EIA/JEDEC STANDARD

Standard Test and Programming Language (STAPL)

JESD71

August 1999

ELECTRONIC INDUSTRIES ALLIANCE

JEDEC Solid State Technology Association

Published by
©ELECTRONIC INDUSTRIES ALLIANCE 1999
Engineering Department
2500 Wilson Boulevard
Arlington, VA 22201-3834

STANDARD TEST AND PROGRAMMING LANGUAGE (STAPL)

(From JEDEC Board Ballot JCB-99-23, formulated under the cognizance of the JC-42.1 Committee on Programmable Logic Devices (PLD).)

CONTENTS

1. INTRODUCTION

The Standard Test And Programming Language (STAPL) is designed to support the programming of programmable devices and testing of electronic systems, using the IEEE Standard 1149.1: "Standard Test Access Port and Boundary Scan Architecture" (commonly referred to as JTAG) interface. As a STAPL file is executed, signals are produced on the IEEE 1149.1 interface, as described in the STAPL file. STAPL operates on a single IEEE 1149.1 chain. STAPL supports the programming of any IEEE 1149.1-compliant programmable device.

STAPL has support for programming and test systems with user interface features. A single STAPL file may perform several different functions, such as programming, verifying, and erasing a programmable device. The STAPL file makes these different high-level functions available through `ACTION` statements that correspond to user controls in an interactive system. STAPL also supports systems that have no user interface features, such as embedded systems. STAPL files may be "filtered" to remove unneeded `ACTION` statements and their associated procedure code, reducing the size of the file. This is advantageous for systems that have strict memory requirements.

STAPL may be implemented as either an interpreted or a compiled language. An interpreted implementation means that the STAPL file statements are executed directly by an interpreter program, without first being compiled into binary executable code. A compiled implementation means that the STAPL file is first pre-processed and then executed.

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

2. LANGUAGE OVERVIEW

2.1 Description

A STAPL file consists of a sequence of program statements. A STAPL 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 STAPL file, but this is not required. Line breaks are not significant to STAPL syntax, except for terminating comments. An apostrophe character (') can be used to signify a comment, which is ignored. The language does not specify any limits for line length, statement length, or file size.

2.2 Overall Structure

A STAPL file consists of the following elements, in the order shown below:

- NOTE statements
- ACTION statements
- PROCEDURE blocks and DATA blocks
- A CRC statement

A `NOTE` statement contains text strings for documentation of the contents and features of the STAPL file.

An `ACTION` statement describes the sequence of steps required to implement a complete operation, such as programming a device. A STAPL file must contain an `ACTION` statement corresponding to each operation that can be selected by a user. Each `ACTION` statement specifies a list of `PROCEDURE` blocks that must be called, in the specified order, to complete the operation. Some of these listed `PROCEDURE` blocks may be identified as recommended or optional and may be included or excluded as desired by the end user, such as verifying a device after programming.

A `PROCEDURE` block contains STAPL statements describing computations and other processing, including interactions with IEEE 1149.1-compliant devices. Statements inside a `PROCEDURE` block may use data variables contained in other `DATA` blocks, and they may call other `PROCEDURE` blocks.

2.2 Overall Structure (cont'd)

A DATA block contains variable declaration statements. A PROCEDURE block that uses this DATA block (referenced via the USES keyword) may contain STAPL statements that use these variables.

The CRC statement contains the cyclic redundancy code of the entire STAPL file that verifies the data integrity of the file.

Variable declaration statements must appear inside a PROCEDURE block or a DATA block. NOTE, ACTION, and CRC statements must appear outside all blocks. All other statements may appear only inside a PROCEDURE block. A PROCEDURE or DATA block may not contain another PROCEDURE or DATA block.

2.3 Program Flow

A STAPL session is the execution process of a STAPL file. A STAPL session starts with the ACTION statement selected by the user and ends with the completion of that statement. The PROCEDURE blocks listed for that ACTION are called in order. Within each PROCEDURE block, statements are executed from the beginning of the block until the ENDPROC statement is encountered. The CALL statement begins execution of a different PROCEDURE block, and the GOTO statement causes execution to branch to another statement within the same PROCEDURE block. To call a PROCEDURE block, the PROCEDURE block containing the CALL statement must be linked to the called block by the USES keyword in the PROCEDURE statement.

No facility exists within a STAPL file for linking multiple STAPL files together, or for including the contents of another file into a STAPL file.

2.4 Data Management

All variables in a STAPL file must be declared in order to be used. Variables declared inside a PROCEDURE block are only available inside that block. Variables declared inside a DATA block are available in and shared by any PROCEDURE block that uses that DATA block. STAPL files 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.

STAPL does not support string variables. However, string constants and string representations of Boolean and 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.

STAPL 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 9.) In this document, STAPL file 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 Annex A on page 38 for a sample STAPL file.

2.5 Input & Output

The only input and output mechanisms supported in STAPL are the IEEE 1149.1 hardware interface, parallel vector hardware interface, user-specified action and optional procedure, the PRINT statement for debugging messages, and the EXPORT statement for sending data values to the calling program.

The EXPORT statement transmits information from the STAPL file 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 13 on page 33 for a list of defined key strings. STAPL does not provide access to any other input or output files or devices.

3. CREATING & INTERPRETING STAPL FILES

3.1 Overview

The STAPL Composer and the STAPL Player are software programs that write and interpret STAPL files. Figure 1 shows how the STAPL Composer writes STAPL files, and the STAPL Player interprets STAPL files.

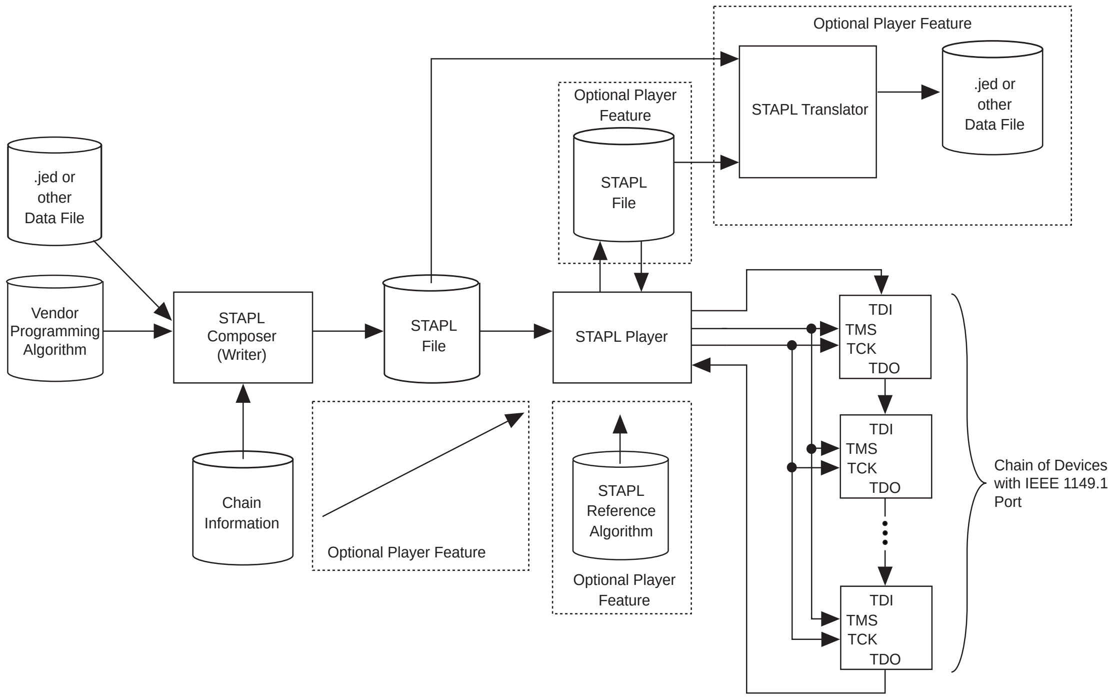

Figure 1 - Flow of STAPL Composer and Player

3.2 The STAPL Composer

The STAPL Composer generates the STAPL files in accordance with this specification. The STAPL Composer may be implemented as a stand-alone utility or as part of an integrated software tool.

The only required feature of the STAPL Composer is to create a STAPL file compliant with this language specification. The required contents of a STAPL file are:

- All mandatory `NOTE` fields
- One or more `ACTION` statements
- One or more `PROCEDURE` or `DATA` blocks
- One `CRC` statement

3.2 The STAPL composer (cont'd)

The STAPL Composer may create STAPL files with many different types of information. STAPL files may contain any or all of the following:

- Design data (single device or chain of devices)
- ISP programming algorithm information
- IEEE 1149.1 chain configuration
- Test vectors
- Any IEEE 1149.1-based instructions
- Parallel vectors via `VECTOR`/`VMAP` extended instructions

For example, a STAPL Composer for device programming may create STAPL files whose contents vary based on the platform for which the STAPL file is targeted. A STAPL file created for programming a device via an embedded processor may contain design data for the devices in the chain, programming algorithms for each device, and information on the chain configuration. A STAPL file for use on a standard socket-based programmer may contain design data and programming algorithms. A STAPL file created for programming with an IEEE 1149.1-based tester may contain design data and programming algorithms and test algorithms; chain information would then be specified in the STAPL Player on these testers.

3.3 The STAPL Player

STAPL supports implementations in either an interpreted or a compiled mode. In an interpreted implementation, the STAPL file statements are executed directly without first compiling these statements into binary executable code. The interpreter program that reads and executes a STAPL file is called the STAPL Player. In a compiled implementation, the STAPL file statements are first pre-processed and then executed.

The mechanism by which the STAPL Player reads the contents of the STAPL file is platform-dependent—it may use a file system, or it may simply read characters from a memory buffer. The STAPL Player has access to the IEEE 1149.1 signals that are used for all instructions based on the IEEE 1149.1 interface. When executing a STAPL file that requires the use of the `VECTOR` and `VMAP` instructions, the STAPL Player may also need access to several other signal pins. The hardware access to the IEEE 1149.1 and other signal pins is platform dependent. The mapping of pin names to hardware channels may be read from a wiring file or it may be built directly into the STAPL Player.

