

# IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data

Sponsor

**Test Technology Standards Committee  
of the  
IEEE Computer Society**

Approved 18 March 1999

Reaffirmed 16 June 2011

**IEEE-SA Standards Board**

Approved 16 November 1999

Reaffirmed 25 July 2012

**IEEE-SA Standards Board**

**Abstract:** Standard Test Interface Language (STIL) provides an interface between digital test generation tools and test equipment. A test description language is defined that: (a) facilitates the transfer of digital test vector data from CAE to ATE environments; (b) specifies pattern, format, and timing information sufficient to define the application of digital test vectors to a DUT; and (c) supports the volume of test vector data generated from structured tests.

**Keywords:** automatic test pattern generator (ATPG), built-in self-test (BIST), computer-aided engineering (CAE), cyclize, device under test (DUT), digital test vectors, event, functional vectors, pattern, scan vectors, signal, structural vectors, timed event, waveform, waveshape

---

The Institute of Electrical and Electronics Engineers, Inc.  
3 Park Avenue, New York, NY 10016-5997, USA

Copyright © 1999 by the Institute of Electrical and Electronics Engineers, Inc.  
All rights reserved. Published 1 September 1999. Printed in the United States of America.

Print: ISBN 0-7381-1646-7 SH94734  
PDF: ISBN 0-7381-1647-5 SS94734

*No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.*

**Notice and Disclaimer of Liability Concerning the Use of IEEE Documents:** IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

Use of an IEEE Standard is wholly voluntary. IEEE disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon any IEEE Standard document.

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained in its standards is free from patent infringement. IEEE Standards documents are supplied “**AS IS.**”

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE standard is subjected to review at least every ten years. When a document is more than ten years old and has not undergone a revision process, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE standard.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

**Translations:** The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE should be considered the approved IEEE standard.

**Official Statements:** A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered the official position of IEEE or any of its committees and shall not be considered to be, nor be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position of IEEE.

**Comments on Standards:** Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important to ensure that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to comments or questions except in those cases where the matter has previously been addressed. Any person who would like to participate in evaluating comments or revisions to an IEEE standard is welcome to join the relevant IEEE working group at <http://standards.ieee.org/develop/wg/>.

Comments on standards should be submitted to the following address:

Secretary, IEEE-SA Standards Board  
445 Hoes Lane  
Piscataway, NJ 08854  
USA

**Photocopies:** Authorization to photocopy portions of any individual standard for internal or personal use is granted by The Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

## Introduction

(This introduction is not part of IEEE Std 1450-1999, IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data.)

Standard Test Interface Language (STIL) was initially developed by an ad-hoc consortium of test equipment vendors, computer-aided engineering (CAE) and computer-aided design (CAD) vendors, and integrated circuit (IC) manufacturers, to address the lack of a common solution for transferring digital test data from the generation environment to the test equipment.

The need for a common interchange format for large volumes of digital test data was identified as an overriding priority for the work; as such, the scope of the work was constrained to those aspects of the test environment that contribute significantly to the volume issue, or are necessary to support the comprehension of that data. Binary representations of data were a key consideration in these efforts, resulting in a proposal to incorporate the compression of files as part of this standard.

Limiting the scope of any standards project is a difficult thing to do, especially for a room full of engineers. However, issues that did not impact the scope as identified were dropped from consideration in this version of the standard. Subclause 1.1 covers, specifically, the capabilities that are not intended to be part of this first standard.

Early work in this consortium consisted of identifying the requirements necessary to address this problem and reviewing existing options and languages in the industry. All options proposed fell short of addressing the requirements, and the consortium started to define a new language. This work was executed with heavy leverage from some existing languages and environments, and STIL owes much to the groundwork established by these other languages.

## Participants

When the STIL Working Group approved this standard, it had the following membership:

**Greg Maston**, *Co-chair*

**Tony Taylor**, *Co-chair*

The technical subgroup consisted of the following members:

Dave Dowding  
Brady Harvey  
Brad Hinckle

Larry Moran  
Gary Murray  
Chris Nelson  
Don Organ

Mike Purtell  
Jim Ward  
Gregg Wilder

Other working group members were as follows:

Srinivas Ajjarapu  
Phil Barch  
Ajit Bhawe  
Fred Berneche  
Joe Carbone

Don Denburg  
Givargis A. Danialy  
Tom Munns  
Eric Parker  
Frank Peyton

Gary Raines  
Tim Wagner  
Tom Williams  
Peter Wohl  
Stefan Zschiegner

The following members of the balloting committee voted on this standard:

Phil Barch	Brion Keller	Hira Ranga
Kenneth M. Butler	Fadi Maamari	Gordon Robinson
Joe Carbone	Gregory A. Maston	John W. Sheppard
Donald Denburg	Colin Maunder	William R. Simpson
Dave Dowding	Larry L. Moran	Tony Taylor
Grady Giles	Tom Munns	Jon Udell
Gary Ginn	Wayne Needham	Jim Ward
Gary Hancock	Chris Nelson	Gregg Wilder
Brady Harvey	Jim O'Reilly	Peter Wohl
Mitsuaki Ishikawa	Frank Peyton	Stefan Zschiegner
	Mike Purtell	

The Working Group thanks those companies that contributed concepts and ideas to this effort, in addition to people and time. These contributions helped to define the language presented in this standard. In particular, the Working Group would like to thank: LTX Corporation, for providing information about the enVision environment; Motorola, Incorporated, for providing information about the Universal Test Interface Code (UTIC); Test Systems Strategies, Incorporated, for providing information about the Waveform Generation Language (WGL); and Texas Instruments, Incorporated, for providing information about the Test Description Language (TDL).

When the IEEE-SA Standards Board approved this standard on 18 March 1999, it had the following membership:

**Richard J. Holleman**, *Chair*

**Donald N. Heirman**, *Vice Chair*

**Judith Gorman**, *Secretary*

Satish K. Aggarwal	James H. Gurney	Louis-François Pau
Dennis Bodson	Lowell G. Johnson	Ronald C. Petersen
Mark D. Bowman	Robert J. Kennelly	Gerald H. Peterson
James T. Carlo	E. G. "Al" Kiener	John B. Posey
Gary R. Engmann	Joseph L. Koepfinger*	Gary S. Robinson
Harold E. Epstein	L. Bruce McClung	Akio Tojo
Jay Forster*	Daleep C. Mohla	Hans E. Weinrich
Ruben D. Garzon	Robert F. Munzner	Donald W. Zipse

\*\*Member Emeritus

Also included is the following nonvoting IEEE-SA Standards Board liaison:

Robert E. Hebner

Janet Rutigliano  
*IEEE Standards Project Editor*

# Contents

1.	Overview.....	1
1.1	Scope.....	3
1.2	Purpose.....	4
2.	References.....	4
3.	Definitions, acronyms, and abbreviations.....	4
3.1	Definitions.....	4
3.2	Acronyms and abbreviations.....	7
4.	Structure of this standard.....	7
5.	STIL orientation and capabilities tutorial (informative).....	8
5.1	Hello Tester.....	8
5.2	Basic LS245.....	13
5.3	STIL timing expressions/"Spec" information.....	17
5.4	Structural test (scan).....	22
5.5	Advanced scan.....	26
5.6	IEEE Std 1149.1-1990 scan.....	32
5.7	Multiple data elements per test cycle.....	37
5.8	Pattern reuse/direct access test.....	41
5.9	Event data/non-cyclized STIL information.....	45
6.	STIL syntax description.....	55
6.1	Case sensitivity.....	55
6.2	Whitespace.....	55
6.3	Reserved words.....	55
6.4	Reserved characters.....	57
6.5	Comments.....	58
6.6	Token length.....	58
6.7	Character strings.....	58
6.8	User-defined name characteristics.....	59
6.9	Domain names.....	59
6.10	Signal and group name characteristics.....	60
6.11	Timing name constructs.....	60
6.12	Number characteristics.....	60
6.13	Timing expressions and units (time_expr).....	61
6.14	Signal expressions (sigref_expr).....	63
6.15	WaveformChar characteristics.....	64
6.16	STIL name spaces and name resolution.....	65
7.	Statement structure and organization of STIL information.....	67
7.1	Top-level statements and required ordering.....	68
7.2	Optional top-level statements.....	70
7.3	STIL files.....	70

8.	STIL statement.....	70
8.1	STIL syntax.....	70
8.2	STIL example.....	70
9.	Header block.....	71
9.1	Header block syntax.....	71
9.2	Header example.....	71
10.	Include statement.....	71
10.1	Include statement syntax.....	72
10.2	Include example.....	72
10.3	File path resolution with absolute path notation.....	72
10.4	File path resolution with relative path notation.....	72
11.	UserKeywords statement.....	73
11.1	UserKeywords statement syntax.....	73
11.2	UserKeywords example.....	73
12.	UserFunctions statement.....	73
12.1	UserFunctions statement syntax.....	74
12.2	UserFunctions example.....	74
13.	Ann statement.....	74
13.1	Annotations statement syntax.....	74
13.2	Annotations example.....	74
14.	Signals block.....	74
14.1	Signals block syntax.....	75
14.2	Signals block example.....	77
15.	SignalGroups block.....	77
15.1	SignalGroups block syntax.....	77
15.2	SignalGroups block example.....	78
15.3	Default attribute values.....	78
15.4	Translation of based data into WaveformChar characters.....	79
16.	PatternExec block.....	80
16.1	PatternExec block syntax.....	81
16.2	PatternExec block example.....	81
17.	PatternBurst block.....	81
17.1	PatternBurst block syntax.....	82
17.2	PatternBurst block example.....	83

18.	Timing block and WaveformTable block .....	83
	18.1 Timing and WaveformTable syntax .....	84
	18.2 Waveform event definitions.....	87
	18.3 Timing and WaveformTable example .....	89
	18.4 Rules for timed event ordering and waveform creation.....	90
	18.5 Rules for waveform inheritance.....	93
19.	Spec and Selector blocks .....	94
	19.1 Spec and Selector block syntax.....	94
	19.2 Spec and Selector block example .....	96
20.	ScanStructures block.....	97
	20.1 ScanStructures block syntax .....	98
	20.2 ScanStructures block example .....	99
21.	STIL Pattern data .....	100
	21.1 Cyclized data.....	100
	21.2 Multiple-bit cyclized data .....	101
	21.3 Non-cyclized data .....	102
	21.4 Scan data .....	102
	21.5 Pattern labels.....	103
22.	STIL Pattern statements .....	103
	22.1 Vector (V) statement.....	103
	22.2 WaveformTable (W) statement.....	104
	22.3 Condition (C) statement.....	104
	22.4 Call statement.....	105
	22.5 Macro statement.....	105
	22.6 Loop statement.....	106
	22.7 MatchLoop statement.....	106
	22.8 Goto statement .....	107
	22.9 BreakPoint statements.....	107
	22.10 IDDQTestPoint statement.....	107
	22.11 Stop statement.....	108
	22.12 ScanChain statement.....	108
23.	Pattern block .....	108
	23.1 Pattern block syntax.....	108
	23.2 Pattern initialization.....	109
	23.3 Pattern examples .....	109
24.	Procedures and MacroDefs blocks.....	109
	24.1 Procedures block.....	110
	24.2 Procedures example .....	111
	24.3 MacroDefs block.....	111
	24.4 Scan testing .....	111
	24.5 Procedure and Macro Data substitution.....	112

Annex A (informative) Glossary .....	116
Annex B (informative) STIL data model.....	117
Annex C (informative) GNU GZIP reference .....	122
Annex D (informative) Binary STIL data format.....	123
Annex E (informative) LS245 design description .....	127
Annex F (informative) STIL FAQs and language design decisions.....	129

# IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data

## 1. Overview

Standard Test Interface Language (STIL) is a standard language that provides an interface between digital test generation tools and test equipment. STIL may be directly generated as an output language of a test generation tool, or it may be used as an intermediate format for subsequent processing. Figure 1 shows STIL usage in a “pipe” format. This is meant solely as a visual analogy to emphasize the high-volume/high-throughput requirements. It is not meant to represent physical structures or implementation requirements.

STIL is a representation of information needed to define digital test operations in manufacturing tests. STIL is not intended to define how the tester implements that information. While the purpose of STIL is to pass test data into the test environment, the overall STIL language is inherently more flexible than any particular tester. Constructs may be used in a STIL file that exceed the capability of a particular tester. In some circumstances, a translator for a particular type of test equipment may be capable of restructuring the data to support that capability on the tester; in other circumstances, separate tools may operate on that data to provide that restructuring. In all circumstances, it is desirable to provide the capability to check the data against the constraints of a tester. This capability is referred to as Tester Rules Checking and is the domain of tools that operate on STIL data. As such, Tester Rules Checking operations are outside the scope of this standard.

Figure 2 shows how STIL fits into the data flow between computer-aided engineering (CAE)/simulation and the test environment. In this figure, STIL is shown as both the input and output of “STIL Manipulation Tools.” STIL represents patterns as a series of cyclized waveforms that are executed sequentially. The waveform representation can be as simple as a “print-on-change” set of events, or a complex set of parameterized events. Hence, tools may be required to manipulate the data according to the requirements of a particular class of device, simulation, or tester. The output of that manipulation is still represented in STIL.

Another issue presented in Figure 2 is the need for data from the tester to be transmitted back to the CAE/simulation environment for the purpose of correlating simulation data to tester data. Although this is recognized as an important aspect of testing digital devices, it does not represent the data volume that the patterns themselves do, and is not specifically supported in this version of the standard.

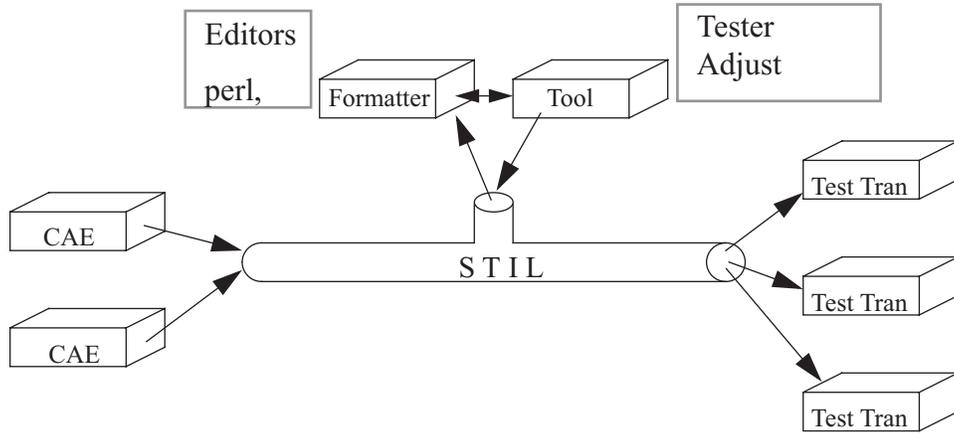


Figure 1—A conduit for transporting data from CAE to ATE

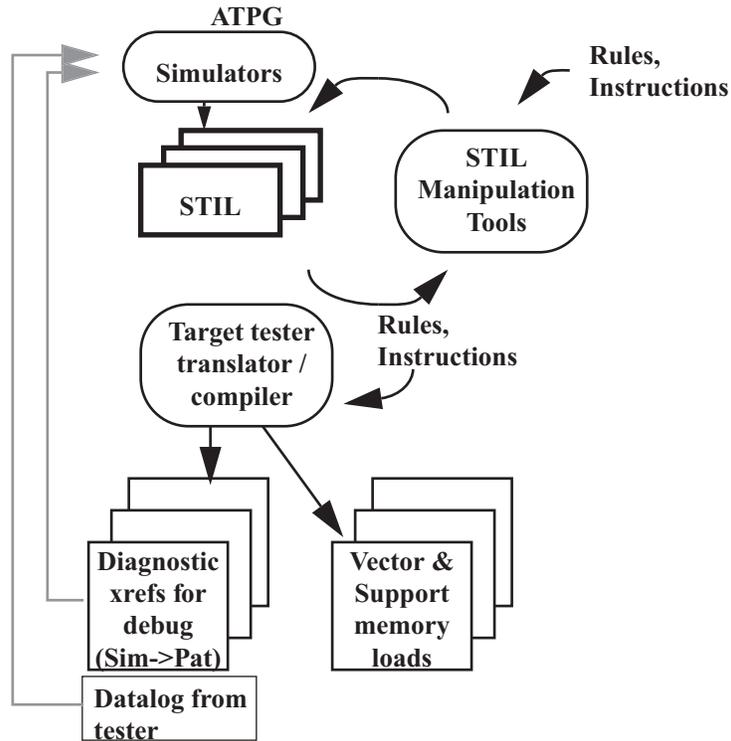


Figure 2—STIL usage model

## 1.1 Scope

This standard defines a test description language that:

- a) Facilitates the transfer of large volumes of digital test vector data from CAE environments to automated test equipment (ATE) environments;
- b) Specifies pattern, format, and timing information sufficient to define the application of digital test vectors to a device under test (DUT);
- c) Supports the volume of test vector data generated from structured tests such as scan/automatic test pattern generation (ATPG), integral test techniques such as built-in self test (BIST), and functional test specifications for IC designs and their assemblies, in a format optimized for application in ATE environments.

In setting the scope for any standard, some issues are defined to not be pertinent to the initial project. The following is a partial list of issues that were dropped from the scope of this initial project:

- Levels: A key aspect of a digital test program is the ability to establish voltage and current parameters (levels) for signals under test. Level handling is not explicitly defined in the current standard, as this information is both compact (not presenting a transportation issue) and commonly established independently of digital test data, requiring different support mechanisms outside the current scope of this standard. Termination values may affect levels.
- Diagnostic/fault-tracing information: The goal of this standard is to optimally present data that needs to be moved onto ATE. While diagnostic data, fault identification data, and macro/design element correspondence data can fall into this category (and is often fairly large), this standard is also focused on integrated circuit and assemblies test, and most debug/failure analysis occurs separately from the ATE for these structures. Note that return of failure information (for off-ATE analysis) is also not part of the standard as currently defined.
- Datalogging mechanisms, formatting, and control usually are not defined as part of this current standard.
- Parametric tests are not defined as an integral part of this standard, except for optional pattern labels that identify potential locations for parametric tests, such as  $I_{DDQ}$  tests or alternating current (AC) timing tests.
- Program flow: Test sequencing and ordering are not defined as part of the current standard except as necessary to define collections of digital patterns meant to execute as a unit.
- Binning constructs are not part of the current standard.
- Analog or mixed-signal test: While this is an area of concern for many participants, at this point transfer of analog test data does not contribute to the same transportation issue seen with digital data.
- Algorithmic pattern constructs (such as sequences commonly used for memory test) are not currently defined as part of the standard.
- Parallel test/multisite test constructs are not an integral part of the current environment.
- User input and user control/options are not part of the current standard.
- Characterization tools, such as shmoo plots, are not defined as part of the current standard.

## 1.2 Purpose

This standard addresses a need in the integrated circuit (IC)<sup>1</sup> test industry to define a standard mechanism for transferring the large volumes of digital test data from the generation environment through to test. The environment today contains unique output formats of existing CAE tools, individual test environments of IC manufacturers, and proprietary IC ATE input interfaces. As each of these three arenas solves individual problems, together they have created a morass of interfaces, translators, and software environments that provide no opportunity to leverage common goals and result in much wasted efforts re-engineering solutions. As device density increases, the magnitude of test data threatens to shift the test bottleneck from the generation process to the processes necessary solely to maintain and transport this data. These two factors threaten to eliminate any productive work performed in this area unless a viable standard is defined.

With a common standard for CAE and IC ATE environments, the generation, movement, and processing of this test data is greatly facilitated. This standard also allows for immediate access to test equipment supporting this standard, which benefits both ATE and IC vendors reviewing this equipment.

This standard also serves as a catalyst for the development of a set of standard third party interface tools to both test and design aspects of IC device generation.

## 2. References

This standard shall be used in conjunction with the following standards. If the following publications are superseded by an approved revision, the revision shall apply.

IEEE Std 100-1996, *The IEEE Standard Dictionary of Electrical and Electronics Terms*, Sixth Edition.<sup>2</sup>

IEEE Std 260.1-1993, *American National Standard Letter Symbols for Units of Measurement (SI Units, Customary Inch-Pound Units, and Certain Other Units)*.

ISO 2955:1983, *Information processing—Representation of SI and other units in systems with limited character sets*.<sup>3</sup>

ISO/IEC 9899:1990, *Programming languages—C*.<sup>4</sup>

## 3. Definitions, acronyms, and abbreviations

### 3.1 Definitions

For the purposes of this standard, the following terms and definitions apply. Additional terminology specific to this standard is found in Annex A. IEEE Std 100-1996, *The IEEE Standard Dictionary of Electrical and Electronics Terms*, Sixth Edition, should be referenced for terms not defined in this document.

<sup>1</sup>The use of this term in this standard is meant only as a point of reference and not to indicate an explicit limitation or restriction of focus.

<sup>2</sup>IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (<http://www.standards.ieee.org>).

<sup>3</sup>ISO publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembe, CH-1211, Genève 20, Switzerland/Suisse (<http://www.iso.ch/>). ISO publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA (<http://www.ansi.org/>).

<sup>4</sup>IEC publications are available from the Sales Department of the International Electrotechnical Commission, Case Postale 131, 3, rue de Varembe, CH-1211, Genève 20, Switzerland/Suisse (<http://www.iec.ch/>). IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

**3.1.1 automatic test pattern generator (ATPG):** Any tool that generates test information for a device based on structural analysis of the device.

**3.1.2 breakpoint:** A position within a pattern set where the pattern may be segmented into multiple independent bursts while still achieving predictable behavior of the device.

**3.1.3 built-in self-test (BIST):** A test paradigm that incorporates circuitry in the device for executing and resolving test information about the device.

**3.1.4 burst:** Tester execution of a pattern or set of patterns. Generally controlled by “start” and “stop” definitions.

**3.1.5 computer-aided engineering (CAE):** A computer-based set of tools to assist in the design and development of integrated circuits.

**3.1.6 cyclize:** To drive a tester, data must be provided in uniform, consistent, repeatable collections. These collections are termed “cycles” or “tester cycles.” The process of constructing these collections, generally from simulation environments, is called “cyclizing.”

**3.1.7 device:** A reference to an integrated circuit or other design structure.

**3.1.8 device under test (DUT):** The device to be placed in a test fixture and tested.

**3.1.9 float-state:** A logic value that indicates the lack of an active drive condition, generally used in an environment with multiple drivers connected to a single signal, and commonly referenced in digital simulation as a “Z” state.

**3.1.10 functional vector:** A pattern generated to exercise a device’s functional behavior. Generally defined to run the device at system speeds to verify system behavior of a design. *Contrast with: structural vectors.*

**3.1.11  $I_{DDQ}$ :** Current measurement taken at the ground rail during quiescent operation.

**3.1.12 incremental vector:** A representation of test vectors containing only the changing signals and new signal values in each vector. Parallel vectors can be generated from incremental vectors by maintaining test-specified state information for signals that did not change.

**3.1.13 metatype:** A collection of defined linguistic entities that share some common features. *Note:* In this standard, all defined metatypes represent collections of entities which may be used interchangeably in the language.

**3.1.14 newline:** The character or characters necessary to generate the start of the next line of ASCII text. May also be known as a carriage-return (CR), linefeed (LF), or a CR-LF combination.

**3.1.15 parallel vector:** A representation specifying a set of waveforms across all primary signals, to be applied to those signals in a parallel fashion (i.e., simultaneously).

**3.1.16 parametric test:** A test that is performed to verify device behavior such as output drive current, input leakage current, or output voltage.

**3.1.17 pattern:** One or more vectors comprising a functionality test for a specific portion of a device under test (DUT).

**3.1.18 primary signal:** A signal at the interface between the physical device and the physical tester. Any and all information meant for test is defined on these signals; test translators need process these signals only.

**3.1.19 pseudo signal:** A signal other than that at the interface between the device and the tester. This includes internal signals, derived signals, and any other signals that may be required by tools other than test translators to generate tests or test constructs.

**3.1.20 scan input signal:** A primary signal which may be used to serially precondition the scan register latches of the DUT.

**3.1.21 scan output signal:** A primary signal which may be used to serially observe the contents of the scan register latches of the DUT.

**3.1.22 scan test methodology:** A test methodology that utilizes shift register latches to precondition and observe modeled faults within the DUT. Scan tests typically consist of a serial preconditioning (load via scan inputs), parallel vectors to clock/transition the DUT, and then a serial observation (unload via the scan outputs).

**3.1.23 scan vectors:** A representation of test information containing lists of states that are to be shifted into or out of the scan pins on the device. *Note:* Scan vectors imply the use of scan test methodology in the design of the device under test.

**3.1.24 signal:** A point in the design from which a stimulus may be directly applied or a response directly measured.

**3.1.25 standard test interface language (STIL):** A syntax for the description of device stimulus and expected response used for stimulus development, as well as input to automated test equipment (ATE).

**3.1.26 structural vectors:** A pattern generated to exercise a device's structural elements (e.g., scan-based ATPG test generation). *Contrast with:* **functional vectors**.

**3.1.27 termination:** A constant impedance and digital logic state that a signal is held at during some or all of a test.

**3.1.28 tester cycle:** *See:* **vector**.

**3.1.29 T0 (pronounced "tee-zero"):** A reference to a MASTER clock that synchronizes all events across all signals to a common starting point. Initiates the start of each test vector.

**3.1.30 valid compare:** A condition on output response when the precise state of the response is not important to the test, but the fact that the output is a valid state value is pertinent.

**3.1.31 valid input:** A condition on input stimulus when the state of that stimulus will not affect the current test. In the simulator perspective, this condition is often identified as an unknown, or X, state.

**3.1.32 vector:** Every signal's stimuli/response to be applied/observed in the smallest integral "step" of a device test. Contains a collection of waveforms to be applied to the primary signals. *See:* **T0**.

**3.1.33 waveform:** A stream of defined events containing both state and timing information.

**3.1.34 waveshape:** A stream of defined states or transitions with no associated timing.

### 3.2 Acronyms and abbreviations

<b>ATE</b>	automated test equipment
<b>ATPG</b>	automatic test pattern generator
<b>BIST</b>	built-in self-test
<b>BNF</b>	Backus-Naur form
<b>CAE</b>	computer-aided engineering
<b>DFT</b>	design for test
<b>DAT</b>	direct access test
<b>DMA</b>	direct memory access
<b>DUT</b>	device under test
<b>FSM</b>	finite state machine
<b>IC</b>	integrated circuit
<b>I/O</b>	input/output
<b>LSB</b>	least significant bit
<b>MSB</b>	most significant bit
<b>TAP</b>	test access port
<b>TCK</b>	test clock
<b>TDI</b>	test data in
<b>TDO</b>	test data out
<b>TMS</b>	test mode select

### 4. Structure of this standard

This standard is partitioned into several clauses to assist those who are just discovering the language to learn the constructs and capabilities of STIL, and to facilitate those experienced with the language to find the particular element they need.

Clause 5 is structured as an informative tutorial to the language, and serves to introduce STIL concepts, starting with the basics and expanding into special purpose or more elaborate constructs. This clause elaborates on what is happening (and why); however, it is not intended to be a complete presentation on each construct, nor is it a normative part of the specification.

Following the tutorial are the language definition clauses. These clauses present the entire language, with all requirements and capabilities delineated completely.

The following conventions are used in this standard.

Different fonts are used as follows:

- a) SMALL CAP TEXT is used to indicate user data;
- b) `courier text` is used to indicate code examples.

In the syntax definitions:

- a) SMALL CAP TEXT is used to indicate user data;
- b) **bold text** is used to indicate keywords;
- c) *italic text* is used to reference metatypes;
- d) () indicates optional syntax which may be used 0 or 1 time;
- e) ()+ indicates syntax which may be used 1 or more times;
- f) ()\* indicates optional syntax which may be used 0 or more times;
- g) <> indicates multiple choice arguments or syntax.

In the syntax explanations, the verb “shall” is used to indicate mandatory requirements. The meaning of a mandatory requirement varies for different readers of the standard:

- To developers of tools that process STIL (“readers”), “shall” denotes a requirement that the standard imposes. The resulting implementation is required to enforce this requirement and issue an error if the requirement is not met by the input.
- To developers of STIL files (“writers”), “shall” denotes mandatory characteristics of the language. The resulting output must conform to these characteristics.
- To the users of STIL, “shall” denotes mandatory characteristics of the language. Users may depend on these characteristics for interpretation of the STIL source.

The language definition clauses contain statements that use the phrase “it is an error,” and “it may be ambiguous.” These phrases indicate improperly-defined STIL information. The interpretation of these phrases will differ for the different readers of this standard in the same way that “shall” differs, as identified in the dashed list above (Clause 4).

Waveforms represented in the diagrams use symbols defined in Table 9 through Table 12. Use the information in these tables to help understand waveform diagrams.

## 5. STIL orientation and capabilities tutorial (informative)

This clause presents an overview of STIL through a layered tutorial that explains the language constructs. This clause is informative and is not a part of IEEE Std 1450-1999, Standard Test Interface Language (STIL) for Digital Test Vector Data.

### 5.1 Hello Tester

Figure 3 represents a complete STIL program to exercise a subset of behavior for an octal bus transceiver design, modeled after a TTL LS245. Details of this design are found in Annex E. This example defines the LS245 as “unidirectional.” To simplify this example, the “A” bus signals are defined as inputs, and the “B” bus signals are defined as outputs. Figure 3 is annotated and explanations (notes) for each of the marked sections follow the figure.

NOTE—Figure notes follow all of the annotated figures in this standard.

```

1 STILL 1.0;
  Signals {
2     DIR In;
      OE_ In;
      A0 In; A1 In; A2 In; A3 In;
      A4 In; A5 In; A6 In; A7 In;
      B0 Out; B1 Out; B2 Out; B3 Out;
      B4 Out; B5 Out; B6 Out; B7 Out;
  }

3 SignalGroups {
      ABUS='A7 + A6 + A5 + A4 + A3 + A2 + A1 + A0';
      BBUS='B7 + B6 + B5 + B4 + B3 + B2 + B1 + B0';
      ALL ='DIR + OE_ + ABUS + BBUS';
}

4 Timing "hello tester timing" {
      WaveformTable one {
5         Period '500ns';
          Waveforms {
6             DIR    { 01 { '0ns' ForceDown/ForceUp; }}
              OE_    { 01 { '0ns' ForceUp; '200ns' ForceDown/ForceUp;
                          '300ns' ForceUp; }}
              ABUS   { 01 { '10ns' ForceDown/ForceUp; }}
              BBUS   { HLZ { '0ns' ForceOff;'0ns' CompareUnknown;
                          '260ns' CompareHighWindow/CompareLowWindow/CompareOffWindow;
                          '280ns' CompareUnknown;}}
          }
      } // end WaveformTable one 7
  } // end Timing "hello tester timing"

8 PatternBurst "hello tester burst" {
      PatList { "hello tester pattern"; {
  } // end PatternBurst "hello tester burst"

9 PatternExec {
      Timing "hello tester timing";
      PatternBurst "hello tester burst";
  } //end PatternExec

10 Pattern "hello tester pattern" {
      W one;
      V { ALL=0000000000LLLLLLLLL; }
      V { ALL=0010000000HLLLLLLLL; }
      V { ALL=0001000000LHLLLLLLLL; }
      V { ALL=0000100000LLHLLLLLLLL; }
      V { ALL=0000010000LLLHLLLLL; }
      V { ALL=0000001000LLLLLHLLL; }
      V { ALL=0000000100LLLLLHLL; }
      V { ALL=0000000010LLLLLHL; }
      V { ALL=0000000001LLLLLLH; }
  } //end Pattern "hello tester pattern"

```

The numbers in the circles (e.g., ①) correspond to the figure notes that follow.

