

EIA/JEDEC STANDARD

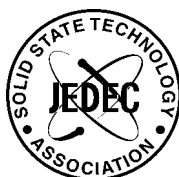
Standard Test and Programming Language (STAPL)

JESD71

August 1999

ELECTRONIC INDUSTRIES ALLIANCE

JEDEC Solid State Technology Association



NOTICE

EIA/JEDEC standards and publications contain material that has been prepared, reviewed, and approved through the JEDEC Board of Directors level and subsequently reviewed and approved by the EIA General Counsel.

EIA/JEDEC standards and publications are designed to serve the public interest through eliminating misunderstandings between manufacturers and purchasers, facilitating interchangeability and improvement of products, and assisting the purchaser in selecting and obtaining with minimum delay the proper product for use by those other than JEDEC members, whether the standard is to be used either domestically or internationally.

EIA/JEDEC standards and publications are adopted without regard to whether or not their adoption may involve patents or articles, materials, or processes. By such action JEDEC does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the EIA/JEDEC standards or publications.

The information included in EIA/JEDEC standards and publications represents a sound approach to product specification and application, principally from the solid state device manufacturer viewpoint. Within the JEDEC organization there are procedures whereby an EIA/JEDEC standard or publication may be further processed and ultimately become an ANSI/EIA standard.

No claims to be in conformance with this standard may be made unless all requirements stated in the standard are met.

Inquiries, comments, and suggestions relative to the content of this EIA/JEDEC standard or publication should be addressed to JEDEC Solid State Technology Association, 2500 Wilson Boulevard, Arlington, VA 22201-3834, (703)907-7560/7559 or www.jedec.org

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

This document may be downloaded free of charge, however EIA retains the copyright on this material. By downloading this file the individual agrees not to charge or resell the resulting material.

**PRICE: Please refer to the current
Catalog of JEDEC Engineering Standards and Publications or call Global Engineering
Documents, USA and Canada (1-800-854-7179), International (303-397-7956)**

Printed in the U.S.A.
All rights reserved

PLEASE!

DON'T VIOLATE
THE
LAW!

This document is copyrighted by the Electronic Industries Alliance and may not be reproduced without permission.

Organizations may obtain permission to reproduce a limited number of copies through entering into a license agreement. For information, contact:

JEDEC Solid State Technology Association
2500 Wilson Boulevard
Arlington, Virginia 22201-3834
or call (703) 907-7559

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.....	1
2. Language Overview	1
2.1 Description	1
2.2 Overall Structure	1
2.4 Program Flow	2
2.5 Data Management	2
2.6 Input & Output	2
3. Creating & Interpreting STAPL files.....	3
3.1 Overview.....	3
3.2 The STAPL Composer.....	3
3.3 The STAPL Player.....	4
4. Statements	6
4.1 Overview.....	6
4.2 Labels (Optional)	6
4.3 Instructions.....	6
4.4 Comments	7
5. Program Flow	8
5.1 Overview.....	8
5.2 The Stack	8
5.3 ACTION	8
5.4 CALL	8
5.5 GOTO	8
5.6 FOR Loops.....	8
5.7 Recursion	9
6. Data Management.....	9
6.1 Identifier Names.....	9
6.2 Types.....	9
6.3 Initialization	9
6.4 Literal Values.....	10
6.5 Constants.....	10
6.6 Advanced Compression Algorithm (ACA)	10
7. Expressions & Operators	15
7.1 Expressions	15
7.2 Integer & Boolean Operations	15
7.3 Array Operations.....	17
7.4 String Operations	17
8. STAPL Statement Specifications.....	18
8.1 Overview.....	18
8.2 ACTION	18
8.3 Assignment	19
8.4 BOOLEAN	19