Executing STAPL statements based on the IEEE 1149.1 interface requires information on the location of the target device(s) along the serial chain. The method for conveying this chain information is platform-dependent—it may be specified via a user interface, in the STAPL file using the `POSTDR`, `POSTIR`, `PREDR`, and `PREIR` instructions, or read from a chain file. When a file is used to store this chain information, the STAPL Player must support chain files as specified in the EIA/JEDEC Standard: "Standard for Chain Description File (EIA/JESD32)".

If the STAPL 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 STAPL file.

3.3.1 Required Features of the STAPL Player

The STAPL Player shall have the following capabilities:

- Execute a STAPL file
- Process the user-specified actions and procedures (if the specified action or procedure does not exist in the file or no action is specified, an error will result)
- Check the CRC of a STAPL file (without executing the STAPL file)
- Extract information from the `NOTE` fields of a STAPL file (without executing the STAPL file)
- Access to the signals of an IEEE 1149.1 interface
- Access to the signals specified by `VECTOR`/`VMAP` instructions as supported by the platform
- Reliable mechanism for creating accurate real-time delays
- Report exit status information following the execution of a STAPL file (e.g., an exit code)

3.3.2 Optional Filtering Feature

A STAPL file may support many ACTION statements. In some situations, only a subset of these supported ACTION statements may be desired in a STAPL file. For these cases, the original file may be filtered down to preserve only the desired actions. All ACTION statements, and DATA and PROCEDURE blocks not necessary for these desired ACTION statements, are removed from the STAPL file.

To filter a STAPL file to the desired subset of actions, identify the list of PROCEDURE blocks called by the desired ACTION statements, and recursively identify all other DATA and PROCEDURE blocks that are used (USES keyword) by that PROCEDURE block.

Finally, a new STAPL file is created by preserving only those PROCEDURE and DATA blocks that have been identified. All NOTE fields must be preserved and a new CRC is calculated. This new STAPL file is called the “filtered” STAPL file.

Implementing this optional filtering feature of the STAPL Player allows STAPL Composers to generate full-featured STAPL files while allowing users to minimize file size by selecting only the actions that they desire.

3.3.3 Optional Update Feature for Device Programming

A STAPL file that programs a programmable 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 provide an updated programming algorithm for the STAPL file, while leaving the data pattern undisturbed. In particular, this updating process may be required to permit the programming of a new device version. For dedicated device programming systems, the update process should be done automatically by the STAPL Player.

For STAPL files to be updated automatically, the STAPL Player must have access to a library of STAPL files 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 STAPL file is loaded into the STAPL Player, if the same three NOTE statements are defined in the loaded STAPL file, it is possible to update the STAPL file. If the library of reference programs contains a program whose DEVICE string matches that of the loaded STAPL file, and whose SAVE_DATA string matches that of the loaded STAPL file, 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, the loaded STAPL file should be updated.

To update the loaded STAPL file, the list of preserved DATA blocks must be extracted from the value string of the SAVE_DATA statement. This value string must contain a comma-separated list of DATA block names. The STAPL Player must then process the loaded STAPL file to find these DATA blocks. Finally, a new STAPL file is created, using the reference program as a basis, and substituting the preserved DATA blocks from the loaded STAPL file with the corresponding DATA blocks in the reference program and calculating the new CRC. This new STAPL file is called the “updated” STAPL file.

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

Implementing this optional feature of the STAPL Player allows two locations for the algorithm: the version of the algorithm contained in the STAPL file and the version of the algorithm from a library of reference programs. This implementation of selecting the newer algorithm is useful because it permits the STAPL Player to support new programmable devices easily and quickly, using the algorithm contained in the STAPL file. Older devices for which a reference program exists will automatically use the newest algorithm available.

3.3.4 Optional Read Feature for Device Programming

In some situations, it may be necessary to read the data pattern from a device or chain of devices to program other boards with the same device(s). The common use of this feature is where a “master board” or “master device” is used to store programming data for programming many copies of this board.

Support of this read feature requires an algorithm for reading the data from the master device(s) and a corresponding algorithm that uses the read data to program copies of the master device(s). The algorithm for reading the device(s) will be designated by the ACTION READ statement, and the algorithm for programming the device(s) will be designated by the ACTION PROGRAM statement. The algorithm for reading the device will contain EXPORT statements to pass the data back to the STAPL Player. The STAPL file that contains these algorithms must have NOTE statements to define the following:

- SAVE_DATA_VARIABLES attribute to list all variables that will store programming data.
- SAVE_DATA attribute (the same attribute described in the optional update feature) to list all data blocks that contain variables listed in the SAVE_DATA_VARIABLES attribute.

All listed variables must also be defined within the STAPL file. Thus, once the SAVE_DATA and SAVE_DATA_VARIABLES notes are read, the STAPL Player will know which variable(s) will be exported from the STAPL file during the READ action.

To read the data pattern from one or more devices, the STAPL file is executed by selecting the READ action. As the STAPL file is executing, EXPORT statements are processed, sending data to the STAPL Player via a callback function. Each time the EXPORT statement is called, the STAPL Player stores the key string, which contains a variable name, and the associated data, which represents the data value for that variable. When the STAPL file execution is complete, if the exit code is zero (indicating success), the STAPL Player compares the variable names specified in the SAVE_DATA_VARIABLES note field to the key strings exported during the STAPL file execution. If all the variables listed in the SAVE_DATA_VARIABLES note field were received by the export callback function, the STAPL Player will create an updated STAPL file containing the new data.

The updated STAPL file is created using a reference program as a basis. (See the “Optional Update Feature for Device Programming” section.) The variables listed in the SAVE_DATA_VARIABLES note are located in the reference program, and a new STAPL file is created, substituting the new data values for those variables. The STAPL Player can store the algorithm with the new data in either of the following ways:

- Temporarily for immediate programming of the master device copies.
- Permanently for later programming by writing the information into a new output STAPL file.

4. STATEMENTS

4.1 Overview

Each statement in a STAPL file 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.

4.2 Labels (Optional)

Labels provide a means of branching within the program. A unique label can begin each STAPL file 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).

4.3 Instructions

Each STAPL statement (except the Assignment statement) begins with one of the following instruction names. “STAPL Statement Specifications” on page 18 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 STAPL file.

- Assignment (=)
- ACTION
- BOOLEAN
- CALL
- CRC
- DATA
- DRSCAN
- DRSTOP
- ENDDATA
- ENDPROC
- EXIT
- EXPORT
- FOR
- FREQUENCY *(1)*
- GOTO
- IF
- INTEGER
- IRSCAN
- IRSTOP
- NEXT
- NOTE
- POP
- POSTDR
- POSTIR
- PREDR
- PREIR
- PRINT
- PROCEDURE
- PUSH
- STATE
- TRST
- WAIT
- VECTOR *(2)*
- VMAP *(2)*

Notes:
(1) This instruction allows the changing of the TCK frequency on hardware that can support it.
(2) This instruction allows the application of parallel vectors if the hardware exists to support it.

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

- The `GOTO` instruction takes labels as arguments.
- The `CALL` instruction takes procedure names as arguments.
- The `PRINT` instruction takes a string expression as an argument.
- The `DRSCAN`, `IRSCAN`, and `VECTOR` instructions take Boolean array expressions as arguments.
- The `DATA` and `PROCEDURE` instructions take identifiers representing names of `DATA` and `PROCEDURE` blocks as arguments.
- The `ENDDATA` and `ENDPROC` instructions take no arguments at all.
- The `ACTION` instruction takes identifiers representing the name of the `ACTION`, a list of `PROCEDURE` blocks, and a string.

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 STAPL. These keywords correspond to the state names specified in the IEEE 1149.1 specification.

Table 1 - Reserved State Names

IEEE 1149.1 State Names	**STAPL 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 strings are also reserved keywords in STAPL, due to their significance in STAPL statements or expressions:

- `BOOL`
- `CAPTURE`
- `CHR$`
- `COMPARE`
- `CYCLES`
- `INT`
- `MAX`
- `OPTIONAL`
- `RECOMMENDED`
- `STEP`
- `THEN`
- `TO`
- `USEC`
- `USES`

4.4 Comments

A comment is a part of a STAPL file that is ignored during processing. Comments can be placed anywhere in the program and are 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.

5. PROGRAM FLOW

5.1 Overview

When a STAPL Player executes a STAPL file, exactly one ACTION will be executed. Because a STAPL file may contain multiple ACTION statements, execution of a STAPL file begins by searching the STAPL file for the ACTION statement whose name matches the one specified by the user. Execution continues with calls to the list of PROCEDURE blocks listed in the ACTION statement. Execution terminates either when the end of the ACTION statement is reached or when an EXIT statement is processed. The flow of execution within each of the called PROCEDURE blocks is controlled using three methods: branches, calls to other PROCEDURE blocks, and loops.

5.2 The Stack

STAPL manages subroutine calls and loops using a stack. The stack is a repository for information about all activities that can be nested. These nested activities are ACTIONs, CALL and ENDPROC, FOR and NEXT, and PUSH and POP. When an ACTION, CALL, FOR, or PUSH statement is encountered, information about the operation is added to the stack. When the corresponding ENDPROC, 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.) Recursion using the stack is allowed; however, the resulting stack depth must be determinate.

5.3 ACTION

When a STAPL file is executed, the user must specify the action to perform and may also specify changes to the default state for the optional or recommended PROCEDURE statements that comprise the action. The ACTION statement causes execution to sequentially jump to each PROCEDURE listed in the statement and the progress through the ACTION statement is saved on the stack. When execution of the called PROCEDURE block has reached completion by encountering an ENDPROC statement, execution jumps back to the next procedure listed in the ACTION statement and the record is deleted from the stack.

5.4 CALL

The CALL statement causes execution to jump to a PROCEDURE and the location of the CALL statement is saved on the STACK. When execution of the called PROCEDURE block has reached completion by encountering an ENDPROC statement, execution jumps to the statement following the CALL statement, and the record is deleted from the stack. If an ENDPROC statement is executed when the stack is empty or does not have a CALL record on the top, an error occurs. The program will terminate with a corresponding error code.