**Figure 3—Hello Tester**

### Notes for Figure 3.

NOTE 1—The very first statement in a STIL file is the **STIL** statement. This statement defines the version of the STIL language following this statement.

NOTE 2—The **Signals** block defines a name for each signal used in the test vectors and identifies the signal type, such as In, Out, InOut, Supply, or Pseudo. Remember that in this example the bidirectional busses of the LS245 design are defined as unidirectional and, therefore, only In and Out are used here.

NOTE 3—The **SignalGroups** block defines an ordered set of signals to be referenced in subsequent operations. In this example, three groups are defined: a collection of all bits of the “A” bus, called ABUS; a collection of all bits of the “B” bus, called BBUS; and a collection of all signals in the design called ALL. ALL has been defined using the two previous group definitions. The operators “+” and “-” are used to define these ordered groups in objects called “pin expressions.”

NOTE 4—The **Timing** block defines sets of “WaveformTables.” Each WaveformTable defines the waveforms to be applied to each signal used in a vector. After the Timing keyword is the quoted string “Hello Tester Timing.” This quoted string becomes the name of this Timing block. By enclosing the name with double-quotes, characters such as spaces can be made to be part of the name.

NOTE 5—The first statement in a WaveformTable is the period of the test vector to be applied to all signals. All signals defined in a single WaveformTable must have the same period. In this example, the tester period is 500 ns long.

Each signal may have several different waveforms defined in a single WaveformTable. Each waveform defined for a signal will be referenced with a single character, called a WaveformChar, or “WFC.” Within each WaveformTable, each signal’s WaveformChars must be unique across all waveforms defined for the signal. However, different signals may define the same WaveformChar for different waveforms.

A waveform needs some explanation. In STIL, a waveform is a series of “time” and “event” pairs. Each pair is defined with a single STIL statement; these statements are also referred to as “timed events.” The “event” may be a special single character defined to have a particular operation, or it may be a longer identifier as used in this example. This example used the events “ForceDown,” “ForceUp,” “ForceOff,” “CompareUnknown,” “CompareHighWindow,” “CompareLowWindow,” and “CompareOffWindow.” “ForceDown” and “ForceUp” are input or drive events; “ForceDown” forces a logic low on an input, and “ForceUp” forces a logic high. “ForceOff” forces a logic float-state, or turns off any input drivers. “CompareHighWindow,” “CompareLowWindow,” “CompareOffWindow,” and “CompareUnknown” are output, or expect, events. “CompareHighWindow” expects a logic high, “CompareLowWindow” expects a logic low, and “CompareOffWindow” expects a logic float-state value. To close a window strobe, the event “CompareUnknown” is used.

There are four distinct types of signals in this design. Each has its own waveform to represent the input or output information required to test this design.

NOTE 6—The first waveform definition is for the signal DIR. This signal controls the “direction” of the bused signals, which is fixed in this test. Even though it is fixed, information is defined for signal DIR to allow this signal to be driven high or driven low at the start of each test cycle. Figure 4 shows graphically the two waveforms defined for this signal and the STIL syntax.

Because both waveforms have the same timing, they can be merged into a single STIL statement. This shorthand syntax allows multiple WaveformChars to be defined to the same event in the waveform, with states for each WaveformChar to apply at that time. The relationship of WaveformChar to event characters is direct: the first WaveformChar (in the example above, “0”) maps to the first waveform event (“ForceDown”), the second (“1”) maps to the second event (“ForceUp”), and so on for each WaveformChar present. Note that a slash must be present to separate the event references when more than one is present.

The signal OE<sub>1</sub> has additional events defined to create a pulsed behavior. The merging process for this signal is not as intuitive as DIR, as the process here requires defining events that do not cause a change of state on the signal. The only reason to define these events is to support a single waveform definition for this signal.

The ABUS signals are defined similarly to DIR, except that they are offset 10 ns into the vector boundary. Note the use of a Group here to reference a collection of signals to be defined.

The BBUS signals are defined as outputs; at entry to the cycle, any test drivers are explicitly turned off with the “ForceOff” event. The WaveformChar characters H, L and Z are mapped to the expected states “CompareHighWindow,”

“CompareLowWindow,” and “CompareOffWindow,” respectively. The strobe window is opened at 260 ns. At 280 ns, the strobe is closed with an “CompareUnknown” event.

NOTE 7—STIL supports two styles of comments: “block comments,” which are delimited by a “/\*” and “\*/”, and “comments to the newline,” which are delimited by a “//” and terminated with a newline. The comments annotating the closing braces in this example use the style of comments to the newline.

NOTE 8—The **PatternBurst** block defines a collection of pattern names to be executed sequentially. (In this example, there is only one pattern defined.) All patterns defined in a single PatternBurst are executed under a similar context, the context being defined by the subsequent PatternExec statement. The references to pattern names in this block are one of the few forward references allowed in STIL; patterns are not defined until the end of the STIL data.

NOTE 9—The **PatternExec** block defines how PatternBurst and Timing information is assembled to create the set of tests to execute. The references in this block to Timing and PatternBurst names must have been defined before this block.

NOTE 10—Finally, the pattern data is defined. Pattern data constitutes the bulk of data in the STIL data set, and is generally processed one-vector-at-a-time. In order to support processing this data as it is read, it is necessary to define pattern data as the last data in a STIL test environment.

In this example, the first statement in the **Pattern** block is a reference to a WaveformTable; the following vectors (until another ‘W’ statement) will use the timing defined in the WaveformTable named “one.”

Note that while this pattern contains references to WaveformChars, and to names of WaveformTables, it does not contain any direct references to the Timing block. This resolution is provided by the PatternExec statement. The PatternExec can define references to different Timing blocks; as long as the WaveformChars and WaveformTable names are defined in the referenced timing set, they can be applied to these same patterns.

Each **V** statement defines one test vector. In this example, each Vector defines the state to be applied to each signal using the group reference ALL. The declaration order of signals in the group ALL is critical, as the mapping of WaveformChars to signals in the group is performed linearly.

Waveforms to be defined for signal DIR:

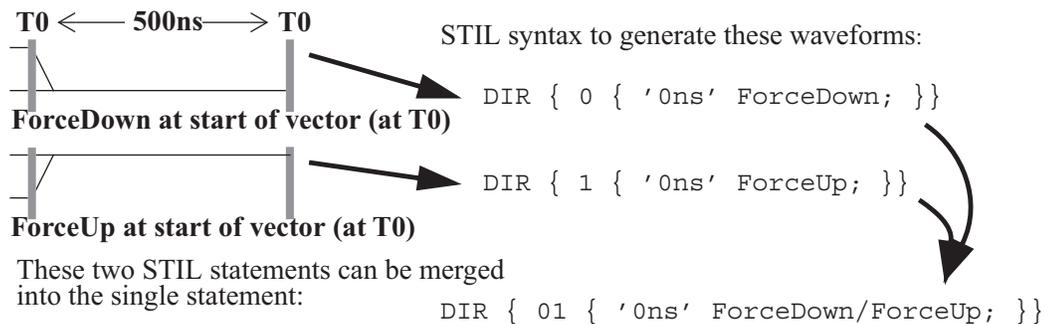


Figure 4—Waveforms associated with signal DIR

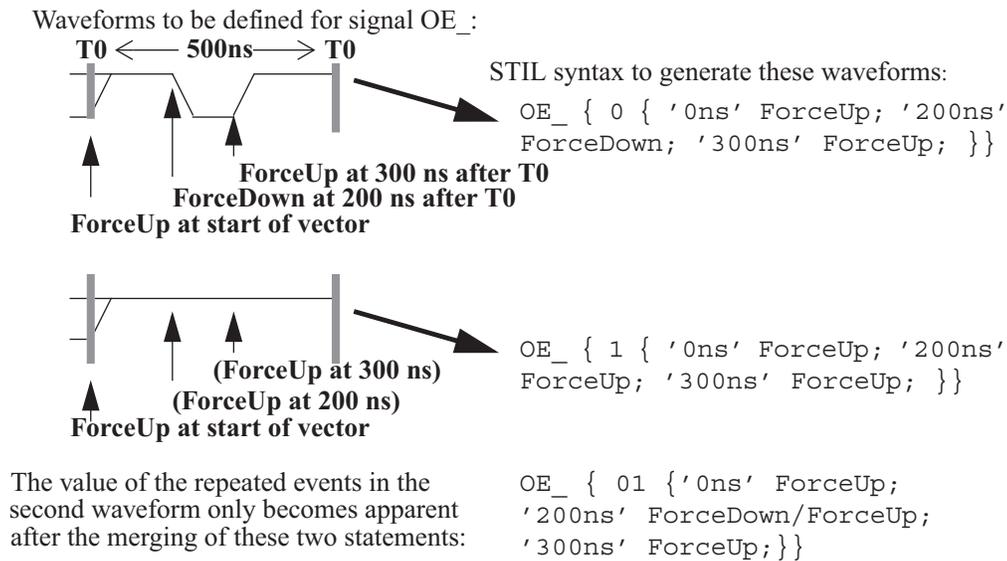


Figure 5—Waveforms associated with the signal OE\_

### 5.1.1 STIL grammatical constructs

STIL has two basic grammatical constructs for statements in the language. The first is a “simple statement,” featuring a keyword, zero or more other tokens, and terminated by a semicolon. The second is a “block statement,” which again starts with a keyword, may again be followed by zero or more tokens, and then has an open-brace. After the open-brace, additional STIL language statements may occur. This statement is terminated by a closing brace. These two statement formats are shown in Figure 6.

Simple statement:

**Keyword** (OPTIONAL\_TOKENS)\*;

Block statement:

**Keyword** (OPTIONAL\_TOKENS)\* { (OPTIONAL\_MORE\_STATEMENTS)\* }

Figure 6—STIL statement structure

Figure 6 is intended to represent a simplification of the STIL syntax. Some statements, such as the assignment statements in Figure 3 (ALL=0000000000LLLLLLLL), also require the “=” sign to be present.

STIL is a case-sensitive language. All STIL keywords start with an uppercase letter, and some may have additional uppercase letters inside.

### 5.1.2 Complexity and language subsets

The previous example of STIL data has been reduced to present the basic language constructs only. As this tutorial progresses through additional examples, different aspects of the STIL language will be presented.

There are many ways in which digital test information may be developed or provided. Some types of tests may be more concerned about testing device specifications. Such tests may not care about how the device

implements those specifications, only that the specifications are satisfied. Other types of tests may be concerned with device operation, such as functional tests, and still others may only be testing the device from a structural perspective (i.e., the elements present in the device and their interconnections) and not even have a concept of how the device is meant to be used. STIL supports all of these perspectives.

Another issue with passing information into the test environment is the definition of the test environment itself. Test equipment is varied in performance, capability, and capacity. If the goal of STIL is to provide information to be used in the test environment, how does the language ensure that this can happen? This is a major issue in the test industry, and a standard language is not going to address the problem. It was deemed critical that STIL attempt to represent tester capability relevant to digital IC and assembly test, in order to provide a mechanism to move information onto “capable” test environments, and not to constrain the language to the lowest common denominator of test capability.

Intentionally, STIL does not cause or enforce constraints on what can be represented. For example, you could legally specify a waveform with eight events. Tools reading STIL, such as a tester vendor's pattern compiler, will enforce the target tester constraints, or possibly translate the request into something that the tester can support (e.g., two four-event waveforms multiplexed together).

These issues, and several others that will be presented as STIL is discussed in this standard, lead to the inevitable conclusion that STIL, in some aspects, is rather complex. The important perspective that should be maintained is that not all of the complexity of STIL may be needed to represent device test information. Use only those constructs that are appropriate to the needs.

While this tutorial presents STIL in a “phased” aspect, from “simple” or mandatory information to more “advanced” constructs, it is important to remember that there are no actual classifications in the language. This presentation is structured in this fashion solely to facilitate presentation of the concepts.

## 5.2 Basic LS245

The previous example demonstrated a subset of the LS245 behavior. In this example, we present a complete STIL test for this device.

```

STIL 1.0;

Signals {
    DIR In;
    OE_ In;
    A0 InOut; A1 InOut; A2 InOut; A3 InOut; ①
    A4 InOut; A5 InOut; A6 InOut; A7 InOut;
    B0 InOut; B1 InOut; B2 InOut; B3 InOut;
    B4 InOut; B5 InOut; B6 InOut; B7 InOut;
}

SignalGroups {
    ABUS = 'A7 + A6 + A5 + A4 + A3 + A2 + A1 + A0';
    BBUS = 'B7 + B6 + B5 + B4 + B3 + B2 + B1 + B0';
    ALL = 'DIR + OE_ + ABUS + BBUS';
}

SignalGroups more { ②
    ABUS_I = 'ABUS' { Base Hex 01; } ③
    BBUS_I = 'BBUS' { Base Hex 01; }
    ABUS_O = 'ABUS' { Base Hex LHZX; }
    BBUS_O = 'BBUS' { Base Hex LHZX; }
}

Timing basic {
    WaveformTable one {
        Period '500ns';
        Waveforms {
            DIR { 01 { '0ns' ForceDown/ForceUp; }}
            OE_ { 01 { '0ns' ForceUp; '200ns' ForceDown/ForceUp;
                    '300ns' ForceUp; }}
            ④ ABUS { 01 { '10ns' ForceDown/ForceUp; }
                LHZX { '0ns' ForceOff; '0ns' CompareUnknown; '260ns'
CompareLowWindow/CompareHighWindow/CompareOffWindow/CompareUnknown;
                    '280ns' CompareUnknown; }}
            BBUS { 01 { '10ns' ForceDown/ForceUp; }
                LHZX { '0ns' ForceOff; '0ns' CompareUnknown; '260ns'
CompareLowWindow/CompareHighWindow/CompareOffWindow/CompareUnknown;
                    '280ns' CompareUnknown; }}
        } // end Waveforms
    } // end WaveformTable one
} // end Timing basic

```

The numbers in the circles (e.g., ①) correspond to the figure notes that follow.

**Figure 7—Basic LS245**

```

PatternBurst basic_burst {
    SignalGroups more;
    PatList { basic; }
} //end PatternBurst basic
PatternExec {
    Timing basic;
    PatternBurst basic_burst;
} //end PatternExec

Pattern basic {
    W one;
    // No default states defined;
    // the first vector must specify states on all signals.
    6 V { ALL=00ZZZZZZZZZZXXXXXXXX; }
    7 V { ABUS_I=00;BBUS_O=0000; }
    V { ABUS_I=80;BBUS_O=4000; }
    V { ABUS_I=40;BBUS_O=1000; }
    V { ABUS_I=20;BBUS_O=0400; }
    V { ABUS_I=10;BBUS_O=0100; }
    V { ABUS_I=08;BBUS_O=0040; }
    V { ABUS_I=04;BBUS_O=0010; }
    V { ABUS_I=02;BBUS_O=0004; }
    V { ABUS_I=01;BBUS_O=0001; }

    8 V { OE_=1; BBUS_O=FFFF; }

    V { DIR=1;OE_=0;ABUS_O=FFFF;BBUS_O=AAAA; }

    V { ABUS_O=0000;BBUS_I=00; }
    V { ABUS_O=0001;BBUS_I=01; }
    V { ABUS_O=0004;BBUS_I=02; }
    V { ABUS_O=0010;BBUS_I=04; }
    V { ABUS_O=0040;BBUS_I=08; }
    V { ABUS_O=0100;BBUS_I=10; }
    V { ABUS_O=0400;BBUS_I=20; }
    V { ABUS_O=1000;BBUS_I=40; }
    V { ABUS_O=4000;BBUS_I=80; }

    V { OE_=1; ABUS_O=FFFF; }
} //end Pattern basic

```

**Figure 7—Basic LS245 (continued)**

Notes for Figure 7:

NOTE 1—In this example, the “A” and “B” buses are now defined as bidirectional, or InOut in STIL terminology.

NOTE 2—Another SignalGroup definition has been added to this example. This SignalGroup has a domain name (“more” without the quotes) after the SignalGroup keyword. In STIL, names may optionally occur before the opening brace of a block section. Named blocks are referenced differently than unnamed blocks. Unnamed blocks are considered to contain “global” information; the information defined in that block may be used by any other sections after that block. Named blocks are “local” information; in order to use that information, that domain name must be explicitly referenced in a block after that declaration. The referencing mechanism for Timing blocks was already presented; the referencing mechanism for SignalGroups is discussed below.

NOTE 3—This SignalGroup adds four more group definitions to groups previously defined in the unnamed SignalGroup. These definitions contain the same signals, in the same order, but add references about a “Base” to each declaration.

The “Base” statement is used to define a default number base to be used for assignment statements referencing this group name. STIL supports the “WFC” base, which is the default mapping of WaveformChars one-to-one to signals in the group; the “hex” base, which uses hexadecimal notation for defining WaveformChar mapping; or the “decimal” base, which uses decimal notation to define WaveformChar references.

To define “hex” or “decimal” mapping, the mapping of WaveformChars to bit values in the hex or decimal number must be defined. This definition is provided by WaveformChar references after the hex or decimal word. In this example, the first group defined is ABUS\_I. ABUS\_I is defined to use a hex base for signal assignment, and the hex values are defined to map to the WaveformChars 0 and 1.

The number of bits in the hex value required to specify the WaveformChar for each signal in a group is determined by the number of the WaveformChar references present in the Base statement. In the definition of ABUS\_I, two WaveformChars are referenced. This requires one bit of a hex character to define the WaveformChar reference. The bit value to WaveformChar mapping is performed linearly: the first WaveformChar reference is assigned the bit value 0, the second WaveformChar reference is defined the value 1, and so on. Note that the values increase from left-to-right in this process; the left-most WaveformChar is assigned zero, and each subsequent WaveformChar is incremented. This process may be extended for as many WaveformChar characters desired.

It is critical to remember here that the only thing being defined is a relationship of bit-values inside a hex (or decimal) value, to WaveformChars.

The number of bits used for a hex or decimal value is always discrete for each signal in a group. If three WaveformChars are defined in a hex or decimal Base, then two bits are required to define those three states. Unused values of the binary field (such as the value “3” in a three-WaveformChar definition) cannot be specified.

In the third group definition, the group ABUS\_O is defined with four WaveformChar references. The mapping of a hex value assigned to this group is demonstrated in Figure 8.

NOTE 4—In this Timing block, the signals ABUS and BBUS are given multiple waveform definitions using two waveform statements each. Note that a single WaveformChar can only be used once in a WaveformTable per signal. Both ABUS and BBUS are given WaveformChars: 0 and 1 for input waveforms, and H,L,Z, and X for output waveforms.

Note that while the ordered definition of WaveformChars in the WaveformTable matches the order defined in the “base” statements contained in SignalGroups “more,” there is no relationship between these two ordered sets. The order of WaveformChars in WaveformTables is aligned with the events defined in the waveform. The order of WaveformChars in base statements in a SignalGroup defines a *value* used to map based-numbers to WaveformChars in vector statements.

NOTE 5—The SignalGroups block named “more” must be explicitly referenced to be used. The SignalGroups statement here provides for the definitions in this named block to be available to any Patterns referenced here. Also note that the Timing and Pattern names are the same (both are blocks named “basic”). Even though these names are the same, the name space for each block type is different and, therefore, they refer to separate, unique blocks.

NOTE 6—As stated in the comment above the first `V{}` statement, the first vector in this Pattern must define states for all signals that will be used in this Pattern because there were no `DefaultState` values defined for these signals.

This vector uses the group `ALL`. `ALL` was defined without a base statement and, therefore, defaults to a one-to-one mapping of `WaveformChars` to signals in the group.

NOTE 7—The next vector is an incremental data vector. In STIL, only the data that changes from one vector to the next needs to be identified. This vector makes use of the bus definitions in the `SignalGroup` “more” even though most of the bits of these busses do not always change. `ABUS_I` is assigned the value `00` in this second vector, which will be interpreted as a hex value because of the definition of this group. Hex `00` maps to the bits `00000000`. `ABUS_I` was defined with two `WaveformChar` references, so each bit of this value is a reference to a `WaveformChar` value for a signal in this group. All bits of `ABUS` are assigned the `WaveformChar` “0.”

`BBUS_O` is assigned the value `0000`, which is again interpreted as a hex value because of the declaration of `BBUS_O`. This expands to 16 0’s; however, `BBUS_O` was defined with four `WaveformChar` references, and so every two bits of this value corresponds to a `WaveformChar` value for each signal in this bus. Each signal in `BBUS_O` is assigned the `WaveformChar` “L” for this statement.

The next eight vectors are essentially the same vectors as in “hello tester,” as the A-bus is being driven and the B-bus is being sampled. The walking-bit pattern is repeated in this sequence.

NOTE 8—On the 11th vector, the individual signal `OE_` is referenced directly. This signal is assigned the `WaveformChar` “1,” which will hold the output disabled for this test cycle. `BBUS` is ignored in this vector from the “X” state, as all signals are assigned the bit-values “11.”

The vector data then continues, testing the opposite direction of data propagation in this device.

Group `ABUS_O` is defined to map the following `WaveformChar` references:

WFC:        L=00    H=01    Z=10    X=11

For example:

Hex Value	“B”	Bits	<u>1</u> <u>0</u> <u>1</u> <u>1</u> ”	Z for the first signal ref (value 10)
				X for the second signal (value 11)

**Figure 8—Mapping of a hex value to the group `ABUS_O`**

### 5.3 STIL timing expressions/“Spec” information

This example defines a test for the LS245 design using spec timing information. Spec timing parameters are defined using STIL constructs, and waveforms and stimulus are created to test device response against those parameters. This test validates timing against “typical” parameters, which are defined here to be an arbitrary amount less restrictive than the Max values.

The numbers in the circles (e.g., ①) correspond to the figure notes that follow.

```

STIL 1.0;

Signals {
  DIR In;
  OE_ In;
  A0 InOut; A1 InOut; A2 InOut; A3 InOut;
  A4 InOut; A5 InOut; A6 InOut; A7 InOut;
  B0 InOut; B1 InOut; B2 InOut; B3 InOut;
  B4 InOut; B5 InOut; B6 InOut; B7 InOut;
}

SignalGroups {
  ABUS = 'A7 + A6 + A5 + A4 + A3 + A2 + A1 + A0';
  BBUS = 'B7 + B6 + B5 + B4 + B3 + B2 + B1 + B0';
  BUSES= 'ABUS + BBUS';
  ALL = 'DIR + OE_ + BUSES';
}

SignalGroups more {
  ABUS_I = 'ABUS' { Base Hex 01; }
  BBUS_I = 'BBUS' { Base Hex 01; }
}

Spec tmode_spec { ①
  Category tmode { ②
    tplh      { Typ '13.00ns';Max '12.00ns'; }
    tphl      { Typ '13.00ns';Max '12.00ns'; }
    tpz1      { Typ '41.00ns';Max '40.00ns'; }
    tpzh      { Typ '41.00ns';Max '40.00ns'; }
    tplz      { Typ '26.00ns';Max '25.00ns'; }
    tphz      { Typ '26.00ns';Max '25.00ns'; }
    strobe_width = '20ns';
    tperiod    = '500ns';
  }
}

Selector tmode_typ { ③
  tplh      Typ;
  tphl      Typ;
  tpz1      Typ;
  tpzh      Typ;
  tplz      Typ;
  tphz      Typ;
}

```

**Figure 9—Spec timing tests LS245**

```

Timing to_specs {
  WaveformTable pulsed_oe {
    Period 'tperiod';
    Waveforms {
      DIR { 01 { '0ns' D/U; }} ④
      OE_ { 01 { '0ns' U; OE_MARK: '200ns' D/U;
                OE_CLOSE: 'OE_MARK+100ns' U; }}
      BUSES{ 01 { '10ns' D/U; } ⑤
            L { '0ns' Z;'0ns' X; 'OE_MARK+tpz1' l;
              '@+strobe_width' X;}
            H { '0ns' Z;'0ns' X; 'OE_MARK+tpzh' h;
              '@+strobe_width' X;} ⑥
            D { '0ns' Z;'0ns' X; 'OE_CLOSE+tplz' t;
              '@+strobe_width' X;}
            U { '0ns' Z;'0ns' X; 'OE_CLOSE+tpzh' t;
              '@+strobe_width' X;}
            X { '0ns' Z;'0ns' X; }}
    } // end Waveforms
  } // end WaveformTable pulsed_oe
  WaveformTable const_oe {
    Period 'tperiod';
    Waveforms {
      DIR { 01 { '0ns' D/U; }} ⑦
      OE_ { 01 { '0ns' D; 'tperiod-strobe_width' U; }}
      BUSES{ 01 { IN_MARK: 'tperiod/10' D/U; }
            L { '0ns' Z;'0ns' X; 'IN_MARK+tpl1' l;
              '@+strobe_width' X;}
            H { '0ns' Z;'0ns' X; 'IN_MARK+tplh' h;
              '@+strobe_width' X;}
            X { '0ns' Z;'0ns' X; }}
    } // end Waveforms
  } // end WaveformTable const_oe
} // end Timing to_specs

```

**Figure 9—Spec timing tests LS245 (continued)**

```
PatternBurst spec_check_burst {
    SignalGroups more;
    PatList { spec_check; }
} //end PatternBurst spec_check_burst

PatternExec {
    Timing to_specs;
    Selector tmode_typ, 8
    Category tmode;
    PatternBurst spec_check_burst;
} //end PatternExec

Pattern spec_check {
    W pulsed_oe;
    // the first vector must specify states on all signals.
    V { ALL=00DDDDDDDDXXXXXXXX; }
    // first set of tests check delays from OE_ signal
    V { ABUS_I=00;BBUS=LLLLLLLL; } //check BBUS tpz1 spec
    V { ABUS_I=FF;BBUS=HHHHHHHH; } //check BBUS tpzh spec
    V { ABUS_I=00;BBUS=DDDDDDDD; } //check BBUS tplz spec
    V { ABUS_I=FF;BBUS=UUUUUUUU; } //check BBUS tphz spec
    V { DIR=1; ABUS=XXXXXXXX; BBUS=DDDDDDDD; }
    V { BBUS_I=00;ABUS=LLLLLLLL; } //check ABUS tpz1 spec
    V { BBUS_I=FF;ABUS=HHHHHHHH; } //check ABUS tpzh spec
    V { BBUS_I=00;ABUS=DDDDDDDD; } //check ABUS tplz spec
    V { BBUS_I=FF;ABUS=UUUUUUUU; } //check ABUS tphz spec
    W const_oe;
    // second set of tests check data propagation delays
    V { BBUS_I=00;ABUS=LLLLLLLL; } //check ABUS tph1 spec
    V { BBUS_I=FF;ABUS=HHHHHHHH; } //check ABUS tplh spec
    V { DIR=0; BBUS=XXXXXXXX; ABUS=XXXXXXXX; }
    V { ABUS_I=00;BBUS=LLLLLLLL; }
    V { ABUS_I=FF;BBUS=HHHHHHHH; } //check BBUS tplh spec
    V { ABUS_I=00;BBUS=LLLLLLLL; } //check BBUS tph1 spec
} //end Pattern spec_check
```

**Figure 9—Spec timing tests LS245 (continued)**

## Notes for Figure 9:

NOTE 1—The **Spec** block defines spec variables in STIL. All spec variables are defined under categories in this block. Categories are used to reference sets of spec variables, and it is the **Category** block name that is important when referencing variables. The Spec block name is not significant.

NOTE 2—This **Category** defines seven variables. The first six variables are parameters representing propagation delays for the A and B bus signals. Each of these delays has two values defined in this example: a **Typ** (typical) value, and a **Max** (maximum) value. A variable may be assigned only one value for each of these fields in a Category, but may be defined to a different value when using a different Category.

The last parameter defines the tester strobe width value. This parameter has only one value, which is interpreted to be the typical value for this parameter.

NOTE 3—The **Selector** block defines which value of a Spec variable to use. The selector may specify one of four indexes to reference a Spec value: **Min**, **Typ**, **Max**, or **Meas**. Meas values are determined during test execution and are not explicitly specified in the Spec information. Note that the selector does not indicate which Category to use to identify a Spec value. The PatternExec provides that information.

The **Timing** block for this environment contains two WaveformTable definitions. The first definition defines a pulsed OE\_ signal and evaluates the propagation delay when transitioning into and out-of a float-state condition. The second definition holds the OE\_ signal low for most of the test cycle to keep the outputs active, and it changes the data with the outputs enabled to test propagation delays from one bus to the other.

NOTE 4—These Waveform definitions use the single character “event” operation vs. the long “event” operation names. Event “D” is equivalent to “ForceDown,” and “U” is equivalent to “ForceUp.” Either notation (single-character or full-name event description) may be used interchangeably.

The waveform defined for signal OE\_ includes two event\_label definitions: one for OE\_MARK, and the other for OE\_CLOSE. These labels operate in the same fashion as Spec variables except they are scoped only to the current WaveformTable. Once defined, they may be referenced in subsequent timing definitions in that WaveformTable to provide a time reference from one waveform event to another waveform event. Labels defined in Timing waveforms only have one value, which is the current value of the labeled event given the environment defined for that Timing block.

NOTE 5—The waveform defined for WaveformChar “L” in the “BUSES” group contains two timing expressions. The first expression, “OE\_MARK+tpzl,” uses the OE\_MARK label defined with respect to the OE\_ signal in the previous waveform. This defines a tester window strobe to check for a logic low state ‘tpzl’ nanoseconds after the time of OE\_MARK, which is the time that the OE\_ signal went low (active). The second expression, “@+strobe\_width,” uses the special event\_label @, which is the time of the previous timed event in the waveform; this timed event causes the window strobe to terminate “strobe\_width” nanoseconds after the previous event.

NOTE 6—This design defines two different propagation times to float-state: one delay when coming from a logic-low, and a different delay when coming from a logic-high. Checking the design to this specification requires two separate waveforms to be declared: one for the transition from the low state, and another for the transition from the high state. These are associated with the WaveformChars D and U, respectively.

Figure 10 shows graphically the waveforms associated with signals in the WaveformTable pulsed\_oe, and their association to timing specifications in the design.

NOTE 7—There are still two remaining timing specifications to validate. These are the ‘tplh’ and ‘tphl’ parameters. To validate these two specifications, the OE\_ signal must be enabled and held constant while the “input” bus is changed. This is the purpose of the WaveformTable const\_oe. In this WaveformTable, the bused signals (in output mode) are validated relative to the specified time of the input bus events in order to check the propagation delay.

NOTE 8—The **PatternExec** is responsible for defining the resolution of all timing variables by referencing both a spec **Category** name and a **Selector** name. Note this particular test references all “typical” values for this design; any device that is slower than this typical timing will fail this test.

