpreter at run-time. These values override the initialization values found in the Jam program. This mechanism permits a single Jam file to perform multiple operations (e.g., device programming and fault testing). To use this feature, the software that invokes the Jam interpreter must know the names and values to supply in the initialization list to obtain the desired result. The initialization list is described in greater detail in “Initialization List Conventions” on page 30.

The Jam language is not case sensitive. All labels, variable names, instruction names, and other language elements are processed without regard to case. (The only exception is the encoded format used for compressed Boolean array initialization data, which is described in “Data Management” on page 6.) In this document, Jam code examples use uppercase instruction and keyword names and lowercase label and variable names, but the language does not require this convention. For string constants in `PRINT` statements, the case is preserved when printing the string.

Goto Appendix A on page 33 for a sample Jam File.

I/O

The only input and output mechanisms supported in the Jam language are the JTAG hardware interface, the initialization list for run-time variable initialization, the `PRINT` instruction for output messages, and the `EXPORT` instruction for sending data values to the calling program. The `EXPORT` instruction transmits information from the Jam program to the calling program using a callback function. The `EXPORT` instruction can be used to relay the current execution status, or to pass other information. The information transmitted by the `EXPORT` instruction consists of a key string and an integer value. The significance of the integer value depends on the key string. See Table 15 on page 30 for a list of defined key strings.

The Jam language does not provide access to any other input or output files or devices. In addition, no facility exists within the Jam language for linking multiple Jam programs together, or for including the contents of another file into a Jam program.

Statements

Each statement in a Jam program contains up to three elements: a label (optional), an instruction, and arguments. The number and type of arguments depends on the instruction. A semicolon (;) terminates the statement.

Labels (Optional)

Labels provide a means of branching within the program. A unique label can begin each Jam program statement and must be followed by a colon (:). Label names are not case sensitive (i.e., two label names that differ only by case are considered equal).

Instructions

Each instruction begins with one of the following instruction names. “Jam Instruction Specifications” on page 19 provides a detailed description of each instruction name.

- BOOLEAN
- CALL
- CRC
- DRSCAN
- DRSTOP
- EXIT
- EXPORT
- FOR
- GOTO
- IF
- INTEGER
- IRSCAN
- IRSTOP
- LET
- NEXT
- NOTE
- PADDING
- POP
- PRINT
- PUSH
- REM
- RETURN
- STATE
- WAIT

These instructions take arguments in the form of variables or expressions, except for the following:

- The GOTO and CALL instructions take labels as arguments.
- The PRINT instruction takes a string expression as an argument.
- The JTAG instructions, DRSCAN and IRSCAN, take Boolean array expressions as arguments.
- The RETURN instruction takes no arguments at all.

The Jam interpreter checks each argument for a valid variable or expression type.

Table 1 shows the sixteen state names that are reserved keywords in the Jam language. These keywords correspond to the state names specified in the JTAG 1149.1 specification.

Table 1. Reserved State Names	
JTAG 1149.1 State Names	**Jam Reserved State Names**
Test-Logic-Reset	RESET
Run-Test-Idle	IDLE
Select-DR-Scan	DRSELECT
Capture-DR	DRCAPTURE
Shift-DR	DRSHIFT
Exit1-DR	DREXIT1
Pause-DR	DRPAUSE
Exit2-DR	DREXIT2
Update-DR	DRUPDATE
Select-IR-Scan	IRSELECT
Capture-IR	IRCAPTURE
Shift-IR	IRSHIFT
Exit1-IR	IREXIT1
Pause-IR	IRPAUSE
Exit2-IR	IREXIT2
Update-IR	IRUPDATE

The following sixteen strings are also reserved keywords in the Jam language, due to their significance in Jam statements or expressions:

- ABS
- BIN
- CAPTURE
- CEIL
- CHR$
- COMPARE
- CYCLES
- FLOOR
- HEX
- LOG2
- RLC
- STEP
- SQRT
- THEN
- TO
- USEC

Comments

Comments, which are ignored by the Jam interpreter, can be placed anywhere in the program using either a single quotation mark (‘) or the REM instruction. A single quotation mark (‘) indicates the beginning of a comment and a line break indicates the end of the comment.

The REM instruction also signifies a comment, or remark, statement. Remark Statements can span multiple lines and are terminated with a semicolon (;). The REM instruction is useful for temporarily removing a statement from a program without regard to the location of line breaks.

Program Flow

A Jam program always begins with the first line and terminates with the EXIT instruction. If the interpreter encounters the end of the file with no EXIT instruction, an error occurs. The flow of execution in a Jam program is controlled using three methods: branches, subroutine calls, and loops.

The Stack

The Jam interpreter manages subroutine calls and loops using a stack. The stack is a repository for information about all activities that can be nested. These nested functions are CALL and RETURN, FOR and NEXT, and PUSH and POP. When a CALL, FOR, or PUSH instruction is encountered, information about the function is added to the stack. When the corresponding RETURN, NEXT, or POP instruction is encountered, the record is removed from the stack. (For the NEXT instruction, the stack record is removed only when the loop has run to completion.)

GOTO

The GOTO instruction causes execution to jump to the instruction corresponding to the label. This label may or may not have been encountered already in the Jam program. If the label was not encountered, the remainder of the Jam program will be processed (without executing any instructions) until the label is found, or until the end of the program is reached. If the label is found, execution of the program will continue from that point.

The IF instruction can be used with the GOTO instruction to create a conditional branch.

CALL & RETURN

The CALL instruction is like the GOTO instruction, but the location of the instruction following the CALL is saved on the stack inside the Jam interpreter. When a RETURN instruction is executed, execution will jump to the statement following the CALL instruction, and the record is deleted from the stack. If a RETURN instruction is executed when the stack is empty or does not have a CALL record on top, an error occurs. The program will terminate with a corresponding error code.

The IF instruction can be used with the CALL and RETURN instructions to call a subroutine conditionally, or to return conditionally.

FOR Loops

The FOR instruction is used for iteration or "looping". Each FOR instruction has an associated integer variable called the "iterator", which maintains a count of the iterations. When a NEXT instruction using the same iterator variable is encountered, the iterator is incremented (or stepped, if the STEP keyword is used with the FOR instruction). If the iterator has reached its terminal value and the body of the loop has been executed for the last time, the FOR loop is complete and control is passed to the instruction following the NEXT instruction. Otherwise, control jumps back to the instruction following the FOR instruction.

