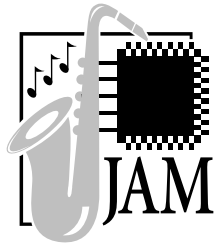
Jam Programming & Test Language Specification



Version 1.1
September 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, and testing of electronic systems, using the IEEE 1149.1 JTAG interface. As a Jam program is executed, signals are produced on the IEEE 1149.1 JTAG interface, as described in the Jam program. The Jam language supports programming any IEEE 1149.1 JTAG-compliant programmable logic or memory device.

The Jam language may be implemented as an interpreted language, meaning that the Jam program source code is executed directly by an interpreter program, without first being compiled into binary executable code.

The Jam language also provides an extended instruction set that allows the Jam program to drive any parallel vectors to the system. Jam compliance does not require support of this extension.

Language Overview

A Jam program consists of a sequence of program statements. A Jam statement consists of a label, which is optional, an instruction, and arguments, and terminates 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, but this is not required. Line breaks are not significant to the Jam language syntax, except for terminating comments. An apostrophe character (‘) can be used to signify a comment, which is ignored by the interpreter. The language does not specify any limits for line length, statement length, or program size.

Program Flow

Execution of a Jam program starts at the beginning of the program. The program flow is controlled using GOTO, CALL/RETURN, and FOR/NEXT structures. The GOTO and CALL statements 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.

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.

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”, which is 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 program 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 program 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.

Go to Appendix A on page 35 for a sample Jam File.

I/O

The only input and output mechanisms supported in the Jam language are the IEEE 1149.1 JTAG hardware interface, the initialization list for run-time variable initialization, the PRINT statement for output messages, and the EXPORT statement for sending data values to the calling program. The EXPORT statement transmits information from the Jam program to the calling program using a callback function. The EXPORT statement can be used to relay the current execution status, or to pass other information. The information transmitted by the EXPORT statement consists of a key string and an integer value. The significance of the integer value depends on the key string. See Table 10 on page 29 for a list of defined key strings. The Jam language does not provide access to any other input or output files or devices.

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 Jam statement begins with one of the following instruction names. “Jam Statement Specifications” on page 16 provides a detailed description of each instruction name. The instruction names, including the names of the optional instructions, are reserved keywords and cannot be used as variable or label identifiers in a Jam program.

- BOOLEAN
- CALL
- CRC
- DRSCAN
- DRSTOP
- EXIT
- EXPORT
- FOR
- GOTO
- IF
- INTEGER
- IRSCAN
- IRSTOP
- LET
- NEXT
- NOTE
- POP
- POSTDR
- POSTIR
- PREDR
- PREIR
- PRINT
- PUSH
- RETURN
- STATE
- WAIT
- VECTOR *(1)*
- VMAP *(1)*

Note:
(1) This instruction name is an optional language extension.

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 statement takes a string expression as an argument.
- The DRSCAN, IRSCAN, and VECTOR statements take Boolean array expressions as arguments.
- The RETURN statement takes no arguments at all.

When a statement is processed, each argument is checked 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 IEEE 1149.1 JTAG specification.

Table 1. Reserved State Names

IEEE 1149.1 JTAG 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

A comment is a part of a Jam program that is ignored during processing. Comments can be placed anywhere in the program. A comment is made using the apostrophe character (‘). The apostrophe, and all characters following it on the same line are ignored. A line break indicates the end of a comment.

Program Flow

Execution of a Jam program always begins with the first line and terminates when the EXIT statement is processed. If the end of a file is reached, 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 language 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 statement is encountered, information about the function is added to the stack. When the corresponding RETURN, NEXT, or POP statement is encountered, the record is removed from the stack. (For the NEXT statement, the stack record is removed only when the loop has run to completion.)

GOTO

The GOTO statement causes execution to jump to the statement that corresponds 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 statements) 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 statement can be used with the GOTO statement to create a conditional branch.

CALL & RETURN

The CALL statement is like the GOTO statement, but the location of the statement following the CALL is saved on the stack. When a RETURN statement is executed, execution jumps to the statement following the CALL statement, and the record is deleted from the stack. If a RETURN statement 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 statement can be used with the CALL and RETURN statements to call a subroutine conditionally, or to return conditionally.