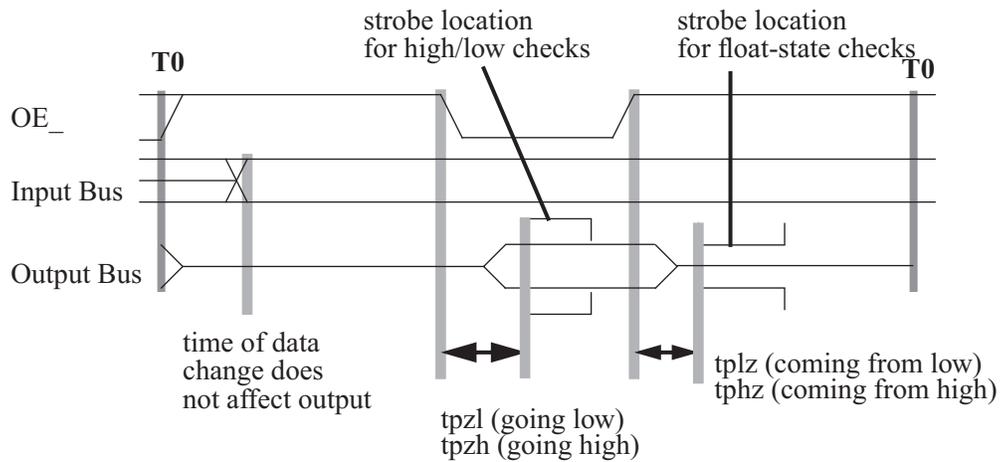


Figure 10—Waveform characteristics of WaveformTable pulsed\_oe

#### 5.4 Structural test (scan)

STIL supports structural test (scan-based) in addition to functional-based testing. To illustrate the scan constructs available in STIL, the previous LS245 example will hypothetically include scan test structures; namely, some signals will be identified as scan signals, and both procedures and macros will be provided to perform the load and unload operations.

```

STIL 1.0;

Signals {
  DIR In;
  OE_ In;
  A0 InOut; A1 InOut; A2 InOut; A3 InOut;
  A4 InOut; A5 InOut; A6 InOut; A7 InOut;
  B0 InOut; B1 InOut; B2 InOut; B3 InOut;
  B4 InOut; B5 InOut; B6 InOut; B7 InOut;
}

SignalGroups {
  ABUS = 'A0 + A1 + A2 + A3 + A4 + A5 + A6 + A7';
  BBUS = 'B0 + B1 + B2 + B3 + B4 + B5 + B6 + B7';
  ALL = 'DIR + OE_ + ABUS + BBUS';
  SI1 = 'A0' { ScanIn 30; }
  SI2 = 'A1' { ScanIn 34; }
  SO1 = 'B0' { ScanOut 30; }
  SO2 = 'B1' { ScanOut 34; }
  MASTER= 'A6';
  SLAVE = 'A7';
}

Timing {
  WaveformTable one {
    Period '500ns';
    Waveforms {
      DIR { 10 { '0ns' U/D; }}
      OE_ { 10 { '0ns' U; '200ns' U/D; '300ns' U; }}
      ABUS { 10 { '10ns' U/D; }}
      BBUS { 10 { '10ns' U/D; }}
      ABUS { HLZX { '0ns' Z; '0ns' X; '260ns' H/L/T/X; '280ns' X; }}
      BBUS { HLZX { '0ns' Z; '0ns' X; '260ns' H/L/T/X; '280ns' X; }}
    }
  } // end WaveformTable one
  WaveformTable two {
    Period '100ns';
    Waveforms {
      ALL { 10 { '0ns' U/D; }}
      ALL { HLZX { '0ns' Z; '50ns' H/L/T/X; }}
      MASTER { P { '0ns' D; '10ns' U; '40ns' D; }}
      SLAVE { P { '0ns' D; '60ns' U; '90ns' D; }}
    }
  } // end WaveformTable two
} // end Timing

```

The numbers in the circles (e.g., ①) correspond to the figure notes that follow.

**Figure 11—LS245 with hypothetical scan**

```

PatternBurst "scan_burst" { ③
    PatList { "scan"; }
}
PatternExec {
    PatternBurst "scan_burst";
}

MacroDefs { ④
    "scan" {
        W two;
        C { MASTER=P; SLAVE=P; SI1=0; SI2=0; SO1=X; SO2=X; }
        Shift { V { SI1=#; SI2=#; SO1=#; SO2=#; } }
        W one;
        C { MASTER=0; SLAVE=0; }
    }
} // end MacroDefs

Procedures { ⑤
    "scan" {
        W two;
        V { ALL=0011ZZZZZZZZZZZZZZZZZZZZ; } // define all signals
        Shift { V { MASTER=P; SLAVE=P; SI1=#;SI2=#;SO1=#;SO2=#;}}
    }
} // end procedures

Pattern "scan" {
    W one;
    V { ALL=00ZZZZZZZZZZZZZZZZZZZZ; } // define all signals
    Macro "scan" {
        SI1=00000000000000000000000000000000; ⑥
        SI2=111111111111111111111111111111111111111111111111111; }
    V { MASTER=1; OE_=1; B0=H; } ⑦
    Call "scan" {
        SO1=LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL; ⑧
        SO2=\110 HHHHHHHHHH; }
} //end Pattern "scan"

```

**Figure 11—LS245 with hypothetical scan (*continued*)**

## Notes for Figure 11:

NOTE 1—Here we hypothetically define A0 and A1 as scan inputs, and B0 and B1 as scan outputs. The scan chain lengths are also defined to be 30 and 34 via the ScanIn and ScanOut keywords.

NOTE 2—The global Timing block now defines two WaveformTables. The first WaveformTable (one) is used to define the tester cycles for non-scan activity. These cycles typically strobe outputs (measure) at the end of the cycle. The second WaveformTable (two) is used to define the tester cycles for scan activity. These cycles typically have a faster period, and strobe outputs before shifting the results in the scan chain. This table defines pulses (WaveformChar “P”) for the MASTER and SLAVE clocks to perform the shifting. Default waveforms are specified for ALL signals.

NOTE 3—The example uses only the global SignalGroups, the global Timing, the global MacroDefs, and the global Procedures blocks. Therefore, the PatternBurst only needs to define the patterns to execute (“scan”), and the PatternExec only needs to reference the PatternBurst (as Timing is global).

NOTE 4—The MacroDefs block defines a single macro (“scan”). This macro first switches to the scan timings (“two”). This macro then defines a Condition statement, which defines what the current WaveformChar for a signal is assumed to be, but doesn’t output a vector. The first vector following this macro will output this change. The Condition statement is used to start pulsing the MASTER and SLAVE clocks.

Condition statements are useful when setup information is available; however, if this setup is applied as a vector, then the subsequent data becomes difficult to align. A typical situation in which to use a Condition statement is to enable the scan clocks preceding a Shift operation, as is done in this macro. Condition statements are also useful at the end of a macro to set up information for the return. Note that Condition statements would not be useful at the end of procedures, because procedures return to the state before the procedure call, and any condition information would be discarded.

This macro also defines the “pad state” for the scan pins. The last defined state before a “#” reference is used as the pad state for scan pins. When SI1 and SI2 need to be pre-padded to normalize all chains to the same length, 0 is used. When SO1 and SO2 need to be post-padded during scan length normalization, X’s are used.

Scan testing introduces a special Shift block, which contains one or more vectors required to shift one scan event in or out of scan chains. Vectors within the Shift block use a special WaveformChar (“#”) in signal assignments to indicate that scan data is to be substituted in the vector for the signal. The values to be substituted are defined in the macro invocation, discussed below.

The macro then switches the WaveformTable reference back to “one.”

Lastly, the macro specifies a Condition statement to turn the MASTER and SLAVE clocks to their off states (“0”).

Note that this block does not have a domain\_name. The macro defined in this block is treated as a “global” name; any Pattern can reference this macro unless the macro name “scan” is defined in another (named) Macro block.

NOTE 5—The Procedures block defines a single procedure called “scan.” Note that the same name may be used for different blocks when the name spaces are unique (pattern, macros, and procedures are all called “scan”).

The procedure must first define all signals and the current WaveformTable, since nothing is assumed from the calling environment (the environment may be different for each call of the procedure). This is the major difference between procedures and macros.

The procedure defines a Shift block to apply the scan data. In this procedure, however, the C statement was not used to condition signals before the first V statement (for the sake of this example). Therefore, the MASTER and SLAVE clocks are explicitly defined as pulsing (“P”) in the Shift block’s vector. The pad event for scan signals is the last defined WaveformChar from the previous vector (“1” for inputs, “X” for outputs).

Remember that upon return from a procedure call, the environment prior to the procedure invocation is reinstated, as discussed in NOTE 8.

NOTE 6—The Macro statement is used to invoke a macro. It allows data to be defined for substitution within the macro. Specifically, it defines the scan input data for SI1 and SI2. Since no data was defined for the scan outputs, but the macro defined the substitute WaveformChar (“#”) for them (SO1 and SO2), then the last defined WaveformChar is used instead. (The pad states X, as defined in the macro’s first Condition statement.)

NOTE 7—The WaveformChars in effect after the macro include the last states from the macro. Therefore, the last state for SI1 is “0” and for SI2 is “1” (the last states of the load data). The states for SO1 and SO2 will be X, per the macro’s first Condition statement. The Macro’s second Condition statement defines the MASTER and SLAVE clocks as off (“0”). However, the vector immediately after the macro defines MASTER as a force up (“1”). Therefore, this vector outputs a “1” for MASTER and a “0” for SLAVE. This illustrates how the Condition statement may be used to define default values, and how the Vector can override it.

NOTE 8—The procedure Call performs the unload operation (only defines the scan out pins). SO2 is an “incomplete” unload, only performing 10 observations. Note the length is specified before the data, using the \ construct (“1” is a lowercase “ell”), followed by the number of bits specified (10). The length of the scan data for a signal must be either explicitly defined via “\,” or it may default to the “ScanIn” or “ScanOut” length if the length information is specified. “\10” precedes the actual scan data, since only 10 observations are required and the default length is 34. Therefore, the scan is normalized to the length of SO1 (30). The padding used is defined in the procedure. Upon returning from the Call, the environment prior to the Call is active; that is, the current WaveformTable is “one,” and MASTER is “P,” SLAVE is “0,” SI1 is “0,” SI2 is “1,” and SO1 and SO2 are “X.” (Because there is no Vector statement after this Call, however, the restored state is not important to these test vectors.)

## 5.5 Advanced scan

This example illustrates advanced scan features, including:

- Scan data that cannot be merged by STIL translators;
- Scan data that may be merged by STIL translators;
- Scan data that has been merged by a STIL creator;
- Scan data with a skewed load;
- Scan data with a skewed unload.

Scan testing typically consists of a set of “tests.” Each “test” typically consists of:

- Device preconditioning: Scanning states (loading) into the internal latches;
- Device test: Applying stimuli and clocking to inputs, and observing results on outputs;
- Device observation: Scanning states (unloading) from the internal latches.

Merged data refers to combining a previous test’s observation (unload) with the next test’s preconditioning (load). The primary reason to merge scan tests is to decrease the volume of tester vectors that impacts both tester usage and test time.

Scan data merging may be performed by a STIL translator or by a STIL writer. Merging data by the STIL translator allows the translator to safely maximize the utilization of the tester’s resources (merging as many tests as can fit within hardware limitations). Also, the translator is bound by rules defining when scan data may be merged. Merging data by a STIL writer may bypass these rules.

A skewed load test refers to applying an extra MASTER clock after loading a scan chain. This will shift the contents of the SLAVE latches into the next MASTER latches, potentially skewing the contents of individual shift register latch’s MASTER/SLAVE pairs.

A skewed unload test refers to applying an extra SLAVE clock prior to performing the scan unload. This will result in observing the contents of the MASTER latches vs. the SLAVE latches during the unload.

```

STIL 1.0;
Signals {
    reset In; scan_mode In;
    sys_clk In; MASTER In; SLAVE In; capture In;
    scan_in1 In; scan_in2 In; scan_out1 Out; scan_out2 Out;
    pi1 In; pi2 In; pi3 In; pi4 In;
    po1 Out; po2 Out; po3 Out; po4 Out;
}
SignalGroups {
    shift_clks = 'MASTER + SLAVE';
    capture_clks = 'SLAVE + capture';
    all_scan_clks = 'MASTER + SLAVE + capture';
    pis = 'pi1 + pi2 + pi3 + pi4';
    pos = 'po1 + po2 + po3 + po4';
    si1 = 'scan_in1' { ScanIn 10; }
    si2 = 'scan_in2' { ScanIn 10; }
    so1 = 'scan_out1' { ScanOut 10; }
    so2 = 'scan_out2' { ScanOut 10; }
    sins = 'scan_in1 + scan_in2';
    souts = 'scan_out1 + scan_out2';
    all = 'reset+scan_mode+sys_clk+all_scan_clks+sins+souts+pis+pos';
}
Timing one {
    WaveformTable wft_base { Period '1000ns';
        Waveforms {
            reset { 10 { '500ns' U/D; } }
            scan_mode { 10 { '500ns' U/D; } }
            sys_clk { 10 { '0ns' U; '500ns' U/D; } }
            all_scan_clks { 10 { '0ns' U/D; } }
            sins { 10 { '500ns' U/D; } }
            souts { XHL { '0ns' X; '900ns' X/H/L; } }
            pis { 10 { '500ns' U/D; } }
            pos { HL { '900ns' H/L; } }
        }
    } // end WaveformTable wft_base
    WaveformTable scan { Period '100ns';
        Waveforms {
            reset { 10 { '50ns' U/D; } }
            scan_mode { 10 { '50ns' U/D; } }
            sys_clk { 10 { '0ns' U; '50ns' U/D; } }
            MASTER { 10 { '0ns' U; '10ns' U/D; '20ns' U; } }
            SLAVE { 10 { '0ns' U; '30ns' U/D; '40ns' U; } }
            capture { 10 { '0ns' U; '10ns' U/D; '20ns' U; } }
            sins { 10 { '50ns' U/D; } }
            souts { HLX { '0ns' X; '25ns' H/L/X; } }
            pis { 10 { '50ns' U/D; } }
            pos { HL { '90ns' H/L; } }
        }
    } // end WaveformTable scan
} // end Timing one

```

The numbers in the circles (e.g., ①) correspond to the figure notes that follow.

**Figure 12—Advanced scan**

```
Procedures {
  unload_load { ①
    W scan;
    C{ all=100 001 00 XX 0000 XXXX; }
    Shift {
      V{si1=#; si2=#; so1=#; so2=#;}
    }
  } // unload_load

  unload_skewedload { ②
    W scan;
    C{ all=100 001 00 XX 0000 XXXX; }
    Shift {
      V{si1=#; si2=#; so1=#; so2=#;}
    }
    V{MASTER=0; SLAVE=1; si1=#; si2=#; so1=X; so2=X;}
  } // unload_skewedload

  skewedunload_load { ③
    W scan;
    V{ all=100 101 00 XX 0000 XXXX; }
    Shift {
      V{MASTER=0; si1=#; si2=#; so1=#; so2=#;}
    }
  } // skewedunload_load
} // Procedures
```

**Figure 12—Advanced scan (continued)**

```

PatternBurst scanpats{
  PatList {
    reset;
    scan_loads;
  }
}

PatternExec {
  Timing one;
  PatternBurst scanpats;
}

Pattern reset {
  W wft_base;
  //      Mode Clks Si  So PIs  POs
  V { all = 010  111  00  XX 0000 XXXX; }
  V { reset = 1; }
}

Pattern scan_loads {
  W wft_base;
  V { all = 110 111 00 XX 0000 XXXX; }
  Call unload_load { ④
    si1=1101110101;
    si2=0000111011;
  }
  V { pis = 1010; } // FORCE PIs
  V { pos = HLHL; } // MEASURE POs
  // FIRE CAPTURE CLK (SUBSET OF all_scan_clks)
  V { capture_clks = 00; pos = XXXX; }
  Call unload_load { ⑤
    so1=HLHHHLHLHL;
    so2=HHHLLHHLHL;
  }

  BreakPoint;
  V { pis = 0101; }
  Call unload_load {
    si1=1101110101;
    si2=0000111011;
  }
  V { pis = 1111; } // FORCE PIs
  V { pos = LLLH; } // MEASURE POs
  V { capture_clks = 00; pos = XXXX; } // FIRE CAPTURE CLK

```

**Figure 12—Advanced scan (continued)**

```
Call unload_load {
    s01=HLHLLLHLHL;
    s02=HLHLHHHLHL;
}
        6
BreakPoint;
Call unload_load {
    si1=1110010101;
    si2=0110110011;
}
V { pis = 1011;} // FORCE PIs
V { pos = HHLH;} // MEASURE POs
V { capture_clks = 00; pos = XXXX;} // FIRE CAPTURE CLK
Call unload_load {
    s01=HLHLHLHLHL;
    s02=HLHHHHHLLL;
    si1=0010010011;
    si2=0100010011;
}
        7
V { pis = 1001;} // FORCE PIs
V { pos = HLLH;} // MEASURE POs
V { capture_clks = 00; pos = XXXX;} // FIRE CAPTURE CLK
Call unload_load {
    s01=LLLLHLHLLL;
    s02=HHHHLHHLLL;
}
        8
BreakPoint;
Call unload_skewedload {
    si1=\111 11100101010;
    si2=\111 01101100111;
}
V { pis = 0011;} // FORCE PIs
V { pos = LLHH;} // MEASURE POs
V { capture_clks = 00; pos = XXXX;} // FIRE CAPTURE CLK
Call skewedunload_load {
    s01=LLHLLLHLLL;
    s02=HLHLLHHLHH;
}
        9
} // Pattern scan_loads
```

**Figure 12—Advanced scan (continued)**

## Notes for Figure 12:

NOTE 1—The `unload_load` procedure is used to apply non-skewed scan data. It utilizes the WaveformTable named Scan to execute the scan tester cycles with a 100 ns period, and to strobe the scan output signals prior to shifting. The initial Condition statement defines the background states for all signals during application of the scan data in the Shift block, including activating the MASTER and SLAVE clocks, and defining the possible padding states for scan inputs (0) and scan outputs (X).

NOTE 2—The `unload_skewedload` procedure is identical to the `unload_load` procedure, except that it contains an extra Vector after the Shift block. This Vector explicitly specifies a MASTER clock (for clarity), disables the SLAVE clock (1), and specifies to substitute the final states of the scan data. All other scan data will be applied by the Shift block. A skewed load implies the need for one extra scan input state. Therefore, standard scan data normalization would pad an extra “X” state to scan outputs if the scan data is merged.

NOTE 3—The `skewedunload_load` procedure replaces the initializing condition with a Vector which applies an initial SLAVE clock. This will cause the MASTER latches to be observed during the unload instead of the SLAVE latches. Also, the Vector in the Shift block now specifies a MASTER clock, since the previous vector had disabled the MASTER clock.

NOTE 4—The first call to the `unload_load` procedure will precondition the device for a test. The scan outputs will be at X because no scan output data was specified in the Call, which then uses the pad state to specify data for those signals.

NOTE 5—This is an example of non-mergeable scan data. A Vector exists between the call to `unload_load` for the scan output data and the call to `unload_load` with the scan input data. This means that the Vector must be performed after the scan output and before the scan input may occur.

NOTE 6—This is an example of mergeable scan data. The same procedure is called back to back for the scan output data and the scan input data, with no Vectors in-between, and no overlap of the scan signals from the two Call statements. This condition ensures that the state of all device signals are the same during the unload and load shift blocks, and the same pre-shift and post-shift vectors are applied. Therefore, STIL translators may merge the scan data and apply the inputs while simultaneously observing the previous tests’ results.

NOTE 7—This is an example of pre-merged data, generated by a STIL creator. It explicitly has specified both the scan input and output data to be applied together.

NOTE 8—This is an example of skewed load. Two things should be noted. First, this load is not mergeable with the previous unload because different procedures are used. Second, the scan input data has an extra state to be applied. This requires using the `\1` flag on the scan input data to override the default scan data length.

NOTE 9—This is an example of skewed unload.

### 5.5.1 Scan data merging

A common test time reduction technique, referred to here as Scan Merging, combines scan-unloads with subsequent scan-loads; the load data preconditions the internal latches, while simultaneously unloading and observing the previous latch states. To perform merging, the STIL translator needs to normalize all scan outputs with subsequent scan inputs to a common length.

STIL translators may perform merging if the following conditions are true:

- a) The state of the device is unchanged between the two Call or Macro statements (e.g., there are no V, C, or W statements between the unload and the load).
- b) The same Procedure or Macro is referenced by the Call or Macro statements.
- c) No Signal is defined in the two Call or two Macro statements.

Scan data may be specified already merged. In this case, one or more scan inputs and one or more scan outputs would be specified in a Call or Macro block. Merging scan data in STIL, however, may lead to errors at the tester. If a large Pattern had to be split into multiple tester buffers, then the preconditioning for a buffer may be dependent on the previous buffer. If the tester powers the DUT down between buffers, or if the buffers are executed out of sequence, then their vectors might fail.

## 5.6 IEEE Std 1149.1-1990 scan

This example (Figure 13) illustrates IEEE Std 1149.1-1990<sup>5</sup> scan features, including applying scan data outside of the Shift block.

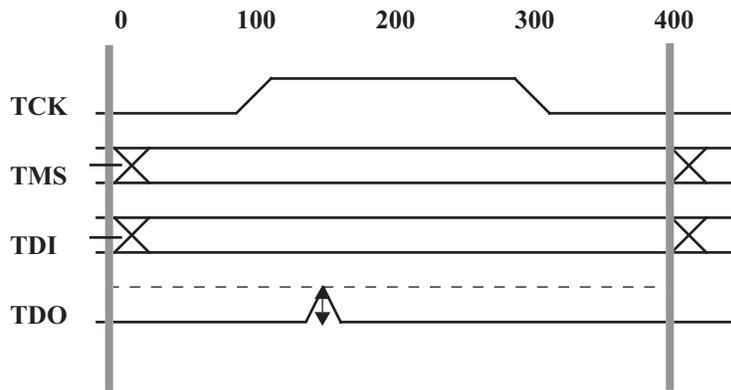


Figure 13—IEEE Std 1149.1-1990 example timings

Notes for Figure 13:

NOTE 1—State changes occur based on TMS signal present at rising edge of TCK.

NOTE 2—Output (TDO) becomes valid after the falling edge of TCK within either the “SHIFT-DR” or “SHIFT-IR” states, and is valid until the next falling edge.

NOTE 3—TAP controller instructions are 1 byte long and are represented by two hex characters (e.g., 7F = 01111111).

<sup>5</sup>IEEE Std 1149.1-1990, IEEE Standard Test Access Port and Boundary-Scan Architecture, is available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (<http://www.standards.ieee.org/>).

```
STIL 1.0;
```

The numbers in the circles (e.g., ①) correspond to the figure notes that follow.

```
Signals {
```

```
    TDO          Out;
    TCK          In;
    TDI          In;
    TMS          In;
    TRST         In;
    SYSCLK       In;
    HRESET       In;
}
```

```
SignalGroups {
```

```
    all = 'TDO + TCK + TDI + TMS + TRST + SYSCLK + HRESET';
    si  = 'TDI' { ScanIn 62; Base Hex 01; }
    so  = 'TDO' { ScanOut 62; Base Hex LHX; }
}
```

```
Timing {
```

```
    WaveformTable base {
        Period '400ns';
        Waveforms {
            TCK          { 0P { '0ns' D; '100ns' D/U; '300ns' D; } }
            SYSCLK       { 0P { '0ns' D; '100ns' D/U; '300ns' D; } }
            TDI          { 01 { '0ns' D/U; } }
            TMS          { 01 { '0ns' D/U; } }
            TRST         { 01 { '0ns' D/U; } }
            HRESET       { 01 { '0ns' D/U; } }
            TDO          { LHX { '0ns' Z; '150ns' L/H/X; } }
        }
    } // WaveformTable base
} // Timing
```

```
Procedures {
```

```
    reset {
        W base;
        // RESET - TRST AND HRESET ACTIVE (LOW)
        Loop 4095 {
            V { all = X0 0 1 0P0; }
        }

        // DEASSERT TRST
        V { all = XP 0 0 1P0; } // TEST-LOGIC-RESET
    } // reset
```

**Figure 14—IEEE Std 1149.1-1990 scan**

```
// Scan starts and ends in run_test_idle state and assumes either
// the WAITR or BIST_RESULTS instruction is loaded
scan {
    // PUT DEVICE IN SCAN STATE(SHIFT-DR)
    V { all = XP 0 1 1P1; } // SELECT-DR-SCAN
    V { all = XP 0 0 1P1; } // CAPTURE-DR
    V { all = XP 0 0 1P1; } // SHIFT-DR

    // LOAD / UNLOAD SCAN CHAIN
    Shift {
        V { all = #P # 0 1P1; } // SHIFT-DR
    }

    // LAST BIT LOADED WITH TRANSITION STATE
    V { all = #P # 1 1P1; } // EXIT1-DR

    // RESUME RUN-TEST/IDLE
    V { all = XP 0 1 1P1; } // UPDATE-DR
    V { all = XP 0 0 1P1; } // RUN-TEST/IDLE
} // scan
} // Procedures

PatternBurst pats{
    PatList {
        bist;
    }
}
PatternExec {
    PatternBurst pats;
}

Pattern bist {
    W base;
    // RESET THE FSM
    Call reset;
}
```

**Figure 14—IEEE Std 1149.1-1990 scan (continued)**

```

// LOAD INSTRUCTION 'FFRZ' = 05
V { all = XP 0 0 1P0; } // RUN-TEST/IDLE
V { all = XP 0 1 1P0; } // SELECT-DR-SCAN
V { all = XP 0 1 1P0; } // SELECT-IR-SCAN
V { all = XP 0 0 1P0; } // CAPTURE-IR
V { all = XP 0 0 1P0; } // SHIFT-IR
V { all = HP 1 0 1P0; } // SHIFT-IR
V { all = LP 0 0 1P0; } // SHIFT-IR
V { all = LP 1 0 1P0; } // SHIFT-IR
V { all = LP 0 0 1P0; } // SHIFT-IR
V { all = LP 0 0 1P0; } // SHIFT-IR
V { all = LP 0 0 1P0; } // SHIFT-IR
V { all = LP 0 0 1P0; } // SHIFT-IR
V { all = LP 0 1 1P0; } // EXIT1-IR
V { all = XP 0 1 1P0; } // UPDATE-IR
V { all = XP 0 0 1P0; } // RUN-TEST/IDLE
V { all = XP 0 0 1P0; } // RUN-TEST/IDLE

// ALLOW FFRZ TO SET
Loop 14 {
  V { all = XP 0 0 1P0; } // RUN-TEST/IDLE
}

// DEASSERT HRESET/LOAD 'WAITR' = A4
V { all = XP 0 0 1P1; } // RUN-TEST/IDLE
V { all = XP 0 1 1P1; } // SELECT-DR-SCA
V { all = XP 0 1 1P1; } // SELECT-IR-SCA
V { all = XP 0 0 1P1; } // CAPTURE-IR
V { all = XP 0 0 1P1; } // SHIFT-IR
V { all = HP 0 0 1P1; } // SHIFT-IR
V { all = LP 0 0 1P1; } // SHIFT-IR
V { all = HP 1 0 1P1; } // SHIFT-IR
V { all = LP 0 0 1P1; } // SHIFT-IR
V { all = LP 0 0 1P1; } // SHIFT-IR
V { all = LP 1 0 1P1; } // SHIFT-IR
V { all = LP 0 0 1P1; } // SHIFT-IR
V { all = LP 1 1 1P1; } // EXIT1-IR
V { all = XP 0 1 1P1; } // UPDATE-IR
V { all = XP 0 0 1P1; } // RUN-TEST/IDLE
V { all = XP 0 0 1P1; } // RUN-TEST/IDLE

```

5

6

Figure 14—IEEE Std 1149.1-1990 scan (continued)

```
// LOAD THE SCAN CHAIN
Call scan { si = 3096A5B66CF42E8C; } ⑦

// LOAD INSTRUCTION 'RUN_BIST' = 0A
V { all = XP 0 0 1P1; } // RUN-TEST/IDLE
V { all = XP 0 1 1P1; } // SELECT-DR-SCA
V { all = XP 0 1 1P1; } // SELECT-IR-SCA
V { all = XP 0 0 1P1; } // CAPTURE-IR
V { all = XP 0 0 1P1; } // SHIFT-IR
V { all = HP 0 0 1P1; } // SHIFT-IR
V { all = LP 1 0 1P1; } // SHIFT-IR
V { all = HP 0 0 1P1; } // SHIFT-IR
V { all = LP 1 0 1P1; } // SHIFT-IR
V { all = LP 0 0 1P1; } // SHIFT-IR
V { all = LP 0 0 1P1; } // SHIFT-IR
V { all = LP 0 0 1P1; } // SHIFT-IR
V { all = LP 0 1 1P1; } // EXIT1-IR
V { all = XP 0 1 1P1; } // UPDATE-IR
V { all = XP 0 0 1P1; } // RUN-TEST/IDLE

// START RUNNING WITH 1.58 MILLION CLOCKS (MINIMUM)
Loop 1580000 {
  V { all = XP 0 0 1P1; } // RUN-TEST/IDLE ⑧
}

// LOAD INSTRUCTION 'BIST_RESULTS' = CE
V { all = XP 0 1 1P1; } // SELECT-DR-SCA
V { all = XP 0 1 1P1; } // SELECT-IR-SCA
V { all = XP 0 0 1P1; } // CAPTURE-IR
V { all = XP 0 0 1P1; } // SHIFT-IR
V { all = HP 0 0 1P1; } // SHIFT-IR
V { all = LP 1 0 1P1; } // SHIFT-IR
V { all = HP 1 0 1P1; } // SHIFT-IR
V { all = HP 1 0 1P1; } // SHIFT-IR
V { all = LP 0 0 1P1; } // SHIFT-IR
V { all = LP 0 0 1P1; } // SHIFT-IR
V { all = LP 1 0 1P1; } // SHIFT-IR
V { all = LP 1 1 1P1; } // EXIT1-IR
V { all = XP 0 1 1P1; } // UPDATE-IR
V { all = XP 0 0 1P1; } // RUN-TEST/IDLE

// UNLOAD THE SCAN CHAIN ⑨
Call scan { so = AAA24600418956A000664215012AAAA; }
} // Pattern
```

**Figure 14—IEEE Std 1149.1-1990 scan (continued)**

Notes for Figure 14:

NOTE 1—The reset procedure is used to reset the TAP controller Finite State Machine (FSM).

NOTE 2—The scan procedure is used to precondition (load) the scan chain, and to observe (unload) the scan chain. The procedure first puts the device in the scan state (SHIFT-DR). The unique concept illustrated in this example is how extra scan data can be applied outside of the Shift block. STIL allows the special scan substitute character (#) to be used in pre-shift and/or post-shift vectors. Pre-shift vector(s) containing # substitutions consume the first (left-most) state(s). Post-shift vector(s) containing # substitutions consume the last (right-most) state(s). All remaining scan data states are applied in the Shift block. Therefore, this example shows that the last state (62) is applied in the Vector following the Shift block, and all remaining states (1 - 61) are applied in the Shift vector. Note also that the last state is applied when TMS transitions to complete the scan state (EXIT1-DR). The procedure then returns the device to the run-test/idle state.

NOTE 3—This example is a simple case of a single pattern contained in a Patlist. The PatternBurst only contains the Patlist block, because global signal groups are used and no special overrides are required. Similarly, the PatternExec only contains the PatternBurst name because global timings are used.

NOTE 4—The BIST pattern begins by defining the waveform table to use, and then calling the reset procedure to initialize the device's FSM.

NOTE 5—This is an example of loading an instruction into the TAP controller. Note that all pins are defined in each vector. Although this is not required, it improves the clarity of both the example and the state transitions of the TAP controller.