FOR loops can be nested. When a FOR instruction is encountered, the Jam interpreter pushes a FOR record onto the stack. This record stores the name of the iterator variable and the location of the FOR instruction. When the corresponding NEXT instruction is encountered, the iterator variable is incremented, and the terminating condition is evaluated. If the FOR loop has reached its terminal value and the body of the loop has been executed for the last time, the FOR loop record is deleted from the stack and control jumps to the instruction following the NEXT instruction. If the FOR loop has not reached its terminal value, control continues at the instruction following the FOR instruction. If a NEXT instruction is encountered and the top record on the stack is not a FOR record with the same iterator variable, or if the stack is empty, an error occurs. When nesting one FOR loop inside another, the inner loop must run to completion before the NEXT instruction of the outer loop is encountered. When nesting a FOR loop inside a subroutine, the FOR loop must run to completion before the RETURN instruction is encountered.

Data Management

Variable Names

Variable names are limited to 32 characters, and must begin with an alphabetic character (not a number). Variable names consist of alphabetic characters, numeric characters, and the underscore (_) character—no other characters are allowed. Variable names are not case sensitive (i.e., two variable names that differ only by case are considered equal).

Types

The two data types available in the Jam language are integer and Boolean. These types may be used to declare "scalar" variables and one-dimensional arrays. Any variable or array must be declared before any reference to it is made.

Initialization

By default, the Jam interpreter initializes all variables and arrays to zero when they are created. Variables and arrays can also be explicitly initialized at the time of declaration. Arrays with explicit initialization are always "read-only" (i.e., the Jam program cannot modify any element of the array). For initialization of Boolean arrays, the initial array data can be specified in one of four ways:

- A comma-separated list of values
- Binary (one bit per character)
- Hexadecimal (four bits per character)
- Modified hexadecimal format containing run-length compression codes (more than four bits per character)
- Advanced Compression Algorithm (ACA)

To initialize integer arrays, the initial array data must be specified as a comma-separated sequence of decimal numbers. In both cases, the size of the initialization data must exactly equal the size of the array to be initialized; otherwise, an error occurs. Array data can be accessed three ways:

- Indexing (using an integer) resulting in a single scalar value
- Sub-range indexing (using two integers) resulting in a smaller array
- Collectively as an array

Boolean arrays and sub-range indexed arrays can only be used as arguments with `DRSCAN` and `IRSCAN` instructions, which accept Boolean array arguments. No arithmetic, logical, relational, or assignment operators are provided for whole arrays or sub-range indexed arrays.

Literal Values

Literal data values may appear in integer or Boolean expressions. For example, in the statement `LET a = a + 1`, the number one is a literal value. The literal values 0 and 1 may be used in either integer or Boolean expressions; other signed decimal numbers between –2147483648 and 2147483647 can be used only in integer expressions. Only decimal format is supported.

For Boolean array expressions, a literal Boolean array value can be expressed as a Hexadecimal string. Such literal arrays can be used as arguments with DRSCAN and IRSCAN instructions, which expect Boolean arrays as arguments. In this case, as with initialization, if the size of the literal array is different from the expected size, an error occurs. Hexadecimal Boolean array constants must begin with a numeric character to avoid confusion with variable names. For example, "FF" must be expressed as "0FF". No format is supported for literal use of integer arrays.

Text strings must be specified as literal values for the PRINT statement, since the Jam language does not support any character or string variable types.

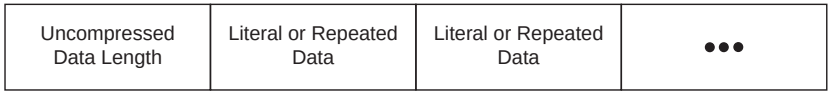
Constants

Since the Jam language is an interpreted language, no special syntax is provided for integer or Boolean constants. A variable should be declared with an initialized value when a symbolic name for a quantity is desired.

Advanced Compression Algorithm (ACA)

The algorithm and syntax for converting raw binary data into ACA compressed format is described in this section. ACA format achieves compression by storing raw data in two sections: literal data and repeated data. Both sections are preceded by an uncompressed data length block that specifies the length of the data when it is uncompressed. Figure 1 shows the overall structure of data in the ACA format.

Figure 1. ACA Data Structure



The uncompressed data length block is a binary, scalar value that can be up to 4 bytes long. Thus, the total length of binary data that can be compressed by the ACA algorithm is $2^{32\ bytes} - 1$ bytes of raw data. The bytes are ordered in the Little Endian format, meaning that the order of the bytes are flipped around within the block, but the order of the bits within the byte remain intact. For example, the data HEX 12345678 would be ordered HEX 78563412. Figure 2 shows the uncompressed data length block.

Figure 2. ACA Uncompressed Data Length Block

Original Byte Order	31..24	23..16	15..8	7..0
Little Endian Format	7..0	15..8	23..16	31..24

The literal data section begins with a bit whose value is 0 and is followed by 3 bytes of uncompressed data (see Figure 3). When inflated, this data section is copied directly to the output stream.

Figure 3. ACA Literal Data Section

Bits	0	1..8	9..16	31..24
Position of Data Bytes in Block	0	Byte 0 (LSB)	Byte 1	Byte 2 (MSB)

Sections of raw data that are repetitions of previous data are stored in the repeated data section. The repeated data section begins with a bit whose value is 1. The next 1 to 13 bits compose the scalar offset, which is followed by a byte specifying the length. When inflating the compressed information, data is copied to the output stream. The offset value, represented by a variable number of bits, specifies the number of bytes back in the output stream where the repeated data begins. The length variable provides the number of bytes of data that are repeated. For example, if the offset value is 8 and the length value is 5, the data pointer goes back 8 bytes in the uncompressed output stream and copies the 5 bytes from that location. Figure 4 illustrates the form of the repeated data block.

Figure 4. ACA Repeated Data Section

Bits	1	N..1	8..1
Bit Function	Constant Bit	Offset	Length
Limits	1 bit	Up to 13 bits $1 \le N \le (2^{13}) - 1$	Up to 8 bits $4 \le \text{Length} \le 255$

The following example illustrates how the ACA algorithm would treat a set of binary data. In this case, the raw binary data is composed of the 24 bytes shown in Table 2.