FOR Loops

The FOR statement is used for iteration or "looping". Each FOR statement has an associated integer variable called the "iterator", which maintains a count of the iterations. When a NEXT statement using the same iterator variable is encountered, the iterator is incremented (or stepped, if the STEP keyword is used with the FOR statement). 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 statement following the NEXT statement. Otherwise, control jumps back to the statement following the FOR statement.

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).

Declaration of a variable whose name exceeds 32 characters in length, contains illegal characters, or conflicts with a previously defined identifier (a variable or label name) or reserved keyword is an error.

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.

All arrays are zero-based (i.e., valid indices range from zero to one less than the total number of elements in the array).

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)
- Advanced Compression Algorithm (ACA)

To initialize integer arrays, the initial array data must be specified as a comma-separated sequence of decimal numbers.

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

Arrays and sub-range indexed arrays can only be used as arguments with `LET`, `DRSCAN`, `IRSCAN`, and `VECTOR` statements, which accept array arguments. No arithmetic, logical, or relational 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 integers.

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 `LET`, `DRSCAN`, `IRSCAN`, and `VECTOR` statements, which accept Boolean arrays as arguments. If the size of the literal array is less than the expected size, an error occurs. literal Boolean arrays must begin with a numeric character to avoid confusion with variable names. For example, “FF” must be expressed as “0FF”. The array elements are ordered from right to left, i.e., the least significant bit (LSB) of the right-most hexadecimal digit corresponds to index zero of the array.

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

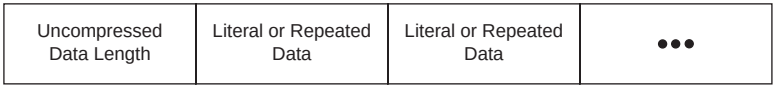
No facility is provided for integer or Boolean constants. A variable should be declared with an initialized value when a symbolic name for a quantity is desired.

Advanced Compression Algorithm (ACA)

The ACA format uses text characters to store Boolean array data in a compressed form. The algorithm and syntax for recovering raw binary data from 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 is 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 $1 \leq \text{Offset} \leq (2^{13}) - 1$	Up to 8 bits $4 \leq \text{Length} \leq 255$

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.

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 below.

```
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 program, 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)).

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 expected 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 4 and 5 take one or two integer arguments and produce an integer result.

Table 4. Operators Yielding an Integer Result

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

Table 5. Functions 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 6 take two integer arguments and produce a Boolean result.

Table 6. 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 7 take two Boolean arguments and produce a Boolean result (except the unary inversion operator, which takes one Boolean argument).

Table 7. 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 8 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 8. 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 LET, DRSCAN, IRSCAN, and VECTOR commands. For example:

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

If no indexing expression is given inside the brackets, this is equivalent to a subrange index spanning the entire array. Thus, the expression `vect[]` is equivalent to `vect[0..n-1]` where *n* is the total number of elements in array `vect[]`.

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, if message text is acquired from a device during test or programming, it can be stored and manipulated as integer data, and displayed as text characters.

Jam Statement Specifications

The Jam language supports 28 types of statements corresponding to the 28 instruction names listed in "Language Overview" on page 1. The following section describes each statement type.

BOOLEAN

The BOOLEAN statement declares a variable or an array of Boolean type. Boolean variables can be initialized to 0 or 1. Arrays can be initialized using binary, hexadecimal or ACA compressed format. By default, a comma-separated list of data values is expected. The keywords BIN, HEX, and ACA may be used to select other formats for initialization data. Initialized arrays are read-only (i.e., the Jam program cannot modify any element of the array). If the size (number of elements) of the initialization data is less than that of the initialized array, the array elements whose initialization data is unspecified will be set to zero. If the size of the initialization data is greater than that of the initialized array, the excess is ignored. If no initialization is specified, the variable or array will be initialized to zero.

Initialization data specified using the BIN and HEX format is always ordered from left to right (i.e., the left-most binary digit or the LSB of the left-most hexadecimal digit corresponds to index zero of the array).

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 flags[3]= 0,1,0;

BOOLEAN address[32] = BIN
01011010010110100101101001011010;

BOOLEAN data[32] = HEX 34B4CDB7;
```

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

CALL

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

Syntax: CALL <*label*>;

CRC

The CRC statement is used to verify the data integrity of the Jam program; it is not an executable statement. The CRC statement should be located at the end of the Jam file, after all executable Jam statements—including the EXIT statement. To check the integrity of the Jam program, the CRC (cyclic redundancy code) of all characters in the file, including comments and white-space characters but excluding carriage-return (CR) characters, must be calculated, 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.