NOTE 6—Other instruction loads were removed to limit this example size.

NOTE 7—The scan procedure is called with just load data. Therefore, because the TDO will have no data to substitute, the last defined waveform character (X) will be applied. Since the scan length was defined as 62 in the SignalGroup, and the base is hex with two possible waveform characters, then 16 hex characters are required to encode the 62 states (plus two unused hex bits). Also note that the signal group "si" is used to define the waveform characters, but the waveforms are defined using the corresponding signal "TDI" in the Timing block.

NOTE 8—This illustrates a vector repeat with a Vector count of 1 and a Repeat of 1580000.

NOTE 9—The BIST pattern completes by unloading the scan chain. TDI has no data to substitute, so the last defined Waveform character (0) is used. Since the scan length was defined as 62 in the SignalGroup, and the base is hex with three possible waveform characters, then 31 hex characters are required to encode the 62 states in two-bit hex. Also note that the signal group "so" is used to define the waveform characters, but the waveforms are defined using the corresponding signal "TDO" in the Timing block.

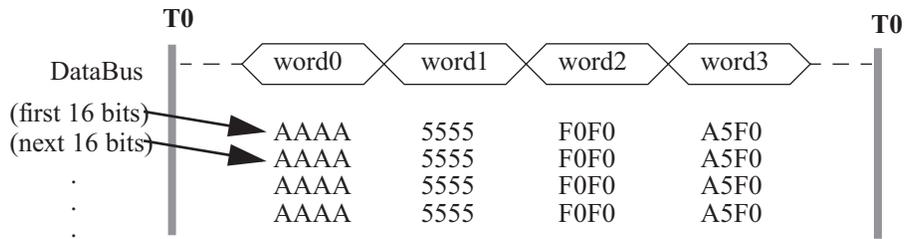
## 5.7 Multiple data elements per test cycle

One of the features of STIL is the ability to present more than one bit of data per test cycle. These constructs differ from the scan constructs presented in the previous example, although some situations may be able to use either of these constructs effectively to represent test data.

There are two differing needs for naturally representing multiple-bit data values in STIL: pipelined data and serial data streams. The following examples for these two cases demonstrate the STIL structures for supporting this data.

### 5.7.1 Burst or pipelined data

Figure 15 describes a 64-bit data bus executing a burst read cycle containing four words.



**Figure 15—Burst or pipelined data bus**

Note that the entire burst (all four reads) have been mapped into a single tester cycle.

As a pattern implementation within STIL, it is natural to specify the bus data as four sets of 64-bit hex values. The content of each of the words is presented in a format designed to preserve intent. The code in Figure 16 demonstrates the constructs in STIL necessary to support this representation. This example is partial, to emphasize multiple data elements only.

The numbers in the circles (e.g., ①) correspond to the figure notes that follow.

```
SignalGroups {
  dburst = 'D[63 .. 00]' { DataBitCount 256; Base Hex 01; }
}
WaveformTable Func {
  Waveforms {
    dburst { 01 {
      '5ns' D/U [0]; // use 0th data bit
      '10ns' D/U [1]; // use 1st data bit
      '15ns' D/U [2]; // use 2nd data bit
      '20ns' D/U [3]; // use 3rd data bit
    } }
  }
}
Pattern {
  W Func;
  V { dburst { AAAAAAAAAAAAAAAAAA; 5555555555555555; F0F0F0F0F0F0F0F0;
              A5F0A5F0A5F0A5F0; } }
}
```

**Figure 16—STIL code to support pipelined data**

Notes for Figure 16:

NOTE 1—A special group must be defined to support referencing multiple-bit data. This group must have a **DataBitCount** attribute defined with it; the number of bits will be the total number of expanded WaveformChars that will be present when this group is used in a Vector.

NOTE 2—The waveform definition for each multiple-bit signal contains a square-bracketed integer as part of the timed event data. The integer inside the square brackets defines which data bit to apply for that timed event. It is not a requirement to use the same group name in the waveform; the definition above could have used the signal expression 'D[63 .. 0]' in the waveform definition instead of the group “dburst.” However, it is a requirement that all WaveformChars to be applied under the context of a multiple-bit environment be defined collectively in a single waveform definition.

NOTE 3—Finally, the Vector is defined with a reference to the multiple-bit group “dburst.” In this representation, the data to be applied to “dburst” is specified inside braces. The data is collected across all signals defined in the group. Each set of data across all signals is defined in a separate STIL statement, terminated by a semicolon. There must be the same number of statements (sets of data) as there are indexed numbers in the waveform definition, and the total number of WaveformChars represented in this data must match the specified DataBitCount attribute of the group.

### 5.7.2 Serial data

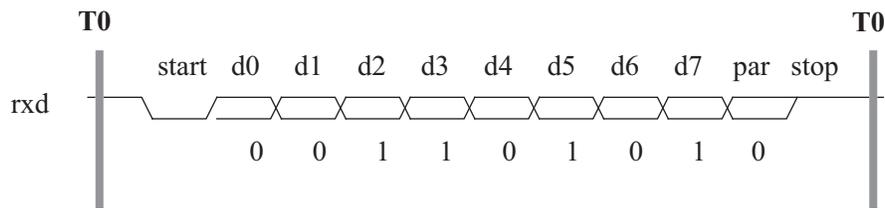


Figure 17—Serial data

An alternate representation is also supported in STIL to present serial or stream data. This format is similar to scan data in concept, but is defined as a single Vector rather than a sequence of Vectors.

Consider the above representation (Figure 17) of a serial data stream being driven onto a serial port RXD pin. The natural way to represent this data in the Vectors block is as serial bit or hex data, as shown in Figure 18.

```

STIL 0.0;
Signals {
    RXDSIG In;
}
SignalGroups {
    rxd='RXDSIG' { DataBitCount 9; Alignment MSB; Base Hex 01; }
}
Timing blk1 { WaveformTable wft1 {
    Waveforms {
        rxd { 01 {
            '5ns' D;          '10ns' D/U[0];
            '15ns' D/U[1]; '20ns' D/U[2];
            '25ns' D/U[3]; '30ns' D/U[4];
            '35ns' D/U[5]; '40ns' D/U[6];
            '45ns' D/U[7]; '50ns' D/U[8];
            '55ns' U;
        } }
    }
}
}
Pattern one {
    W wft1;
    V { rxd = \w001101010; }
    V { rxd = 350; }
}

```

The number in the circle (e.g., ①) corresponds to the figure note that follows.

1

**Figure 18—STIL code to support serial data**

NOTE (Figure 18)—The same requirements apply to the group and waveform definitions as previously presented. The difference here is that instead of applying data to the group through a STIL statement with braces, the data is directly defined. Each bit of the data in linear fashion is applied to the next indexed timed event in the waveform.

If several signals are defined in the multiple-bit group, in this notation, all indexed data associated with the first signal defined in the group would be applied first, then all indexed data associated with the second signal, until all the data was applied. This differs from the Pipelined Data approach in that with the Pipelined approach, all data to be applied to the first index value (not signal) is defined in the first statement.

### 5.7.3 Multiple bit restrictions

Valid multiple-bit data has the following semantic constraints:

First, the signal expression used in the Vector statement must reference a group that identifies a DataBitCount attribute for data to be applied when this group is used. The DataBitCount attribute must be an integral multiple of the number of signals defined in the group.

Second, vectors for multiple-bit signals must equate to WaveformChars, which define multiple-bit timing events.

Third, these constructs can only be used when all WaveformChars referenced for a particular signal are defined in a single waveform definition. In other words, the WaveformChar references in an assignment cannot mix single-bit and multiple-bit waveform definitions in the same reference.

## 5.8 Pattern reuse/direct access test

### 5.8.1 Background

This example demonstrates mechanisms in STIL that can support the reuse of pattern files between several designs that use common functional blocks, modules, megacells, cores, etc. Specifically, one strategy for support of the Direct Access Test (DAT) Design For Test (DFT) concept in STIL is demonstrated.

For this example, the design approach defines a 32-bit DAT bus at the full chip level. While in DAT test mode, the modules within the design map their module level internal input/output (I/O) signals onto any convenient sub-set of the DAT bus. Note that from one design to the next, a given module's signals may map onto different lines within the 32-bit DAT bus. Few standard cell modules will use all 32-bits of the DAT bus. In this example, the design standard cell library provides a counter module and a Direct Memory Access (DMA) controller module, and the DAT bus is wired to a portion of the design's external databus. Figure 19 shows the configuration used for this example.

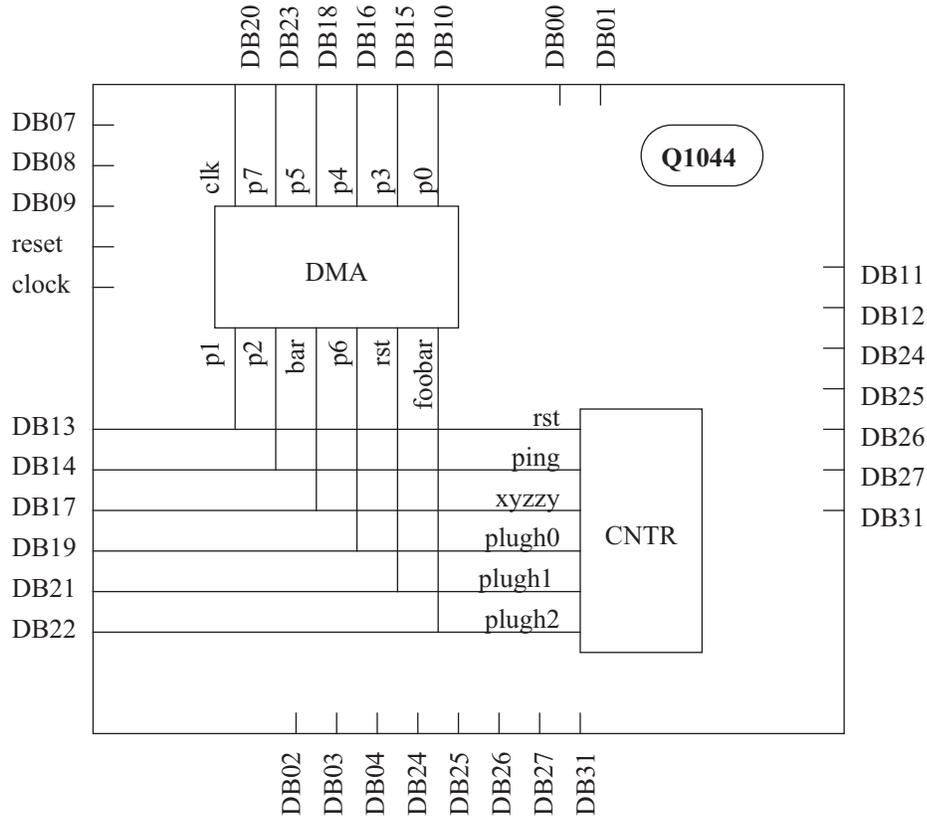
The goal, as demonstrated by this example, is to have a library of completely static tests, or patterns, for each standard cell module. These “golden” pattern source files must not require customization from one design to the next.

The example begins with two of the golden patterns: one for the DMA controller module, and one for the counter module. Each pattern is in its own file in the golden patterns directory. Note that these patterns reference only the signals for their respective modules, and that within the pattern files there are no Signals or SignalGroups blocks. The Signals and SignalGroups block declarations only exist in the design-specific STIL files and map the DAT module test signal references to the actual design's primary I/O signals.

Note that the timing blocks have been left out of this example because their content is unimportant.

The DMA\_1 and CNTR\_1 STIL files contain Vectors for testing their respective modules in the standard cell library. Signal data assignments are made only to signals on the internal and external boundaries of the given module.

Note that Figure 19, Figure 20, and Figure 21 are only code fragments to represent the data present in those files. As separate STIL files, however, they must have a STIL version statement as the first statement.



**Figure 19—Design configuration**

```
STIL 1.0;
Ann {* Pattern DMA_1 include file *}
    V { foobar=u; bar=x; clk=d; rst=d; portin=FF; }
    V { foobar=d; }
    V { rst=u; bar=1; portout=08; }
    ...
```

**Figure 20—DMA\_1.stil**

```
STIL 1.0;
Ann {* Pattern CNTR_1 include file *}
    V { xyzzy=w; plugh=0; ping=x; rst=q; }
    V { plugh=5; rst=Q; ping=1; }
    V { xyzzy=d; plugh=7; }
    ...
```

**Figure 21—CNTR\_1.stil**

The Q1044.stil file is particular to the Q1044 design. This file contains the design-specific primary I/O Signals and SignalGroups. Also included are domained (named) SignalGroups for the modules in the design that are tested through the DAT bus. For each DAT-tested module, the module-specific signals are mapped onto the primary I/Os.

```

STIL 1.0;
// Main file for the Q1044 design

Signals {           // Defines the primary I/Os for this design
  reset  In ;
  clock  In ;
  DB[31..00] InOut;
  ...           // other signals deleted
}

SignalGroups {     // Global groups declaration
  DBar = 'DB[31..00]' {Base Hex x;} ①
  // Used in the initial vector in each pattern
}

SignalGroups DMA { // DMA DAT pattern signal reference map to the
  // actual I/Os for this design.
  foobar = 'DB[22]';
  bar    = 'DB[17]';
  clk    = 'DB[20]';
  rst    = 'DB[21]';
  portin = 'DB[23]+DB[19..18]+DB[16..13]+DB[10]' {Base Hex du;}
  portout= 'DB[23]+DB[19]+DB[18]+DB[16]+DB[15]+
           DB[14]+DB[13]+DB[10]' {Base Hex LH;}
}

SignalGroups CNTR { // Counter DAT pattern signal reference map to
  // the actual I/Os for this design.
  xzyzy = 'DB[17]';
  plugh = 'DB[22]+DB[21]+DB[19]' {Base Hex du;}
  ping  = 'DB[14]';
  rst   = 'DB[13]';
}

```

The number in the circle (e.g., ①) corresponds to the figure note that follows.

**Figure 22—Q1044.stil**

```
PatternBurst CNTR_Patterns {
  SignalGroups CNTR;
  PatList {
    Q1044_CNTR_1; // Only CNTR_1 is shown in this example
    Q1044_CNTR_2;
    Q1044_CNTR_3;
    ...}
}

PatternBurst DMA_Patterns {
  PatList {
    Q1044_DMA_1 { SignalGroups DMA; }
    Q1044_DMA_2 { SignalGroups DMA; }
    Q1044_DMA_3 { SignalGroups DMA; }
    ...}
}

PatternBurst All_Patterns { PatList { DMA_Patterns; CNTR_Patterns; }

PatternExec AllPats {
  Timing Global_Timing; // Not defined in this example
  PatternBurst All_Patterns;
}

Pattern Q1044_DMA_1 {
  V { reset=r; clock=r; DBus=\R x;... }
  Include "../golden_patternsets/DMA_1.stil";
}

Pattern Q1044_CNTR_1 {
  V { reset=r; clock=r; DBus=\R x;... }
  Include "../golden_patternsets/CNTR_1.stil";
}
```

**Figure 22—Q1044.stil (continued)**

NOTE (Figure 22)—The group definition of DBus to a single WaveformChar. Because there is only one WaveformChar in the list, the hex data must be all zero values to reference this WaveformChar. This is the diminutive case for hex mapping; that is, providing a single value because only a single WaveformChar is referenced in the definition.

Design-specific versions of the DMA\_1 and CNTR\_1 patterns are created by providing a pattern that sets the state of all primary I/O signals, and then includes the golden PatternSet.

Files ../golden\_patternsets/DMA\_1.stil and ../golden\_patternsets/CNTR\_1.stil contain only V (Vector) statements.

It is within the PatternBurst block that the signal names used within the golden patterns are formally affiliated with the actual primary I/Os of the design. This affiliation is accomplished by selecting a domained SignalGroup. In the PatternBurst of Figure 22, the Q1044\_CNTR\_1 pattern is selected (which includes the golden pattern CNTR\_1.stil) and the CNTR SignalGroup is selected for resolving the signal references within the golden pattern.

The DMA PatternBurst demonstrates an alternative method for affiliating the SignalGroups with the golden patterns by using a hierarchical PatternBurst. The benefit is that the SignalGroup is only listed once. Both techniques are equivalent.

This PatternExec defines an executable set of patterns with associated timing. If the timing needs between the DMA and CNTR patterns differ, and an overlapping set of pattern data characters are used, then separate PatternExec blocks must be defined which reference different Timing blocks.

## 5.9 Event data/non-cyclized STIL information

While it is not a primary format for data presentation, STIL may be used to contain event data in the Pattern block. Event data in the Pattern block has not been cyclized and, therefore, this format is not meant to be used directly for test without additional processing. This capability is meant to be used only to provide additional data for consistency-checking, or to be used by tools designed to cyclize.

This subclause contains three examples that demonstrate one methodology for resolving event data into tester stimulus. The LS245 design of Annex E is used for the context of the information presented here.

### 5.9.1 Pure event data

The stimulus presented in Figure 23 defines the input stimulus to a LS245 design. Figure 24 defines both the input stimulus and the output response to this stimulus. Note this response includes references to two internal signals of the design, busBEN and busAEN. These signals will be used to define the direction of the A and B busses in subsequent processing.

The stimulus, while appearing to be relatively free-form, is designed to be cyclized. This will be explored in 5.9.2 and 5.9.3.

```

/* signals in column
order:
    DOAAAAAAAAABBBBBBBB
    IE0123456701234567
    R_
first column is time in
ns.
*/
0      11ZZZZZZZZZZZZZZZZZZ
1000  1110101010ZZZZZZZZ
1200  1010101010ZZZZZZZZ
1300  1110101010ZZZZZZZZ
1500  1101010101ZZZZZZZZ
1700  1001010101ZZZZZZZZ
1800  1101010101ZZZZZZZZ
2000  01ZZZZZZZZ01010101
2200  00ZZZZZZZZ01010101
2300  01ZZZZZZZZ01010101

```

**Figure 23—Stimulus used to generate STIL data**

```

/*
DO AbAAAAAAAAAbBBBBBBB
IE 0u12345670u1234567
R_  s      s
   B      A
   E      E
   N      N

Time */
0.00 ns 11 xxxxxxxxxxxxxxxxxxxxxxx
3.06 ns 11 x0xxxxxxxxxxxxxxxxxxxx
3.30 ns 11 x0xxxxxxxxx0xxxxxxxx
4.64 ns 11 z0zzzzzzzx0xxxxxxxx
4.88 ns 11 z0zzzzzzzz0zzzzzzz
1000.00 ns 11 100101010z0zzzzzzz
1200.00 ns 10 100101010z0zzzzzzz
1206.76 ns 10 100101010z1zzzzzzz
1258.52 ns 10 100101010z10z0z0z0
1268.12 ns 10 100101010110101010
1300.00 ns 11 100101010110101010
1303.30 ns 11 100101010100101010
1304.66 ns 11 10010101010z1z1z1z
1304.88 ns 11 100101010z0zzzzzzz
1500.00 ns 11 001010101z0zzzzzzz
1700.00 ns 10 001010101z0zzzzzzz
1706.76 ns 10 001010101z1zzzzzzz
1758.52 ns 10 00101010101z0z0z0z
1768.12 ns 10 001010101011010101
1800.00 ns 11 001010101011010101
1803.30 ns 11 001010101001010101
1804.66 ns 11 001010101z01z1z1z1
1804.88 ns 11 001010101z0zzzzzzz
2000.00 ns 01 z0zzzzzzz001010101
2200.00 ns 00 z0zzzzzzz001010101
2206.76 ns 00 z1zzzzzzz001010101
2258.52 ns 00 01z0z0z0z001010101
2268.12 ns 00 011010101001010101
2300.00 ns 01 011010101001010101
2303.30 ns 01 001010101001010101
2304.66 ns 01 z01z1z1z1001010101
2304.88 ns 01 z0zzzzzzz001010101
2500.00 ns 01 z0zzzzzzz100101010
2700.00 ns 00 z0zzzzzzz100101010
2706.76 ns 00 z1zzzzzzz100101010
2758.52 ns 00 z10z0z0z0100101010
2768.12 ns 00 110101010100101010
2800.00 ns 01 110101010100101010
2803.30 ns 01 100101010100101010
2804.66 ns 01 10z1z1z1z100101010
2804.88 ns 01 z0zzzzzzz100101010
3000.00 ns 01 z0zzzzzzzz0zzzzzzz

```

Figure 24—Stimulus and response data for the LS245

The numbers in the circles (e.g., ①) correspond to the figure notes that follow.

```

STIL 1.0;
Signals { DIR In; OE_ In; A0 InOut; A1 InOut; A2 InOut; A3 InOut;
  A4 InOut; A5 InOut; A6 InOut; A7 InOut; B0 InOut; B1 InOut;
  B2 InOut; B3 InOut; B4 InOut; B5 InOut; B6 InOut; B7 InOut;
  busAEN Pseudo; busBEN Pseudo; } ①
Pattern "basic_functional" { TimeUnits '10ps'; ②
  V {
    ③ @ 0      { DIR=U; OE_=U; A0=?; A1=?; A2=?; A3=?; A4=?; A5=?; A6=?; A7=?;
              B0=?; B1=?; B2=?; B3=?; B4=?; B5=?; B6=?; B7=?; }
    @ 306    { busBEN=A; }
    @ 330    { busAEN=A; }
    @ 464    { A0=F; A1=F; A2=F; A3=F; A4=F; A5=F; A6=F; A7=F; }
    @ 488    { B0=F; B1=F; B2=F; B3=F; B4=F; B5=F; B6=F; B7=F; }
    @ 100000 { A0=B; A1=A; A2=B; A3=A; A4=B; A5=A; A6=B; A7=A; }
    @ 120000 { OE_=D; }
    @ 120676 { busAEN=B; }
    @ 125852 { B1=A; B3=A; B5=A; B7=A; }
    @ 126812 { B0=B; B2=B; B4=B; B6=B; }
    @ 130000 { OE_=U; }
    @ 130330 { busAEN=A; }
    @ 130466 { B1=F; B3=F; B5=F; B7=F; }
    @ 130488 { B0=F; B2=F; B4=F; B6=F; }
    @ 150000 { A0=A; A1=B; A2=A; A3=B; A4=A; A5=B; A6=A; A7=B; }
    @ 170000 { OE_=D; }
    @ 170676 { busAEN=B; }
    @ 175852 { B0=A; B2=A; B4=A; B6=A; }
    @ 176812 { B1=B; B3=B; B5=B; B7=B; }
    @ 180000 { OE_=U; }
    @ 180330 { busAEN=A; }
    @ 180466 { B0=F; B2=F; B4=F; B6=F; }
    @ 180488 { B1=F; B3=F; B5=F; B7=F; }
    @ 200000 { DIR=D; A0=F; A1=F; A2=F; A3=F; A4=F; A5=F; A6=F; A7=F;
              B0=A; B1=B; B2=A; B3=B; B4=A; B5=B; B6=A; B7=B; }
    @ 220000 { OE_=D; }
    @ 220676 { busBEN=B; }
    @ 225852 { A0=A; A2=A; A4=A; A6=A; }
    @ 226812 { A1=B; A3=B; A5=B; A7=B; }
    @ 230000 { OE_=U; }
    @ 230330 { busBEN=A; }
    @ 230466 { A0=F; A2=F; A4=F; A6=F; }
    @ 230488 { A1=F; A3=F; A5=F; A7=F; }
    @ 250000 { B0=B; B1=A; B2=B; B3=A; B4=B; B5=A; B6=B; B7=A; }
    @ 270000 { OE_=D; }
    @ 270676 { busBEN=B; } @ 275852 { A1=A; A3=A; A5=A; A7=A; }
    @ 276812 { A0=B; A2=B; A4=B; A6=B; } @ 280000 { OE_=U; }
    @ 280330 { busBEN=A; } @ 280466 { A1=F; A3=F; A5=F; A7=F; }
    @ 280488 { A0=F; A2=F; A4=F; A6=F; }
    @ 300000 { B0=F; B1=F; B2=F; B3=F; B4=F; B5=F; B6=F; B7=F; }
  } //end V } //end Pattern

```

Figure 25—LS245 event data

Notes for Figure 25:

NOTE 1—**Pseudo** signals are specified in this example to provide information about the “enable” state of the bidirectional busses in the design. This information may be used to assist subsequent tools in determining the state of the test equipment (i.e., which drivers are turned on in which test Vectors). These signals are internal to the design. (Refer to the design description in Annex E to identify these signals.) The response data in Figure 24 includes information on the state of these signals.

NOTE 2—The **TimeUnits** statement defines the interpretation of all subsequent event statements in this Vector block. All time values in this context are integer, scaled to the value specified in this statement.

NOTE 3—This is an example of “Big Bang” timing. It is comprised of non-cyclized data, which defines Vectors as events at a timing offset (@time {signal = event;}) versus cyclized data, which defines Vectors as WaveFormChars for a Signalref (signalref = wfcs;).

It contains a single Vector and, therefore, all events in this Vector are relative to “time zero,” the start of the simulation run. Because only one period is defined (the period of the entire sequence), there is no need for WaveformTables in this representation.

In this example, the input signals DIR and OE\_ are specified using Driver state values (U,D), while the Pseudo signals and InOuts are specified using Unknown direction state values (A,B,F,?). In this example, there is no notion of “direction” on the bidirectional signals in the design; they are simply given values to represent the state information.

STIL supports integer values not exceeding  $2^{32}-1$ . If time values exceed this amount, then the Pattern must contain multiple Vectors. Each set of events is then relative to the enclosing Vector statement. If multiple Vectors are defined, each Vector must reference a WaveformTable to define the Period of that Vector. Subclause 5.9.2 demonstrates events in the context of multiple Vectors.

### 5.9.2 Mixed event and pattern data in STIL

This example is a continuation of the previous example. In this example, all input data has been cyclized relative to a 500 ns period. The output events, however, including events on the bidirectional signals when in output mode, are still left as events. Time values specified in the Event data are reset relative to each new Vector. Also, direction is now known (in this example) on the bidirectional signals; therefore, the event data is changed to represent input or output events. In other words, states are no longer applied from the Unknown Direction set {A B F ?}; now they are {U D N Z} for input direction, and {L/l H/h X T/t} for output direction.

```
STIL 1.0;
```

The numbers in the circles (e.g., ①) correspond to the figure notes that follow.

```
Signals {
  DIR In; OE_ In;
  A0 InOut; A1 InOut; A2 InOut; A3 InOut;
  A4 InOut; A5 InOut; A6 InOut; A7 InOut;
  B0 InOut; B1 InOut; B2 InOut; B3 InOut;
  B4 InOut; B5 InOut; B6 InOut; B7 InOut;
  busAEN Pseudo; busBEN Pseudo;
}
```

```
① SignalGroups {
  Abus_in = 'A0+A1+A2+A3+A4+A5+A6+A7' { Base Hex du; }
  Bbus_in = 'B0+B1+B2+B3+B4+B5+B6+B7' { Base Hex du; }
  Abus_out = 'A0+A1+A2+A3+A4+A5+A6+A7' { Base Hex z; }
  Bbus_out = 'B0+B1+B2+B3+B4+B5+B6+B7' { Base Hex z; }
}
```

```
② Timing "basic_functional" {
  WaveformTable one {
    Period '500ns';
    Waveforms {
      DIR      { ud { '0ns' U/D; }}
      OE_      { ud { '0ns' U; '200ns' U/D; '300ns' U; }}
      Abus_in  { ud { '0ns' U/D; }}
      Bbus_in  { ud { '0ns' U/D; }}
      Abus_out { z  { '0ns' Z; }}
      Bbus_out { z  { '0ns' Z; }}
    }          // end Waveforms
  }          // end WaveformTable one
}          // end Timing "basic_functional"
```

```
PatternBurst basic {
  PatList { "basic_functional"; }
}          //end PatternBurst "basic_functional"
```

```
PatternExec {
  Timing "basic_functional";
  PatternBurst basic;
}          //end PatternExec
```

**Figure 26—LS245 mixed event and cyclized data**

```

Pattern "basic_functional" {
    TimeUnits '10ps';
    W one;
    3 V { DIR=u; OE_=u; Abus_out=00; Bbus_out=00;
      @ 0      { A0=X; A1=X; A2=X; A3=X; A4=X; A5=X; A6=X; A7=X;
                B0=X; B1=X; B2=X; B3=X; B4=X; B5=X; B6=X; B7=X; }
      @ 306   { busBEN=A; }
      @ 330   { busAEN=A; }
      @ 464   { A0=t; A1=t; A2=t; A3=t; A4=t; A5=t; A6=t; A7=t; }
      @ 488   { B0=t; B1=t; B2=t; B3=t; B4=t; B5=t; B6=t; B7=t; }
      }
    V {}
    V { OE_=d; Abus_in=AA;
      @ 20676 { busAEN=B; }
      @ 25852 { B1=l; B3=l; B5=l; B7=l; }
      @ 26812 { B0=h; B2=h; B4=h; B6=h; }
      @ 30330 { busAEN=A; }
      @ 30466 { B1=t; B3=t; B5=t; B7=t; }
      @ 30488 { B0=t; B2=t; B4=t; B6=t; }
      }
    V { Abus_in=55;
      @ 20676 { busAEN=B; }
      @ 25852 { B0=l; B2=l; B4=l; B6=l; }
      @ 26812 { B1=h; B3=h; B5=h; B7=h; }
      @ 30330 { busAEN=A; }
      @ 30466 { B0=t; B2=t; B4=t; B6=t; }
      @ 30488 { B1=t; B3=t; B5=t; B7=t; }
      }
    V { DIR=d; Abus_out=00; Bbus_in=55;
      @ 20676 { busBEN=B; }
      @ 25852 { A0=l; A2=l; A4=l; A6=l; }
      @ 26812 { A1=h; A3=h; A5=h; A7=h; }
      @ 30330 { busBEN=A; }
      @ 30466 { A0=t; A2=t; A4=t; A6=t; }
      @ 30488 { A1=t; A3=t; A5=t; A7=t; }
      }
    V { Bbus_in=AA;
      @ 20676 { busBEN=B; }
      @ 25852 { A1=l; A3=l; A5=l; A7=l; }
      @ 26812 { A0=h; A2=h; A4=h; A6=h; }
      @ 30330 { busBEN=A; }
      @ 30466 { A1=t; A3=t; A5=t; A7=t; }
      @ 30488 { A0=t; A2=t; A4=t; A6=t; }
      }
    }
    //end Pattern
}

```

Figure 26—LS245 mixed event and cyclized data (*continued*)

Notes for Figure 26:

NOTE 1—This example defines four SignalGroups: two for bus-input states, and two for bus-output states. Because output information has not yet been defined, the output waveforms are defined only to turn the tester driver off (in this example, at T0 in a Vector). Because these are two *separate* sets of groups for input and output states, the proper group reference must be made in the Vectors to get the expected operation.

When the output groups are used, the hex data must be all zero values to reference the single waveform defined in hex mode. This is the diminutive case for hex mapping; that is, providing a single value because only a single WaveformChar is referenced in the definition. Even though only a single bit is defined, each driver must have its own bit of data, and the eight signals of the group still require two hex characters (of zero values) to represent the state for all signals.