Table 2. Sample Uncompressed Data

Offset	Data	ASCII	Binary
0	61	a	01100001
1	62	b	01100010
2	63	c	01100011
3	64	d	01100100
4	65	e	01100101
5	66	f	01100110
6	61	a	01100001
7	62	b	01100010
8	63	c	01100011
9	64	d	01100100
10	65	e	01100101
11	66	f	01100110
12	67	g	01100111
13	68	h	01101000
14	69	i	01101001
15	6A	j	01101010
16	6B	k	01101011
17	6C	l	01101100
18	64	d	01100100
19	65	e	01100101
20	66	f	01100110
21	61	a	01100001
22	62	b	01100010
23	63	c	01100011

Once the ACA algorithm formats the raw data, the data is compressed. Table 3 shows how the sample data appears after it is compressed.

Table 3. Sample Compressed Data

Offset	Data	Binary
0	18	00011000
1	00	00000000
2	00	00000000
3	00	00000000
4	C2	11000010
5	C4	11000100
6	C6	11000110
7	90	10010000
8	95	10010101
9	99	10011001
10	B5	10110101
11	81	10000001
12	33	00110011
13	B4	10110100
14	34	00110100
15	6A	01101010
16	6B	01101011
17	6C	01101100
18	9F	10011111
19	01	00000001

In this example, the original data is compressed from 24 bytes to 19 bytes. Compression is achieved by recognizing that the sequence abc and def are repeated in the original data block. These sequences are then compressed in two repeated data sections. Table 4 shows the compressed data of Table 3 in serial form with its values.

Table 4. Compressed Data (Part 1 of 4)

Section	Uncompressed Data Length				Literal Data			
Binary Data	00011000	0000000	00000000	00000000	0	01100001	01100010	01100011
Hexadecimal Data	18	0	0	0	0	61	62	63
ASCII Value	Length = 24 Bytes				–	a	b	c

Table 4. Compressed Data (Part 2 of 4)

Section	Literal Data				Repeated Data		
Binary Data	0	01100100	01100101	01100110	1	110	00000110
Hexadecimal Data	0	64	65	66	1	6	5
ASCII Value	–	d	e	f	–	–	abcdef

Table 4. Compressed Data (Part 3 of 4)

Section	Literal Data				Literal Data			
Binary Data	0	01100111	01101000	01101001	0	01101010	01101011	01101100
Hexadecimal Data	0	67	68	69	0	6a	6b	6c
ASCII Value	–	g	h	i	–	j	k	l

Table 4. Compressed Data (Part 4 or 4)

Section	Repeated Data		
Binary Data	1	01111	00000110
Hexadecimal Data	1	f	6
ASCII Value	–	–	defabc

Run-Length Data Compression

This section describes the algorithm and syntax for converting initialization data for Boolean arrays into run-length compressed (RLC) form.

Uncompressed data is searched (beginning with the bit that appears at index zero in the array) for any sequence of repeated ones or zeros, which are located at contiguous positions in the array and satisfy a minimum sequence length. These sequences are stored into the compressed array as constant blocks. The random data outside those sequences is stored as random data blocks. Raw binary data is converted to ASCII data characters for storage. To make the conversion, a table is constructed to encode the actual data as a subset of ASCII characters. The character set used for compressed arrays is the set of digits 0-9, uppercase and lowercase alphabetic characters (A-Z) and (a-z), the underscore character (_), and the "at" character (@). These 64 characters are used to represent numeric quantities. The encoding of the binary values as ASCII characters is implemented by the 'C' program code shown in Figure 5.

Figure 5. C Programming Code to Encode Binary Values as ASCII Characters

```
if      ((ch >= '0') && (ch <= '9')) result = (ch - '0');
else if ((ch >= 'A') && (ch<= 'Z')) result = (ch + 10 - 'A');
else if ((ch >= 'a') && (ch <= 'z')) result = (ch + 36 - 'a');
else if (ch == '_') result = 62;
else if (ch == '@') result = 63;
```

With this table, the numeric values from 0 to 63 are encoded as ASCII characters. Thus, a single ASCII character represents 6 bits of raw binary data (i.e., 6 = log2(64)).

Using the encoding program from Figure 1, random data and constant data can be stored in RLC blocks. The structure of the RLC block is as follows. Each block is preceded by an ID byte, followed by characters that represent the number of data bits to be compressed, followed by the compressed form of the actual binary data. Tables 5 and 6 show a single block of data in compressed form.

Table 5. RLC Random Data Block										
Block ID Byte	**Count Data (Up to 7 ASCII Characters)**							**Actual Data (Up to 7×10^{11} ASCII Characters)**		
See Table 7 for values	Char 1	Char 2	Char 3	Char 4	Char 5	Char 6	Char 7	Char 1	...	Char *N*

Table 6. RLC Constant Data Block							
Block ID Byte	**Count Data (Up to 7 ASCII Characters)**						
See Table 7 for values	Char 1	Char 2	Char 3	Char 4	Char 5	Char 6	Char 7

Block ID Byte

The block ID byte functions as a header for the whole block. The ID byte specifies whether the data is constant or random, the number of data bits stored, and, for constant data, the value of the data stored. Tables 7 and 8 show the configuration and function of the ID byte.

Table 7. Block ID Bit Configuration								
Bit Position	8	7	6	5	4	3	2	1
Bit Value	0	1	0	Type	Value	Count 2	Count 1	Count 0

Table 8. Block ID Bit Functions			
Bit	**Description**	**Function**	**Possible Values**
5	Type	Identifies the type of data block	0: Constant block 1: Random block
4	Value	Provides value of constant block	0: Constant zero block 1: Constant one block 0: Random block
3..1	Count 2 - 0	Encodes number of count characters	1 through 7

The three most significant bits (MSBs) of the ID byte always take the constant values shown in Table 4. Thus, the ID byte will always be represented by an upper case ASCII alphabetic character (A-Z).

For random data blocks, the state of the value bit is a zero. Count bits are always presented in decreasing order of significance. The count bits specify the number of count data characters that follow. Since there are 3 count bits, the maximum number of count data characters that can follow the ID byte are 7.

Count Data

The count data characters specify the number of data bits. Each count character encodes six bits of the complete count value. For example, three count characters can store an 18-bit binary count value. The count data section can store up to a 42-bit scalar value. The number of data characters can be calculated from the count value by simply dividing by six, and adding one if the remainder is not zero (i.e., if the number of data bits is not an integer multiple of six).