The IF statement can be used with the CALL statements to call a subroutine conditionally.

5.5 GOTO

The GOTO statement causes execution to jump to the statement that corresponds to the label. This label must be located within the same PROCEDURE block as the GOTO statement. This label may or may not have been encountered already in that PROCEDURE block. If the label was not encountered, the remainder of the PROCEDURE block will be processed (without executing any statements) until the label is found, or until the end of the PROCEDURE block 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.

5.6 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 compared to its terminal value. 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, the iterator is incremented (or stepped, if the STEP keyword is used with the FOR statement) and control jumps back to the statement following the FOR statement.

5.7 Recursion

A PROCEDURE block may call itself. This technique, known as recursion, must be used with care because PROCEDURE blocks are not normally re-entrant (i.e., every call to the PROCEDURE block will share the same copy of all data variables). To make a PROCEDURE block re-entrant, the PUSH and POP statements may be used to save and restore the values of data variables on the stack. The PUSH statement is used to save data variables before the call, and the POP statement is used to restore values after the return. In that case, the PROCEDURE block may call itself to implement recursion. The "Initialization" section describes the rules for initializing variables.

6. DATA MANAGEMENT

6.1 Identifier Names

STAPL uses identifiers to represent ACTION names, PROCEDURE block names, DATA block names, label names, and variable names. Identifier names are limited to 32 characters, and must begin with an alphabetic character—not a number or underscore character (_). Identifier names consist of alphabetic characters, numeric characters, and the underscore (_) character—no other characters are allowed. Identifier names are not case sensitive (i.e., two identifier names that differ only by case are considered equal).

Declaration of an identifier whose name exceeds 32 characters in length, contains illegal characters, or conflicts with a previously defined identifier or reserved keyword is an error. All identifiers must be unique. For example, a variable may not have the same name as a PROCEDURE block. No two variables may have the same name even if they occur in different blocks.

6.2 Types

The two data types available in STAPL 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).

6.3 Initialization

By default, all variables and arrays are initialized to zero when they are created. Variables and arrays can also be initialized explicitly at the time of declaration.

When a variable declaration statement inside a PROCEDURE block is executed, the variable is initialized to the specified value, or to zero if no initial value is specified. If the variable declaration statement is executed multiple times, for example, inside a loop, the variable is initialized each time the declaration statement is executed.

When a variable declaration statement inside a DATA block has an initial value, the variable is initialized to the specified value only once, at any time before the DATA block is used for the first time within the current STAPL session.

For initialization of Boolean arrays, the initial array data can be specified in one of three ways:

- 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
- Subrange indexing (using two integers) resulting in a smaller array
- Collectively as an array

Arrays and subrange indexed arrays can only be used as arguments with Assignment, `DRSCAN`, `IRSCAN`, `POSTDR`, `POSTIR`, `PREDR`, `PREIR`, and `VECTOR` statements, which accept array arguments. No arithmetic, logical, or relational operators are provided for whole arrays or subrange indexed arrays.

6.4 Literal Values

Literal data values may appear in integer or Boolean expressions. For example, in the statement 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 (-2^{31}) and 2147483647 ($2^{31} - 1$) 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 in one of three ways:

- Binary (one bit per character)
- Hexadecimal (four bits per character)
- Advanced Compression Algorithm (ACA)

Such literal arrays can be used as arguments with Assignment, `DRSCAN`, `IRSCAN`, `POSTDR`, `POSTIR`, `PREDR`, `PREIR`, and `VECTOR` statements, which accept Boolean arrays as arguments. 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. If the size of a literal array is greater than the expected size, the excess most significant bits are ignored. If the size of the literal array is less than the expected size, an error occurs. Literal Boolean arrays must begin with a format symbol to avoid confusion with variable names. These format symbols are the pound symbol (#) for a binary array, the dollar sign ($) for a hexadecimal array, and the ‘at’ symbol (@) for the ACA compression format. For example, the 9-bit value of 101101111 can be expressed as a binary value of “#101101111”, as a hexadecimal value of “$16F”, or as an ACA value of “@30000uj000”. Since literal values for large Boolean arrays may become long, white space (space, tabs, or carriage returns) may appear anywhere in the literal Boolean array value.

No format is supported for literal use of integer arrays.

Text strings must be specified as literal values for the `PRINT` statement, because STAPL does not support any character or string variable types.

6.5 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.

6.6 Advanced Compression Algorithm (ACA)

The ACA format uses text characters to store Boolean array data in a compressed form. This section describes the algorithm and syntax for recovering raw binary data from ACA compressed format.

The ACA decompression process is performed in two steps. Because ACA format uses text characters to store the compressed representation of a Boolean array, the first step is to convert the text characters into an array of binary values, called the compressed binary array. This compressed binary array is then processed using the ACA decompression algorithm to recover the original uncompressed data bytes.

The character set for ACA 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' symbol (@). These 64 text characters are used to represent numeric quantities, with each character representing six bits of compressed data. Each character in an ACA compressed array (excluding white-space characters) must be one of these text characters. White-space characters may occur anywhere in an ACA compressed array and are ignored. The decoding of binary values from text characters is shown in Table 2.

Table 2 - Character Decoding into Binary Values

Character	Decimal Value	Binary Value
'0'..'9'	0..9	000000..001001
'A'..'Z'	10..35	001010..100011
'a'..'z'	36..61	100100..111101
'_'	62	111110
'@'	63	111111

To convert an ACA compressed array from text format to binary format, each text character is processed sequentially, beginning with the initial format symbol character, which must be the 'at' character (@). Subsequent characters in the compressed array are processed as follows: all white space characters are ignored, others are converted into six-bit binary values according to the decoding rules shown in Table 2. The resulting six-bit binary values are packed into bytes, such that four six-bit values pack into three bytes of the compressed binary array. For example, the first six-bit value is stored in the lower six bits of the first byte; the second six-bit value is stored in the two remaining bits of the first byte and the lower four bits of the second byte; the third six-bit value is stored in the remaining four bits of the second byte and the lower two bits of the third byte; and the fourth six-bit value is stored in the remaining six bits of the third byte.

The overall structure of the ACA format consists of a data length block, followed by any number of data blocks. The data length block and data blocks are composed of binary values, located in bit-fields within the compressed binary array. Compression is achieved by storing data in two types of data blocks: literal data blocks and repeated data blocks. To uncompress the data, these data blocks are processed sequentially, producing the uncompressed data bytes. Figure 2 shows the overall structure of data in the ACA format.

Uncompressed Data Length (Number of Bytes)	Literal Data	Literal or Repeated Data	Literal or Repeated Data	●●●

Figure 2 - ACA Data Structure

The uncompressed data length block contains a numeric value that is 32 bits long, representing the length (in bytes) of the data when it is uncompressed. Thus, the maximum length of binary data that can be compressed by the ACA algorithm is
$2^{32} - 1$ bytes of raw data. The four bytes that make up the uncompressed data length block are ordered in the Little Endian format, meaning that the least-significant byte is stored at the lowest memory address, followed by bytes in increasing significance. For example, if the length is Hex 12345678, the first byte is Hex 78, followed by Hex 56, etc. Figure 3 shows the uncompressed data length block.

Byte Position	Byte 0	Byte 1	Byte 2	Byte 3
Bits of 32-Bit Value	7..0 (LSB)	15..8	23..16	31..24 (MSB)

Figure 3 - ACA Uncompressed Data Length Block

As the ACA data blocks are processed, uncompressed data is copied to an output buffer. The current position in the output buffer is advanced as data is copied. Before the first byte is processed, the current position in the output buffer is zero.

Each literal data block begins with a bit whose value is 0 and is followed by three bytes of uncompressed data (see Figure 4). When this block is processed, the three literal data bytes are copied directly to the output buffer, starting at the current position. The current position is advanced by three bytes.

Bit Position in Block	0	8..1	16..9	24..17
Literal data byte:		Byte 0	Byte 1	Byte 2

Figure 4 - ACA Literal Data Block

The repeated data block indicates a sequence of data bytes that is a repetition of previous data. Each repeated data block begins with a bit whose value is 1. The next 1 to 13 bits comprise the offset, which is followed by a byte specifying the length. To process the repeated data block, the offset value and length value are extracted from the block, and the corresponding repeated data is copied from the reference position in the output buffer (specified by the offset value) to the current position in the output buffer. The offset value, represented by a variable number of bits, specifies the number of bytes back in the output buffer (relative to the current position) where the repeated data begins. The length value indicates the number of bytes of data that are repeated. The size of the offset field (variable *N* in Figure 5) depends on the current position in the output buffer; it is the minimum number of bits required to represent the current position. For example, if the current position is 1025, $N = 11$. Since the first data block must always be a literal data block, the size of the offset field is always greater than zero.

As an example of the repeated data block, if the offset value is eight and the length value is five, the reference position is eight bytes less than the current position in the output buffer, and five bytes are to be copied from that position to the current position in the output buffer. Figure 5 illustrates the structure of the repeated data block.

Bit Position in Block	0	N..1	N + 8..N + 1
Function	Constant Bit	Offset	Length
Size	1 bit	Up to 13 bits	8 bits
Limits		$1 \le \text{Offset} \le (2^{13}) - 1$	$4 \le \text{Length} \le 255$

Figure 5 - ACA Repeated Data Block

The following example illustrates how uncompressed data is recovered from a compressed Boolean array. In this example, the compressed data consists of the text string “O00008Cn63PbPMRWpGBDgj6RV60”.

The first step in uncompressing the data is to convert each character into the corresponding binary value, using the conversion mapping described in Table 2. This step produces the compressed binary array. See Table 3.

Table 3 - Sample Compressed Data

Offset	ASCII Data	Binary Data
0	O	011000
1	0	000000
2	0	000000
3	0	000000
4	0	000000
5	8	001000
6	C	001100
7	n	110001
8	6	000110
9	3	000011
10	P	011001
11	b	100101
12	P	011001
13	M	010110
14	R	011011
15	W	100000
16	p	110011
17	G	010000
18	B	001011
19	D	001101
20	g	101010
21	j	101101
22	6	000110
23	R	011011
24	V	011111
25	6	000110
26	0	000000

The second step in uncompressing the data is to find and process the data blocks in the compressed binary array. Note that the offset field for the first repeated data block is three bits long, because at that stage the current position in the output buffer is six, which may be represented in three bits. The offset field for the second repeated data block is five bits long, because at that stage the current position in the output buffer is 18, which may be represented in five bits. Table 4 shows the compressed binary array data for this example, with annotations to show the uncompressed data length block and the literal and repeated data blocks.