NOTE 2—The Timing block defines one WaveformTable, which is then explicitly referenced in the Pattern. In this WaveformTable, waveforms are defined using the WaveformChars “u” and “d” for all input states. For the DIR signal, and the Abus or Bbus signals when being driven as an input, “u” and “d” directly map to the Drive states U and D, which are asserted at the start of the Vector (“T0”). Note that even though the characters are similar, “u” and “d” are aliases that reference waveforms, not drive states themselves. The OE\_ signal is defined as a low-going pulse, but it also is defined with the WaveformChars “u” and “d”. When “u” is applied in the Vectors, the OE\_ signal is held high; when “d” is applied, a low-going pulse is generated on the OE\_ signal.

NOTE 3—The Vectors in this data are relative to the 500 ns period defined in the WaveformTable. Therefore, the time value for each event is reset relative to the start of the current Vector.

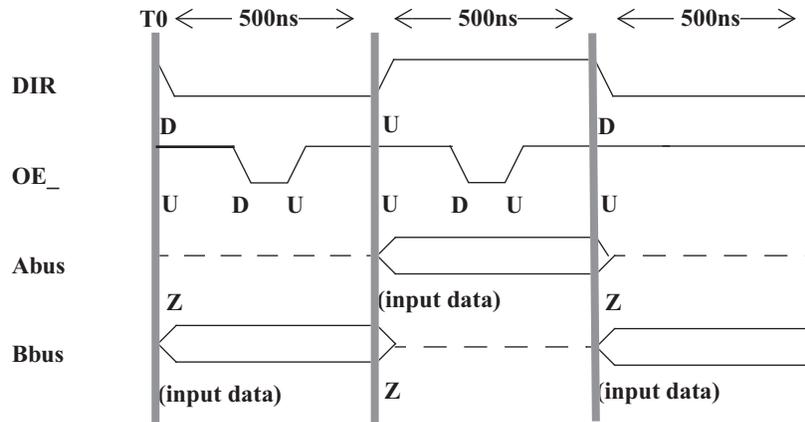


Figure 27—Cyclized input data from the events

### 5.9.3 Fully cyclized data

This is a continuation of the previous example. This example is tester-compatible; all data is provided through the application of waveforms defined in the Timing block.

In this example, there are no longer any references to pseudo signals in the Vector (although the definitions are still provided). The output bus SignalGroups have changed; now there are four waveforms specified in the output groups. This requires two bits to represent each signal in hex when these groups are used, which now requires four hex characters to represent the state of the eight signals in the group. The mapping of the two bits is shown in Figure 28.

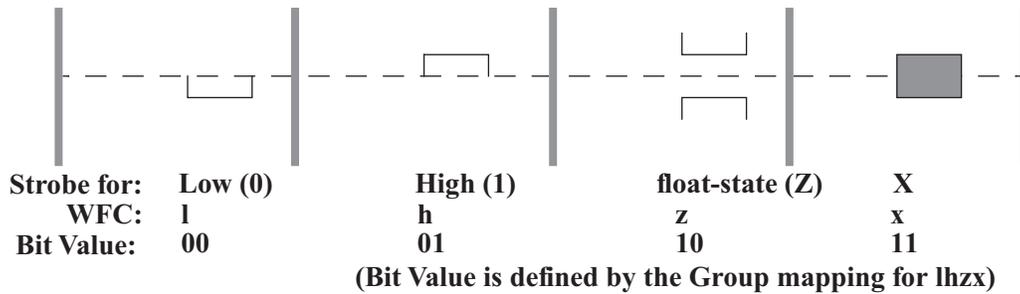


Figure 28—Output values of the bidirectional buses

```

STIL 1.0;

Signals {
  DIR In;
  OE_ In;
  A0 InOut; A1 InOut; A2 InOut; A3 InOut;
  A4 InOut; A5 InOut; A6 InOut; A7 InOut;
  B0 InOut; B1 InOut; B2 InOut; B3 InOut;
  B4 InOut; B5 InOut; B6 InOut; B7 InOut;
  busAEN Pseudo;
  busBEN Pseudo;
}

SignalGroups {
  Abus_in = 'A0+A1+A2+A3+A4+A5+A6+A7' { Base Hex du; }
  Bbus_in = 'B0+B1+B2+B3+B4+B5+B6+B7' { Base Hex du; }
  Abus_out = 'A0+A1+A2+A3+A4+A5+A6+A7' { Base Hex lhzx; }
  Bbus_out = 'B0+B1+B2+B3+B4+B5+B6+B7' { Base Hex lhzx; }
}

Timing "basic_functional" {

  WaveformTable one {
    Period '500ns';
    Waveforms {
      DIR      { ud { '0ns' U/D; }}
      OE_      { ud { '0ns' U; '200ns' U/D; '300ns' U; }}
      Abus_in  { ud { '0ns' U/D; }}
      Bbus_in  { ud { '0ns' U/D; }}
      Abus_out { hlzx { '0ns' Z; '0ns' X; '220ns' h/l/t/X; '280ns' X; }}
      Bbus_out { hlzx { '0ns' Z; '0ns' X; '220ns' h/l/t/X; '280ns' X; }}
    } // end Waveforms
  } // end WaveformTable one
} // end Timing "basic_functional"

PatternBurst basic {
  PatList { "basic_functional"; }
} //end PatternBurst "basic_functional"

PatternExec {
  Timing "basic_functional";
  PatternBurst basic;
} //end PatternExec

```

1

Figure 29—LS245 fully cyclized data

```
Pattern "basic_functional" {

W one;
V { DIR=u; OE_=u; Abus_out=AAAA; Bbus_out=FFFF; }

// In the Vector above, both buses are in output direction.

// Abus is being sampled for valid float-state.
// hex A = binary 1010; Each signal gets two bits of value in order = `1
// `10' = WFC `z' from the Abus_out SignalGroup definition.
// Bbus is not being sampled.
// hex F = binary 1111; Each signal gets two bits of value in order = `1
// `11' = WFC `x' from the Bbus_out SignalGroup definition.

V {}
V { OE_=d; Abus_in=AA; Bbus_out=4444; }

// In the Vector above, Abus is driven A0=1,A1=0,A2=1,A3=0, etc.
// Bbus, during the sample period, will follow Abus.
// During the sample period, B0=1 and B1=0.
// For B0 to be `high', it must have the value `01'
// from the SignalGroup order.
// For B1 to be `low', it must have the value `00'
// from the SignalGroup order.
// `0100' = hex value `4'.

V { Abus_in=55; Bbus_out=1111; }
V { DIR=d; Abus_out=1111; Bbus_in=55; }
V { Bbus_in=AA; Abus_out=4444; }

} //end Pattern
```

**Figure 29—LS245 fully cyclized data (*continued*)**

NOTE (Figure 29)—Notice that the order of the WaveformChars in the Base statement for the group declaration of Abus\_out (lhzx) is different than the order presented in the Waveform declaration for Abus\_out (hlzx). These two fields are independent of each other, as was identified previously in Figure 7. (See NOTE 4 for Figure 7.)

## 6. STIL syntax description

This clause describes, in general, the basic syntax and semantic constructs of STIL.

### 6.1 Case sensitivity

STIL is case-sensitive; all tokens, including identifiers, are manipulated in a case-sensitive fashion. For instance, “Dbus” and “dbus” are two different identifiers.

### 6.2 Whitespace

Whitespace in STIL is one or more of the following:

	space
\t	tab
\n	newline character

### 6.3 Reserved words

All keywords are reserved for the explicit use as defined for the keyword. STIL keywords have the first character of each word in upper case, and no underscores or spaces are used. For instance, WaveformTable is a reserved word in STIL.

Reserved words are generally the first token in a STIL statement. They are used only in the context of that word, except for single-character reserved words, which may also appear in WaveformChar contexts. See Clause 11 for the definition of how to extend STIL by adding additional reserved words.

Table 1 lists the reserved words, including reserved single characters, in STIL.

**Table 1—STIL reserved words**

A, Alignment, Ann
B, Base, BreakPoint
Call, Category, CompareHigh, CompareHighWindow, CompareLow, CompareLowWindow, CompareUnknown, CompareValid, CompareValidWindow, CompareZ, CompareZWindow, Condition
DataBitCount, Date Dec, DefaultState
ExpectHigh, ExpectLow, ExpectOff
F, ForceDown, ForceOff, ForcePrior, ForceUnknown, ForceUp
G, Goto
H, Header, Hex, History
I <sub>DDQ</sub> TestPoint, IfNeed, In, Include, Infinite, InheritWaveform, InheritWaveformTable, InOut
L, LogicHigh, LogicLow, LogicZ, Loop, LSB
M, Macro, MacroDefs, Marker, MatchLoop, Max, Meas, Min, MSB
N
Out
P, PatList, Pattern, PatternBurst, PatternExec, Period, Procedures, Pseudo
Q
R
ScanCells, ScanChain, ScanIn, ScanInversion, ScanLength, ScanMasterClock, ScanOut, ScanOutLength, ScanSlaveClock, ScanStructures, Selector, Shift, SignalGroups, Signals, Source, Spec, Start, STIL, Stop, SubWaveforms, Supply
T, TerminateHigh, TerminateLow, TerminateOff, TerminateUnknown, Termination, TimeUnit, Timing, Title, Typ
U, Unknown, UserFunctions, UserKeywords
V, Variable, Vector
W, Waveforms, WaveformTable
X
Z

## 6.4 Reserved characters

Table 2 lists the reserved characters in STIL.

**Table 2—STIL reserved characters**

Character	Usage
;	Semicolon is used as a statement delimiter.
{}	Left and right braces are used as block delimiters.
[]	Left and right square brackets are used to denote numeric indexes.
@	“At” sign is used to identify timing in vectors and waveforms.
“ “	Double quote character is used to denote literal strings.
‘ ‘	Single quote character is used to denote timing and signal expressions.
:	Colon is used to terminate labels.
/	Forward slash is used to separate state characters in waveforms, and as the division operator in timing expressions.
//	Double slash defines a comment-to-newline.
/* */	Defines a comment block.
{* *}	Contains an annotation (Ann) block.
.	Period separates hierarchical names in timing references, and is used to concatenate strings.
,	Comma delimits list of arguments in timing expressions.
!	Exclamation (NOT sign) defines scan cell inversion in scan chain definitions.
#	Pound sign defines incremental parameter data replacement in macros/procedures.
%	Percent sign defines fixed parameter replacement in macros/procedures.
\	Back slash delimits vector flags (used to modify vec_data).
()	Parentheses reserved in timing and signal expressions.
*	Multiply in timing expressions.
+	Add in timing and signal expressions.
-	Subtract in timing and signal expressions.
<	Less than in timing expressions.
>	Greater than in timing expressions.
<=	Less than or equal to in timing expressions.
>=	Greater than or equal to in timing expressions.
=	Equal to in timing expressions.
!=	Not equal to in timing expressions.
?:	Conditional selection in timing expressions.
=	Assignment in: timing expressions, vector expressions, groupname expressions, spec category expressions, and spec variable expressions (5 contexts).

**Table 2—STIL reserved characters (continued)**

Character	Usage
..	Range indication for signal expressions.
	Whitespace set (space, tab, newline).
E, P, T, G, M, k, m, u, n, p, f, a	Engineering prefixes may modify SI units in timing expressions (see Table 4).
A, Cel, F, H, Hz, m, Ohm, s, W, V	SI units may appear in timing expressions (see Table 3).
min, max	Functions may appear in timing expressions.

## 6.5 Comments

There are two styles of comment in STIL:

```
// line comment           line comments are terminated by newline
/* block comment */      block comments may span multiple lines
```

Comments may appear at any legal whitespace location and are treated as whitespace. Nested block comments shall not be allowed (e.g., “/\* /\* /\* /\* \*/”), but line comments may be contained in block comments. Comments defined using these constructs may not be preserved through STIL processes. See Clause 13 for annotations, which are a type of comment that is preserved through processes.

## 6.6 Token length

Tokens are defined to be the block of text between reserved characters, or reserved characters themselves (other than whitespace and comment delimiters). Tokens are limited to a maximum length of 1024 characters. Longer sequences of character strings may be defined by segmenting the character string into sections and placing a period between the sections (see 6.7).

## 6.7 Character strings

Blocks of text containing reserved characters or STIL-defined reserved words may be passed through STIL by double quoting the text. Signal names that contain reserved characters or match STIL-defined reserved words, and text strings that contain whitespace, are maintained in a STIL file by enclosing the text in double quotes. Double-quoted strings are constrained to a maximum length of 1024 characters (including quotes). Longer character strings may be defined by partitioning the string into segments, quoting each segment, and placing a “.” (period) between each consecutive string. Only a single period and whitespace may occur between multiple segments of a character string. The complete character string is defined as a concatenation of all quoted strings separated by periods. For example, the character strings “acell.” . “bpart” are internally handled as the single character string “acell.bpart.” Be aware that handling of references in the Timing block (Clause 18) may not eliminate the intervening period.

## 6.8 User-defined name characteristics

There are several categories of user-defined names in STIL: signal and group references, WaveformChar references, WaveformTable references, variable references, UserKeywords, labels, and domain names.

If a user-defined name contains STIL reserved characters or is identical to a STIL reserved word, then that name shall be quoted in double quotes.

User-defined names shall be unique to their respective domains (as defined in Table 6).

User-defined names may be declared either unquoted or enclosed in double quotes. Once declared, all references to that name shall use the same convention to reference that name; for instance, the name “Xyz” is always referenced as “Xyz” (with quotes present).

Unquoted, user-defined names have the following naming restrictions. The first character in unquoted names shall be alphabetic or an underscore. The remaining characters may be alphanumeric or underscores. Names that contain any other character or character sequence shall be enclosed in double-quotes.

The following Backus-Naur Form (BNF) represents these above-stated options for user-defined names:

```

name ::= name_segment | name “.” name_segment
name_segment ::= simple_identifier | escaped_identifier (The maximum length of a name_segment is
1024 characters.)
simple_identifier ::= letter_or_underline simple_characters
simple_characters ::= simple_characters simple_character | (null)
letter_or_underline ::= letter | underline
simple_character ::= letter | digit | underline
letter ::= upper_case_letter | lower_case_letter
upper_case_letter ::= “A” | “B” | ... | “Z”
lower_case_letter ::= “a” | “b” | ... | “z”
underline ::= “_”
escaped_identifier ::= “”“” escaped_characters””””
escaped_characters ::= escaped_characters escaped_character | escaped_character
escaped_character ::= simple_character | special_character | whitespace_character
special_character ::= !@#$%^&*()-+=|`~{ } ; , < . > / ? \
whitespace_character ::= “ ” | “\t” | “\n”

```

Names may not contain a double-quote character. Signal or Group names may contain square brackets, with integer values inside, at the end of the name string. (See 6.10 for more information.)

## 6.9 Domain names

Certain STIL block statements support the option of a user-defined “domain name” before the opening brace of the block statement. Domain names provide a mechanism to reference the data defined in a named block. When a domain name is present for a SignalGroups, Procedures, MacroDefs, PatternBurst, Timing, Selector, Pattern or ScanStructures block, that domain name shall be specified in a “reference” statement in order to make use of the data in that block. (See Clause 16 for an example of a “reference” statement to a named PatternBurst block inside a PatternExec block.)

All domain names for a single type of block shall be unique; for instance, all Pattern blocks need unique names. (See 6.16 for more information about domain names in STIL.)

## 6.10 Signal and group name characteristics

Signal and group names are user-defined names and follow the requirements listed previously for user-defined names. In addition, a set of signals with a common name and a numeric index may be expressed using a double-period ellipsis (..) operator. To use the ellipsis operator, the signal names shall be appended with an index number in square brackets. For example, the signals referenced by the statement `data[0..36]` would include the range of signals from `data[0]` through `data[36]`. If signal names are quoted (because of characters used in the name), the quotes occur before the bracketed part of the name; for example, `"a&b"[0..7]`. This defines signals `"a&b"[0]` through `"a&b"[7]`. The brackets, when present, become part of the name reference, and the values inside the bracket are interpreted as integer values only. For example, the signal `data[0]` is the same as the signal `data[00]`, but is not the same as `data00`. The values may be defined in either ascending (`[0..7]`) or descending (`[7..0]`) order. The square-bracket operation is allowed any place a signal expression may occur. It is allowed as the name of a series of signals in a Signals block, but is not allowed as the name of a group in the SignalGroups block.

## 6.11 Timing name constructs

Timing blocks allow an additional mechanism to support the importing of timing data defined in one block into a different Timing block. The period (".") is used to reference specific information through hierarchical levels of Timing information. For example, a Timing block with the name ALL, containing a WaveformTable with the name ONE, may be referenced in a subsequent Timing block with the statement `WaveformTable ALL.ONE`; (see Clause 18 for more details on this capability).

## 6.12 Number characteristics

Certain fields in STIL expect a numeric value. The following types are defined:

- Integer numbers are contiguous sets of the characters 0-9 (e.g., "77"). These numbers may be preceded by a minus sign to indicate a negative value. A conforming reader shall accept any integer number capable of being represented in a 32-bit twos complement binary value.
- Signed real numbers are contiguous sets of the characters 0-9. A decimal point may appear once in the number (e.g., "123.456"). These numbers may be preceded by a minus sign to indicate a negative value. The precise value of a Signed real number may round to the precision/resolution of the machine interpreting that number.
- Exponential numbers are signed real numbers, followed by either an uppercase or lowercase "e," and followed by a signed decimal number for the exponent (e.g., "1.0e-9"). Exponential numbers may not contain embedded whitespace. The precise value of an exponential number may round to the precision/resolution of the machine interpreting that number.
- Hex numbers are contiguous sets of the characters 0-9,A-F,a-f (e.g., "HA4"). Hex numbers are always positive.

The following BNF representation restates these characteristics:

```

digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
hexdigit ::= digit | "a" | "A" | "b" | "B" | "c" | "C" | "d" | "D" | "e" | "E" | "f" | "F"
hex_number ::= hexdigit | hexdigits hexdigit
integer ::= digit | integer digit
signed_integer ::= integer | "-" integer
number ::= signed_integer
          | signed_integer "." integer
          | signed_integer "e" signed_integer
          | signed_integer "." integer "e" signed_integer

```

### 6.13 Timing expressions and units (*time\_expr*)

Timing expressions are enclosed in single quotes. Expressions consist of integers, real numbers, exponential numbers, spec variables, event\_labels, and operators. Exponential numbers may be expressed using real numbers or exponential numbers, followed by engineering notation using the prefixes and units identified in Table 3 and Table 4. There shall be no whitespace between the value and the engineering notation. Only simple units are allowed in STIL (e.g., 3V, 44.5mA, 22ns). Complex units like (3.5V/ns) are not allowed; however, they may be represented as a ratio of two values (3.5V/1ns). Only expressions that compute to units of time or scientific numbers may be used in a timing expression. The following are examples of timing expressions that may be used in the construction of a waveform:

'5ns'	// 5ns
'5.0ns'	// 5ns
'15.0ns/3'	// 5ns
'5.0e-9s'	// 5ns
'1/200MHz'	// 5ns
'txx'	// simple variable from a spec sheet or labeled event
'txx*5'	// expression
'txx+5ns'	// expression
'@+5ns'	// expression relative to previous event
'@3+5ns'	// expression relative to event 3 of current waveform
'wft_x.tlab+3ns'	// expression relative to another WaveformTable

Table 5 specifies the operators and functions supported for timing expressions in STIL. These operators follow the same precedence as equivalent operators in the C-language, with parentheses having the highest precedence. Horizontal dark lines in this table indicate operators of equal precedence; precedence decreases going down the table. Complex statements may use parentheses.

The SI units in Table 3 are derived from ISO 2955:1983 and IEEE Std 260.1-1993.<sup>6</sup> An engineering prefix from Table 4, when present, shall be used in conjunction with an SI unit from Table 3.

There are two contexts where time expressions may occur. The first context is in the Spec block as part of a spec variable definition. The second context is in the waveform block, to define the timing associated with an event.

A time expression occurring in a waveform block supports additional constructs not supported with time expressions in spec variable definitions. These additional constructs provide the ability to reference other time events as part of a new time expression. There are two mechanisms defined to support referencing other time events:

<sup>6</sup>Information on references can be found in Clause 2.

- Use of `event_labels`. A waveform definition may contain `event_labels`, which may be referenced in other timing expressions just like a `spec` variable. The time value of that labeled event is the value of the `event_label` used in the time expression. References to `event_labels` may be resolved in different waveform definitions in a single `WaveformTable`, but event labels are scoped to the current `WaveformTable`. All referenced `event_labels` shall be defined in the `WaveformTable` in which they are referenced. Inherited `WaveformTables` become part of the current `WaveformTable`; hence, they may include labels to be referenced and label references.
- Use of '@' and '@n' time marks. These constructs are used to reference events in the current waveform definition only. The '@' construct references the event previously defined (in a syntactic sense), that is, the previous event to the left of the current definition. The '@n' construct references the n<sup>th</sup> event statement in the current waveform; '@1' references the first event statement in the waveform. These constructs may reference events yet to be defined. These constructs shall not reference an undefined event, nor reference from any path one event back to itself (circular references).

NOTE—The term *time\_expr* is used in this standard as part of the syntax descriptions to indicate the presence of timing expressions in statements.

**Table 3—SI units**

Unit	Description
A	Amperes
Cel	Degrees Celsius
F	Farads
H	Henries (inductance)
Hz	Hertz
m	Meter
Ohm	Ohms
s	Seconds
W	Watts
V	Volts

**Table 4—Engineering prefixes**

Prefix	Description	Multiplier
E	exa	$10^{18}$
P	peta	$10^{15}$
T	tera	$10^{12}$
G	giga	$10^9$
M	mega	$10^6$
k	kilo	$10^3$
m	milli	$10^{-3}$
u	micro	$10^{-6}$
n	nano	$10^{-9}$
p	pico	$10^{-12}$
f	femto	$10^{-15}$
a	atto	$10^{-18}$

### 6.14 Signal expressions (*sigref\_expr*)

Signal expressions define an ordered list of signals; they are either a single token, or an expression enclosed in single quotes. Signal expression operators are plus (+), minus (-), ellipsis (..), and parentheses. These operators are not extendable. Expressions are evaluated left-to-right, with parentheses used to override this order. Signals referenced in signal expressions may occur only once in the sub-expressions generated during evaluation of the expression.

A Signal expression shall not “remove” a signal (using the minus operator) that is not part of an expression as currently defined, and an expression shall not “add” a signal (using the plus operator) that is already part of an expression as currently defined.

For instance, if sig1, sig2, sig3, sig4, and sig5 are defined, and two groups are defined:

```
SignalGroups {
    grp1 = 'sig3+sig2+sig1';
    grp2 = 'sig3+sig4+sig5'; }
```

In this example, then, it would be incorrect to define a group that combines these two groups, because sig3 is a member of both groups:

```
grp3 = 'grp1+grp2';           //error
```

However, this group could be assembled by removing the repeated element first. There are two different ways this could be done, depending on what final grouping is desired. If the desire is to keep the sig3 reference at the “end” (right-hand-side) of the final expression, then the following expression could be used:

```
grp3 = 'grp1-sig3+grp2';     // 'sig2+sig1+sig3+sig4+sig5'
```

**Table 5—Operators and functions allowed in a timing expression**

Op	Definition
min ()	Minimum value
max ()	Maximum value
()	Parentheses
Table 3, Table 4	SI units and prefixes
/	Divide
*	Multiply
+	Add
–	Subtract
<	Less than
>	Greater than
<=	Less or equal
>=	Greater or equal
==	Equal
!=	Not equal
?:	Conditional expression
=	Assignment

If the desire is to keep the sig3 reference in the beginning of the final expression, then parenthesis are used to override the left-to-right evaluation:

```
grp3 = 'grp1+(grp2-sig3)'; // 'sig3+sig2+sig1+sig4+sig5'
```

The ellipsis (..) operator is a shorthand notation for expressing signals as a range. The signals shall be identified as a name followed by an index number in square brackets:

```
databus = 'data[0..31]'; // data[0], data[1],... data[31]
dbus2 = 'dbus2[20..1]'; // dbus2[20], dbus2[19],... dbus2[1]
```

NOTE—The term *sigref\_expr* is used in this standard as part of the syntax descriptions to indicate the presence of signal expressions in statements.

## 6.15 WaveformChar characteristics

WaveformChar characters are used to assign waveform information to specific signals in Vector statements. Each signal's WaveformChar characters are defined in Timing blocks. A WaveformChar shall be a single alphanumeric character, from the set of characters: [0-9][a-z][A-Z].

A WaveformChar list is a list of WaveformChar characters that apply one for one with the corresponding signals of a SignalGroup. (For example, if a SignalGroup contains five Signals, then its corresponding WaveformChar list would contain five WaveformChar characters.) A WaveformChar list may contain whitespace,

including newlines. A complete assignment is built by reading the individual tokens into a complete assignment value. For example, the following assignments are equivalent:

```
group_of_10 = XXXXXXXXXXX; // 10 X's
group_of_10 = XXXXX
                XX X XX;    // 10 X's
```

A WaveformChar list may be encoded into a hexadecimal or decimal base for data compaction (see 15.4). The default base (WaveformChar, Hex, or Dec) is determined by each Signal or SignalGroup's definition. Vector flags (\w, \h, and \d) may be used to override the default base of assignment values. In addition, repeated character sequences may be compacted using the \r vector flag or changed in length by the \l (lower-case ell) (see 21.1). The following rules define the interaction of vector flags in a WaveformChar list:

- The last-specified base value (the default value, or \w, \h, or \d if the default was changed) is applied to a WaveformChar list until another base value is specified.
- The repeat flag \r is applied to a list until terminated by the end of the expression, or by the first whitespace after the \r and count fields.
- The length flag \l is applied to the subsequent WaveformChar list up to the end of the expression after expansion of any \r constructs in that list.

For example, the WaveformChar list “f \r2 f\w0000 0101” would expand into (assuming the default was Hex encoding to the WaveformChars “0” and “1”):

- The initial “f” represents the WaveformChar list “1111.”
- The \r2 sequence is applied to the whitespace-separated list “f\w0000.” This list represents the WaveformChars “11110000,” and after the repeat, is “1111000011110000.”
- The \w (changed in middle of the previous \r expression) is still in effect for the final segment. So this remains as “0101.”
- The entire segment would be equivalently represented as: “\w1111 1111000011110000 0101.”

Since the \r construct is terminated by whitespace, it is not possible to define a repeat construct around a list that specifies \h or \d options. For example, the list “\r2 \hwW f0” attempts to apply the WaveformChars ‘w’ and ‘W’ to the hex value f0. However, the whitespace required for \h also terminates the \r operation. This situation is resolved by putting the hex flag first as follows: “\hwW \r2 f0.”

WaveformChar lists are referred to in the syntax explanations as *vec\_data* or *serial\_data*.

## 6.16 STIL name spaces and name resolution

Information defined in a block with a domain name requires a reference to that domain name to use the information in that block. Information defined in a block without a domain name is available to be used without an explicit reference; this information is considered “global” after it has been defined.

All information in a named domain becomes available when that name is referenced. Because most STIL domain blocks contain only data definitions, there is no method to access a subset of domain information, except for Timing information, which may have several levels of hierarchy and scoping mechanisms. (See 6.11 for more information.) If a subset of information is desired, then the information may be partitioned into separate named blocks.

Table 6 identifies STIL blocks, the type of information contained in each block, and the function of the domain operation for that information.

**Table 6—STIL name spaces**

STIL block	Type of name	Domain restrictions
Signals	Signal and SignalGroup names <sup>a</sup>	Any signal defined in the Signal block is global.
SignalGroups	Signal and SignalGroup names <sup>b</sup>	Supports a single unnamed global block and domain name (restricted) blocks.
ScanStructures	Scan names and scanchain names	A single ScanStructures block is optionally named. Multiple ScanStructures blocks shall be uniquely named. ScanChain names shall be unique inside a ScanStructures block.
Procedures	Procedure names	Supports a single unnamed global block and domain name (restricted) blocks.
MacroDefs	Macro names	Supports a single unnamed global block and domain name (restricted) blocks.
Timing	WaveformTable names <sup>c</sup>	Supports a single unnamed global block and domain name (restricted) blocks.
Timing	Event_labels	Event_label names shall be unique with respect to Spec variables, but may be multiply declared if scoped within Timing information.
Spec	Spec names	There may be multiple Spec blocks present. The names of all Spec blocks shall be unique.
Spec	Spec variables, Spec categories	Any Spec variable or Spec category defined in any Spec block is global, irrespective of the presence of an explicit domain name on the Spec block.
Selector	Selector names	Each Selector block defines an entity. The domain name is required.
PatternBurst	Pattern/PatternBurst names	Each PatternBurst or Pattern block defines an entity. The domain name is required, and the name space of Pattern and PatternBurst blocks is shared.
Pattern	Pattern/PatternBurst names	
PatternExec	PatternExec names	Supports a single unnamed global block and domain name blocks.

<sup>a</sup>The Signals block defines only signals, but the name space is shared for both groups.

<sup>b</sup>The SignalGroups block defines only groups, but the name space is shared for both groups.

<sup>c</sup>Also contains hierarchical data items, discussed subsequently.

There are three environments for name spaces in STIL. The first supports both unnamed (global) definitions and domain named definitions. This environment is used for signals and groups (although the name space of signals is global only, it is combined with group names that support this environment), procedures, macros, and timing names. The second name space environment is global only, used for spec variables (and the signal name subset of signal and group names). The third environment is where the domain name itself serves as the name. This environment is used by the selector, patternburst, and pattern names.

The first environment requires mechanisms to resolve potential conflicts between unnamed (global) information and names declared under a domain name. These mechanisms are as follows:

- A name defined in a SignalGroups, Timing, Procedures, and MacroDefs block with a domain name for that block shall override any identical names of the same type defined from a block with no domain name, when that domain name is referenced.
- It is an error to refer to a name defined in two different (domain named) blocks, even if both definitions for the name are the same.
- Specifically for the SignalGroups information: it is an error to define a group name in a SignalGroups block with no domain name, with the same name as a name defined in a Signals block. Signal names may be overridden only from SignalGroups with domain names.
- For spec variables, it is an error to redefine a previously defined category for a spec variable.
- For selector, patternburst, and pattern name spaces: it is an error to redefine a previously defined name.
- It is an error to redefine a previously defined signal name, although a domain named SignalGroup may override a named signal. In the case that a domained SignalGroup changes the definition of a signal name, that new definition shall be reflected in all signal groups that use that name, including groups defined in the global SignalGroup block, when that domained SignalGroup is referenced.

## 7. Statement structure and organization of STIL information

There are two general forms of STIL statements: simple and block statements. Both forms start with a STIL keyword, followed by a number of tokens (depending on the statement). The simple statement is terminated by a semicolon. The block statement contains open and close braces; additional STIL statements may occur inside these braces. The statement forms are presented in Figure 30.

Simple statement:

**Keyword** (OPTIONAL\_TOKENS)\*;

Block statement:

**Keyword** (OPTIONAL\_TOKENS)\* { (OPTIONAL\_MORE\_STATEMENTS)\* }

**Figure 30—STIL statement structure**

The remaining clauses of this standard detail each STIL keyword, the type of STIL statement used with that keyword, and any STIL statements associated with that statement (for block statements).

Before each statement is presented, it is important to define the overall organization of data in a STIL environment. Subclauses 7.1 and 7.2 reference STIL keywords that are defined subsequently. For more information about these keywords, refer to Clause 8 through Clause 24.