Actual Data

The actual data characters represent the original binary information for random data blocks. The value for a constant data blocks are stored in the block ID byte. The actual data contains up to 7×10^{11} ASCII characters (with 7 count data characters, representing 42 bits: 2^{42} bits / 6 bits / character = 7×10^{11} characters). Each ASCII character represents 6 bits of the random data for the block, making 4×10^{12} bits of binary storage. The six bits are taken LSB first, meaning that the LSB has the least array index. The character following the expected actual data characters should be interpreted as the block ID character for the next block.

Once the binary data is compressed, an array will exist that contains random and constant data blocks. This array of compressed data can then be used to initialize a `BOOLEAN` variable. Run-length compression is only supported in the `BOOLEAN` statement, when declaring a Boolean array variable with initialization data. The syntax for the statement is shown below.

`BOOLEAN vector[`<*size*>`] = RLC` <*compressed array data*>`;`

where <*size*> is the number of bits present when the array is uncompressed.

Expressions & Operators

Expressions

An expression in the Jam language is a collection of variables, literal data values, or other expressions joined together by operators to describe a computation. Parentheses may be used to control the precedence of evaluation. The result of every expression, applied as an instruction argument, must match the instruction type.

Integer & Boolean Operations

The Jam language offers a complete set of arithmetic, logical, and relational operators. The character codes used for these operators are similar to the operators used in the 'C' programming language. The assignment operator (=) is not included in this list because it is considered to be part of the LET statement. The ternary operator in the 'C' language (A = B ? C : D) is not supported in the Jam language. Arithmetic and logical operators always produce the same type of result as used by the arguments (i.e., integer arguments produce integer results, and Boolean arguments produce Boolean results). The relational operators always produce a Boolean result.

The arithmetic and logical operators described in Tables 9 and 10 take one or two integer arguments and produce an integer result.

Table 9. Operators Yielding an Integer Result *Note (1)*

Operator	Description
~	Bitwise unary inversion
* / %	Multiplication Division Modulo
+ –	Addition Subtraction and unary negation
<< >>	Left shift Right shift
&	Bitwise logic AND
^	Bitwise logical exclusive OR
|	Bitwise logical OR

Note:
(1) Each row holds operators with identical precedence. Operators that are in higher rows take precedence over operators in lower rows.

Table 10. Function with a Single Integer Argument & an Integer Result

Function	Description
ABS( )	Absolute value
LOG2( )	Logarithm base 2
SQRT( )	Square root
CEIL( )	Ceiling (least integer which is greater than..)
FLOOR( )	Floor (greatest integer which is less than..)

The results of division and square-root operations are rounded down to the nearest integer value. The ceiling (CEIL) function can be used on the result of a division or square-root operation to round the result up. For division, the result may be a negative number. In this case, the result is rounded toward zero by default. The CEIL function can be used to round the result to the more negative value.

The result of LOG2 is rounded up to the nearest integer value. The floor (FLOOR) function can be used on the result of LOG2 to round the result down.

The relational operators described in Table 11 take two integer arguments and produce a Boolean result.

Table 11. Operators with Integer Arguments & a Boolean Result

Operator	Description
==	Equality comparison
!=	Inequality comparison
>	Greater comparison
<	Less comparison
>=	Greater or equal comparison
<=	Less or equal comparison

The logical and relational operators described in Table 12 take two Boolean arguments and produce a Boolean result (except the unary inversion operator, which takes one Boolean argument).

Table 12. Operators with Two Boolean Arguments & a Boolean Result

Operator	Description
&&	Logical AND
||	Logical OR
!	Unary inversion
==	Equality comparison
!=	Inequality comparison

Note that the equality and inequality comparison operators (== and !=) are used for both integer and Boolean arguments. However, both arguments must be either Boolean or integers (i.e., an integer argument cannot be directly compared to a Boolean argument).

Table 13 shows the precedence of operations, in descending order of priority. However, parentheses can be used to force the precedence in any expression. The precedence of operations in the Jam language closely resembles the precedence in the 'C' programming language.

Table 13. Operator Precedence

Precedence	Operator	Description
1	!	Unary inversion
2	*, /, %	Multiplication, division, and modulo
3	+, -	Addition, subtraction
4	<<, >>	Shift
5	<, <=, >, >=	Magnitude comparison
6	==, !=	Equality comparison
7	&	Bitwise logical AND
8	^	Bitwise logical exclusive OR
9	|	Bitwise logical OR
10	&&	Logical AND
11	||	Logical OR

Integers and Booleans are never automatically converted; a relational operator must be used to convert an integer to Boolean, since relational operators always give a Boolean result. To convert a Boolean bit to an integer, use an `IF` statement to test the value of the Boolean, and then assign a value to the integer accordingly. The constant literal numbers 0 and 1 can be used either as an integer or as a Boolean value, according to the context.

Array Operations

Square brackets ([ ]) are used to index arrays. The result of indexing is either a single element (integer or Boolean) or a smaller array, representing a subset of the original array. To gain access to a single element of an array, the index consists of a single integer expression. For example, one element of an array can be assigned to another element as follows:

```
LET vect[52] = vect[0];
```

To copy a group of elements from one array to another:

```
FOR i = 0 TO 255;
LET dest[i + 256] = source[i];
NEXT i;
```

An array expression can consist of a range of elements from another array variable. The syntax for this expression is the same as for indexing, but with a start index and stop index, separated by two periods (..). This method is used to provide Boolean array expressions for DRSCAN and IRSCAN commands. For example:

```
DRSCAN length invect[start..stop] CAPTURE outvect;
```

Indexing is the only operation supported for array variables.

String Operations

String operations can be used only in PRINT statements. Integers are converted to strings automatically in the PRINT statement. For example, the following statement prints out the value of an integer variable:

```
PRINT "The signed integer value of a is ", a;
```

The following statement displays the character represented by an integer variable:

```
PRINT "The character in a is ", CHR$(a), " and you can
depend on it.";
```

The CHR$() function coverts an integer value to its ASCII code, allowing the Jam language to print ASCII characters in a more elegant manner. For example, the ^G(BEL) character could be printed to make the system beep when a programming error occurs. But, instead of printing ^G, it is easier to print CHR$(7), where 7 is the integer equivalent of the ASCII character.

Jam Instruction Specifications

This section describes each instruction in the Jam language.