Table 4 - Sample Compressed Binary Array

Block Type	Binary Data	Notes
Uncompressed data length	00011000 00000000 00000000 00000000	Uncompressed data length = 24 bytes
Literal data	0 01100001 01100010 01100011	Type = literal data ‘a’ ‘b’ ‘c’
Literal data	0 01100100 01100101 01100110	Type = literal data ‘d’ ‘e’ ‘f’

Repeated data	1
	110
	00000110
		Type = repeated data
		Offset = 6
		Length = 6

Literal data	0 01101010 01101011 01101001	Type = literal data ‘g’ ‘h’ ‘i’
Literal data	0 01101010 01101011 01101100	Type = literal data ‘j’ ‘k’ ‘l’

Repeated data	1
	01111
	00000110
		Type = repeated data
		Offset = 15
		Length = 6

After processing the data blocks in sequential order, the output buffer contains the uncompressed data corresponding to the compressed input data. Although the uncompressed data in this example contains text characters, it is not required. Table 5 shows the uncompressed data for this example.

Table 5 - Sample Uncompressed Data

Offset	Hex Data	ASCII Data	Binary Data
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

In this example, the uncompressed data consists of the text string “abcdefabcdefghijkldefabc”. Compression was achieved by recognizing that the sequences “abcdef” and “defabc” were repeated in the original data array. These sequences were then compressed in two repeated data blocks. During decompression, the repeated data blocks were processed, causing the desired repetition of those sequences and resulting in an exact reproduction of the original data array in the output buffer.

7. EXPRESSIONS & OPERATORS

7.1 Expressions

An expression in STAPL 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.

7.2 Integer & Boolean Operations

STAPL offers a complete set of arithmetic, logical, and relational operators. The character codes and behavior 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 Assignment statement. The ternary operator in the ‘C’ language (A = B ? C : D) is not supported in STAPL. 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 Table 6 take one or two integer arguments and produce an integer result.

Table 6 - Operators Yielding an Integer Result

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

The relational operators described in Table 7 take two integer arguments and produce a Boolean result.

Table 7 - Operators with Integer Arguments and 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 8 take two Boolean arguments and produce a Boolean result (except the unary inversion operator, which takes one Boolean argument).

Table 8 - Operators with Boolean Arguments and a Boolean Result

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

For the logical AND and logical OR, both operands are always evaluated.

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).

The functions described in Table 9 allow type conversions between Boolean arrays and integers. Note that the conversion is using two's complement Boolean values.

Table 9 - Functions for Converting Types

Function	Description
`BOOL()`	Integer to 32-bit Boolean array conversion
`INT()`	32-bit Boolean array to integer conversion

The `INT()` function converts Boolean array arguments of 32 or fewer bits. For arguments of fewer than 32 bits, zeros are used to extend the argument prior to conversion, producing a non-negative integer result.

Table 10 shows the precedence of operations, in descending order of priority. However, parentheses can be used to force the precedence in any expression.

Table 10 - 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

7.3 Array Operations

Square brackets ([ ]) are used to index arrays. The result of indexing is either a single element (integer or Boolean) or an 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:

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

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 (..). The indices of array variables may be listed in increasing order (i.e., `vect[0..9]`) or in decreasing order (i.e., `vect[9..0]`). Decreasing order is preferred. When increasing order is specified, it represents a reversal of the preferred order of the bits. For example, to copy a group of elements from one array to another while reversing the order:

```
dest[255..0] = source[0..255];
```

If no indexing expression is given inside the brackets, this is equivalent to a subrange index spanning the entire array in decreasing order. Thus, the expression `vect[]` is equivalent to `vect[`*n*`-1..0],` where *n* is the total number of elements in array `vect[]`. This method is used to provide Boolean array expressions for Assignment, `DRSCAN`, `IRSCAN`, `POSTDR`, `POSTIR`, `PREDR`, `PREIR`, and `VECTOR` statements. For example:

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

7.4 String Operations

String operations can be used only in `PRINT` statements. Integer and Boolean expressions are converted to strings automatically in the `PRINT` statement. Boolean expressions are converted to a string of “0” and “1”. 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 converts an integer value to its ASCII (ANSI X3.4-1986(R1997)) code, allowing STAPL to print ASCII characters. 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.

8. STAPL STATEMENT SPECIFICATIONS

8.1 Overview

The following section describes each STAPL statement type.

8.2 ACTION

The ACTION statement specifies the sequence of steps required to implement a complete operation, such as programming a device. A STAPL file must contain an ACTION statement corresponding to each operation that can be selected by a user. Each ACTION statement specifies a list of PROCEDURE blocks that must be called, in the specified order, to complete the operation. Each of these listed PROCEDURE blocks may be identified as optional or recommended. Optional and recommended PROCEDURE blocks may be included or excluded as desired by the end user.

The ACTION statement contains the name of the operation, and an optional text string describing the operation. To support localization, the string should not be used with reserved ACTION names. After the equal sign, the names of the PROCEDURE blocks are listed in the order in which they will be called. Each PROCEDURE block name may be followed by one of the keywords RECOMMENDED or OPTIONAL. PROCEDURE blocks with these notations may be included or excluded from the execution of the ACTION. The RECOMMENDED keyword indicates that the PROCEDURE block will be called unless explicitly declined by the user, while the OPTIONAL keyword indicates that the PROCEDURE block will not be called unless explicitly requested by the user. A PROCEDURE block listed without these keywords will always be included. One PROCEDURE block must exist for each PROCEDURE name listed in ACTION statements; however, multiple ACTION statements may call a PROCEDURE block. All ACTION statements must be located after the NOTE statements and before all PROCEDURE and DATA blocks.

Syntax:
```
ACTION <action name> ["<string>"] =
<procname> [ OPTIONAL | RECOMMENDED ]
{, <procname> [ OPTIONAL | RECOMMENDED ] };
```

Examples:
```
ACTION SECURE =
     DO_ENTER_ISP,
     DO_SECURE,
     DO_EXIT_ISP;

ACTION VERIFY "Verify only" =
     DO_ENTER_ISP,
     DO_VERIFY,
     DO_EXIT_ISP;

ACTION PROGRAM "Program all devices" =
     DO_ENTER_ISP,
     DO_ERASE,
     DO_BLANK_CHECK OPTIONAL,
     DO_PROGRAM,
     DO_VERIFY RECOMMENDED,
     DO_EXIT_ISP;

ACTION READ_IDCODE = DO_READ_IDCODE;
```

8.3 Assignment

The Assignment 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 an array or a subrange of such an array. The array expression being assigned to the array may be a Boolean array variable or literal Boolean array value. When assigning an array variable subrange to another array variable subrange, the ordering may be the same or opposite; however, the width must be the same. All Assignment statements must be located within PROCEDURE blocks.

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

Examples:

```
i = i + 1;            ' i is an integer variable
b = !c;               ' b and c are Boolean variables
ia[2] = 3;            ' ia[] is an integer array
ba[2] = 0;            ' ba[] is a Boolean array
ia[7..0] = ia[15..8];' copy array subrange
ba[0..8] = $0AB;      ' literal Boolean array-reverse order
```

8.4 BOOLEAN

The BOOLEAN statement declares a Boolean variable or array. All variables in a STAPL file must be declared before they are used. Variables declared inside a PROCEDURE block are only available inside that block. Variables declared inside a DATA block are available in any PROCEDURE block that uses (USES keyword) that DATA block. Boolean variables can be initialized to 0 or 1. Arrays can be initialized using binary, hexadecimal, or ACA compressed format. To select the format for initialization data, the pound (#), dollar ($), and 'at' (@) symbols can be used to represent binary, hexadecimal, or ACA compression, respectively. The size of a Boolean array is an integer expression. Initialization data specified using binary or hexadecimal formats is always ordered from right to left (i.e., the right-most binary digit or the LSB of the right-most hexadecimal digit corresponds to index zero of the array). The size (number of elements) of the initialization data must not be less than that of the initialized array. If the size of the initialization data is greater than that of the initialized array, the excess bits must be zero. If no initialization data is specified, the variable or array is initialized to zero. All BOOLEAN statements must be located within either PROCEDURE or DATA blocks.

Syntax:

BOOLEAN *<variable name>*;