## 7.1 Top-level statements and required ordering

STIL follows a “define before use” paradigm, with several exceptions discussed below. For example, the timing data is defined before it may be referenced, and a name used to reference a group of signals defines what signals it contains before it is used. Since all data is defined inside “top-level” blocks, these requirements are satisfied by properly ordering these top-level blocks.

“Top-level” blocks are blocks that occur outside the context of any other STIL statement. The first columns of Table 7 and Table 8 list all possible “top-level” blocks and statements in STIL.

Inside some top-level blocks are “reference” statements. Reference statements are used to access data defined in specific other top-level blocks. Reference statements use the domain name of a block being referenced as the referencing mechanism; generally, all data (of whatever type is defined in that block) may be used once a reference to that block has been defined. The exception to this is references to Timing data, which may include mechanisms to use specific subsets of previously-defined timing (such as a specific WaveformTable).

To satisfy the “define before use” constraint, a STIL block defining a type of information shall be present before the reference is made to that block. The order of the first column in Table 7 is one example of an order of blocks that satisfies this constraint.

It is not necessary to define all types of STIL data together. For instance, all the Timing blocks for a test do not need to be defined consecutively. However, it is necessary to define each particular Timing block before any references are made to that block from other blocks. Again, the order of data presented in Table 7 satisfies the most rigorous constraints of referencing in STIL. However, the only STIL requirement is that the data be defined before it is referenced.

There are several legal (and expected) exceptions to the “define before use” paradigm. These exceptions are:

- The `pattern_domain_name`, Procedures, and MacroDefs domain name references in the PatternBurst block. The `pattern_name` is always a forward reference, as Pattern blocks are always last in a STIL file; however, the Procedures and MacroDefs may or may not be forward references.
- Any SignalGroups or Timing data referenced in a Procedures or MacroDefs block (as these references cannot be resolved until the reference statements have been identified in the PatternBurst or PatternExec block).
- Any forward references to labels made in Pattern Goto statements.

Incomplete or unresolved variables or expression values are handled at run-time.

**Table 7—STIL top-level statements and ordering requirements**

Statement	Purpose
STIL	Defines the version of STIL present in the file. This is the first statement of any STIL file, including files opened from the Include statement.
Header	Contains general information about the STIL file being parsed. This block is optional; if present, it shall be the first statement after the STIL statement for a file.
Signals	Defines all primary signals under test.
SignalGroups	Defines collections of Signals. Requires reference to use if domain_name present.
ScanStructures	Defines internal scan chain information. The ScanStructures block or blocks are optional. If there are multiple ScanStructures blocks, they must be named. The PatternBurst may contain a reference to a named ScanStructures block, and the Pattern may contain a reference to a named ScanChain inside a ScanStructures. These blocks shall be defined before the PatternBurst if the PatternBurst contains references; otherwise, these blocks are defined before the Pattern blocks.
Spec	Defines values of variables to be applied in Timing Expressions. Multiple values may be assigned to variables; variable values are not resolved until the PatternExec statement. All Spec blocks shall precede the first Timing block definition.
Timing	Defines the waveforms to be applied to each signal in the test. Timing expressions in this block may reference variables defined in Spec blocks, but timing variables are not resolved until the PatternExec. Timing blocks may reference other timing blocks; those blocks shall be defined before they are referenced.
Selector	Selects min/typ/max/meas value of variables defined in Spec. Although this references variables defined in Spec blocks, variable values are not resolved until the PatternExec statement. This block shall precede the PatternExec block.
PatternBurst	Defines all Patterns to be executed collectively; any operation performed on this data (such as timing assignment) is performed on all Patterns referenced in this domain. The domain_name_pat reference, MacroDefs reference, and Procedures reference may all be forward references. Any references to SignalGroups shall have those blocks defined first. This block shall precede the PatternExec block.
PatternExec	Resolves timing variables and waveforms to apply with pattern references from PatternBursts. This block binds all information into a form to be applied to patterns as they are parsed.
Procedures	Defines a set of test data to be used multiple times in a Pattern; at the end of each execution, the state of the test before this call is restored for the next test vector. Procedure data may be processed after a PatternBurst referencing this block is parsed.
MacroDefs	Defines a set of test data to be used multiple times in a Pattern; at the end of each execution, the state of the test at the end of macro is in force. Macro data cannot be processed until Patterns are processed; pattern context may affect macro processing.
Pattern	Defines test data. All references shall be complete at the point pattern data is processed.

## 7.2 Optional top-level statements

There are several statement types in STIL that do not have ordering requirements as presented in 7.1. Some of these statements may appear any place a legal STIL statement may occur, including as top-level statements. These statements are listed in Table 8.

**Table 8—Optional top-level statements**

Statement	Purpose
Include	Opens the specified file for interpretation as a STIL file at the point the Include is parsed. At the end of the included file, parsing is resumed in the current file. This statement may appear any place a legal STIL statement may occur after the initial STIL (version) statement.
UserKeywords	Defines additional words as STIL keywords. The statements referencing these keywords shall be consistent with STIL statement formats, as presented in Figure 30. This statement shall appear at the top level only.
UserFunctions	Defines additional words as Timing Expression functions that are parsed and ignored. This statement shall appear at the top level only.
Ann	An annotation which is a preserved comment. This statement may appear any place a legal STIL statement may occur after the initial STIL (version) statement.

## 7.3 STIL files

Each STIL file is comprised of ASCII source statements. Optionally, any STIL file may be compressed using the GNU GZIP software, or GZIP-compatible software. Therefore, all readers shall be able to decompress a STIL file, if required, using the GNU GUNZIP or a compatible program. Readers shall incorporate the decompression in order to process compressed include files. See Annex C for information about obtaining and/or integrating the GUNZIP program into readers.

## 8. STIL statement

The STIL statement shall be the first statement of a STIL file. The version number refers to the revision of STIL that the writer is designed to support.

### 8.1 STIL syntax

```
STIL STIL_VERSION_NUMBER;
```

**STIL:** A statement at the beginning of each STIL file.

**STIL\_VERSION\_NUMBER:** The current version of STIL. This is used by a STIL writer to indicate the version of STIL for which it was written. The version contains a major and minor revision number separated by a period. This standard defines version 1.0.

### 8.2 STIL example

```
STIL 1.0;
```

## 9. Header block

The Header block may appear only once at the beginning of a STIL file, and is used to specify data that pertains to the creation of the file.

### 9.1 Header block syntax

```
Header {
  ( Title "TITLE_STRING"; )
  ( Date "DATE_STRING"; )
  ( Source "SOURCE_STRING"; )
  ( History {} )
}
```

**Header:** Start of the header block.

**Title:** String used to identify this STIL block or file.

**Date:** The date the file was generated. The format of the `date_string` is as defined for the `ctime()` and `asctime()` functions in the C programming language (see ISO/IEC 9899-1990), except the `date_string` contains no embedded newline (`\n`) or null (`\0`) characters and shall be enclosed in double-quotes to be processed as a single string.

**Source:** A special annotation used to indicate how and/or where the file was generated.

**History:** A block used to contain annotations as the history of the data in the file.

### 9.2 Header example

```
Header {
  Date "Tue Apr 28 12:23:48 EST 1996";
  Source "VHDL simulation on April 22, 1996";
  History {
    Ann { * rev1 - 4/21/96 - made some change * }
    Ann { * rev2 - 4/22/96 - made it work * } } }
```

## 10. Include statement

The Include statement allows reference to be made to an external file.

The Include statement may occur at any point at which a legal STIL statement may be defined after the initial STIL (version) statement.

If the included file contains specific information (e.g., a complete STIL section, like Signals), then the optional `IfNeed` clause may be used to indicate this. If this is indicated, and the type of information indicated to be present in this file is not required for current processing, then the STIL parser may choose to skip parsing this *entire file*. There is no required checking performed on the contents of the include file, or on whether the contents match the indicated block type. If the information in the include file contains a mix of blocks, or parse-optimization is not desired, then the block type should not be specified.

Compressed files from GNU GZIP have a “.gz” extension. The included file reference may have “.gz” as part of the specified file name; or if the specified file is not found, a STIL parser shall also look for that file name with a “.gz” extension.

## 10.1 Include statement syntax

**Include** “FILE\_NAME” (**IfNeed** <BLOCKTYPE>);

**Include:** Keyword for the file include statement.

**FILE\_NAME:** The name of file to be accessed. All file names are parsed as strings, with double quotes enclosing the name. The double-quotes are removed before the file is accessed, as specified from the current directory of the file that contains this include statement. This specified file is accessed from the current location of the file containing this include statement; path information, if necessary, is provided as part of the name. Be aware of the implications of absolute and relative path notations in the target operating environment, if STIL files are moved into different locations. If this file is not found, the extension “.gz” is added to the file name and a compressed format of the file is accessed.

**IfNeed:** An option to allow readers to override reading the file if the current processing environment may not need the specified blocktype. This directive may not check the file for each object; it skips processing the whole file if the specified blocktype is not needed for current processing. It is up to the generator or writer of STIL to make sure the specified information is present in the file, and no more.

**BLOCKTYPE:** The allowed block types are listed in Table 7 and Table 8.

## 10.2 Include example

```
Include "../all_timing.stil";  
Include "$DESIGN_STUFF/STIL_scan.stil" IfNeed ScanStructures;
```

## 10.3 File path resolution with absolute path notation

If an Include statement defines a file reference that is complete (i.e., is specified relative a top-level directory or contains a device name), then that name is passed directly into the operating system to locate that file. The STIL environment may not provide any additional mechanisms or search paths for file path resolution, nor should STIL make explicit use of any that may be present in the operating environment of the computer system. The name specified in the Include statement is passed directly into the operating system without any other manipulations, and the operating system resolves that name appropriately.

## 10.4 File path resolution with relative path notation

The one extension to this operation is that the context of the Include statement is taken, if necessary, relative to the current STIL file being parsed. This is important if STIL files are collected from different directories and “relative path” notation is used to reference those files. Once a STIL file is being parsed, any Include statements that use relative path notation use the location of that file being parsed as the starting point of the relative path. Be aware that once a directory is changed by including a file from a different directory, any additional includes in that file are processed relative to that new location until that file has been completely parsed.

## 11. UserKeywords statement

The UserKeywords statement allows for extensions to the STIL language. The specified keywords are added to the allowed set in the language. Any implementation of STIL should be written such that the extended keywords may be processed without error, even if they are not known to the reader. The UserKeywords statement appears at the top level of block statements. Any user-defined keywords shall be defined in a UserKeywords statement before the keyword is encountered in a STIL statement. Once defined, user keywords apply to all code that is processed following their occurrence. The UserKeywords statement may occur multiple times, and subsequent occurrences may repeat previous definitions. UserKeywords definitions add to the STIL reserved word name space. The intent or purpose of UserKeywords defined by a user shall be specified by the user.

STIL statements referencing user-defined keywords shall conform to STIL statement structure, but they may occur as either simple statements or block statements. These UserKeyword sections may occur any place a STIL statement may occur after the initial STIL (version) statement. Block statements may contain information not constrained to STIL statement rules; however, if the information contains braces, then all braces shall be matched inside the block in order to be properly ignored. User-defined keyword STIL statements may occur inside other STIL block statements. User-defined keyword sections shall be maintained through a STIL process as appropriate for that process. (In particular, a STIL output should contain any User-defined keyword sections that were present in the input.)

### 11.1 UserKeywords statement syntax

```
UserKeywords (USER_KEYWORDS)+;
```

**UserKeywords:** Keyword for the UserKeywords statement.

**USER\_KEYWORDS:** One or more names to be supported as STIL keywords.

### 11.2 UserKeywords example

```
UserKeywords tchn diepad;
```

## 12. UserFunctions statement

The UserFunctions statement allows for extensions to the STIL language. The specified functions are added to the allowed set of functions supported in timing expressions. Any implementation of STIL should be written such that the extended functions may be processed without syntax errors in timing expressions, even though the resulting expression cannot be evaluated. (Errors at this point are environment-dependent; if the expression is passed to an environment-specific expression processor, then the expression may be properly processed.) The UserFunctions statement appears at the top level. Any user-defined functions shall be defined in a UserFunctions statement before the function is encountered in a timing expression. Once defined, user functions apply to all code subsequently processed. The UserFunctions statement may occur multiple times, and subsequent occurrences may repeat previous definitions. UserFunctions definitions add to the set of known time expression operators, and as such affect the name space of time expression operators only.

## 12.1 UserFunctions statement syntax

```
UserFunctions (USER_KEYWORDS)+;
```

**UserFunctions:** Keyword for the UserFunctions statement.

USER\_KEYWORDS: One or more names to be supported as STIL timing expression functions.

## 12.2 UserFunctions example

```
UserFunctions abs_min abs_max;
```

## 13. Ann statement

Annotations are text strings that are maintained through a STIL process as appropriate for that process. (In particular, a STIL output shall contain any annotations that were present in the input.) Annotations may contain any desired user information or comments. The Ann statement may occur any place a STIL statement may occur after the initial STIL (version) statement. It may occur as a top-level statement or inside STIL block statements.

The Ann statement uses two-character delimiters to identify an annotation block. The Ann statement block starts after the token “{\*” and is terminated by the token “\*}”. These delimiters shall be separated with whitespace from the annotation text.

### 13.1 Annotations statement syntax

```
Ann {* ANNOTATION *}
```

**Ann:** Keyword for the annotation statement.

ANNOTATION: Text string of annotation.

### 13.2 Annotations example

```
Ann {* signals 1-10 are high-level inputs *}  
Ann {* all other signals are low levels *}
```

## 14. Signals block

The Signals block is used to define individual signal names. Only one Signals block is allowed in a STIL file set; any other Signal block parsed is ignored. This is to facilitate the collection of several separate STIL programs for a DUT into a complete test.

## 14.1 Signals block syntax

```

Signals {
  ( SIG_NAME < In | Out | InOut | Supply | Pseudo >; )*
  ( SIG_NAME < In | Out | InOut | Supply | Pseudo > {
    ( Termination < TerminateHigh | TerminateLow | TerminateOff | TerminateUnknown >; )
    ( DefaultState < U | D | Z | ForceUp | ForceDown | ForceOff >; )
    ( Base < Hex | Dec > WAVEFORM_CHARACTER_LIST ; )
    ( Alignment < MSB | LSB >; )
    ( ScanIn (DECIMAL_INTEGER) ; )
    ( ScanOut (DECIMAL_INTEGER) ; )
    ( DataBitCount DECIMAL_INTEGER ; )
  } )*
}

```

**Signals:** Start of block defining the device signal names.

SIG\_NAME: Name of a signal (see 6.10).

**In:** A signal that has only input data (i.e., input to the DUT).

**Out:** A signal that has only output data (i.e., output from the DUT).

**InOut:** A signal that is bidirectional.

**Supply:** A power supply signal (i.e., power or ground).

**Pseudo:** A signal that is not a primary DUT signal.

The next set of attributes are defined for both Signals and SignalGroups (see Clause 15). The attributes defined here apply to both environments.

**Termination:** This optional attribute defines the global default tester termination required for the signal. This global default may be overridden by an individual PatternBurst termination specification (See Clause 17 for more information.) The specified termination value should be provided by the tester in order to define the proper device test. This attribute, if present, shall be defined only once for a signal.

Tester termination may be assumed during test generation as a mechanism to eliminate noise and speed logic state transitions. In addition, for bidirectional signals, a known termination value may increase fault coverage and reduce the pattern size by simulating the terminated driver state as input to the receiver. When tester termination is assumed during test generation, the assumed termination needs to be applied to the test vectors in order for the Vectors to execute successfully. This termination value may also need to be provided in a resimulation environment, in order to properly simulate the device response.

**TerminateHigh:** Indicates that the signal was simulated with high impedance terminated to a 1. This requires testers to be programmed with the comparator load reference voltage set at the “high” voltage level. Test generation systems would expect “highs” in place of their expect float-state measures.

**TerminateLow:** Indicates that the signal was simulated with high impedance terminated to a 0. This requires testers to be programmed with the comparator load reference voltage set at the “low” voltage level. Test generation systems would expect “lows” in place of their expect float-state measures.

**TerminateOff:** Indicates that the signal was simulated with non-terminated float-state measures. This requires testers to be programmed with the comparator load reference voltage set at the “float-state” voltage level. Test generation systems would expect float-state measures.

**TerminateUnknown:** Indicates that the signal was simulated with unknown termination of float-state measures (receiver feedback is not simulated). This requires testers to be programmed with the comparator load reference voltage set at the “float-state” voltage level. Test generation systems would expect float-state measures from this situation.

**DefaultState:** This optional attribute defines a Drive Event value to apply to this signal if this signal is unused in a Pattern. The only valid Drive Events for the Default state are D, U, or Z (or their equivalent names). See Table 9 for a description of these events. This attribute, if present, shall be defined only once for a signal.

**Base:** An optional statement indicating the default base used for this signal or group when referenced in the Pattern block. The allowed bases are Hex and Dec. If no base is specified, the default base is WaveformChar. This attribute is meaningful only when defined on a signal with a ScanIn or ScanOut attribute, or on a multiple-bit signal definition or group.

**Hex:** A keyword indicating that the signal’s WaveformChar assignments shall be interpreted as hexadecimal encoding.

**Dec:** A keyword indicating that the signal's WaveformChar assignments shall be interpreted as decimal encoding.

**WAVEFORM\_CHARACTER\_LIST:** The set of WaveformChar characters valid for this signal, or any signal in a multiple-signal definition or group. Each WaveformChar character has a binary value associated to its position in the list (left to right, 0 to n-1). Assignments to the group consist of the binary value used to select the appropriate WaveformChar for each signal, expressed in either the Hexadecimal or Decimal base. (See 15.4 for details.)

**Alignment:** An optional statement indicating how to map the bits of a non-WaveformChar numeric value into the individual signals of a multiple-signal definition or group. This attribute is significant only for multiple-signal definitions or groups, and is applied only in the context of Vector assignments (see 21.1 and 21.2). It is not applied to scan environments (21.4), as scan relies on padding operations to resolve data length issues.

**LSB:** A keyword indicating to align the least significant (right most) bit (LSB) of the data with the right most signal in the group. This attribute only has an effect when the number of bits in the data is larger than the number of signals in the group.

**MSB:** A keyword indicating to align the most significant (left most) bit (MSB) of the data with the left most signal in the group. This attribute only has an effect when the number of bits in the data is larger than the number of signals in the group.

**ScanIn:** Identifies this signal or group as a scan input (implies that scan data is pre-padded during scan data normalization). Also optionally specifies the default length (number of scan WaveformChars) assumed in the scan data. The ScanIn statement is required if this signal is used to reference scan data (see 21.4).

**ScanOut:** Identifies this signal or group as a scan output (implies that scan data is post-padded during scan data normalization). Also optionally specifies the default length (number of scan WaveformChars) assumed in the scan data. The ScanOut statement is required if this signal is used to reference scan data (see 21.4).

**DataBitCount:** Indicates the number of bits of data required to complete waveforms containing multiple-bit data references. This statement is required whenever a reference is made to a waveform containing multiple-bit constructs. (See 21.2 for complete semantics.)

Default attribute values are defined in 15.3.

## 14.2 Signals block example

(NOTE—Signals named A0 and A[0] are both defined.)

```

Signals {
  DIR In;
  A0 InOut;
  A1 InOut;
  A2 InOut;
  A3 InOut;
  A[0..7] InOut;
  B[0..7] InOut;
}

```

## 15. SignalGroups block

The SignalGroups block is used to create named references to zero or more signals. A group may be empty. A group of one signal becomes a rename for that signal. A group name may be used anywhere that an individual signal name may be used. The group name may be assigned to a scan signal, in which case, the default base for the scan data may be specified.

Only one global SignalGroups block shall be allowed in STIL. A global SignalGroups block is a SignalGroups block with no DOMAIN\_NAME specified.

Any number of SignalGroups blocks with domain\_names are allowed. All domain\_names shall be unique across all SignalGroups. (A name in one domain may be the same as a name in another domain without conflicting. (See 6.16 for details on name conflict resolution.)

A SignalGroup domain to be used by a Pattern is defined in the PatternBurst block, thereby allowing the reuse of patterns. The statement to select a domain is “**SignalGroups** DOMAIN\_NAME;”.

### 15.1 SignalGroups block syntax

```

SignalGroups (DOMAIN_NAME) {
  ( GROUPNAME = sigref_expr; ) *
  ( GROUPNAME = sigref_expr {
    ( Termination < TerminateHigh | TerminateLow | TerminateOff | TerminateUnknown > ;
  )
  ( DefaultState < U | D | Z | ForceUp | ForceDown | ForceOff > ; )
  ( Base < Hex | Dec > WAVEFORM_CHARACTER_LIST ; )
  ( Alignment < MSB | LSB > ; )
  ( ScanIn (DECIMAL_INTEGER); )
  ( ScanOut (DECIMAL_INTEGER); )
  ( DataBitCount DECIMAL_INTEGER; )
  } ) *
}

```

**SignalGroups:** Start of block defining signal group names.

DOMAIN\_NAME: An optional name given to a group block allowing it to be referenced by a Timing Block, Pattern Block, or PatternBurst (see SignalGroups statement in Timing, Pattern, or PatternBurst block).

GROUPNAME: The name given to a group. (See naming rules in 6.10.)

*sigref\_expr*: An expression creating an ordered list of signals (see 6.14).

**Termination, TerminateHigh, TerminateLow, TerminateOff, TerminateUnknown, DefaultState, U, D, Z, ForceUp, ForceDown, ForceOff, Base, Hex, Dec, WAVEFORM\_CHARACTER\_LIST, Alignment, LSB, MSB, ScanIn, ScanOut, and DataBitCount** are defined in Clause 14.

## 15.2 SignalGroups block example

```
// Example of a global (unnamed) SignalGroups block:

SignalGroups {
    abus_pins   = 'A[0]+A[1]+A[2]+A[3]+A[4]+A[5]+A[6]+A[7]';
    bbus_pins   = 'B[0]+B[1]+B[2]+B[3]+B[4]+B[5]+B[6]+B[7]';
    bbus_odd    = 'bbus_pins-B[0]-B[2]-B[4]-B[6]';
    xbus        = 'x[0]+x[1]+x[2]+x[3]' { Base Hex wW; Alignment
    LSB; }
    scan_out    = 'scan1' { ScanOut 289; Base Hex LHX; Alignment
    MSB; }
}

// Example of a SignalGroups block for domain "quality."
// Same names declared as above, except the ellipsis operator is used:

SignalGroups quality {
    abus_pins   = 'A[0..7]';
    bbus_pins   = 'B[0..7]';
    xbus        = 'x[0..3]' { Base Hex wW; Alignment LSB; }
}
```

## 15.3 Default attribute values

When a new group is declared, it is assigned the following set of default property values. These values may be overridden only by explicit declaration of property values:

- Base WaveformChar;
- Alignment MSB;
- ScanIn is *not* present;
- ScanOut is *not* present;
- DataBitCount 0;
- Termination TerminateUnknown;
- DefaultState ForceOff.

A new group declared from a collection of previous groups shall redefine any property value to be used that is different than the default value. For instance, in the following example the combination of groups a and b defines the Base Hex environment that is applied across all bits of that group. Even though 'a' should never go to a 0 or 1, and 'b' should never go to an L, H, or X, the combination has to reserve space for those states.

```
SignalGroups {
    a='pin1' {Base Hex LHX;}
    b='pin2' {Base Hex 01;}
    "a&b" = 'a+b' { Base Hex LHX01;}
}
```

## 15.4 Translation of based data into WaveformChar characters

STIL supports representation of state assignment to signals using either individual WaveformChars or two “based” options, Hex or Decimal. Hex options are presented first.

If a signal only uses two states, a “high” and a “low” state, then any hex data mapping should obviously map those two states into a single bit of hex. But general test data often has additional states, an “x” state, and sometimes even a “z.” If hex is to be used with these states as well, somehow the hex base needs to know how many bits are required to determine what state it’s in.

The term “WaveformChar” as used in STIL refers to the single character used in the Vectors to reference a waveform definition. STIL allows the user to define what characters are used; the set is not fixed or constrained (beyond being a single alphanumeric character). So the simple application of a single “bit” of data says that the single character specified in the Vectors defines a direct mapping to a waveform reference; for example:

```
WaveformTable one {
    Waveforms {
        CLKS {
            wW { '0ns' D; '10ns' D/U; '20ns' D; }
        } } }
}
```

defines a high-going pulse if any given signal in the group CLKS is using a W in the Vectors, and defines no events (constant low) if a w is used.

The process of mapping *numeric* hex values to a signal with multiple waveforms defined from ASCII characters is explained next.

This is where the “explicit WaveformChar definition” comes into play. By defining a Base Hex wW in the groups, the STIL environment maps the derived hex values into these two waveforms, as in the group definition:

```
SignalGroups { hexd_CLKS = 'CLK1+CLK2+CLK3+CLK4' { Base Hex wW; } }
```

A subsequent reference to the group called “hexd\_CLKS” in the Vectors interprets the data following to be using a hex base (unless the data following has an explicit base defined) and, furthermore, maps that hex value into a single bit where 0=w and 1=W.

So the following Vector statements are equivalent:

```
V { hexd_CLKS=A; }
V { CLK1=W; CLK2=w; CLK3=W; CLK4=w; }
V { hexd_CLKS=\wWwWw; }
// (Note:\w indicates WaveformChars)/(WwWw)
```

Adding more references to WaveformChar characters causes less compaction but more waveform flexibility; for example:

```
SignalGroups { hexd_CLKS = 'CLK1+CLK2+CLK3+CLK4' { Base Hex wWX; } }
```

now requires two bits of hex data to hold the values to map for this set of WaveformChar characters, with a mapping of: 0=w, 1=W, and 2=X. (This would now be a “two-bit hex.”) Since each signal takes two bits of value, it now takes two hex characters to hold sufficient data to define the states on all signals. For instance, using this group definition, the following are equivalent:

```
V { hexd_CLKS=AA; }
V { CLK1=X; CLK2=X; CLK3=X; CLK4=X; }
V { hexd_CLKS=\wXXXX; }
```

One last issue remains with regard to the hex expansion, and that is what to do with undefined states. For example, in the case of three WaveformChar characters defined for hex, the first WaveformChar maps to the bits 00, the second to the bits 01, and the third to 10. 11 is unused. It is an error to assign this unused bit value to a signal.

Decimal operations proceed in exactly the same fashion, except a decimal value (rather than a hex value) is used to represent the bit settings. For example, if a group was defined as:

```
SignalGroups { decd_CLKS = 'CLK1+CLK2+CLK3+CLK4' { Base Dec wW; } }
```

the following Vector statement

```
V { decd_CLKS=10; }
```

would apply the same states to the signals as:

```
V { hexd_CLKS=A; }
```

## 16. PatternExec block

The PatternExec block is the “glue” that defines all of the pieces needed in order to execute patterns on a tester. It defines the Category names to be used to resolve spec variables, the selector names to indicate which value (Min, Typ, Max, or Meas) of the spec variables to apply, which Timing block to find the WaveformTable references under, and what PatternBurst to use.

Only one global PatternExec block shall be allowed in STIL. A global PatternExec block is a PatternExec block with no domain name specified. Any number of PatternExec blocks with domain\_names are allowed. All domain\_names shall be unique across all PatternExecs.

If the Timing block referenced contains spec variables that have multiple categories, then one or more Category statements shall be specified in the PatternExec block. If the Timing block references spec variables that contain multiple values (i.e., Min, Typ, or Max values), the variables shall be specified in an unambiguous manner, either by resolving which value to apply via a Selector block, or qualifying the variable name in the reference (for example, ‘var.Min’). The named Timing block shall resolve all WaveformTable names that are referenced in all Pattern blocks that are referenced.

## 16.1 PatternExec block syntax

```

PatternExec (PAT_EXEC_NAME) {
    ( Category CATEGORY_NAME ; ) *
    ( Selector SELECTOR_NAME ; ) *
    ( Timing TIMING_NAME ; )
    ( PatternBurst PAT_BURST_NAME ; )
}

```

**PatternExec**: Start of block defining pattern execution.

**Category** CATEGORY\_NAME: Selection of a category, which defines the values of spec variable to be used for this pattern execution.

**Selector** SELECTOR\_NAME: Selection of a selector which defines the Min, Typ, Max, or Meas values to be used for each spec variable that is referenced.

**Timing** TIMING\_NAME: Selection of the Timing block that is to be used to resolve all WaveformTable references that appear in the Pattern blocks. If a named Timing block is not referenced with this statement (if this statement is not present), then the timing shall be found in an unnamed Timing block.

**PatternBurst** PAT\_BURST\_NAME: A reference to a named PatternBurst block. This statement may not be present for contexts that are passing Timing information only.

## 16.2 PatternExec block example

```

PatternExec maintest_fast {
    Category fast;
    Selector typel;
    Timing simple_wave;
    PatternBurst one_functional;
}

```

## 17. PatternBurst block

The PatternBurst block is used to specify a list of patterns that are to be executed in a single execution. Refer to the PatternExec block for the definition of the other necessary information to fully specify the timing and waveform information needed to support a pattern burst.

The PatternBurst block contains references to Pattern blocks that appear later in the file. It may also contain references to Procedures or MacroDefs that have not yet been defined. (See 7.1 for more details.)

A PatternBurst block may reference another PatternBurst block. These references shall be defined before they are referenced. This eliminates potential recursion in PatternBurst definitions, and facilitates identification of PatternBurst or Pattern names.

The PatternBurst block shall have a domain name. This name-space is shared with the Pattern names. The set of names across both PatternBurst and Pattern blocks shall be unique.

## 17.1 PatternBurst block syntax

```

PatternBurst PAT_BURST_NAME {
    ( SignalGroups GROUPS_DOMAIN ; ) *
    ( MacroDefs MACROS_DOMAIN ; ) *
    ( Procedures PROCEDURES_DOMAIN ; ) *
    ( ScanStructures SCAN_NAME ; ) *
    ( Start PAT_LABEL ; )
    ( Stop PAT_LABEL ; )
    ( Termination { ( sigref_expr
        < TerminateHigh | TerminateLow | TerminateOff | TerminateUnknown > ; ) * } ) *
    ( PatList {
        ( PAT_NAME_OR_BURST_NAME ; ) *
        ( PAT_NAME_OR_BURST_NAME {
            ( SignalGroups GROUPS_DOMAIN ; ) *
            ( MacroDefs MACROS_DOMAIN ; ) *
            ( Procedures PROCEDURES_DOMAIN ; ) *
            ( ScanStructures SCAN_NAME ; ) *
            ( Start PAT_LABEL ; )
            ( Stop PAT_LABEL ; )
            ( Termination { ( sigref_expr
                < TerminateHigh | TerminateLow | TerminateOff | TerminateUnknown > ; ) * } ) *
            } ) *
        } ) +
    }
    // end of Pat_name_or_Burst_name
    // end of PatList
    // end of PatternBurst

```

**PatternBurst** PAT\_BURST\_NAME: Start of a block defining a list of Pattern blocks that are to be run as a continuous burst.

PAT\_NAME\_OR\_BURST\_NAME: Name of a Pattern block or PatternBurst block to execute.

**SignalGroups** GROUPS\_DOMAIN: Optional statement indicating the name of a SignalGroups block that is to be used in resolving Pattern signal group references. When this statement occurs external to the PatList, all Patterns specified by the PatList shall use this SignalGroups block in resolving their signal group references. When this statement is used for a specific pat\_name\_or\_burst\_name, then only those Patterns shall use this SignalGroups block in resolving signal group references.