BOOLEAN

The BOOLEAN instruction declares a variable or an array as being a Boolean type. Boolean variables can be initialized to 0 or 1. Arrays can be initialized using binary, Hexadecimal,RLC, or ACA compressed format. Arrays are used to provide the data that will be scanned into the targeted device, or devices. In many cases, it is desirable to represent the initialized array data as compressed data, in order to be most efficient with system resources. If no initialization is specified, the variable or array will be initialized to zero.

Syntax: BOOLEAN <*variable name*>;

BOOLEAN <*variable name*> = <*Boolean expression*>;

BOOLEAN <*array name*> [<*array size*>];

BOOLEAN <*array name*> [<*array size*>] = <*Boolean array initialization data*>;

Examples: `BOOLEAN status = 0;`

```
BOOLEAN address[32] = BIN
01011010010110100101101001011010;

BOOLEAN data[32] = HEX 34B4CDB7;

BOOLEAN vector[256] = RLC
J2gR1My@x_Vl@NpvTs@h2;

BOOLEAN verify{128] = ACA hd30000t@ztV;
```


CALL

The CALL instruction causes execution to jump to the instruction corresponding to the label, and saves a CALL record on the stack. The RETURN instruction is used to return to the instruction after the CALL instruction.

Syntax: CALL <*label*>;

CRC

The CRC instruction is used to verify the data integrity of the Jam program; it is not an executable instruction. The CRC instruction should be located at the end of the Jam file, after all executable Jam instructions—including the EXIT instruction. When checking the integrity of the Jam program, the Jam interpreter calculates the CRC (cyclic redundancy check) of all characters in the file or memory buffer, including comments and white-space characters, up to (but not including) the CRC statement. The CRC value obtained is then compared to the value found in the CRC statement. If the CRC values agree, the data integrity of the Jam program is verified. The CRC computation performed is based on the CCITT standard.

If the CRC statement is encountered during execution of the Jam program, an error occurs.

Syntax: CRC <*4-digit hexadecimal number*>;

Example: `CRC 9C4A;`

DRSCAN

The DRSCAN instruction specifies a data register scan pattern to be applied to the target data register. The scan data shifted out of the target data register may be captured (in a Boolean array variable), compared (to a Boolean array variable or constant), or ignored. The data register length is an integer expression. DRSCAN can be used to gather data that needs to be branched upon (CAPTURE), or it can be used to perform a verify (COMPARE), for example. The scan data array contains the data to be loaded into the data register. The data is shifted in increasing order of the array index, that is, beginning with the least index. The capture array is a writable Boolean array variable (i.e. not an initialized array). The compare array and mask array are Boolean arrays, and the result is a Boolean variable which receives the result of the comparison. An unsuccessful comparison will cause a zero (or FALSE) value to be stored in the result variable, but will not interrupt the Jam program execution. To abort in the case of an error, a conditional (IF) statement must be used to test the result value, and the EXIT statement called to stop the program.

The DRSCAN instruction will only take variables as its arguments, with exception to the <*length*>.

Syntax: DRSCAN <*length*>, <*scan data array*>;

DRSCAN <length>, <*scan data array*>, CAPTURE <*capture array*>;

DRSCAN <*length*>, <*scan data array*>, COMPARE <*compare array*>, <*mask array*>, <*result*>;

Examples:
```
DRSCAN 15, add[14..0];

DRSCAN 20, datain[20..1], CAPTURE
dataout[20..1];

DRSCAN 41, indata[41..0], COMPARE
expecteddata[41..0], maskdata[41..0],
verify_error;
```


DRSTOP

Specifies the JTAG end state for data register scan operations. The default state is IDLE, when no state name is provided. Once an end state is specified, all subsequent data register scan operations will park in that end state, until another DRSTOP command is encountered.

Syntax: DRSTOP <*state name*>;

EXIT

Terminates the Jam program with specified error code. By convention, an error code of zero indicates success, and non-zero values indicate error conditions.

Syntax: EXIT <*integer expression*>;

EXPORT

The EXPORT instruction exports a key string and an integer value to the calling program via a callback function. The calling program should ignore exported data if the key string is not recognized. A set of standard key strings is defined in “Conventions” on page 29.

Syntax: EXPORT <*key string*>, <*integer expression*>;

Example:
```
EXPORT "Percent done", (done * 100) / total;
```

FOR

The FOR instruction initiates a loop. When a FOR statement is encountered, a FOR record is added to the stack, identifying the iterator variable and storing the line of the FOR statement itself. The NEXT statement continues or terminates the loop. When the NEXT statement is encountered, if the loop has not yet run to completion, control jumps to the statement after the FOR statement. If the loop has run to completion, control jumps to the statement following the NEXT instruction, and the FOR record is removed from the stack.

Syntax: FOR <*integer variable*> = <*integer-expr*> TO <*integer-expr*>;

FOR <*integer variable*> = <*integer-expr*> TO <*integer-expr*>
STEP <*integer-expr*> ;

Example:
```
FOR index = 0 TO (maximum - 1);

LET accumulator = accumulator + vector[index];
NEXT index;
```


GOTO

The GOTO instruction causes execution to jump to the instruction corresponding to the label. If the label is not already known, the Jam interpreter will search ahead in the Jam program for the desired label.

Syntax: GOTO <*label*>;

IF

Evaluates a Boolean expression, and if the expression is true, executes a statement. In general, IF statements are used to branch based on information that has been read from the targeted device.

Syntax: IF <*Boolean expression*> THEN <*statement*>;

Examples:
```
IF a > b THEN GOTO greater;

IF a < b THEN CALL less;

IF a == b THEN RETURN;
```

INTEGER

The INTEGER instruction declares an integer variable or array. Integer variables may be initialized to a value between –2147483648 (-2^{31}) and 2147483647. Integer arrays can be initialized using a comma-separated list of decimal integer values. Arrays with explicit initialization data are read-only. By default, any variable or array without initialization data is initialized to zero.

Syntax: INTEGER <*variable name*>;

INTEGER <*variable name*> = <*integer-expr*>;

INTEGER <*array name*> [<*size*>];

INTEGER <*array name*> [<*size*>] = <*integer-expr*>, ... <*integer-expr*>;

Examples:
```
INTEGER column = -32767;

INTEGER array[10] = 21, 22, 23, 24, 25, 26,
27, 28, 29, 30;
```


IRSCAN

The IRSCAN instruction specifies an instruction register scan pattern to be applied to the instruction register. Data shifted out of the instruction register is ignored. The instruction register length is an integer expression. The instruction data array is a Boolean array expression. The instruction data is shifted into the device in increasing order of the array index. The IRSCAN instruction will only take variables as its arguments, with exception to the <*length*>.