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

See Appendix B for details on how the CRC is computed.

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

Example: `CRC 9C4A;`

DRSCAN

The DRSCAN statement 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 both, or it may be ignored. The data register length is an integer expression. 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 (these may be initialized Boolean arrays or literal Boolean array values) 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.

Syntax: DRSCAN <*length*>, <*scan data array*> [,CAPTURE <*capture array*>] [,COMPARE <*compare array*>,<*mask array*>, <*result*>];

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

DRSCAN 20, datain[0..19], CAPTURE
dataout[0..19];

DRSCAN 41, indata[0..40], COMPARE
expecteddata[0..40], maskdata[0..40],
verify_error;
```

DRSTOP

The DRSTOP statement specifies the IEEE 1149.1 JTAG end state for data register scan operations. This end state must be one of the IEEE 1149.1 JTAG stable states: RESET, IDLE, IRPAUSE, or DRPAUSE. 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 statement is encountered.

Syntax: DRSTOP <*state name*>;

EXIT

The EXIT statement terminates the Jam program with the specified error code. By convention, an error code of zero indicates success, and non-zero values indicate error conditions. A set of standard EXIT codes is defined in “Conventions” on page 29.

Syntax: EXIT <*integer expression*>;

EXPORT

The EXPORT statement 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 statement initiates a loop. Each FOR statement has an associated integer variable called the "iterator", which maintains a count of the iterations. The NEXT statement continues or terminates the loop. When the NEXT statement is encountered, the value of the iterator variable is compared to the terminal value. If the loop has not yet run to completion, the iterator is "stepped" by adding the specified step value. (If no value is specified, the default step value is 1). Then, control jumps to the statement after the FOR statement. If the loop has run to completion, control jumps to the statement following the NEXT statement.

FOR loops can be nested. When a FOR statement is encountered, a FOR record is pushed onto the stack. This record stores the name of the iterator variable and the location of the FOR statement. When the corresponding NEXT statement is encountered, the iterator variable is incremented (or stepped), 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 statement following the NEXT statement. If the FOR loop has not reached its terminal value, control continues at the statement following the FOR statement. If a NEXT statement 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 statement of the outer loop is encountered. When nesting a FOR loop inside a subroutine, the FOR loop must run to completion before the RETURN statement is encountered.

Since the terminating condition is not evaluated unit the NEXT statement is processed, the body of the loop will always be executed at least once, even if the initial value of the iterator is equal to the terminal value.

Syntax: 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 statement causes execution to jump to the statement corresponding to the label. If the label is not already known, the reminder of the Jam program will be processed (without executing any statements) 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.

Syntax: GOTO <*label*>;

IF

The IF statement evaluates a Boolean expression, and if the expression is true, executes a statement. The THEN statement can be any executable statement type (i.e., not NOTE or CRC).

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 statement 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. If the size (number of elements) of the initialization data is less than that of the initialized array, the array elements whose initialization data is unspecified will be set to zero. If the size of the initialization data is greater than that of the initialized array, the excess is ignored. 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 statement specifies an IEEE 1149.1 JTAG instruction register scan pattern to be applied to the instruction register. Data shifted out of the instruction register may be captured in a Boolean array variable, compared to a Boolean array variable or constant, or both, or it may be ignored. The capture array is a writable Boolean array variable (i.e., not an initialized array). The compare array and mask array are Boolean arrays (these may be initialized Boolean arrays or literal Boolean array values), and the result is a Boolean variable that 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 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 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.

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

IRSTOP

The IRSTOP statement specifies the IEEE 1149.1 JTAG end state for instruction register scan operations. The end state must be one of the JTAG stable states: RESET, IDLE, IRPAUSE, or DRPAUSE. 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 statement is encountered.

Syntax: IRSTOP <*state name*>;

LET

The LET statement assigns the value of an expression to a variable. It may be used to assign integer or Boolean values, and it may be used with scalar (single) quantities or arrays. When assigning arrays or array subranges, the variable receiving the assignment must be a writable (i.e. not initialized) array or a subrange of such an array. The array expression being assigned to the array may be writable, or read-only, or it may be a literal Boolean array value.

Syntax: LET <*integer variable*> = <*integer-expr*>;
LET <*Boolean variable*> = < *Boolean-expr*>;
LET <*element of integer array*> = <*integer-expr*>;
LET <*element of Boolean array*> = < *Boolean-expr*>;
LET <*integer array or array subrange*> = <*integer-array-expr*>;
LET <*Boolean array or array subrange*> = <*Boolean-array-expr*>;

Examples:
```
LET i = i + 1; ' i is an integer variable
LET b = !c; ' b and c are Boolean variables
LET ia[2] = 3; ' ia[] is an integer array
LET ba[2] = 0; ' ba[] is a Boolean array
LET ia[0..7] = ia[8..15]; ' copy array subrange
LET ba[0..7] = 0FF; ' use literal Boolean array
```

NEXT

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

Syntax: NEXT <*variable name*>;

NOTE

The NOTE statement 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. Note statements are ignored during program execution.

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 must be enclosed in quotation marks. The note text string must also be enclosed in quotation marks. (The 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" "1997/05/19";`

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)
```