8.5 CALL	20
8.6 CRC	20
8.7 DATA	20
8.8 DRSCAN	21
8.9 DRSTOP	21
8.10 ENDDATA	21
8.11 ENDPROC	22
8.12 EXIT	22
8.13 EXPORT	22
8.14 FOR	22
8.15 GOTO	23
8.16 IF	23
8.17 INTEGER	24
8.18 IRSCAN	24
8.19 IRSTOP	25
8.20 NEXT	25
8.21 NOTE	25
8.22 POP	25
8.23 POSTDR	26
8.24 POSTIR	26
8.25 PREDR	26
8.26 PREIR	27
8.27 PRINT	27
8.28 PROCEDURE	27
8.29 PUSH	28
8.30 STATE	28
8.31 TRST	29
8.32 WAIT	29
9. STAPL Extension Specifications	30
9.1 Overview	30
9.2 VMAP	30
9.3 VECTOR	31
9.4 FREQUENCY	32
10. Structural Requirements	33
10.1 Overview	33
10.2 Reserved Key Strings for NOTE Fields	33
10.3 Reserved Identifiers for ACTIONS	35
10.4 Reserved Identifiers for PROCEDURE Blocks	35
10.5 Reserved Key Strings for EXPORT Strings	36
10.6 EXIT Codes	36
ANNEX A EXAMPLES	38
ANNEX B CALCULATING THE CRC FOR A STAPL FILE	41

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.

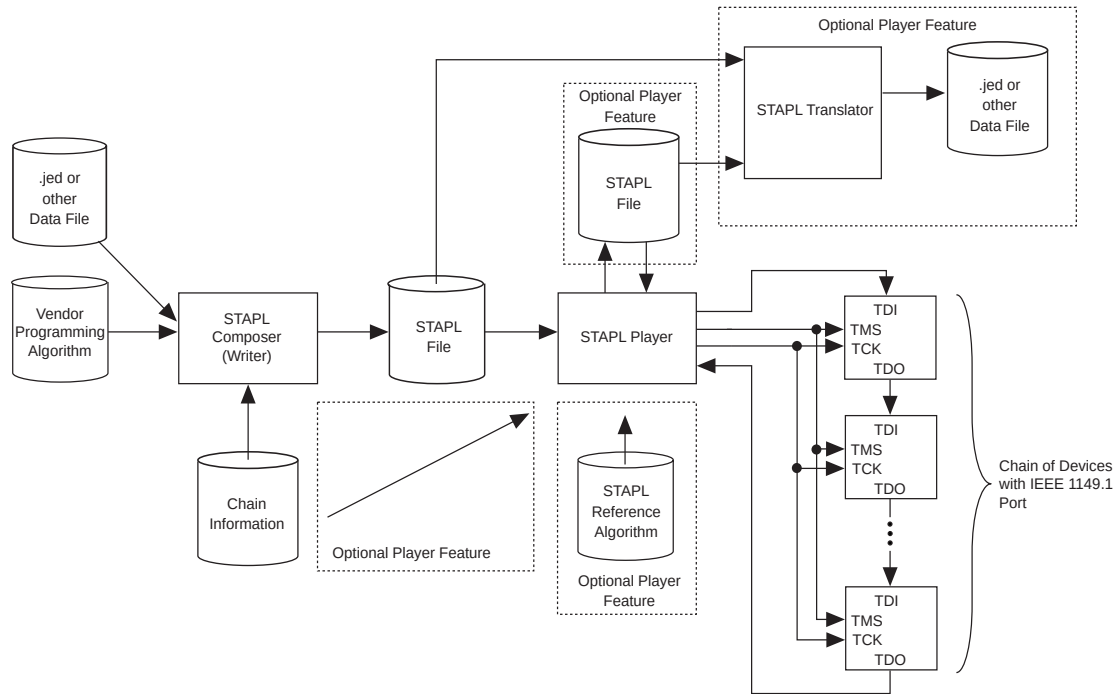


Figure 1 - Flow of STAPL Composer and Player

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

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 | • MAX | • USEC |
| • CAPTURE | • OPTIONAL | • USES |
| • CHR\$ | • RECOMMENDED | |
| • COMPARE | • STEP | |
| • CYCLES | • THEN | |
| • INT | • TO | |

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.