Syntax: IRSCAN <*length*>, <*instruction data array*>;

IRSTOP

The IRSTOP instruction specifies the JTAG end state for instruction register scan operations. When no state name is provided, the default is IDLE. Once an end state is specified, all subsequent instruction register scan operations will park in that end state, until another IRSTOP command is encountered.

Syntax: IRSTOP <*state name*>;

LET

The LET instruction assigns value of integer expression to integer variable, or value of Boolean expression to Boolean variable. The LET statement can be used to assign individual array elements, but not arrays.

Syntax: LET <*integer variable*> = <*integer-expr*>;

LET <*Boolean variable*> = <*Boolean-expr*>;

NEXT

The NEXT instruction causes the program execution to jump to the corresponding FOR instruction, where the value of the iterator variable will be compared to the terminal value. If the loop is complete, execution proceeds to the line following the NEXT instruction, and the corresponding FOR record is deleted from the stack; otherwise, execution proceeds at the line following the FOR instruction.

Syntax: NEXT <*variable name*>;

NOTE

The NOTE statement has no effect on program execution; it is used to store information about the Jam program file that can be extracted without actually executing the Jam program. The information stored in NOTE fields may include any type of documentation or attributes related to the particular Jam program.

The meaning and significance of the NOTE field is determined by a note type identifier string, or “key” string. A set of standard key strings is provided in the section on “Conventions” on page 29. Key strings are not case sensitive, and they may be enclosed in double quotation marks. The note text string also may be enclosed in double quotation marks. (The double quotation marks are not considered part of the text string itself.) Like a comment, the NOTE statement can be placed anywhere in the Jam program.

Syntax: NOTE <*type identifier*> <*note text*>;

Examples:
```
NOTE USERCODE "001EDFFF";

NOTE DATE "5/19/97";
```

PADDING

The PADDING statement specifies the number of padding bits to shift in before and after all subsequent data register scan operations and instruction register scan operations. These padding bits are always ones. The PADDING statement takes four integer arguments: data register pre-padding and post-padding bit counts, and instruction register pre-padding and post-padding bit counts. All arguments must be non-negative integer expressions. The padding values affect all subsequent DRSCAN and IRSCAN statements, until a new PADDING statement is executed to change the padding values.

Syntax: PADDING <*pre-data*>, <*post-data*>, <*pre-instruction*>, <*post-instruction*>;

Example: `PADDING 1, 0, 5, 0;`

POP

The POP statement removes a PUSH record from the stack, storing the data value into an integer or Boolean variable. If a Boolean expression is PUSHed, it will be stored on the stack as an integer 0 or 1. Any value may be POPed into an integer variable. If the stack is POPed into a Boolean variable, the value on the stack must be 0 or 1, otherwise an error will occur.

Syntax: POP <*integer variable*>;

POP <*Boolean variable*>;

Example:
```
PUSH 3 - 2;'Integer expression

POP status; 'Boolean variable gets value of 1
(TRUE)
```

PRINT

The PRINT instruction prints a message on the output device, if one is installed. The specific details of writing to the output device are handled by the Jam interpreter. If no output device exists, the PRINT statement has no effect. A string expression consists of string constants, integer expressions, and characters generated by the character-code-conversion function (CHR$), concatenated with commas.

Syntax: PRINT <*string-expr*>;

Examples:
```
PRINT "The integer value ", a, " corresponds
to the character code ", CHR$(a);
```

PUSH

Adds a PUSH record to the stack storing an integer data value. The subsequent POP statement removes the PUSH record from the stack and stores the data value into the corresponding variable. If a Boolean expression is PUSHed, it will be stored on the stack as an integer 0 or 1. If the stack is POPed into a Boolean variable, the value on the stack must be 0 or 1, otherwise an error will occur.

Syntax: PUSH <*integer-expr*>;

PUSH <*Boolean-expr*>;

Example:
```
PUSH 3 + 2;

POP a;' Integer variable a gets value of 5
```


REM

The interpreter ignores all comments beginning with a REM instruction and ending with a semicolon (;). Comments and quoted strings are recognized inside the REM statement, so the terminating semicolon must not be hidden inside a comment or quoted string.

Syntax: REM <*ignored text*>;

Example:
```
REM This statement will be ignored;
```

RETURN

Jumps to the statement after the corresponding CALL statement, and removes the CALL record from the stack. If the top record on the stack is not a CALL record, an error will occur. In the example below, the Jam program will first print “Scan complete”. Next, the RETURN will cause program execution to jump to the Print_Error label, where “Failed to read silicon id” will be printed.

Syntax: RETURN;

Example:
```
CALL Print_Message;

Print_Error: PRINT “Failed to read silicon
id”;
```

```
Print_Message: PRINT "Scan complete";

RETURN;
```

STATE

The STATE instruction causes the JTAG state machine to go to the specified state. The path to the end state may be delineated explicitly, by providing each state between the start state and the end state. Otherwise, the states traversed will default to the paths outlined in Table 14. If the specified start or end states are not a current or final state, as shown in Table 14, the state machine is undefined.

Syntax: STATE *<state name 1> <state name 2> . . . <state name n>*;

Examples:
```
STATE IRPAUSE;

STATE IREXIT2 IRSHIFT IREXIT1 IRUPDATE IDLE;
```


Table 14. State Table

Current State	Final State	State Path
RESET	RESET	Single TCK cycle applied with TMS = 1
RESET	IDLE	RESET-IDLE
RESET	DRPAUSE	RESET-IDLE-DRSELECT-DRCAPTURE-DREXIT1-DRPAUSE
RESET	IRPAUSE	RESET-IDLE-DRSELECT-IRSELECT-IRCAPTURE-IREXIT1-IRPAUSE
IDLE	RESET	IDLE-DRSELECT-IRSELECT-RESET
IDLE	IDLE	Single TCK cycle applied with TMS = 0
IDLE	DRPAUSE	IDLE-DRSELECT-DRCAPTURE-DREXIT1-DRPAUSE
IDLE	IRPAUSE	IDLE-DRSELECT-IRSELECT-IRCAPTURE-IREXIT1-IRPAUSE
DRPAUSE	RESET	DRPAUSE-DREXIT2-DRUPDATE-DRSELECT-IRSELECT-RESET
DRPAUSE	IDLE	IDLE-DRPAUSE-DREXIT2-DRUPDATE-IDLE
DRPAUSE	DRPAUSE	Single TCK cycle applied with TMS = 0
DRPAUSE	IRPAUSE	DRPAUSE-DREXIT2-DRUPDATE-DRSELECT-IRSELECT-IRCAPTURE-IREXIT1-IRPAUSE
IRPAUSE	RESET	IRPAUSE-IREXIT2-IRUPDATE-DRSELECT-IRSELECT-RESET
IRPAUSE	IDLE	IRPAUSE-IREXIT2-IRUPDATE-IDLE
IRPAUSE	DRPAUSE	IRPAUSE-IREXIT2-IRUPDATE-DRSELECT-DRCAPTURE-DREXIT1-DRPAUSE
IRPAUSE	IRPAUSE	Single TCK cycle applied with TMS = 0