POSTDR

The POSTDR statement modifies the behavior of subsequent DRSCAN statements. It specifies a number of extra bits to shift after all subsequent IEEE 1149.1 JTAG data register scan operations, and optionally specifies the scan data pattern to be used for the extra bits. If no scan data pattern is provided, the default is all ones. Since the POSTDR scan data is shifted after the DRSCAN scan data, the POSTDR scan data does not pass through the data register of the target device (or devices) during the scan operation.

Syntax: `POSTDR` <*integer-expr*> `[,` <*Boolean-array-expr*> `];`

POSTIR

The POSTIR statement modifies the behavior of subsequent IRSCAN statements. It specifies a number of extra bits to shift after all subsequent IEEE 1149.1 JTAG instruction register scan operations, and optionally specifies the scan data pattern to be used for the extra bits. If no scan data pattern is provided, the default is all ones. Since the POSTIR scan data is shifted after the IRSCAN scan data, the POSTIR scan data does not pass through the instruction register of the target device (or devices) during the scan operation.

Syntax: POSTIR <*integer-expr*> [, <*Boolean-array-expr*>];

PREDR

The PREDR statement modifies the behavior of subsequent DRSCAN statements. It specifies a number of extra bits to shift before all subsequent IEEE 1149.1 JTAG data register scan operations, and optionally specifies the scan data pattern to be used for the extra bits. If no scan data pattern is provided, the default is all ones. Since the PREDR scan data is shifted before the DRSCAN scan data, the PREDR scan data always passes through the data register of the target device (or devices) during the scan operation.

Syntax: PREDR <*integer-expr*> [, <*Boolean-array-expr*>];

PREIR

The PREIR statement modifies the behavior of subsequent IRSCAN statements. It specifies a number of extra bits to shift before all subsequent IEEE 1149.1 JTAG instruction register scan operations, and optionally specifies the scan data pattern to be used for the extra bits. If no scan data pattern is provided, the default is all ones. Since the PREIR scan data is shifted before the IRSCAN scan data, the PREIR scan data always passes through the instruction register of the target device (or devices) during the scan operation.

Syntax: PREIR <integer-expr> [, <Boolean-array-expr>];

PRINT

The PRINT statement 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 or Boolean 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

The PUSH statement adds a PUSH record to the stack storing an integer or Boolean 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
```

RETURN

The RETURN statement causes execution to jump 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”;

EXIT exit_code;

Print_Message: PRINT “Scan complete”;

RETURN;
```

STATE

The STATE statement causes the IEEE 1149.1 JTAG state machine to go to the specified state. The path to the end state may be delineated explicitly, by specifying one or more intermediate states between the current state and the end state. Otherwise, the states traversed will default to the paths outlined in Table 9. The final state must be a stable state (i.e., one of RESET, IDLE, DRPAUSE, or IRPAUSE). Non-stable states may be specified as intermediate states.

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

Examples:
```
STATE IRPAUSE;

STATE IREXIT2 IRSHIFT IREXIT1 IRUPDATE IDLE;
```

Table 9. State Table

Current State	Final State	State Path
RESET	RESET	At least one 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	At least one 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	At least one 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	At least one TCK cycle applied with TMS = 0

WAIT

The WAIT statement causes the IEEE 1149.1 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 statement may specify either a clock cycle count or a time delay, or both. When both are specified, the clock cycles and time delay occur concurrently until both are satisfied. 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 may be stopped in the low state.