**MacroDefs** MACROS\_DOMAIN: Optional statement indicating the name of a MacroDefs block that is to be used in resolving Pattern macro references. When this statement occurs external to the PatList, all Patterns specified by the PatList shall use this MacroDefs block in resolving their macro references. When this statement is used for a specific pat\_name\_or\_burst\_name, then only those Patterns shall use this MacroDefs block in resolving macro references.

**Procedures** PROCEDURES\_DOMAIN: Optional statement indicating the name of a Procedures block that is to be used in resolving Pattern procedure references. When this statement occurs external to the PatList, all Patterns specified by the PatList shall use this Procedures block in resolving their procedure references. When this statement is used for a specific PAT\_NAME\_OR\_BURST\_NAME, then only those Patterns shall use this Procedures block in resolving procedure references.

**ScanStructures** SCAN\_NAME: Optional statement indicating the name of the ScanStructures block that contains ScanChain definitions applied for this PatternBurst. When this statement occurs external to the PatList, all Patterns specified by the PatList shall use these ScanStructures references. When this statement is used for a specific PAT\_NAME\_OR\_BURST\_NAME, then only those Patterns shall use this ScanStructures block in resolving references.

**Start** PAT\_LABEL: Optional statement indicating to start execution at the vector following the named label within the Pattern. When the statement is used external to the PatList block, the PAT\_LABEL shall be unique in the set of Patterns referenced in the PatList, and this location defines the start of this PatternBurst. Only one Start statement external to the PatList block may be specified in a hierarchy of PatternBursts (additional PatternBursts being referenced with the PatList statement). When the statement is used in reference with a single name inside a PatList, PAT\_LABEL needs to be unique only inside that name, and this start is applied only to this specific block's execution. Executing a start statement from a PatternBurst to a label on or inside a Loop or MatchLoop statement results in undefined behavior.

**Stop** PAT\_LABEL: Optional statement indicating to stop execution after applying the Vector following the named label within the Pattern. When the statement is used external to the PatList block, the PAT\_LABEL shall be unique in the set of Patterns referenced in the PatList, and this location defines the stop of this PatternBurst. Only one Stop statement external to the PatList block may be specified in a hierarchy of PatternBursts (additional PatternBursts being referenced with the PatList statement). When the statement is used in reference with a single name inside a PatList, PAT\_LABEL needs to be unique only inside that Pattern, and this stop is applied only to this specific block's execution. Executing a stop statement from a PatternBurst to a label on or inside a Loop or MatchLoop statement results in undefined behavior.

**Termination:** Defines tester termination required for signals in the Pattern. PatternBurst terminations, if present, shall override any signal terminations defined in the Signals block. Termination information, if present, shall be specified only once for a signal for a PatternBurst. (See Clause 14 for more information on the use of the Termination statement.)

*sigref\_expr* is a reference to a signal expression (see 6.14).

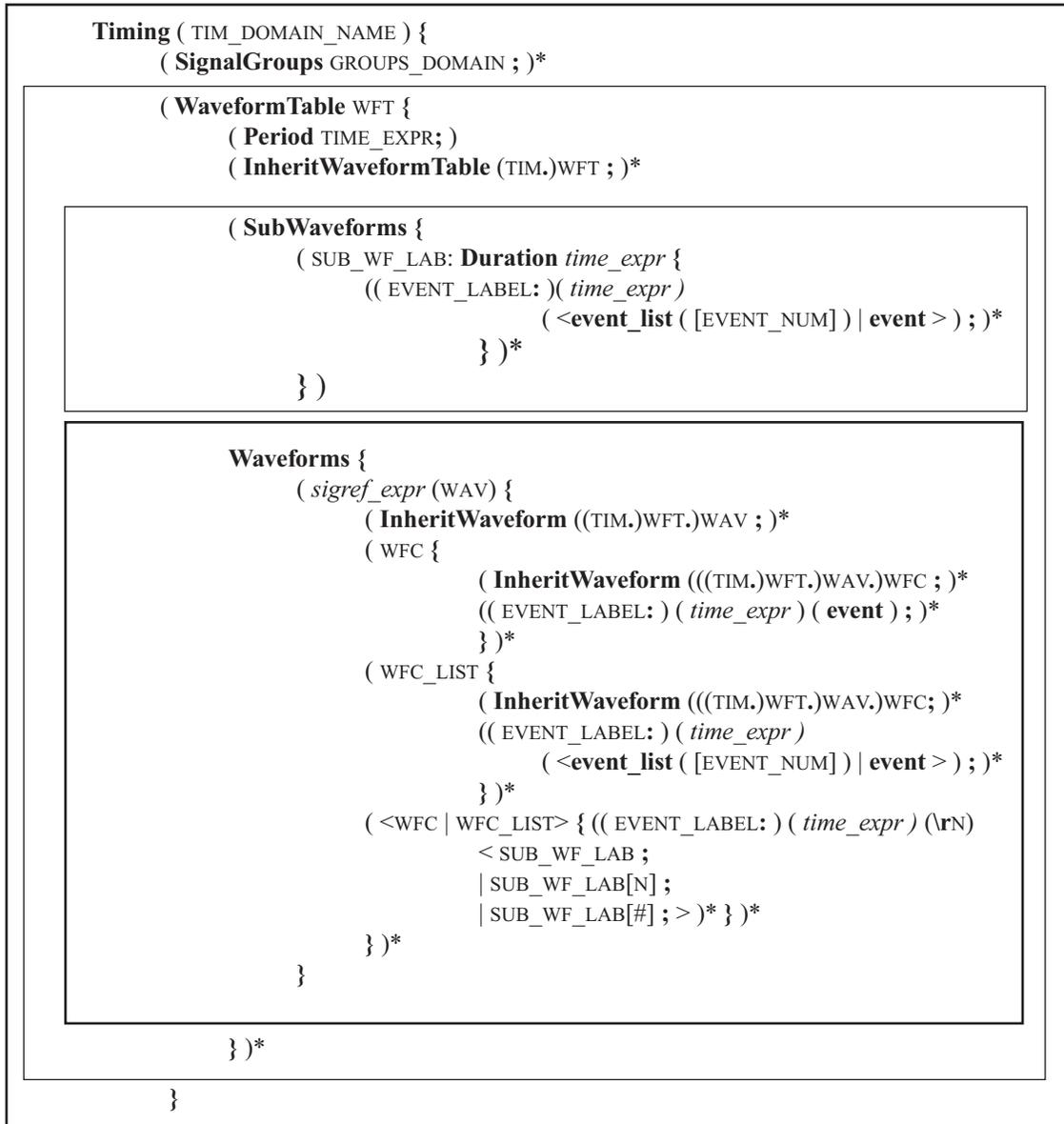
## 17.2 PatternBurst block example

```
PatternBurst one_functional {
    PatList { write_vecs {Start xbegin; } // 'start' for write_vecs only
            read_vecs {Start xend; } // 'start' for read_vecs only
    }
}
```

## 18. Timing block and WaveformTable block

The Timing block defines the placement of the timing edges, and the format of the cyclized waveforms that is referenced by applying WaveformChars to signals in the Vector statements. This block is divided into four parts, as shown by the four boxes around the syntax statements in the next section. The first part is the Timing block itself, and contains global information followed by definitions of the waveform tables. The second part is the Waveform Table block, which names the waveform table and contains common information with regard to all of the contained waveforms. The third section is the SubWaveforms block, which defines partial waveforms to facilitate representation of repetitive or shared timing. The fourth section is the Waveforms block, where event and time information is used to define the shape of the waveform for each signal or group of signals.

## 18.1 Timing and WaveformTable syntax



**Timing:** Start of a Timing bloc, which is a collection of WaveformTables. For a given pattern execution (refer to PatternExec), all WaveformTable references in the PatternBurst shall be resolved in a Timing block. The WaveformTable names shall be locally defined. The waveform definitions shall be either defined locally, or inherited from another waveform table.

TIM\_DOMAIN\_NAME: Optional name of a timing block. If no name is present, then the timing defined in this block is applied to any block without references to named timing.

**SignalGroups** GROUPS\_DOMAIN: This statement is optional. If used, it indicates a named set of groups to be used to resolve group references in the WaveformTables defined in this block.

**WaveformTable** (WFT): This block defines a set of waveforms for each of the signals. The name of the block (WFT in this syntax example) is the name that is used in a Pattern to reference the waveform, or it is used in other WaveformTables for data inheritance.

**Period:** This statement specifies the time duration of the waveform being defined. Events may occur outside of this time duration, as defined in 18.4. The next waveform cycle begins after the period value of the current cycle has expired.

Normally, this statement is required; however, there are two situations where it is optional. One case is where the InheritWaveformTable statement calls for inheritance from a WaveformTable that has the Period defined. The second case is where the waveforms within this WaveformTable are designed to be inherited by another WaveformTable that has the Period defined. In either case, the Period value shall be resolved if the WaveformTable is referenced within any Pattern.

**InheritWaveformTable** (TIM.)WFT: This statement is an optional statement. It is used to identify other waveforms and the Period (if present) as a basis for building new waveforms (i.e., data inheritance). If there are multiple InheritWaveformTable statements, the last waveform definitions for each signal are the initial definitions for each signal. Local waveform statements override the inherited waveform; the last inherited waveform definition is applied to any signal without a local waveform defined. The local Period value, if present, overrides any inherited Period value.

**SubWaveforms:** This is the block where waveform segments are defined. These segments are referenced in the Waveforms section as single or repetitive sections, to define a complete waveform assigned to a signal. The SubWaveform section is optional, but if present it shall only appear once. Because the SubWaveform block contains statements that are referenced in the Waveforms block, if present this section shall appear before the Waveforms block.

SUB\_WF\_LAB: **Duration** *time\_expr*: Each SubWaveform is defined with a Duration statement. The SubWaveform label SUB\_WF\_LAB is a required part of this statement, because the only reference to a SubWaveform in the Waveforms block is through this label. The *time\_expr* following the Duration keyword specifies the length of this SubWaveform; this value is used when the SubWaveform is referenced as a repeating construct in a Waveform. The remainder of this statement follows the semantics defined in the Waveforms block below, with the modification that any reference to WFC or WFC\_LIST is deferred until the SubWaveform is referenced in a Waveform statement.

**Waveforms:** This is the block where waveforms are defined. Only one Waveforms block shall appear in a WaveformTable block. Each waveform is identified by the signals that are being defined and an optional label. The optional label (WAV) is used if this waveform is to be inherited by another one.

*sigref\_expr*: This is defined in 6.14. A particular signal may be referenced in several waveform statements. Each waveform definition shall have a different set of WaveformChars defined for the signal(s) referenced.

**InheritWaveform** (((TIM.)WFT.)WAV.)WFC): This statement is optional and is used to define data inheritance from another Waveform. “TIM.” is optional and is used if the waveform to be inherited is in another Timing block. “WFT.” is optional and is used if the waveform to be inherited is in another WaveformTable block. “WAV.” is the label on the waveform that is to be inherited. If the InheritWaveform statement is outside of a particular WaveformChar definition, then all WaveformChars defined for that signal or group are inherited. If the InheritWaveform statement is within the WaveformChar definition, then only the definitions for those waveform characters are inherited. In this last case, a further option of specifying the WFC exists. It is an error to inherit the same WFC reference for a complete waveform from multiple InheritWaveform statements. (Incomplete WFC definitions may be inherited as shown in 18.5.) A local definition of a WFC overrides any inherited definitions.

**WFC:** The WaveformChar is a single character. The block of information following this character defines a waveform to be applied when this character is referenced in a Vector block.

**WFC\_LIST:** A list of WaveformChar characters that are all represented by the set of waveform events. If a WFC\_LIST is used, then at least one of the events in the waveform should be an event\_list, and the number of characters in each should be equal. A WaveformChar List may apply to a single event or to a list of events. For a single event, each WaveformChar in the list selects the event. For a list of events, each WaveformChar in the list selects a corresponding event from the event list (i.e., the length of the event list shall be either 1 or the length of the WaveformChar list).

**time\_expr:** A timing expression shall precede each event in a waveform to specify where in relation to the beginning of the cycle the event is to take place. There are limitations on the sequencing of events and cycles in time as outlined below. (See 6.13 for details on these expressions.)

**event:** An event is an identifier that is used to signal a potential change in the state of the signal. Whether or not the actual state changes depends on whether the event causes a change from the previous state. The events may be classified as input or drive events, and output or compare events. All events have fixed definitions as specified in 18.2.

**event\_list:** A list of events used when representing waveforms that are similar in time, but where the actual event (e.g., whether it goes Up or Down) is selectable by the WaveformChar character from the Vector block. An event\_list is used in conjunction with a WFC\_LIST. The event characters are defined in 18.2. All events in the event\_list shall be separated by a '/' character.

**event\_list[EVENT\_NUM]:** This form of event list functions the same as the previous event\_list, with the added feature that multiple data values may be specified in the invoking Vector block. The "EVENT\_NUM" represents an index starting from 0, which selects which of the Vector data values to use. (See 5.7 for an example of this construct.) The event characters are defined in 18.2. All events in the event\_list shall be separated by a '/' character.

**EVENT\_LABEL:** This is an optional label that may be used on an event or event list. It serves to give a name to the time of this event that may be used in following timing expressions. The scope of this label is local to the current WaveformTable, or any WaveformTable that inherits this event label. EVENT\_LABELS in the Sub-Waveform block, while useful for documentation, may not be referenced in other timing expressions, as the SubWaveform timing is relative to the usage of the SubWaveform.

**SUB\_WF\_LAB:** This is a reference to a SubWaveform defined in the SubWaveforms block. The events of the referenced SubWaveform are integrated into this Waveform definition.

**\rN:** Repeats (iterates) the next SUB\_WF\_LAB N times, where N is a positive integer. The iteration process increments all time values in the next iteration of the referenced SubWaveform by the Duration of the SubWaveform.

**SUB\_WF\_LAB[N]:** This is a reference to a SubWaveform defined in the SubWaveforms block. The events of the referenced SubWaveform are integrated into this Waveform definition. This construct may be used when the SubWaveform definition contains [EVENT\_NUM] constructs; the value of all [EVENT\_NUM] constructs in the SubWaveform are replaced with the value N. This allows the SubWaveform with [EVENT\_NUM] constructs to be reused in a context where all [EVENT\_NUM] constructs defined in the SubWaveform are constant, given the instance of the SubWaveform in the Waveform.

**SUB\_WF\_LAB[#]:** This is a reference to a SubWaveform defined in the SubWaveforms block. The events of the referenced SubWaveform are integrated into this Waveform definition. This construct may be used when the SubWaveform definition contains [EVENT\_NUM] constructs and the \rN iteration operator is used; the value of all [EVENT\_NUM] constructs in the SubWaveform is replaced with the current iteration count.

This allows the SubWaveform with [EVENT\_NUM] constructs to be reused in a context where the [EVENT\_NUM] values are incremented relative to the current iteration. All [EVENT\_NUM] constructs receive the same value in a single iteration.

## 18.2 Waveform event definitions

Table 9 through Table 12 define the events that may be used in the construction of a waveform. These events (unlike WaveformChar characters in a Vector) are fixed and cannot be redefined by the user. See 18.4 for an explanation of the use of these events in creating a waveform.

**Table 9—Drive events**

	Identifier	Icon	Definition
D	ForceDown		Force logic low. Drive to the low voltage level (VIL). If the driver was previously in the off (Z state), then turn the driver on and drive low.
U	ForceUp		Force logic high. Drive to the high voltage level (VIH). If the driver was previously in the off (Z state), then turn the driver on and drive high.
Z	ForceOff		Force logic high impedance. Turn the driver off. Note: The current U/D state is restored if the next drive state is "P."
P	ForcePrior		Force logic to last driven state. Turn the driver on and go to the last drive state (i.e., If the last drive state was "D," then go to low; if the last drive state was "U" then go to high).

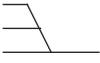
**Table 10—Compare events**

	Identifier	Icon	Definition
L	CompareLow	↓	Compare logic low (edge). Compare for a voltage level lower than the low voltage threshold (VOL).
H	CompareHigh	↑	Compare logic high (edge). Compare for a voltage level higher than the high voltage threshold (VOH).
X x	CompareUnknown		Compare logic unknown. Don't compare. This event is used to terminate any window compare state (X and x may be used interchangeably).
T	CompareOff	∨ ∧	Compare logic high impedance (edge). Compare for a voltage level between the low voltage threshold (VOL) and the high voltage threshold (VOH).

**Table 10—Compare events (continued)**

	Identifier	Icon	Definition
V	CompareValid		Compare logic valid level (i.e., not high impedance) (edge). Compare for a voltage level either lower than the low voltage threshold (VOL) or higher than the high voltage threshold (VOH).
l	CompareLowWindow		Compare logic low (window). Terminated by CompareUnknown.
h	CompareHighWindow		Compare logic high (window). Terminated by CompareUnknown.
t	CompareOffWindow		Compare logic high impedance (window). Terminated by CompareUnknown.
v	CompareValidWindow		Compare logic valid level (i.e., not high impedance) (window). Terminated by CompareUnknown.

**Table 11—Expect events**

	Identifier	Icon	Definition
R	ExpectLow		Expect logic low. The DUT output is expected to go to the low state.
G	ExpectHigh		Expect logic high. The DUT output is expected to go to the high state.
Q	ExpectOff		Expect logic high impedance. The DUT output is expected to turn off.
M	Marker		A marker event is any arbitrary point in time. No DUT activity or tester activity is expected.

**Table 12—Unresolved events**

	Identifier	Icon	Definition
N	ForceUnknown		Force logic unknown. The driver is turned on, but the Up/Down state of the driver is not defined. This event would map into a ForceUp or ForceDown once the logic state of the data is determined.
A	LogicLow		Unknown direction, logic low. The driver is in an unknown direction, but the logic level is defined as low. This event maps into a ForceDown or a CompareLow event once the state of the driver is determined.
B	LogicHigh		Unknown direction, logic high. The driver is in an unknown direction, but the logic level is defined as high. This event maps into a ForceUp or a CompareHigh event once the state of the driver is determined.
F	LogicZ		Unknown direction, logic high impedance. As in the case of the LogicLow and LogicHigh, the state of the driver is not known. This event would map into either a ForceUnknown or a ForceOff event once the state of the driver is determined.
?	Unknown		Unknown direction, logic unknown. Nothing is known about this event. The drive state, compare relevance, and logic level are all unknown.

### 18.3 Timing and WaveformTable example

```

Timing {
  WaveformTable defaults {
    Waveforms {
      allpins { xX { TMARK: 't_anchor' Z; } }
    }
  }
  WaveformTable wft1 {
    Period 'tper' ;
    InheritWaveformTable defaults;
    Waveforms {
      A7M { xX { 'TMARK+tx' Z; } }
      A7 { xX { 'TMARK+tx' Z; } }
      DIR { 01 { 'TMARK+tic1' D/U; } }
      OE_ { rR { 'TMARK+tx' P; '@+tic2' D/U; '@+tic6' U; } }
      A7 {
        rR { 'TMARK+tic1' P;
          '@+tic6' D/U;
          '@+tic4' U;
          'TMARK+tic6+100ns' Z;
        }
        lh { 'TMARK' Z; 'TMARK+tic5' L/H; 'TMARK+tic9+50ns'T }
      }
      bbus_pins bbus_aliases {
        lh { 'TMARK' Z; '@+tic5' L/H; '@1+tic9+50ns' T; }
        fF { 'TMARK' Z; '@+tic2' t; '@+5ns' x; }
      }
    }
  }
}

```

```
WaveformTable misc {
  Waveforms {
    some_pins {
      InheritWaveform wft1.bbus_aliases;
      xy { '0ns' D; '10ns' D/U; }
    }
    some_more_pins {
      ab { InheritWaveform wft1.bbus_aliases.lh; }
    }
  }
}
```

## 18.4 Rules for timed event ordering and waveform creation

In a WaveformTable, a waveform is described by means of a list of event characters, each of which has a time association and is used to specify a change in either the drive waveform (input to the DUT) or the compare state (output from the DUT).

There are four categories of events: drive, compare, expect, and unresolved events. Each category is processed as a separate stream of information, independently from events present for other categories. Unresolved events are a consequence of partial waveform generation, in which case it may not be possible to form the waveform without first applying some kind of pre-processing rules to turn these events into one of the drive or compare events. Expect events reflect information about DUT response that may differ in timing from compare events. Figure 31 is an example of Waveforms defined on two signals, and the resulting waveforms generated from Vector data.

```
STIL 1.0;
Signals { CLK In;  SIG InOut; }
Timing { WaveformTable wft { Period '40ns';
  Waveforms {
    CLK { 1 { '15ns' U; '25ns' D; } }
    SIG { 01 { '10ns' D/U; '20ns' D;
'30ns' Z; } }
    SIG { LH { '15ns' l/h; '30ns' x; } }
  }
} }
```

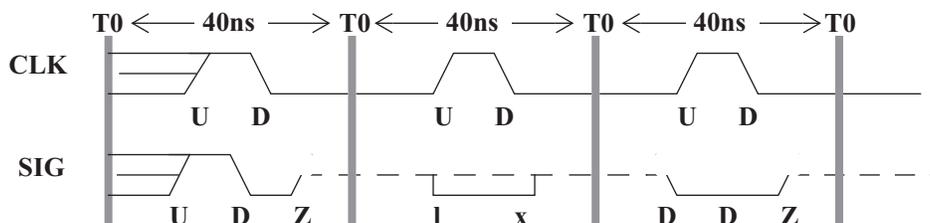


Figure 31—Creating waveforms from events

Each Vector references a WaveformTable that defines the period value and the waveform to be applied to each signal of the DUT. The rules by which the individual waveforms are connected are as follows:

- a) The start of the cycle is referred to as T0. An event may be specified at '0ns', in which case it occurs at T0.
- b) The Period statement establishes the end of the cycle, and hence the beginning (T0) for the subsequent vector.
- c) On each signal, all events present for each collection of event type (drive, compare, expect, and unresolved) shall occur in time in the same sequence that they appear in the Waveform definition. Events of any collection may extend beyond the end of the cycle, as long as the last event of that collection occurs prior to the first event of that collection in the next cycle. Events of any collection may be specified prior to T0 (as a negative value), as long as the first event (most negative value) of that collection occurs after the last event of that collection in the previous cycle. Events between different collections do not need to be in time sequence, and may be combined in any fashion as long as the events in a single collection are not out of time sequence.
- d) The time for any given event may be a constant or an expression. The pertinent category's event order shall be maintained for all applied values of the time expression.
- e) At the point that a waveform is applied in a Vector, all statements in that waveform shall define timed events, consisting of both an event and a time value. (This requirement is in consideration of partially defined waveforms, as described in 18.5.)

Figure 32 details an example of events extending into the next cycle.

```

STIL 1.0;
Signals {
    CLK In;
    SIG1 InOut;
    SIG2 InOut;
}
SignalGroups {
    sigs = 'SIG1+SIG2';
}
Timing {
    WaveformTable wf1 {
        Period '40ns';
        Waveforms {
            CLK { 1 { '0ns' U; '10ns' D; } }
            sigs { 01 { '5ns' P; '15ns' D/U; '35ns' D; '50ns' Z; } }
        }
    }
    WaveformTable wf2 {
        Period '50ns';
        Waveforms {
            CLK { 1 { '0ns' U; '10ns' D; } }
            sigs { LH { '35ns' l/h; '45ns' x; } }
        }
    }
}
Pattern {
    W wf1; V { sigs=11; CLK=1; }
    W wf2; V { sigs=LH; }
    W wf1; V { sigs=01; }
}

```

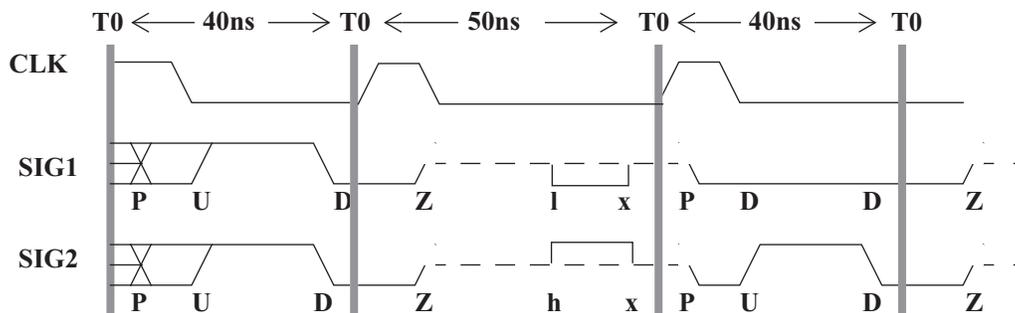


Figure 32—Waveforms extending into the next cycle

## 18.5 Rules for waveform inheritance

STIL supports two mechanisms to re-use previously defined waveform information: the `InheritWaveformTable`, and the `InheritWaveform` statements. The `Inherit` constructs allow data to be incorporated into a new definition, with modifications. Inherited information is applied at the waveform statement level; `WaveformChar` definitions per signal may be modified.

Waveform inheritance has two different means of combining information, depending on the type of waveform statement being inherited. These two types of inherited information shall not be mixed in a single waveform statement.

The first type of inheritance is a complete waveform definition. A complete waveform is defined as one that has both time expressions and events for all statements in the waveform. For a referenced signal, the complete waveform description defined for a `WaveformChar` shall be applied unless that `WaveformChar` is redefined for that signal, by defining another waveform definition for that `WaveformChar` for that signal. Any new definition shall completely replace an inherited definition.

The second type of inheritance is applied to partially defined waveform definitions. A waveform statement may be defined with only time expressions present for each statement, or only event identifiers present for each statement. These statements are combined, through inheritance, to create a complete waveform statement containing “timed event” statements. This process allows waveform “shapes” to be defined independently of timing, as shown in the example below. Waveforms defined this way shall have an equal number of statements in the waveforms defining the same `WaveformChar` for a signal; and statements are combined in a linear fashion across the two definitions and are independent of event-type. Waveforms are combined per signal, based on the common partial definitions for each `WaveformChar` present.

```
WaveformTable shape {
    Waveforms {
        allpins "allpins_shapes" { 01 { ForcePrior; ForceDown/ForceUp;
ForceDown }}
        clock "C_shapes" { C { ForceUp; ForceDown }}
    }
}
WaveformTable ts1 {
    Period '40ns';
    InheritWaveformTable shape;
    Waveforms {
        allpins "allpins_events" { 01 { '0ns'; '25ns'; '35ns'; }}
        clock "C_events" { C { '20ns'; '30ns'; }}
    }
}
WaveformTable ts2 {
    Period '40ns';
    InheritWaveformTable shape;
    Waveforms {
        group_a_pins "a_pins_events" { 01 { '0ns'; '25ns'; '35ns'; }}
        //first half of allpins
        group_b_pins "b_pins_events" { 01 { '5ns'; '30ns'; '40ns'; }}
        //second half of allpins
        clock "C_events" { C { '20ns'; '30ns'; }}
    }
}
```

## 19. Spec and Selector blocks

Spec blocks are used to define the value of the variables and expressions that are used within the waveform definitions. Each Spec block may contain Category blocks (which contain spec variable definitions), or Variable definitions directly.

There may be multiple Spec blocks present. Each Spec block shall have a domain name, and each Spec block name shall be unique across all Spec blocks. Spec blocks are never referenced directly; the information contained in Spec blocks is globally accessible. The defined spec variables are resolved through Category and Selector references.

Each individual variable name may:

- a) Be assigned a single value which defaults to Typ; or
- b) Be assigned one to three values (called Min, Typ, and Max values); or
- c) Be assigned to be a measured value called Meas.

The Min, Typ, Max, or Meas values are selected by means of the Selector block. The provision for defining and selecting values allows spec sheet parameters to be represented. A timing expression may specifically select one of the values by specifying *var-name*.<Min|Typ|Max|Meas>.

The fourth variable name qualifier, “Meas,” is a facility to allow the resulting set of timing spec values to be based upon a measurement that is made at run time. The mechanics of making the measurement and notifying the run time system that manages the expression is not defined as part of STIL.

The Selector block is used to specify for each variable name whether to use the Min, Typ, Max, or Meas values. Refer to Clause 17 to see how the Selector and Category names are selected for a given pattern execution.

There may be multiple Selector blocks present. Each Selector block shall have a domain name, and each Selector block name shall be unique across all Selector blocks.

Spec variable values shall be defined either by variable name within a Category block, or by category name within a Variable block. Any given variable-category value shall be defined only once. In general, a set of spec values would be organized by using either the Category form of the syntax or the Variable form of the syntax only.

### 19.1 Spec and Selector block syntax

```

Spec (SPEC_NAME) {           // this block statement defines variable values for a given category
  ( Category CAT_NAME {
    ( VAR_NAME = time_expr; )*           // define only the Typ value
    ( VAR_NAME { (Min time_expr; ) (Typ time_expr; ) (Max time_expr; ) } )*
  } )+
}

```

```

Spec (SPEC_NAME) {           // this block statement defines category values for a given variable
    ( Variable VAR_NAME {
        ( CAT_NAME = time_expr; )*           // define only the Typ value
        ( CAT_NAME { (Min time_expr;) (Typ time_expr;) (Max time_expr;) } )*
        } )+
    }

Selector SELECTOR_NAME {
    ( VAR_NAME < Min | Typ | Max | Meas >; )*
}

```

**Spec:** Start of a block defining spec variables.

**SPEC\_NAME:** The name of the spec table. This name is for reference only. It is not used in any subsequent references. All defined spec\_names shall be unique.

**Category** CAT\_NAME: Start of the definitions for a list of variables within the named category. This name is used in the PatternExec to identify a category of values for variables to use.

**Variable** VAR\_NAME: Start of the definitions for a list of categories for the named variable.

CAT\_NAME: Name of the category.

VAR\_NAME: Name of the variable.

*time\_expr*: Timing definition. (See 6.13 for details on time expressions.)

**Min**: Minimum value for the variable.

**Typ**: Typical value for the variable (default if no Min, Typ, Max specified).

**Max**: Maximum value for the variable.

**Meas**: Measured value to be assigned to the variable at execution time by the test program.

**Selector** SELECTOR\_NAME: An optional block to allow selection of Min, Typ, Max, and Meas values to be used in conjunction with a PatternExec block.

## 19.2 Spec and Selector block example

```

Spec tmode_spec {
  Category tmode {
    sp1 = '50.00ns';
    sp5 = '40.00ns';
    shmsp5= '50.00ns';
    sp10p= '0.00ns';
    sp10b= '0.00ns';
    dutyb= '0.00ns';
    dutyb= '0.00ns';
  }
  Category tmode_slow {
    sp1 {Min '30.00ns'; Typ '50.00ns'; Max '70.00ns';}
    sp5 {Min '35.00ns'; Typ '40.00ns'; Max '55.00ns';}
    shmsp5 {Min '0.00ns'; Typ '23.00ns'; Max '40.00ns';}
    sp10p {Min '0.00ns'; Typ '0.00ns'; Max '10.00ns';}
    sp10b {Min '-10.00ns'; Typ '0.00ns'; Max '0.00ns';}
    dutyb {Min '-10.00ns'; Typ '0.00ns'; Max '0.00ns';}
    dutyb {Min '-20.00ns'; Typ '0.00ns'; Max '0.00ns';}
  }
}

```

```

Selector tmode_typ {
  sp1 Typ;
  sp5 Typ;
  shmsp5 Typ;
  sp10p Typ;
  sp10b Typ;
  dutyb Typ;
  dutyb Typ;
}

```

```

Selector tmode_mix {
  sp1 Min;
  sp5 Typ;
  shmsp5 Max;
  sp10p Typ;
  sp10b Typ;
  dutyb Min;
  dutyb Max;
}

```

Figure 33 demonstrates how the information in 19.2 might be referenced in a PatternExec, and how the Spec and Selector are used in combination to specify values for variables.