WAIT

The WAIT instruction causes the JTAG state machine to go to the specified stable state for the specified number of TCK clock cycles, and / or for a minimum number of microseconds. A WAIT instruction may specify either a clock cycle count or a time delay, or both. When both are specified, they are processed in the order they appear (i.e., left to right). When a USEC time delay is specified, the delay implemented is not related to the clock rate of TCK. TCK may continue to run during the USEC delay, or it can be turned off.

If either the wait-state or the end-state is not specified, IDLE is assumed. If an ENDSTATE is specified, the JTAG state machine will go to that state immediately after the specified number of clock cycles or the specified amount of real time has elapsed. The valid wait-state and end-states are: IRPAUSE, DRPAUSE, RESET, and IDLE.

Syntax: WAIT [<*wait-state*>,] [<*integer-expr*> CYCLES,] [<*integer-expr*> USEC,][<*end-state*>];

Example:
```
WAIT 3 CYCLES, 10000 USEC;' clock 3 times,
then wait 10ms

WAIT DRPAUSE, 10 CYCLES;' go to DRPAUSE, wait
10 cycles

WAIT 10 CYCLES, DRSELECT;' 10 cycles in IDLE
then DRSELECT
```


Conventions

Conventions in the Jam language are preferred ways of specifying tasks to be performed on the targeted device.

NOTE String Conventions

Each NOTE statement has a key string and a value string. Table 15 defines each key string; others may be defined in the future.

Table 15. NOTE String Conventions

Key String	Value String
FAMILY	Name of the device family supported by the Jam program
DEVICE	Name of the device supported by the Jam program
DATE	Date when the Jam program was created
DESIGN	Design name and revision used to create the Jam program
CREATOR	Name and copyright notice of the software which created the Jam program
REF_DESIGNATOR	Reference designator of the chip on the PCB (example: "U1")
CHECKSUM	"Fuse checksum" of the pattern (if applicable)
UESCODE	JTAG User Electronic Signature of the chip
JAM_VERSION	Version of the Jam language specification used
TITLE	Text used to identify the Jam program
ALG_VERSION	Component algorithm used

Each key string is used to provide additional information about the targeted device. Each key string is optional, and can be used only once within each source file.

Initialization List Conventions

The initialization list can be used to force the initialization value of any integer or Boolean variable in a Jam program. In practice, it is useful to force the value of certain specific variables to specific values which have well-defined influence over the execution of the Jam program. The variable names and values shown in Table 16 are defined to have specific effects for Jam programs for programmable devices. Usage of these reserved variable names is optional.

Table 16. Initialization List Conventions

Variable Name	Value	Description
DO_ERASE	0	Do not perform a bulk-erase
	1 *(1)*	Perform a bulk-erase
DO_BLANKCHECK	0	Do not check the erased state of the device
	1 *(1)*	Check the erased state of the device
DO_PROGRAM	0	Do not program the device
	1 *(1)*	Program the device
DO_VERIFY	0	Do not verify the device
	1 *(1)*	Verify the device
READ_UESCODE	0 *(1)*	Do not read the JTAG UESCODE
	1	Read UESCODE and EXPORT it
DO_SECURE	0 *(1)*	Do not set the security bit
	1	Set the security bit

Notes:
(1) This value is the default value.

Export Conventions

The EXPORT statement transmits a key string and an integer value outside the Jam interpreter to the calling program. The interpretation of the integer value depends on the key string. Table 17 lists the export key strings that are defined.

Table 17. Export Key Strings

Key String	Value
PERCENT_DONE	Percent of program executed so far (range 0-100)
UESCODE	Integer value of JTAG UESCODE
IDCODE	JTAG manufacturer's 32-bit ID

PERCENT_DONE is an optional key. If used, it will allow the calling program to show a "progress" display which indicates the activity of the Jam program while it is running. To support this feature, the Jam program should EXPORT the PERCENT_DONE value at periodic intervals during processing. Some Jam programs may not support this feature; the calling program may ignore this information entirely.

The Jam Player

Jam programs are executed by the Jam interpreter, called the Jam Player, which is an executable program running on a computer system. The mechanism by which the Jam Player reads the contents of the Jam program is platform-dependent—it may use a file system, or it may simply read characters from a memory buffer. The Jam Player has access to the JTAG signals that are used for all scan operations. This hardware I/O interface is also platform dependent. If the Jam Player is running inside a system that has a console or teletype output device, that device can be used to display messages generated by the Jam program.

Capabilities of the Jam Player

The Jam interpreter must have the following capabilities:

- Execute a Jam program, processing the initialization list if one is present
- Check the CRC of a Jam File (without executing it)
- Extract information from the NOTE fields of a Jam File (without executing the Jam File)
- Access to the signals of an IEEE 1149.1 JTAG interface
- Reliable mechanism for creating accurate real-time delays
- Report error status information following the execution of a Jam File (e.g., a return code)

Appendix A
Examples

The following examples illustrate the flexibility and utility of the Jam Programming and Test Language. All of the examples read the IDCODE out of a single device or out of a multi-device JTAG chain. Each example provides increasingly complex code to illustrate the sub-routine and intelligent capabilities of the Jam language.