If either the wait-state or the end-state is not specified, IDLE is assumed. If an ENDSTATE is specified, the IEEE 1149.1 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, DRPAUSE;' 10 cycles in IDLE
then DRPAUSE
```

Jam Extension Specifications

The following instructions are optional extensions to the Jam language. Because these instructions are optional, the Jam Player is not required to support them. However, if these instructions are used, a NOTE field must be provided to indicate the use of the instruction using the following syntax: NOTE "VECTOR" "ON";. These instructions provide access to a hardware interface that does not comply with the IEEE 1149.1 JTAG specification

VMAP

The VMAP statement provides a standard interface that maps the signal order used when asserting or reading data with the VECTOR statement. The signal order provided by the VMAP statement is retained until the next VMAP statement is encountered. Each string following the VMAP statement is surrounded by quotation marks (“) and is seperated by a comma (,).

Syntax: VMAP <*"string0"*>, <*"string1"*>, ..., <*"stringn"*>;

Example: `VMAP "Vpp", "D0", "D1", "A0", "A1";`

VECTOR

The VECTOR statement allows data to be asserted at pins other than the interface defined by the IEEE 1149.1 JTAG specification. This is done by first setting up the order of the signals asserted in the VMAP statement. The VECTOR syntax requires that the direction of the signal be supplied, in the <*dir-vect*> field, followed by the value to be asserted, in the <*in-vect*> field. When <*dir-vect*> specifies an input signal, the value of the input signal will be stored in <*in-vect*>. (When <*dir-vect*>=0, the signal is tri-stated; when <*dir-vect*>=1, the signal is driven with the corresponding value in <*in-vect*>). The state(s) of the signal(s) are held until the next VECTOR statement.

The <*in-vect*>, <*dir-vect*>, <*mask-vect*>, and <*compare-vect*> fields can be literal or compressed arrays. The <*capture-vect*> must be a writable Boolean array. The number of bits supplied in each of these fields must be the same as the number of signals supplied in the VMAP statement. Likewise, these arrays can be sub-indexed when applied within the VECTOR statement. Like the DRSCAN statement, the VECTOR statement can be used to CAPTURE data, COMPARE data, or assert data. For the COMPARE function, the last comma-separated field, <*result*>, must be a BOOLEAN type.

Syntax: VECTOR <*dir-vect*>, <*in-vect*> [,CAPTURE <*capture-vect*>][,COMPARE <*comare-vect*>,<*mask-vect*>,<*result*>];

Since the VECTOR statement updates all signals in parallel, it will be found within a looping structure when multiple vectors must be loaded or read from the pins. The following example illustrates how a clock might be generated while loading data through a non-JTAG hardware interface:

```
BOOLEAN dir[2] = 1, 1; 'Both signals are outputs
BOOLEAN data_array[100] = HEX
14ACD135F00124A9C0341D074; 'Data to be loaded on D0
BOOLEAN in[2];
INTEGER i;