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

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

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

Examples:

```
BOOLEAN status = 0;

BOOLEAN flags[3]= #010;

BOOLEAN verifydata[100];

BOOLEAN address[20] = #10100101101001011010;

BOOLEAN data[31] = $34B4CDBF;

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

8.5 CALL

The CALL statement causes execution to jump to the first statement in the specified PROCEDURE block, and saves a CALL record on the stack. In this case, the ENDPROC statement causes execution to return to the statement following the CALL statement. One PROCEDURE block must exist for all procedure names called by the CALL statement. To call a PROCEDURE block, the PROCEDURE block containing the CALL statement must reference the PROCEDURE block to be called in its USES list. All CALL statements must be located within PROCEDURE blocks.

Syntax: CALL <*procedure name*>

Example:
```
CALL reset_variables;
```

8.6 CRC

The CRC statement is a mandatory statement used to verify the data integrity of the STAPL file; it is not an executable statement. The CRC statement must be located at the end of the STAPL file, after all other STAPL statements. To check the integrity of the STAPL file, the cyclic redundancy code (CRC) 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 STAPL file is verified.

A CRC value of “0” indicates that CRC should not be compared.

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

Example:
```
CRC 9C4A;
```

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

8.7 DATA

The DATA statement marks the beginning of a DATA block. A DATA block may only contain variable declaration statements (INTEGER or BOOLEAN) with or without initialization values. No other statement may be encountered in a DATA block. All DATA statements must have a corresponding ENDDATA statement to indicate the end of the DATA block.

DATA blocks may be used by one or more PROCEDURE blocks with the USES keyword. When a PROCEDURE uses a DATA block, the variables declared in that DATA block are available for use in the PROCEDURE. Any variable declared in a DATA block is initialized only the first time the block is used (USES keyword); all subsequent times the DATA block is used, the initialization value is ignored. If multiple PROCEDURE blocks use a DATA block, those PROCEDURE blocks share the variables in the DATA block. The shared variables will not lose their values at the end of any PROCEDURE block. If a PROCEDURE block does not use a DATA block, the statements inside that PROCEDURE block may not use or modify variables in the DATA block.

All DATA and PROCEDURE blocks must be located after the ACTION statements and before the final CRC statement. However, within that constraint, PROCEDURE and DATA blocks may occur in any order.

Syntax: DATA <*dataname*>;

Example:
```
DATA prog_data;
   INTEGER num_of_address = 10;
   BOOLEAN data[100] = $764396545498705A7545BD8E4;
ENDDATA;
```

8.8 DRSCAN

The DRSCAN statement specifies an IEEE 1149.1 data register scan pattern to be updated into 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 expression, or both, or it may be ignored. The data register length is an integer expression specifying the number of data bits to be shifted. This length must be greater than zero. The scan data array is a Boolean array expression, specifying 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 Boolean array variable. The compare array and mask array are Boolean array expressions and the result is a Boolean variable or a single element of a Boolean array variable that receives the result of the comparison. All Boolean array expressions must have size greater than or equal to the specified data register length (excess bits are ignored). Mask array bit values of "1" represent bits to be compared, and bit values of "0" represent bits not to be compared. A successful comparison will cause a one (or TRUE) value to be stored in the result variable. An unsuccessful comparison will cause a zero (or FALSE) value to be stored in the result variable, but will not interrupt the STAPL file 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. All DRSCAN statements must be located within PROCEDURE blocks.

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

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

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

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

DRSCAN 10, #0111100011, COMPARE #1111011110,
   #1111111110, done[3];
```

8.9 DRSTOP

The DRSTOP statement specifies the IEEE 1149.1 end state for data register scan operations. This end state must be one of the IEEE 1149.1 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. All DRSTOP statements must be located within PROCEDURE blocks.

Syntax:
```
DRSTOP <state name>;
```

Example:
```
DRSTOP DRPAUSE;
```

8.10 ENDDATA

The ENDDATA statement indicates the end of a DATA block. ENDDATA statements may only appear following a previous DATA statement marking the beginning of the block. Exactly one ENDDATA statement must exist per DATA block.

Syntax:
```
ENDDATA;
```

Example:
```
DATA prog_data;
   INTEGER num_of_address = 10;
   BOOLEAN data[100] = $764396545498705A7545BD8E4;
ENDDATA;
```

8.11 ENDPROC

The ENDPROC statement indicates the end of the current PROCEDURE block and execution returns to the calling location, determined by removing the CALL record from the stack. If the record on the stack is not a CALL record, an error will occur. If the PROCEDURE block is called from the list of PROCEDURES in an ACTION statement, execution jumps to the following PROCEDURE in the list. If the PROCEDURE block is called from another PROCEDURE block using a CALL statement, execution jumps to the statement after the CALL statement. Exactly one ENDPROC statement must exist per PROCEDURE block.

Syntax: `ENDPROC;`

Example:

```
PROCEDURE print_message;
   PRINT “Scan Complete”;
ENDPROC;
```

8.12 EXIT

The EXIT statement immediately terminates the execution or processing of the STAPL file with the specified EXIT code. By default, if all the PROCEDURE blocks listed in the user-selected ACTION statement are processed without encountering an EXIT statement, the execution of the STAPL file terminates with a successful EXIT code. An EXIT code of zero indicates success, and non-zero values indicate error conditions. A set of standard EXIT codes is defined in the “Structural Requirements” section. All EXIT statements must be located within PROCEDURE blocks.

Syntax: EXIT *<integer expression>*;

Example:

```
EXIT 0;
EXIT status;
```

8.13 EXPORT

The EXPORT statement exports a key string and a data value to the calling program via a callback function. The data value may be the result of a Boolean expression, an integer expression, or a Boolean array expression. The calling program should ignore exported data if the key string is not recognized. A set of standard key strings is defined in the “Structural Requirements” section. All EXPORT statements must be located within PROCEDURE blocks.

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

Example:

```
EXPORT "PERCENT_DONE", (done * 100) / total;
EXPORT "data", $12AC3F74;
EXPORT "USERCODE", value[31..0];
```

8.14 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. Comparison to a terminal value is determined using the following methods:

- For a positive step value—Greater than or equal to
- For a negative step value—Less than or equal to

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 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, the iterator variable is incremented (or stepped), and 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.

The iterator variable may be assigned a value within the FOR loop. This capability allows the number of iterations to be defined during the execution of the loop. Additionally, the STEP value may be zero, allowing the terminating condition to be independent of the STEP value and be solely dependent on the assigning of values to the iterator variable. The start, end, and STEP values are evaluated when first entering the FOR loop. Although the value of these variables can be changed during the loop, this change will not affect the rest of the loop.

Since the terminating condition is not evaluated until 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. All FOR statements must be located within PROCEDURE blocks and the corresponding NEXT statement must be located in the same PROCEDURE block. Once initiated, a FOR loop must run to completion before the ENDPROC statement is encountered.

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

Examples:

```
FOR index = 0 TO (maximum - 1);
   accumulator = accumulator + vector[index];
NEXT index;

FOR done = 0 TO 1 STEP 0;
   WAIT 10 USEC;
   DRSCAN 100, in[99..0], COMPARE out[99..0], done;
NEXT done

FOR index = 1 TO 5 STEP 1;
   WAIT 100 USEC;
   DRSCAN 20, in[19..0], COMPARE out[19..0], done
   IF done == 1 THEN index = 5;
NEXT index;
```

8.15 GOTO

The GOTO statement causes execution to jump to the statement corresponding to the label. The label must be located within the same PROCEDURE block as the GOTO statement; however, it may occur in the PROCEDURE block either before or after the GOTO statement. The IF statement can be used with the GOTO statement to create a conditional branch. All GOTO statements must be located within PROCEDURE blocks.

Syntax: GOTO <*label*>;

Example: `GOTO set_variable`;

8.16 IF

The IF statement evaluates a Boolean expression, and if the expression is true, executes a statement. The THEN statement can be any statement type (except: ACTION, BOOLEAN, CRC, DATA, ENDDATA, ENDPROC, INTEGER, NOTE, and PROCEDURE statements). The THEN statement may include a GOTO or CALL to a label; however, it may not include a definition of a label (i.e., `IF a == b THEN message: PRINT done";` is not allowed). All IF statements must be located within PROCEDURE blocks.

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

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

IF a < b THEN CALL less;

IF a == b THEN EXIT 1;
```

8.17 INTEGER

The `INTEGER` statement declares an integer variable or array. All variables in a STAPL file must be declared before they are used. Variables declared inside a `PROCEDURE` block are only available inside that block. Variables declared inside a `DATA` block are available in any `PROCEDURE` block that uses (`USES` keyword) that `DATA` block. Integer variables may be initialized to a value between –2147483648 (-2^{31}) and 2147483647 ($2^{31} - 1$). Integer arrays can be initialized using a comma-separated list of decimal integer values. The size of an integer array is an integer expression. The size (number of elements) of the initialization data must be equal to that of the initialized array. If no initialization data is specified, the variable or array is initialized to zero. All `INTEGER` statements must be located within either `PROCEDURE` or `DATA` blocks.

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;

INTEGER addr[50];
```

8.18 IRSCAN

The `IRSCAN` statement specifies an IEEE 1149.1 instruction register scan pattern to be updated into the instruction register. Data shifted out of the instruction register may be captured in a Boolean array variable, compared to a Boolean array expression, or both, or it may be ignored. The instruction register length is an integer expression, specifying the number of data bits to be shifted. This length must be greater than zero. The instruction array is a Boolean array expression, specifying the data to be loaded into the instruction register. The data is shifted in increasing order of the array index (i.e., beginning with the least index). The capture array is a Boolean array variable. The compare array and mask array are Boolean array expressions, and the result is a Boolean variable or a single element of a Boolean array variable that receives the result of the comparison. All Boolean array expressions must have size greater than or equal to the specified instruction register length (excess bits are ignored). Mask array bit values of "1" represent bits to be compared, and bit values of "0" represent bits not to be compared. A successful comparison will cause a one (or `TRUE`) value to be stored in the result variable. An unsuccessful comparison will cause a zero (or `FALSE`) value to be stored in the result variable, but will not interrupt the STAPL file 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. All `IRSCAN` statements must be located within `PROCEDURE` blocks.

Syntax:

`IRSCAN` <*length*>, <*instruction array*> `[,CAPTURE` <*capture array*>`]`
    `[,COMPARE` <*compare array*>`,`<*mask array*>`,`<*result*>`]`

Examples:
```
IRSCAN 3, #111;

IRSCAN 5, i_idcode[4..0], COMPARE #10101, #11111,
    test_valid_tdo;

IRSCAN 40, instr[39..0], CAPTURE irval[39..0];
```

8.19 IRSTOP

The IRSTOP statement specifies the IEEE 1149.1 end state for instruction register scan operations. The end state must be one of the 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. All IRSTOP statements must be located within PROCEDURE blocks.

Syntax: IRSTOP <*state name*>;

Example:

```
IRSTOP IDLE;
```

8.20 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. All NEXT statements must be located within PROCEDURE blocks.

Syntax: NEXT <*variable name*>;

Example:

```
FOR index = 1 TO 5;
   DRSCAN 3, BOOL(index);
NEXT index;
```

8.21 NOTE

The NOTE statement is used to store information about the STAPL file that can be extracted without actually executing the STAPL file. The information stored in NOTE fields may include any type of documentation or attributes related to the particular STAPL file. 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 “Structural Requirements” section. 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.) For NOTE key strings that may require multiple note values, a comma-separated list is used within the quotes (e.g., “1, 2, 3”). When this list references devices along the IEEE 1149.1 chain, the order starts at 1 from the side closest to the TDI of the chain and counts up toward TDO. All NOTE statements must be located at the beginning of the file before any ACTION statement.