Example 1. Reading IDCODE from a Single MAX 7000S or MAX 9000 Device

```
Initialize instruction and data arrays
BOOLEAN read_data[32];
BOOLEAN I_IDCODE[10] = BIN 1001101000;
BOOLEAN ONES_OATA[32} = HEX FFFFFFFF;

INTEGER i;

‘Set up stop state for IRSCAN
IRSTOP IRPAUSE;

‘Initialize device
STATE RESET;

IRSCAN 10, I_IDCODE[0..9]; ‘LOAD IDCODE INSTRUCTION
STATE IDLE;
WAIT 5 USEC, 3 CYCLES;
DRSCAN 32, ONES_DATA[0..31], CAPTURE read_data[0..31];
‘CAPTURE IDCODE

PRINT “IDCODE:”;
FOR i=0 to 31
    PRINT read_data[i];
NEXT i;

EXIT 0;
```


Note that the array variable, I_IDCODE, is initialized with the IDCODE instruction bits ordered LSB first (on the left) to MSB (on the right). This is done since the array field in the IRSCAN instruction is always interpreted, and sent, most significant bit to least significant bit.

Example 2. IDCODE Read from Multiple MAX 7000S & MAX 9000 Devices (Part 1 of 2)

```
‘Initialize instruction and data arrays
BOOLEAN IDCODE_data[32*10];   ‘[IDCODE_LENGTH * MAX_NUM_DEVICES]
BOOLEAN I_IDCODE[10] = BIN 1001101000;
BOOLEAN ONES_DATA[10*32] = HEX
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;
BOOLEAN tmp_ir[10*10];   ‘[IR_LENGTH*MAX_NUM_DEVICES}
BOOLEAN read_data[10+1];   ‘MAX_NUM_DEVICES + 1]

INTEGER MAX_NUM_DEVICES=10;
INTEGER IR_LENGTH=10;
INTEGER IDCODE_LENGTH=32;

INTEGER i;
INTEGER j;
INTEGER number_of_chips;

‘*************************************************
‘   MAIN
‘*************************************************
‘Initialize devices
IRSTOP IRPAUSE;
DRSTOP DRPAUSE;
STATE RESET;

CALL COMPUTE_NUMBER_OF_CHIPS;

‘Assume all devices in chain are either MAX 7000S or MAX 9000
For i=0 to (number_of_chips-1);
    FOR j=0 to 9;
       LET tmp_ir[i*IR_LENGTH)+j] = I_IDCODE[j];
    NEXT i;

IRSCAN (number_of_chips*IR_LENGTH),
tmp_ir[0..((number_of_chips*IR_LENGTH)-1]’
STATE IDLE;
WAIT 5 USEC;
WAIT 3 CYCLES;

DRSCAN (number_of_chips*IDCODE_LENGTH),
ONES_DATA[0..((number_of_chips*IDCODE_LENGTH)-1)], CAPTURE
IDCODE_data[0..((number_of_chips*IDCODE_LENGTH)-1)];
PRINT “IDCODE:”;
FOR i=0 TO (number_of_chips-1);
    PRINT “IDCODE for chip #”, (number_of_chips-i);
    FOR j=0 TO (IDCODE_LENGTH-1);
       PRINT IDCODE_data[j];
```

Example 2. IDCODE Read from Multiple MAX 7000S & MAX 9000 Devices (Part 2 of 2)

```
    NEXT j;
NEXT i;

EXIT 0;
'******************************************************
'   BEGIN: COMPUTE_NUMBER_OF_CHIPS
'******************************************************
COMPUTE_NUMBER_OF_CHIPS:

IRSCAN (IR_LENGTH*MAX_NUM_DEVICES),
ONES_DATA[0..((IR_LENGTH*MAX_NUM_DEVICES)-1)];

DRSCAN(MAX_NUM_DEVICES+1), ONES_DATA[0..MAX_NUM_DEVICES], CAPTURE
read_data[0..MAX_NUM_DEVICES];
FOR i=0 to MAX_NUM_DEVICES];
    IF(read_data[i] ==0) THEN
       LET number_of_chips=number_of_chips+1;
NEXT i;

RETURN;

'******************************************************
'   END: COMPUTE_NUMBER_OF_CHIPS
'******************************************************
```

Appendix B
Calculating the CRC for a Jam File

The CRC for a Jam File is a 16-bit Cyclic Redundancy Code (CRC) computed on all bytes in the Jam File up to (but not including) the CRC statement, and excluding all carriage return characters. The method for computing the CRC is explained below. The CRC statement should always be the last statement in a Jam File—any characters located after the CRC statement will not be included in the CRC computation.

The CRC is a 16-bit convolution code based on a generator polynomial. CRCs for Jam Files are calculated using the generator polynomial used by the CCITT for 16-bit CRCs:

$G(X) = X^{16} + X^{12} + X^5 + 1$

Figure 6 shows the C code for implementing this algorithm.

Figure 6. Generator Polynomial Algorithm

```
#define CCITT_CRC 0x8408/* bit-mask for CCITT CTC polynomial */
unsigned short crc_register; /*global 16-bit shift register */

void init_crc()
{
    crc_register = 0xFFFF; /*start with all ones in shift register */
}

void compute_crc(unsigned char in_byte)
{
    int bit, feedback;

    /* compute for each bit in in_byte */
    for (bit = 0; bit < CHAR_BIT; bit++)
    {
        feedback = (in byte ^ crc_register) & 0x01; /* XOR LSB */
        crc_register >>= 1; /* shift the shift register */
        if (feedback)
           crc_register ^= CCITT_CRC; /*invert selected bits */
        in_byte >>=1; /*get the next bit of in_byte */
    }
}

unsigned short crc_value()
{
    return(~crc_register); /* CRC is complement of shift register */
}
```

The function `init_crc()` must be called first to initialize the CRC shift register. Then, `compute_crc()` must first be called on each byte in the Jam File, except carriage return characters (ASCII 13 or 0D Hex). The characters must be processed in order, from the beginning of the file up to the CRC statement itself (or to the end of the file, if the CRC statement is absent). Finally, `crc_value()` is called to obtain the final CRC value, which is the complement of the current value of the CRC shift register after all characters have been processed.

Carriage return characters are excluded from the CRC calculation to allow a Jam File to have the same CRC when stored in the MS-DOS text file format (with CR-LF characters as line separators) or in the UNIX format (with LF character only). Since a Jam File is a text file, it may be stored in either format, and the CRC will be the same.

If the CRC statement is found, the calculated CRC value should be compared to the expected CRC value represented in the CRC statement. If these values differ, the CRC check fails—the Jam File contents may be corrupted.

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com
Applications Hotline:
(800) 800-EPLD
Customer Marketing:
(408) 894-7104
Literature Services:
(888) 3-ALTERA
lit_req@altera.com