VMAP "Clock", "D0";
```

```
FOR i=1 TO 100;
    LET in[1] = data_array[i]; 'Load data into second
    ‘ bit of intermediate array, in[]
    LET in[0] = 0;'Set clock low
    VECTOR dir[0..1], in[0..1]; 'Assert clock and data
    LET in[0] = 1;'Set clock high
    VECTOR dir[0..1], in[0..1]; 'Assert clock and data
NEXT i;
EXIT 1;
```

This method allows a repetitive signal, such as a clock, to be represented in the smallest possible space while sending large amounts of data to other non-JTAG pins with relatively few lines of code.

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 10 defines each key string; others may be defined in the future.

Table 10. NOTE String Conventions

Key String	Value String
DEVICE	Name of the device supported by the Jam program
DATE	Date when the Jam program was created, in numeric format: *YYYY/MM/DD*
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	User-programmable Electronic Signature Code
USERCODE	User-programmable identification code as captured by the IEEE 1149.1 JTAG USERCODE instruction
VECTOR	“ON” if the Jam program uses the optional VMAP and VECTOR statement types
JAM_VERSION	Version of the Jam language specification used
TITLE	Text used to identify the Jam program
ALG_VERSION	Component algorithm used
SAVE_DATA	List of variable names to be preserved when the Jam program is updated

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 11 are defined to have specific effects for Jam programs for programmable devices. Usage of these reserved variable names is optional.

Table 11. Initialization List Conventions for Device Programming

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

The variable names and values shown in Table 12 are defined to have specific effects for Jam programs for test applications.

Table 12. Initialization List Conventions for Testing

Variable Name	Value	Description
DO_TEST	0	Do not perform the test
	1 (default)	Perform the test
DO_DIAGNOSTICS	0 (default)	Do not diagnose faults
	1	Diagnose faults

Export Conventions

The EXPORT statement transmits a key string and an integer value outside the Jam program to the calling program. The interpretation of the integer value depends on the key string.Table 13 lists the export key strings that are defined.

Table 13. Export Key Strings

Key String	Value
PERCENT_DONE	Percent of program executed so far (range 0-100)
UESCODE	Integer value of User-programmable Electronic Signature code
USERCODE	Integer value of user-programmable identification code captured by IEEE 1149.1 JTAG USERCODE instruction
IDCODE	Identification code captured by IEEE 1149.1 JTAG IDCODE instruction

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.

EXIT Code Conventions

Exit codes are the integer values used as arguments for the EXIT statement. These codes are used to indicate the result of execution of Jam program. An exit code value of zero indicates success, while a non-zero value indicates failure, and identifies the general type of failure that occurred.

Exit codes are not used to indicate errors in the processing of the Jam program. Jam processing errors (such as “Divide by zero” or “Illegal variable name”) are detected and reported by the system which is processing the Jam file. Exit codes indicate the status of a Jam program which has run to completion, including successful processing of the EXIT statement itself (which terminates the execution of the program). Table 14 shows the EXIT codes that are defined.

Table 14. EXIT Codes

EXIT Code	Description
0	Success
1	Illegal flags specified in initialization list
2	Unrecognized device ID
3	Device version is not supported
4	Programming failure
5	Blank-check failure
6	Verify failure
7	Test failure

Note:
(1) For example, DO_ERASE=1, DO_PROGRAM=0, DO_VERIFY=1 is illegal.

The Jam Player

The Jam language supports an interpreted mode of execution, in which the Jam program source code is executed directly by an interpreter program without first being compiled into binary executable code. This interpreter program is called the Jam Player.

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 Player has 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)

Optional Features for Device Programming

A Jam program which programs a logic or memory device must contain both the data pattern to be programmed into the device and the algorithm for programming that data into the device. In some situations it may be necessary to update the part of the Jam program that describes the programming algorithm, while leaving the data pattern undisturbed. In particular, this updating process may be required to permit programming a new version of the device. For dedicated device programming systems, the "update" process should be done automatically by the Jam Player.

For Jam programs to be updated automatically, the Jam Player must have access to a library of Jam programs containing programming algorithms for the supported devices. Each of these "reference" programs must have NOTE statements to define the following attributes: DEVICE, SAVE_DATA, and ALG_VERSION. When a Jam program is loaded into the Jam Player, if the same three NOTE statements are defined in the loaded Jam program, it is possible to update the Jam program. If the library of reference programs contains a program whose DEVICE string matches that of the loaded Jam program, and whose SAVE_DATA string matches that of the loaded Jam program, the ALG_VERSION of the loaded program is compared to the ALG_VERSION of the reference program. If the ALG_VERSION of the reference program is greater, then the loaded program should be updated.

To update the loaded Jam program, the list of data variables to be preserved must be extracted from the value string of the NOTE SAVE_DATA statement. This value string must contain a comma-separated list of variable names. For arrays, only the variable name is used, with no brackets or array index information. The Jam Player must then process the loaded Jam program to find the declaration statements corresponding to these variables. Finally, a new Jam program is created, using the reference program as a basis, and substituting the preserved variable declaration statements from the loaded Jam program for the corresponding variable declaration statements in the reference program. This new Jam program is called the "updated" Jam program.

The updated Jam program can exist only temporarily, or it can be saved for future use. If it is not saved, then the update procedure will occur each time that Jam program is used.

In the case where the library of reference programs does not have a reference program for the specified device, or has a version which precedes the version of the loaded Jam file, the device can still be programmed using the algorithm in the loaded Jam file. This is useful because it permits the Jam Player to support new programmable devices easily and quickly.

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 IDCODE Instruction

```
Initialize instruction and data arrays
BOOLEAN read_data[32];
BOOLEAN I_IDCODE[10] = BIN 1001101000; ‘assumed
BOOLEAN ONES_DATA[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 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; ‘assumed IDCODE instruction
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 j;
NEXT i;

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

Notes:

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 5 shows the C code for implementing this algorithm.

Figure 5. 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.