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

Examples:

```
NOTE "USERCODE" "001EDFFF";

NOTE "DATE" "1997/05/19";

NOTE  "DEVICE" "ABCD123, EFG1234, HIJ12345";
```

8.22 POP

The POP statement removes a PUSH record from the stack, storing the data value into an integer or Boolean variable or a single element of an integer or Boolean array variable. If a Boolean expression is pushed, it will be stored on the stack as an integer 0 or 1. Any value may be popped into an integer variable or a single element of an integer array. If the stack is popped into a Boolean variable or a single element of a Boolean array, the value on the stack must be 0 or 1, otherwise an error will occur. If the record that is popped off the stack is not a PUSH record, an error will occur. If a POP is executed when the stack is empty, an error will occur. A POP statement must be located within the same PROCEDURE block as its corresponding PUSH statement.

Syntax: POP <*integer variable*>;

POP *<Boolean variable>*;

Example:
```
PUSH 3 - 2;'Integer expression
POP status;
'Boolean variable gets value of 1 (TRUE)

PUSH 4;
POP data [6];
‘Integer element with index of 6 gets the value of 4
```

8.23 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 data register scan operations, and optionally specifies the scan data pattern to be used for the extra bits. If the scan data pattern is specified, it must have size greater than or equal to the specified scan data length (excess bits are ignored). If no scan data pattern is provided, the default is all ones. The scan data length must be greater than or equal to zero. If the scan data length is zero, the scan data pattern is ignored. 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. All POSTDR statements must be located within PROCEDURE blocks.

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

Example:
```
POSTDR 20;

POSTDR 10, instr_byp[10..1];
```

8.24 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 instruction register scan operations, and optionally specifies the scan data pattern to be used for the extra bits. If the scan data pattern is specified, it must have size greater than or equal to the specified scan data length (excess bits are ignored). If no scan data pattern is provided, the default is all ones. The scan data length must be greater than or equal to zero. If the scan data length is zero, the scan data pattern is ignored. 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. All POSTIR statements must be located within PROCEDURE blocks.

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

Example:
```
POSTIR 24;

POSTIR 12, #111001100110;
```

8.25 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 data register scan operations, and optionally specifies the scan data pattern to be used for the extra bits. If the scan data pattern is specified, it must have size greater than or equal to the specified scan data length (excess bits are ignored). If no scan data pattern is provided, the default is all ones. The scan data length must be greater than or equal to zero. If the scan data length is zero, the scan data pattern is ignored. 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. All PREDR statements must be located within PROCEDURE blocks.

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

Example:
```
PREDR 10;

PREDR 16, data [32..17];
```

8.26 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 instruction register scan operations, and optionally specifies the scan data pattern to be used for the extra bits. If the scan data pattern is specified, it must have size greater than or equal to the specified scan data length (excess bits are ignored). If no scan data pattern is provided, the default is all ones. If the scan data length is zero, the scan data pattern is ignored. 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. All PREIR statements must be located within PROCEDURE blocks.

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

Example:

```
PREIR 16;

PREIR 8, byp[8..1];
```

8.27 PRINT

The PRINT statement prints a message on the output device, if one is installed. If no output device exists, the PRINT statement has no effect. The PRINT statement is intended for debug purposes only. Under no circumstances should it be used to convey informational messages to the user because such use inhibits localization. It is intended that EXIT and EXPORT statements be used for that purpose. Any usage of the PRINT statement should not be considered portable from one STAPL Player to another. A string expression consists of string constants, integer or Boolean expressions, and characters generated by the character-code-conversion function (CHR$), separated by commas. All PRINT statements must be located within PROCEDURE blocks.

Syntax: PRINT *<string-expr>*;

Examples:

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

8.28 PROCEDURE

The PROCEDURE statement specifies the start of a PROCEDURE block. The end of a PROCEDURE block is marked by an ENDPROC statement. A PROCEDURE block may contain any statement type except NOTE, ACTION, DATA, ENDDATA, PROCEDURE, and CRC. A PROCEDURE block may not contain another PROCEDURE block or a DATA block. The statements inside a PROCEDURE block are executed when the PROCEDURE block is called, either by an ACTION statement or by a CALL statement. A PROCEDURE block must exist for all PROCEDURE names either called by a CALL statement or listed in an ACTION statement. When the ENDPROC statement is reached, execution returns to the ACTION or CALL statement that called the PROCEDURE.

The USES keyword identifies other PROCEDURE or DATA blocks that are associated with this PROCEDURE block. This is required when a CALL statement in this PROCEDURE block calls another PROCEDURE block, or when a statement in this PROCEDURE block uses or modifies a variable that is declared inside a DATA block.

A PROCEDURE block may call itself (see the “Recursion” section). In that case, the PROCEDURE should not be listed in its own USES list.

All DATA and PROCEDURE blocks must be located after the ACTION statements and before the final CRC statement. However, within that constraint, PROCEDURE and DATA blocks may occur in any order. For example, a given PROCEDURE or DATA block may precede or follow another PROCEDURE block that uses it.

Syntax: PROCEDURE *<procname>*
[USES *<procname>*|*<dataname>* {, *<procname>*|*<dataname>* }] ;

Examples:

```
PROCEDURE print_message;
   PRINT “Success”;
ENDPROC;
```

```
PROCEDURE DO_READ_IDCODE USES data_id;
    IRSCAN 5, #10100;
    DRSCAN 32 $00000000 CAPTURE id_value[31..0];
ENDPROC;
DATA data_id;
    BOOLEAN id_value[32];
ENDDATA;
```

8.29 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 popped into a Boolean variable, the value on the stack must be 0 or 1, otherwise an error will occur. All PUSH statements must be located within the same PROCEDURE block as the corresponding POP statement.

Syntax: PUSH <*integer-expr*>;

PUSH <*Boolean-expr*>;

Example:

```
PUSH 3 + 2;
POP a;' Integer variable a gets value of 5
```

8.30 STATE

The STATE statement causes the IEEE 1149.1 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. Alternatively, if only the end state is specified, the states traversed will default to the paths outlined in Table 11. If the Player cannot implement the specified state transition sequence, an error will occur. The final state must be one of the states: RESET, IDLE, DRPAUSE, or IRPAUSE. All STATE statements must be located within PROCEDURE blocks.

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

Examples:

```
STATE IRPAUSE;

STATE IREXIT2 IRSHIFT IREXIT1 IRUPDATE IDLE;
```

Table 11 - State Table

Current State	Final State	Default 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

8.31 TRST

The TRST statement enables the optional IEEE 1149.1 TRST pin for the specified number of TCK clock cycles and/or for a minimum number of microseconds. A TRST statement may specify a clock cycle count, a time delay, or both. When both are specified, the clock cycles and time delay occur simultaneously 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.

Since many IEEE 1149.1 compliant devices do not offer this optional TRST pin, this statement will not ensure that all devices on the chain are reset to the TEST-LOGIC-RESET state. To ensure reset of all devices to this state, use the "STATE RESET;" statement. All TRST statements must be located within PROCEDURE blocks.

Syntax: TRST [<*wait-type*>];

Where <*wait-type*> is either:

<*integer-expr*> CYCLES [,<*integer-expr*> USEC] or
<*integer-expr*> USEC

Example:
```
TRST 10 USEC;
TRST 10 CYCLES, 200 USEC;
TRST;
```

8.32 WAIT

The WAIT statement causes the IEEE 1149.1 state machine to go to the specified wait state for the specified number of TCK clock cycles, and/or for a minimum number of microseconds. A WAIT statement must specify a clock cycle count, a time delay, or both. When both are specified, the clock cycles and time delay occur simultaneously 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.

A maximum number of TCK clock cycles and/or a maximum number of microseconds may also be specified. The STAPL Player should ensure that the number of clock cycles and the number of microseconds used will not exceed the specified limits. If the STAPL Player cannot ensure that these limits will be satisfied, an error will occur.

If either the wait-state or the end-state is not specified, `IDLE` is assumed. If an `ENDSTATE` is specified, the IEEE 1149.1 state machine will go to that state immediately after the specified number of clock cycles and the specified amount of real time has elapsed. The valid wait-state and end-states are: `IRPAUSE`, `DRPAUSE`, `RESET`, and `IDLE`. All `WAIT` statements must be located within `PROCEDURE` blocks.

Syntax: `WAIT [`<*wait-state*>`,]` <*wait-type*> `[<,end-`*state*`>] [MAX <wait-type>];`

Where <*wait-type*> is either:

<*integer-expr*> `CYCLES [,`<*integer-expr*> `USEC]` or
<*integer-expr*> `USEC`

Examples:

```
WAIT 10 CYCLES, 10000 USEC;
' clock 10 times, then wait 10 microseconds

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

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

9. STAPL EXTENSION SPECIFICATIONS

9.1 Overview

The following section describes two sets of extended statement types:

- `VECTOR/VMAP`
- `FREQUENCY`

The `VECTOR/VMAP` extended instructions provide access to additional hardware signals that are not part of the IEEE 1149.1 interface. When control of these additional hardware signals is required, these instructions shall be the means to control these signals. Platforms that do not require access to these additional hardware signals do not need to implement this language extension. If these extended instructions are used, a `NOTE` field with the following syntax must be provided to indicate their use: `NOTE "VECTOR" "ON"`;.

The `VECTOR/VMAP` extended instructions may also be used to control the IEEE 1149.1 signals. This section also discusses how to avoid causing conflicts between the instructions that control the IEEE 1149.1 interface and these extended instructions.

The `FREQUENCY` extended instruction provides the ability to change the IEEE 1149.1 `TCK` clock frequency in systems that allow direct control of the `TCK` clock signal. If this `FREQUENCY` extended instruction is used, a `FREQUENCY_MIN` note field with the following syntax must be provided to indicate the extent to which the `TCK` frequency must be reduced.

9.2 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. The first (left-most) signal name corresponds to the most significant bit (MSB) of the arrays used in the `VECTOR` statement; subsequent signal names correspond to positions of decreasing index. Each string following the `VMAP` statement is surrounded by quotation marks (“) and is separated by a comma (,). All `VMAP` statements must be located within `PROCEDURE` blocks.

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

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

9.3 VECTOR

The VECTOR statement allows hardware signals to be controlled directly, without using the protocol defined by the IEEE 1149.1 specification. Like the DRSCAN statement, the VECTOR statement can be used to assert data, capture data, and compare the captured data to an expected value. The number of vector signals, their names, and their corresponding bit positions must first be specified using the VMAP statement, before the VECTOR statement is called. The *<dir-vect>* and *<in-vect>* arguments are Boolean array expressions that specify the direction and input value for each vector signal. The
<dir-vect> array contains a direction bit corresponding to each vector signal. If the direction is “1”, the signal is driven with the corresponding value in *<in-vect>*; if the direction is “0”, the signal is disabled (high impedance).

If the CAPTURE keyword is used, the state of the vector signals is captured and stored in *<capture-vect>*. The *<capture-vect>* must be a Boolean array variable, not a literal value. If the COMPARE keyword is used, the state of the vector signals is compared to the values stored in *<compare-vect>*. Mask array bit values of “1” represent bits to be compared, and bit values of “0” represent bits not to be compared. The *<mask-vect>* and *<compare-vect>* arguments are Boolean array expressions, and the result is a Boolean variable or a single element of a Boolean array that receives the result of the comparison. A successful comparison will cause a “1” (or TRUE) value to be stored in the result variable. An unsuccessful comparison will cause a “0” (or FALSE) value to be stored in the result variable, but will not interrupt the STAPL file 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. All Boolean array expressions must have size greater than or equal to the number of vector signals specified in the preceding VMAP statement (excess bits are ignored). All VECTOR statements must be located within PROCEDURE blocks.

Syntax: VECTOR *<dir-vect>*, *<in-vect>* [,CAPTURE *<capture-vect>*]
[,COMPARE *<compare-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-IEEE 1149.1 hardware interface:

```
BOOLEAN dir[2] = #11; 'Both signals are outputs
BOOLEAN data_array[100] = $14ACD135F00124A9C0341D074;
'Data to be loaded on D0
BOOLEAN in[2];
INTEGER i;

VMAP "Clock", "D0";

FOR i=0 TO 99;
   in[1] = data_array[i]; 'Load data into second
   ‘ bit of intermediate array, in[]
   in[0] = 0;'Set clock low
   VECTOR dir[1..0], in[1..0]; 'Assert clock and data
   in[0] = 1;'Set clock high
   VECTOR dir[1..0], in[1..0]; 'Assert clock and data
NEXT i;
```

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-IEEE 1149.1 pins with relatively few lines of code.

9.3.1 Use of IEEE 1149.1 Test Signals for Vector Input & Output

Under some circumstances it is necessary for a single hardware signal to be used both as an IEEE 1149.1 test signal and as a general-purpose logic signal. This situation can occur when a STAPL file contains VECTOR interface statements which specify one or more IEEE 1149.1 test signals to be used as general-purpose logic signals.

When a STAPL file is processed, the IEEE 1149.1 test signals are controlled during the processing of DRSCAN, IRSCAN, STATE, and WAIT statements. Likewise, any general-purpose logic signal may be controlled during the processing of VECTOR statements, following a VMAP statement in which that signal is selected for use. In the case where one or more signals selected for use in a VMAP statement is an IEEE 1149.1 test signal, the access to the common hardware signal must be controlled by a multiplexer. This multiplexer may be implemented using hardware circuits or software. The operation of this multiplexer is described below.

Each IEEE 1149.1 test signal is equipped with a two-to-one multiplexer controlled by a mode selector. The mode selector determines whether the signal is currently operating as an IEEE 1149.1 test signal or as a general-purpose logic signal. The two modes are called the "IEEE 1149.1" mode and the "VECTOR" mode. While the mode selector for a particular signal is in the IEEE 1149.1 mode, all logic transitions corresponding to DRSCAN, IRSCAN, STATE, and WAIT statements are applied to the signal. While the mode selector is in the VECTOR mode, all logic transitions corresponding to VECTOR statements (following a VMAP statement in which that signal is selected for use) are applied to the signal.

The mode selector for each signal is set to IEEE 1149.1 mode immediately when any DRSCAN, IRSCAN, STATE, or WAIT statement is processed. The mode selector for a particular signal is set to VECTOR mode immediately when any VECTOR statement is processed for which that signal has been selected in the previous VMAP statement. The processing of the VMAP statement does not directly affect the mode selector. When a VECTOR statement is processed, only the mode selectors for signals which are selected for use (in the previous VMAP statement) are affected.

To use a logic signal with the VECTOR statement, it must be selected by name in the preceding VMAP statement. For this purpose, the names shown in Table 12 should be used in the VMAP statement to select IEEE 1149.1 test signals for use as logic signals.

Table 12 - Reserved Names for IEEE 1149.1 Signals

IEEE 1149.1 Signal	VMAP Signal Name	Direction
TDI	"**TDI**"	Output
TMS	"**TMS**"	Output
TCK	"**TCK**"	Output
TRST	"**TRST**"	Output
TDO	"**TDO**"	Input

9.4 FREQUENCY

The FREQUENCY statement defines the maximum frequency at which the IEEE 1149.1 TCK signal should operate following execution of this statement. This specified frequency remains in effect until a subsequent FREQUENCY statement is executed or STAPL file execution terminates. Using an integer expression sets the frequency to a new value. Omitting this integer expression restores the operating frequency to the value operating prior to execution of the first FREQUENCY statement encountered during the STAPL session.

The STAPL player shall process all FREQUENCY statements. Implementation is optional according to the ability of the underlying hardware to change frequency during execution. The player can choose to run at or below this frequency for the entire operation of the STAPL file, if no dynamic frequency change is possible. If the underlying hardware cannot dynamically change the clock frequency and cannot support a frequency less than or equal to this setting, then an error occurs.

The execution evaluates to an integer representing a frequency in Hertz and must be greater than or equal to zero. Results of scan instructions when the clock frequency equals zero (i.e., the clock is stopped) are undefined.

If this statement is used, an accompanying FREQUENCY_MIN note field is mandatory using the slowest value of all FREQUENCY statements in the STAPL file.

Syntax:

```
FREQUENCY <integer expression>;
FREQUENCY;
```

Example:

```
FREQUENCY 1000000; 'represents 1 MHz
FREQUENCY;
```

10. STRUCTURAL REQUIREMENTS

10.1 Overview

Structural requirements in STAPL are required ways of specifying tasks to be performed on the targeted device(s).

10.2 Reserved Key Strings for NOTE Fields

Each `NOTE` statement has a key string and a value string. Table 13 defines several reserved Note strings. STAPL files may also define other vendor-specific Note strings.

Table 13 - Note Strings (Part 1 of 2)

Key String	Value String	Mandatory
`DEVICE`	Name of the device(s) supported by the STAPL file	Yes
`DATE`	Date when the STAPL file was created, in the format: *YYYY/MM/DD*	Yes
`DESIGN`	Design name(s) and revision(s) used to create the STAPL file	Yes (if applicable)
`CREATOR`	Name and copyright notice of the software which created the STAPL file	Yes
`REF_DESIGNATOR`	Reference designator of the chip(s) on the PCB (example: “U1”)	No
`CHECKSUM`	“Fuse checksums” of the programming pattern, in hexadecimal format (if applicable)	Yes (if applicable)
`UES`	User-programmable Electronic Signatures	Yes (if applicable)
`IDCODE`	Identification code(s) as captured by the IEEE 1149.1 IDCODE instruction, in hexadecimal format	Yes (if applicable)
`USERCODE`	User-programmable identification code(s) as captured by the IEEE 1149.1 USERCODE instruction, in hexadecimal format	Yes (if applicable)
`VECTOR`	“ON” if the STAPL file uses the optional `VMAP` and `VECTOR` statement types	Yes (if applicable)
`STAPL_VERSION`	Version of the STAPL specification used, in string format (example: “JESD00-A”)	Yes
`TITLE`	Text used to identify the STAPL file	No
`ALG_VERSION`	Component algorithm used, in integer format (example: “3”)	Yes
`SAVE_DATA`	List of `DATA` blocks to be preserved when the STAPL file is updated	Yes (if applicable)
`SAVE_DATA_VARIABLES`	List of variables to be exported during a read.	Yes (if applicable)
`STACK_DEPTH`	Number of stack records required by the STAPL Player to support the STAPL file (not bytes)	Yes
`TARGET`	List of device numbers targeted by the STAPL file	Yes

Table 13 - Note Strings (Part 2 of 2)

`MAX_FREQ`	This note defines the maximum frequency at which the IEEE 1149.1 `TCK` signal should operate. This will be the maximum `TCK` frequency of the device in the chain that has the slowest IEEE 1149.1 boundary scan circuitry. The player can operate more slowly than this value. This NOTE field is mandatory. The value is a decimal integer expressed in Hertz, and must be greater than zero.	Yes
`FREQUENCY_MIN`	This note defines the lowest setting of all `FREQUENCY` statements contained within the STAPL file. If the `FREQUENCY` statement is used in the STAPL file, this note is mandatory. If the `FREQUENCY` statement is not used in the STAPL file, then this note shall not be used. The value is a decimal integer expressed in Hertz, and must be greater than or equal to zero. Results of boundary scan instructions when the clock frequency equals zero (i.e., the clock is stopped) are undefined.	Yes (if applicable)

Each key string is used to provide additional information about the STAPL file. Each key string can be used only once within each source file.

Note fields that are mandatory "if applicable" must be used whenever the STAPL file uses the corresponding feature. For example, the `USERCODE` note should be used whenever the STAPL file uses the IEEE 1149.1 USERCODE instruction.

Comparisons involving the `STAPL_VERSION` value string are performed using string comparisons. Comparisons involving the `ALG_VERSION` value string are performed using integer comparisons.

Some note fields will contain multiple elements in the value string. These values are expressed as a single quoted string containing multiple values separated by commas. For example, the `SAVE_DATA` note field lists multiple `DATA` block names:

```
NOTE "SAVE_DATA" "datablock1, datablock2, datablock3";
```

If the elements in the value string correspond to devices in an IEEE 1149.1 chain, they must be listed in order beginning with the device closest to `TDI`. When the devices in the chain are enumerated, the device closest to `TDI` is number 1, and the device closest to `TDO` is number *n*, where *n* is the number of devices in the chain. For example, the `TARGET` note field lists the chain positions of devices that are programmed or tested by the STAPL file. For a STAPL file that programs the first and third devices in a chain, the `TARGET` note field will appear:

```
NOTE "TARGET" "1, 3";
```

Additional note key strings that are not defined in this specification are allowed.

10.3 Reserved Identifiers for ACTIONs

The ACTION statement specifies the operation (such as programming a device) that can be selected by a user. Table 14 defines several reserved ACTION names for programming and testing a chain of devices. The intended use of these reserved ACTION names is to allow a STAPL Player to identify certain expected action names for displaying in a user interface. STAPL Composers may also define other vendor-specific operations.

Table 14 - Reserved ACTION Names

ACTION Name	Description
CHECKCHAIN	Verify the continuity of the IEEE 1149.1 scan chain
READ_IDCODE	Read the IEEE 1149.1 IDCODE and EXPORT it
READ_USERCODE	Read the IEEE 1149.1 USERCODE and EXPORT it
READ_UES	Read the UESCODE and EXPORT it
ERASE	Perform a bulk erase of the device(s)
BLANK_CHECK	Check the erased state of the device(s)
PROGRAM	Program the device
VERIFY	Verify the programming data of the device(s)
READ	Read the programming data of the device(s)
CHECKSUM	Calculate one fuse checksum of the programming data of the device(s)
SECURE	Set the security bit of the device(s)
QUERY_SECURITY	Check whether the security bit is set
TEST	Perform a test. This test can include tests such as boundary scan, internal, vector, and built-in self tests.

Additional ACTION names that are not defined in this specification are allowed. Because ACTION names are identifiers, they must follow the rules for identifiers.

10.4 Reserved Identifiers for PROCEDURE Blocks

The PROCEDURE statement specifies the steps that comprise an ACTION statement (such as verifying a device as part of device programming). Table 15 defines several reserved PROCEDURE names for programming and testing a chain of devices. The intended use of these reserved PROCEDURE names is to allow STAPL Players to identify stages of an ACTION based on certain expected PROCEDURE names either for displaying in a user interface or for allowing users to select optional procedures. STAPL Composers may also define other vendor-specific procedures.

Table 15 **- Reserved PROCEDURE Names**

PROCEDURE Name	Description
DO_CHECKCHAIN	Verify the continuity of the IEEE 1149.1 scan chain
DO_READ_IDCODE	Read the IEEE 1149.1 IDCODE and EXPORT it
DO_READ_USERCODE	Read the IEEE 1149.1 USERCODE and EXPORT it
DO_READ_UES	Read the UESCODE and EXPORT it
DO_ENTER_ISP	Enter the ISP programming mode
DO_ERASE	Perform a bulk erase of the device(s)
DO_BLANK_CHECK	Check the erased state of the device(s)
DO_PROGRAM	Program the device
DO_VERIFY	Verify the programming data of the device(s)
DO_READ	Read the programming data of the device(s)
DO_CHECKSUM	Calculate one fuse checksum of the programming data of the device(s)
DO_SECURE	Set the security bit of the device(s)
DO_QUERY_SECURITY	Check whether the security bit is set
DO_EXIT_ISP	Exit the ISP programming mode
DO_TEST	Perform the system test

Additional PROCEDURE names that are not defined in this specification are allowed. Because PROCEDURE names are identifiers, they must follow the rules for identifiers.

10.5 Reserved Key Strings for EXPORT Strings

The EXPORT statement transmits a key string and an integer or Boolean array value outside the STAPL file to the calling program. The interpretation of the value depends on the key string. Table 16 lists the export key strings that are defined.

Table 16 - Export Key Strings

Key String	Value Type	Value
PERCENT_DONE	Integer	Percent of program executed so far (range 0-100)
UES	Boolean Array	Value of User-programmable Electronic Signature code
USERCODE	Boolean Array (32 Bits)	Value of user-programmable identification code captured by IEEE 1149.1 USERCODE instruction
IDCODE	Boolean Array (32 Bits)	Identification code captured by IEEE 1149.1 IDCODE instruction
CHECKSUM	Boolean Array	Design data checksum

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 STAPL file while it is running. To support this feature, the STAPL file should EXPORT the PERCENT_DONE value at periodic intervals during processing. Some STAPL files may not support this feature; the calling program may ignore this information entirely.

Additional export key strings that are not defined in this specification are allowed.

10.6 EXIT Codes

Exit codes are the integer values used as arguments for the EXIT statement. These codes are used to indicate the result of execution of a STAPL file. An exit code value of zero indicates success, while a non-zero value indicates failure and identifies the general type of failure that occurred. Positive integers are reserved for current and future standard EXIT codes. Negative integers are reserved for vendor-specific EXIT codes.

Exit codes are not used to indicate errors in the processing of the STAPL file. STAPL processing errors (such as "Divide by zero" or "Illegal variable name") are detected and reported by the system that is processing the STAPL file. Exit codes indicate the status of a STAPL file that has run to completion, including successful processing of the `EXIT` statement itself (which terminates the execution of the program). Table 17 shows the `EXIT` codes that are defined.

Table 17 - EXIT Codes

EXIT Code	Description
0	Success
1	Checking chain failure
2	Reading `IDCODE` failure
3	Reading `USERCODE` failure
4	Reading `UESCODE` failure
5	Entering ISP failure
6	Unrecognized device ID
7	Device version is not supported
8	Erase failure
9	Blank check failure
10	Programming failure
11	Verify failure
12	Read failure
13	Calculating checksum failure
14	Setting security bit failure
15	Querying security bit failure
16	Exiting ISP failure
17	Performing system test failure

ANNEX A EXAMPLES

The following examples illustrate the flexibility and utility of the Standard Test and Programming Language (STAPL). All of the examples read the IDCODE out of a single device or out of a multi-device IEEE 1149.1 chain.

Example 1
Reading IDCODE from a Single Device

```
NOTE "CREATOR" "AAAA Tool Version 1.0"
NOTE "DEVICE" "ABCD1234";
NOTE "DATE" "1997/12/31";
NOTE "STAPL_VERSION" "JEDS00-A";
NOTE "ALG_VERSION" "3";
NOTE "STACK_DEPTH" "2";
NOTE "MAX_FREQ" "10000000"; '10MHz
NOTE "TARGET" "1";
NOTE "IDCODE" "00000001";

ACTION READ_IDCODE = DO_READ_IDCODE;

PROCEDURE DO_READ_IDCODE;

‘Declare variables for data arrays
BOOLEAN read_data[32];
BOOLEAN i_idcode[10] = #1001101000;
BOOLEAN ones_data[32] = $FFFFFFFF;

INTEGER i;

‘Initialize device
STATE RESET;

‘Load idcode instruction
IRSCAN 10, i_idcode[9..0];
‘Capture idcode
DRSCAN 32, ones_data[31..0], CAPTURE read_data[31..0];

EXPORT "IDCODE", read_data[31..0];

ENDPROC;
CRC 3759;
```

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

Example 2
IDCODE Read from Multiple Devices

```
NOTE "CREATOR" "bbbb version 6.9"
NOTE "DATE" "1998/01/21";
NOTE "STAPL_VERSION" "JESD00-C";
NOTE "ALG_VERSION" "4";
NOTE "STACK_DEPTH" "3";
NOTE "MAX_FREQ" "10000000" '10MHz
NOTE "TARGET" "1";
'**********************************************************
ACTION READ_IDCODE = DO_READ_IDCODE;
'**********************************************************
DATA data_id;
BOOLEAN idcode_data[32*10]; '[idcode_length * max_num_devices]
BOOLEAN i_idcode[10] = #1001101000; 'assumed IDCODE instruction
BOOLEAN ones_data[10*32] =
       $FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
        FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;
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;
ENDDATA;
'******************************************************
PROCEDURE DO_READ_IDCODE USES compute_number_of_chips, data_id;
'Initialize devices
IRSTOP IRPAUSE;
DRSTOP DRPAUSE;
STATE RESET;

CALL compute_number_of_chips;

'Assume all devices have same idcode instruction bit pattern
FOR i=0 TO (number_of_chips-1);
  FOR j=0 TO 9;
    tmp_ir[(i* ir_length)+j] = i_idcode [j];
  NEXT j;
NEXT i;

IRSCAN (number_of_chips*ir_length),
       tmp_ir[((number_of_chips*ir_length)-1)..0];
DRSCAN (number_of_chips*idcode_length),
       ones_data[((number_of_chips* idcode_length)-1)..0], CAPTURE
       idcode_data[((number_of_chips* idcode_length)-1)..0];

FOR i=0 TO (number_of_chips-1);
  EXPORT "IDCODE", idcode_data[(i*32)+31..(i*32)];
NEXT i;
ENDPROC;
```

```
'*********************************************************
PROCEDURE compute_number_of_chips USES data_id;

IRSCAN (ir_length * max_num_devices),
       ones_data [((ir_length * max_num_devices)-1)..0];
DRSCAN (max_num_devices +1), ones_data [max_num_devices..0],
       CAPTURE read_data[max_num_devices..0];
FOR i=0 TO max_num_devices-1;
    IF(read_data[i] ==0) THEN
       number_of_chips=number_of_chips+1;
NEXT i;
ENDPROC;
'********************************************************
CRC 0374;
```

ANNEX B CALCULATING THE CRC FOR A STAPL FILE

The CRC for a STAPL file is a 16-bit Cyclic Redundancy Code (CRC) computed on all bytes in the STAPL 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 STAPL 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 STAPL files are calculated using the generator polynomial used by the CCITT for 16-bit CRCs:

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

The C code for implementing this algorithm is shown below.

Example 1
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;

  if (in_byte != '\r')
  {
    /* 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 STAPL 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 STAPL 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 STAPL 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 STAPL file contents may be corrupted.