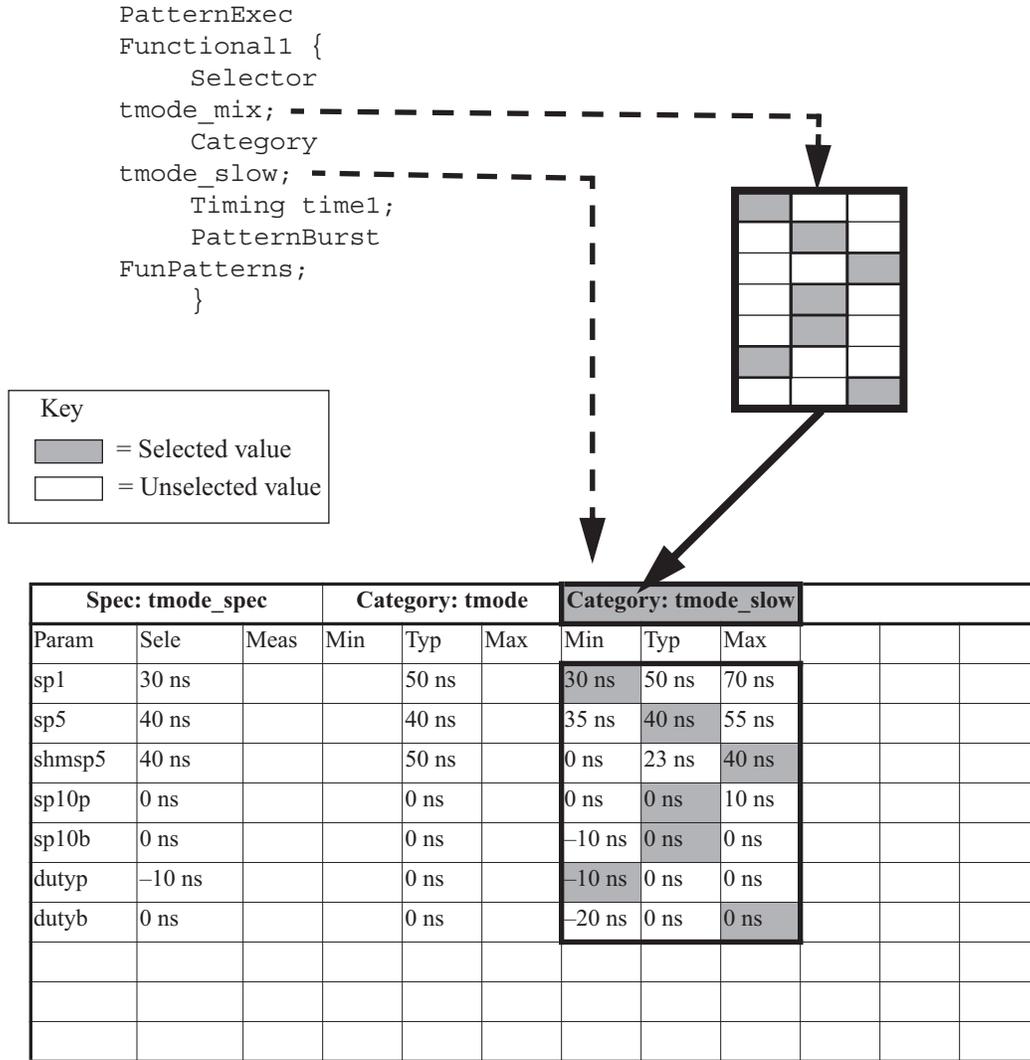


Figure 33—Using the PatternExec to resolve Spec and Selector information

## 20. ScanStructures block

The ScanStructures block is used to define Scan Chains utilized in the scan data. This information is included optionally for documentation and failure analysis. The ScanStructures are not required for translation of STIL to a tester’s format, since the scan data is in tester ready format.

Scan Chains are constructed as a series of Scan Cells. A Scan Cell may be comprised of a single memory element (e.g., a register or flip-flop) or a dual memory element (e.g., MASTER/SLAVE latch). In addition, a Scan Cell may contain other memory elements (shadows).

The Scan Chain definition is limited to indicating the interconnection of Scan Cells and inverters (using the “!” character).

## 20.1 ScanStructures block syntax

```

ScanStructures (SCAN_NAME) {
    ( ScanChain CHAINNAME {
        ScanLength DECIMAL_INTEGER;
        ( ScanOutLength DECIMAL_INTEGER; )
        ( ScanCells CELLNAME-LIST ; )
        ( ScanIn SIGNALNAME; )
        ( ScanOut SIGNALNAME; )
        ( ScanMasterClock SIGNALNAME-LIST ; )
        ( ScanSlaveClock SIGNALNAME-LIST; )
        ( ScanInversion < 0 | 1 >; )
    })*
}

```

**ScanStructures:** Block to define scan chains.

**SCAN\_NAME:** Optional name of this ScanStructures block. Required if multiple ScanStructures blocks are defined. May be referenced in ScanStructures statement in the PatternBurst.

**ScanChain:** Block to define a single scan chain. Multiple scan chains may be defined in the ScanStructures block. CHAINNAME is a unique name associated with each scan chain definition.

**ScanLength:** Identifies the number of scan cells in the scan chain. DECIMAL\_INTEGER is used to audit the ScanCells definition. The length of the actual scan data for a chain may be less than the overall length (incomplete scan, non-observable cells, etc.).

**ScanOutLength:** Identifies the number of observable scan cells in the scan chain. Allows for specification of a scan output in the middle of a scan chain. The default length is **ScanLength** (i.e., all input cells are observable). DECIMAL\_INTEGER is the number of observable latches. It is less than or equal to the ScanLength. The length of the actual scan data for a chain may be less than the overall length (incomplete scan, non-observable cells, etc.).

**ScanCells:** Identifies the scan cells comprising the scan chain. CELLNAME-LIST is **ScanLength** scan cell names separated by whitespace. Inversion is also specified by interleaving the “!” character between scan cell names (also includes before the first scan cell and after the last scan cell). Scan cell names are ordered from the first scan cell to be shifted (input) to the last scan cell to be shifted (output).

**ScanIn:** Identifies the input of the scan chain. A ScanIn SIGNALNAME may be either a Primary Signal (i.e., external source via a tester) or a Pseudo Signal (i.e., internal source via an on-product structure). If no ScanIn is specified, then the first cell’s scan input is indeterminate. Defining a lineheld state on the scan chain would require defining and assigning a pseudo signal with a DefaultState.

**ScanOut:** Identifies the output of the scan chain. A ScanOut signalname may be either a Primary Signal (i.e., external observation via a tester) or a Pseudo Signal (i.e., internal observation via an on-product structure). If no ScanOut is specified, then the scan cells are not directly observable; values would have to be propagated through internal logic to an observation point.

**ScanMasterClock:** Identifies a list of signal names which are MASTER Clocks. MASTER Clocks scan data into single memory elements (e.g., registers or flip-flops) or into the MASTER latch of a dual memory element. SIGNALNAME-LIST is one or more signal names separated by whitespace.

**ScanSlaveClock:** Identifies the signal names which are SLAVE Clocks. A SLAVE Clock scans data into a SLAVE latch of a dual memory element. SIGNALNAME-LIST is one or more signal names separated by whitespace.

**ScanInversion:** Indicates the overall relative inversion from before the first scan cell to after the last scan cell. 0 = no inversion, 1 = inversion.

## 20.2 ScanStructures block example

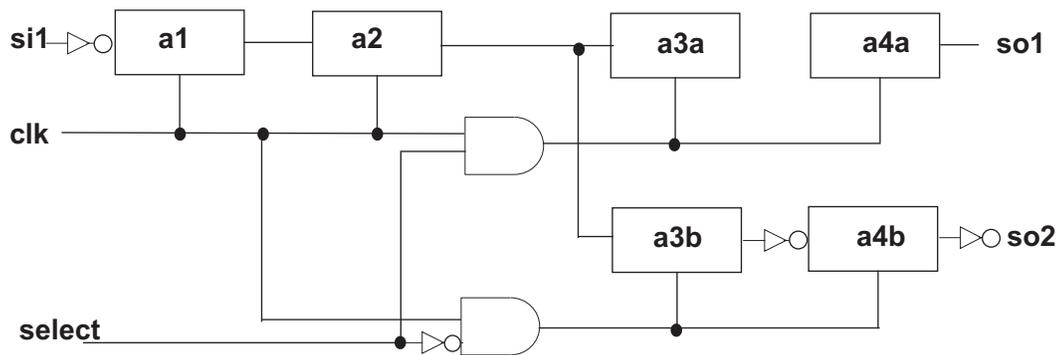


Figure 34—ScanStructures block example

```
ScanStructures {
  ScanChain chain1a {
    ScanLength 4;
    ScanIn si1;
    ScanOut so1;
    ScanMasterClock clk;
    ScanInversion 1;
    ScanCells ! a1 a2 a3a a4a;
  }
  ScanChain chain1b {
    ScanLength 4;
    ScanIn si1;
    ScanOut so2;
    ScanMasterClock clk;
    ScanInversion 1;
    ScanCells ! a1 a2 a3b ! a4b ! ;
  }
}
```

NOTE (Figure 34)—This example assumes that chain1a and chain1b are used independently. Therefore, both scan lengths are 4, and separate **Procedures** or **Macros** are required to load/unload the chains, since the value of the select signal is required to control which scan cells are being loaded/unloaded. Alternatively, a different domain could be defined which assumes that both chains are used concurrently (serially). In this mode, the effective ScanOutLength for the second scanned chain would be 2, and a single Procedure or Macro could perform the load/unload operations.

## 21. STIL Pattern data

Pattern Statements may appear in Pattern blocks, MacroDefs blocks and Procedures blocks. These statements consist of vectors and control information required to implement device tests.

The following pattern constructs contribute to a reduced pattern file size:

- a) Pattern vectors need only define delta changes. Only those signals that change from one vector to the next need to be defined. Optionally, Signals may be re-specified for purposes of clarity at the STIL writer's discretion.
- b) Patterns support scan constructs. Scan data is a primary contributor to pattern file size; STIL supports definition of only the relevant scan data, including incomplete scan and non-normalized data lengths.
- c) Pattern vectors support compaction. Both vector group data and scan data may be defined in a compressed format using Repeats and Hex representations.
- d) Procedures and Macros. Common functions may be defined to eliminate redundant specification within patterns.

### 21.1 Cyclized data

Vector Statements may be defined as cyclized data. Cyclized data consists of waveform specifications per signal applied in a common period of time. Cyclized data is typically tester ready. However, complex waveforms may be defined which require STIL translators to further sub-divide the data into waveforms that may be applied by the tester.

Cyclized data consists of a Signal reference associated to vector data:

```
sigref_expr = vec_data;
```

*sigref\_expr*: This is defined in 6.14.

*vec\_data*: The vector data to be associated to the *sigref\_expr*. The *vec\_data* shall always define at least an individual WaveformChar for each SIGNAL-NAME in *sigref\_expr*. (See also the impact of references to multiple bit waveforms in 21.2.) The vector data may be expressed in different bases with optional repeats for data compaction. The **Signals** or **SignalGroups** may specify a default **Base** when the group is defined. Locally, the following flags may be used to override the **Base** specification:

**\w**: WaveformChar format (e.g., `\w0100 xxxx qqQQQ`). (See 6.15 for more information.)

**\rN XXX**: Repeat the next group of characters N times (e.g., `\r60 01`). N copies of the XXX characters are generated. The XXX characters are delimited by whitespace and may be WaveformChars, Hexadecimal characters, or Decimal characters.

**\h**: Hexadecimal format (e.g., `\h E4B2`). Specifies that the *vec\_data* will switch to a hexadecimal data stream, whose binary value shall select the WaveformChars defined in the **Base Hex** attribute of the **Signals** or **SignalGroups** declaration. Note the required whitespace delimiter between the **\h** and the hexadecimal data stream. The hexadecimal data stream shall consist of valid hexadecimal characters (0..9, a..f, A..F) with optional repeated sequences (e.g., `\h A0 \r5 F8 0E <==> A0F8F8F8F8F80E`).

**\hCHARS**: Hexadecimal format with a local WaveformChar association (e.g., `\h01 69BC`). Specifies that the *vec\_data* will switch to a hexadecimal data stream, whose binary value shall select the locally defined

WaveformChars. Note the whitespace delimiter between the locally defined WaveformChars and the Hexadecimal data stream. The hexadecimal data stream shall consist of valid hexadecimal characters (0..9, a..f, A..F) with optional repeated sequences (e.g., \hLH A0 \r5 F8 0E <==> A0F8F8F8F8F80E).

\d: Decimal format (e.g., \d 79). Specifies that the *vec\_data* will switch to a decimal data stream, whose binary value shall select the WaveformChars defined in the **Base Dec** attribute of the **Signals** or **SignalGroups** declaration. Note the required whitespace delimiter between the \d and the decimal data stream. The decimal data stream shall consist of valid decimal characters (0..9) with optional repeated sequences (e.g., \d 21 \r5 38 21 <==> 21383838383821).

\dCHARS: Decimal format with local WaveformChar association (e.g., \dHL 375). Specifies that the *vec\_data* will switch to a decimal data stream, whose binary value shall select the locally defined WaveformChars. Note the whitespace delimiter between the locally defined WaveformChars and the decimal data stream. The decimal data stream shall consist of valid decimal characters (0..9) with optional repeated sequences (e.g., \dLH 21 \r5 38 21 <==> 21383838383821).

\N: This option is used in two contexts: first, for shift data passed in procedure or macro calls; and second, for multiple-bit data passed into waveforms that contain square-bracket constructs. This may be used to specify the DataBitCount of the data if different from the length specified by the data present.

## 21.2 Multiple-bit cyclized data

Vector statements may contain references to groups that were defined to support multiple-bit data definitions. Multiple-bit data is defined in waveforms through the use of square brackets (See Clause 18 for additional details.)

Multiple bit data may be presented in cyclized data in two formats. Each format has a set of requirements in order to process this multiple-bit data.

The first format is identical to the Cyclized Data statement presented in 21.1, with the extension that the *VEC\_DATA* length is longer than the length necessary to assign a bit to each signal referenced in *sigref\_expr*. The number of WaveformChars is defined by the DataBitCount statement associated with the definition of the Signal or Group being referenced (and the default is 1).

The second statement format is:

```
sigref_expr { ( vec_data; )+ }
```

In this format, there are one or more *vec\_data*; statements. Each statement contains the number of WaveformChar references necessary to apply to all signals defined in the *sigref\_expr*. There are as many statements defined as index values present in the waveform definition applied.

In addition to these constructs, multiple-bit data has the following additional requirements:

- The *sigref\_expr* shall be a group only. The definition of this group shall include a DataBitCount attribute. The definition of this group shall also include a Base attribute identifying the multiple-bit WaveformChars that may be applied to the signals in this group.
- The waveform referenced by the WaveformChars shall define all WaveformChars listed in the Base attribute in a single waveform definition. All waveforms applied across all signals defined in a multiple-bit group shall have the same number of indexed timed events.

### 21.3 Non-cyclized data

Vector statements may define non-cyclized data. Non-cyclized data may be provided during format translation, or as additional information. Specialized tools may be required to process non-cyclized data into cyclized form.

Non-cyclized data consists of a time value, a signal or group reference (or group expression), and a list of Waveform event changes. There shall be a one-to-one relationship of signals in the expression to events in the event\_list. A single time may apply to a block of signal/event pairs.

```
@TIME_VALUE sigref_expr = EVENT_LIST ;
@TIME_VALUE { ( sigref_expr = EVENT_LIST ; )+ }
```

**@TIME\_VALUE:** An integer time offset relative to the start of the Vector. A Vector block may be comprised of a single Vector with all events relative to a common T0. The TIME\_VALUE has an assumed unit type as defined via the TimeUnit statement (see 23.1).

*sigref\_expr:* This is defined in 6.14.

**EVENT\_LIST:** A series of any of the waveform table events allowable in a waveform table. (See Table 9 through Table 12 for a definition of the supported events.)

The following are example non-cyclized data definitions:

```
@30 CLK=U; @50 CLK=D; @100 DATA6=U; @200 { A1=U; A2=D; }
```

### 21.4 Scan data

Serial data may be defined to be applied through an iterative operation on a set of Vectors. The iteration process is defined with the Shift statement, and the serial data is defined only in the context of a procedure or macro call. The syntax for this data is:

```
sigref_expr = serial_data;
```

*sigref\_expr:* A single signal or a GROUPNAME of a single signal with either the ScanIn or ScanOut attribute (see Clause 14). The attribute identifies the required padding; scan inputs are pre-padded during scan length normalization, and scan outputs are post-padded. (See 24.4 for more information about scan functions and 24.5 for parameter-data substitution.)

*serial\_data:* A stream of scan data to be applied serially. The serial data stream may utilize the same Base and Repeat compression flags as used in VEC\_DATA. The length of the serial\_data is typically the length defined by the **ScanIn** or **ScanOut** attributes. However, the length may be less than these values to support incomplete scans (only a subset of the scan cells are scanned).

The following are example scan data definitions:

```
Call shift_one {
    si_1=100100010011100101; // length implied by Scan definition
    so_2=\120 \hLHX 5821940559; // incomplete scan (length=20) w/ base
    // definition. Equivalent to HHXLLXLHXHLLLLHHHHXXH
}
```

## 21.5 Pattern labels

Any Pattern Statement may have an associated label. Labels are used to identify Start/Stop vectors and to identify **Goto** targets. Labeling non-vector statements results in selecting the next executable vector.

Labels consist of a user-defined name followed by a colon. The user-defined name may be double quoted. Labels shall be unique per **Pattern** or **Procedure** block. As the general purpose of defining a Macro is to facilitate re-use of that information in a Pattern, and because labels need to be unique inside a Pattern, labels (if present in MacroDefs blocks) shall be ignored during processing. The following are example labels:

```
mylabel:      V { }           // simple label name.
"mylabel":    V { }           // optionally quoted label,
                                     // different than the unquoted name.
"mylabel+2":  V { }           // required quoted label,
                                     // to protect special character.
```

## 22. STIL Pattern statements

### 22.1 Vector (V) statement

The Vector statement is used to define stimulus and response for one test cycle.

NOTE—A test cycle is typically equivalent to a tester's T0 cycle. However, complex events may be defined which may require the STIL translator to segment the test cycle into multiple tester T0 cycles.

Vector statements typically only specify the delta changes from the previous vector. Vector statements may contain cyclized data and/or non-cyclized data. The syntax of Vector statements is:

```
( LABEL : ) V(ector) { ( cyclized data ) * ( non-cyclized data ) * }
```

The following are example Vector statements:

```
// change Signal "abc" to WaveformChar 'h'
Vector { abc=h; }
V { abc=h; } // 'Vector' may be abbreviated to 'V'

// labeled vector which changes "outs" to WaveformChars
// 'LLLHHHZZZXXX'
label1: V{outs=\hLHZX 015ABF; }

// non-cyclized info with cyclized data
// Select the waveform "1" for the signal "control" and also apply a
// ForceUp event at TimeUnit * 30.
V { control=1; @30 control=U; }

// another cycle exactly like the previous one
V { }
```

## 22.2 WaveformTable (W) statement

The WaveformTable statement is used to specify the current WaveformTable to use during vector translation. Every signal's WaveformChar is translated using the specified WaveformTable for subsequent vectors. A WaveformTable reference is applied to all subsequent Vector statements until changed with another WaveformTable statement. It is recommended that only one WaveformTable statement precede a Vector statement; however, if multiple statements exist, the last one is applied to the Vector statement. The syntax of the WaveformTable statement is:

```
( LABEL : ) W(aveformTable) NAME ;
```

The following are example WaveformTable statements:

```
WaveformTable init;
V { pin1=H; pin2=H; }      // 'H' definition from WaveformTable(WFT)
'init'

W main;                    // "WaveformTable" may be abbreviated to 'W'
V { pin2=Z; }              // "pin2" value of 'Z' in WFT 'main',
"pin1"                     // didn't change, so its value of 'H' is also
                           // in WFT 'main'
```

## 22.3 Condition (C) statement

The Condition statement is used to define stimulus and/or response to be set up, but deferred from being performed until a Vector statement is defined. Condition statements are useful in scan testing to define the "background" states for signals required when applying the scan data. The WaveformTable used to translate the conditions is the waveform in effect at the time of the next Vector, not the waveform in effect at the time of the Condition statement. Multiple condition statements may be defined between vector statements. The last state defined in a Condition or Vector statement for each pin is the state applied to that pin on the Vector statement. A Vector statement defining a WaveformChar to be applied to a signal shall override any WaveformChars defined in preceding Condition statements. The syntax of the Condition statement is:

```
( LABEL : ) C(ondition) { ( cyclized data ) * ( non-cyclized data ) * }
```

The following are example Condition statements:

```
// define "pin1" to be WaveformChar 'H' for next vector
Condition { pin1=H; }
C { pin1=H; }      // 'Condition' may be abbreviated to 'C'

// change the WaveformChar to 'L' ('H' never got output)
W wft1;
C { pin1=L; }      // assumes 'L' is output using wft1 definition
W wft2;
V { }              // "pin1" output is 'L' as defined by wft2
```

See 5.4 for examples illustrating practical usage of Condition statements.

## 22.4 Call statement

The Call statement transfers execution to the named procedure block. The procedure shall be previously defined in a **Procedures** block section. The current WaveformChars for every signal shall be reinstated upon return from the procedure. A procedure call may define values to be substituted in the procedure, typically used to implement scan tests. The syntax of the Call statement is:

```
( LABEL : ) Call PROCNAME ;
( LABEL : ) Call PROCNAME { ( scan-data ) * ( cyclized data ) *
                             ( non-cyclized data ) * }
```

Non-cyclized data, if present, is applied relative to the start of the Call execution. This data is applied directly to referenced signals irrespective of '#' or '%' references. (See 24.5 for information about data substitution via '#' and '%' references.)

The following are example Call statements:

```
Call setup; // self contained procedure

Call scanload { // procedure which implements scan, pass in scan
data
    si1=8C92206; // "si1" defined as ScanIn 28; Base Hex 01
    si2=\114 9CC0; } // "si2" defined as Base Hex 01;

Call stability { cnt1=0; } // procedure with values substituted
```

## 22.5 Macro statement

The Macro statement instantiates the specified macro. The macro shall be previously defined in a **MacroDefs** block section. Upon completion of the macro, the current WaveformChars for every signal are retained for subsequent Vectors. A Macro statement may define values to be substituted in the macro, typically used to implement scan tests. The syntax of the Macro statement is:

```
( LABEL : ) Macro MACRONAME ;
( LABEL : ) Macro MACRONAME { ( scan-data ) * ( cyclized data ) *
                             ( non-cyclized data ) * }
```

Non-cyclized data, if present, is applied relative to the start of the Macro execution. This data is applied directly to referenced signals irrespective of '#' or '%' references. (See 24.5 for information about data substitution via '#' and '%' references.)

The following are example Macro statements:

```
Macro reset;

Macro scanunload {
    so1=8C92206; // "so1" defined as ScanOut 28; Base Hex LH
    so2=\114 9CC0; } // "so2" defined as Base Hex LH;

Macro clksoff { clk1=0; clk2=0; clk3=1; }
```

## 22.6 Loop statement

The Loop statement defines a block of Pattern statements to be looped upon. The loop count is a positive integer that specifies how many times the block of pattern statements is to be executed. The syntax of the Loop statement is:

```
( LABEL : ) Loop LOOPCNT { ( pattern-statements ) * }
```

Executing a Start or Stop statement from a PatternBurst to a label on or inside a Loop statement results in undefined behavior.

The following are example Loop statements:

```
outer: Loop 10 { // label "outer" serves only as documentation
  V { clk1=P; clk2=0; }
  inner: Loop 100 {
    V { clk1=0; clk2=P; }
  } // inner
} // outer
```

## 22.7 MatchLoop statement

The MatchLoop statement defines a block of Pattern statements to be looped (repeated) until all Pattern statements are executed without differences between the output information present in the Pattern data, and the actual response of the device, or until a specified number of loops are exceeded.

The MatchLoop construct includes the definition of a set of Patterns to be executed after all Patterns have passed in the loop body. These Patterns may also be looped, and are intended to hold the device in an “idle” state until the test hardware is ready to progress to the next statement. These Patterns are enclosed in a BreakPoint block in the MatchLoop syntax. The syntax of the MatchLoop construct is:

```
( LABEL : ) MatchLoop < LOOPCNT | Infinite > {
  (pattern-statements)+
  ( LABEL : ) BreakPoint { (pattern-statements)+ }
}
```

Pattern-statements shall be present in a MatchLoop. The BreakPoint block shall be present. The LOOPCNT is either an integer or the keyword “**Infinite**.” An integer value shall be greater than or equal to one, which defines the maximum number of loops before the execution of this loop is stopped and this Pattern is terminated. The keyword “**Infinite**” defines that this loop shall execute until all Patterns agree with the device response.

BreakPoint vectors are used to allow test hardware to re-establish the pattern flow after all MatchLoop conditions have been met. BreakPoint patterns may execute zero or more times depending on test hardware.

Label statements inside a MatchLoop may be referenced only from inside that MatchLoop; it is illegal to jump (Goto) into a MatchLoop. Executing a Start or Stop statement from a PatternBurst to a label on or inside a MatchLoop statement results in undefined behavior.

A MatchLoop statement shall not be nested inside another MatchLoop.

Pattern-statements in the BreakPoint block shall define information for all signals active in the test; partial or incremental statements are not sufficient.

The following is an example MatchLoop statement:

```
MatchLoop 1024 { V { clk=1; out=1;} V { clk=0; }
                BreakPoint { V { clk=0; out=X;}}
}
```

## 22.8 Goto statement

The Goto statement defines an unconditional branch to a vector label. It is invalid to branch outside of the Pattern block (i.e., into or out of a procedure). The syntax of the Goto statement is:

```
( LABEL : ) Goto LABELNAME ;
```

The following is an example Goto statement:

```
infinite_loop: V { }
               Goto infinite_loop;
```

## 22.9 BreakPoint statements

The BreakPoint statement defines a location in the Pattern where the device is in a stable state. This defines locations where a large Pattern block may be segmented into multiple Bursts if tester resources are becoming constrained (e.g., tester memory is full, or timing information is exceeded). Specifying BreakPoints is optional, but is useful for very large Patterns. The BreakPoint statement has two forms:

```
( LABEL : ) BreakPoint;
( LABEL : ) BreakPoint { ( PATTERN-STATEMENTS ) * }
```

The first form of the BreakPoint identifies a Pattern location where the test may be stopped during the BreakPoint, and restarted after the BreakPoint, without affecting the test. The second form identifies a location where some Vector activity is required (such as a keep-alive clock) to allow the device to continue properly after the BreakPoint. The pattern statements defined in this block are implicitly looped for an undefined amount of time until the test is ready to resume execution. The pattern statements iterate completely during the BreakPoint; the patterns are not exited until the end of the BreakPoint block even if the processing has completed. The following is an example BreakPoint statement:

```
Call scanunload { scan-data }
BreakPoint; // scan is typically at stability before the next load
Call scanload { scan-data }
```

The BreakPoint (with pattern-statements) is required at the end of the MatchLoop construct to define patterns that may be applied after a MatchLoop has met its exit criteria. In a similar fashion, a BreakPoint with pattern-statements may be defined at the end of a Loop block. These statements may then be applied during the exit process from a Loop operation.

## 22.10 I<sub>DDQ</sub> TestPoint statement

The I<sub>DDQ</sub> TestPoint statement defines a place in the Pattern where an I<sub>DDQ</sub> measurement may be performed. I<sub>DDQ</sub> TestPoint statements are optional. The syntax of the I<sub>DDQ</sub> TestPoint statement is:

```
( LABEL : ) IDDQ TestPoint;
```

The following is an example  $I_{DDQ}$  TestPoint statement:

```
V { new-delta-change-data }
IDDQ TestPoint; // perform IDDQ measurement between 2 vectors
V { new-delta-change-data }
```

## 22.11 Stop statement

The Stop statement specifies that execution of the pattern is to stop. The Stop statement is optional. The syntax of the Stop statement is:

```
( LABEL : ) Stop ;
```

The following is an example Stop statement:

```
V { new-delta-change-data }
Stop;
V { }
```

All execution is halted when the Stop statement is encountered in a Pattern. Any subsequent Patterns or PatternBursts referenced in the PatternExec will not be executed after a Pattern Stop statement has been executed.

## 22.12 ScanChain statement

The ScanChain statement specifies the name of a ScanChain that is active for the next set of pattern operations. The syntax of the ScanChain statement is:

```
( LABEL : ) ScanChain CHAINNAME ;
```

The following is an example ScanChain statement:

```
V { new-delta-change-data }
ScanChain config_A;
V { }
```

## 23. Pattern block

The Pattern block shall have a domain name. The name-space of the Pattern is shared with the PatternBurst names. The set of names across both PatternBurst and Pattern blocks shall be unique.

### 23.1 Pattern block syntax

```
Pattern PATTERN_NAME {
  (( LABEL : ) TimeUnit 'TIME_DEF';)
  ( PATTERN-STATEMENTS )*
}
```

**Pattern:** Identifies the start of the block defining pattern data.

**TimeUnit:** Defines the default time units associated to non-cyclized data. The TimeUnit statement is required if non-cyclized data is specified. The syntax of the TimeUnit statement is:

```
TimeUnit 'UNITTYPE';
```

The following is an example TimeUnit statement:

```
TimeUnit '1ns'; // timings are in nanoseconds
```

PATTERN-STATEMENTS: Any pattern statement may be used in a Pattern block (see Clause 22).

## 23.2 Pattern initialization

Every signal used in a Pattern shall have an initial WaveformChar defined in the first vector of the Pattern. Unused signals (i.e., ones not in the first vector) default to their assigned DefaultState as defined in Clause 14, and shall not appear in any pattern statements within this pattern block.

## 23.3 Pattern examples

See 5.1 for Pattern examples.

## 24. Procedures and MacroDefs blocks

Procedures and MacroDefs are very similar, so they are discussed and contrasted together.

**Table 13—Comparison of procedures and MacroDefs**

Procedures	MacroDefs
May contain any Pattern statements.	May contain any Pattern statements.
Supports Scan testing using Shift block.	Supports Scan testing using Shift block.
Serves as a data reduction vehicle for STIL writers.	Serves as a data reduction vehicle for STIL writers.
Possible reuse if no Signal substitution. Procedures called without Signals defined in the Call { } have no data substitution. Therefore, these vectors may be implemented as a subroutine by STIL translators. However, translators may elect to always expand procedures.	Macros are always expanded.
Possible expansion if Signal substitution. Procedures called with Signals defined in the Call { } have data substitution. ATE testers may not be capable of supporting variable data, resulting in vector expansion vs. using a common subroutine.	Macros are always expanded.
Invoked via Call statement.	Invoked via Macro statement.
Reused vectors require that all Signals be initially defined. Therefore, all signals used in the procedure shall be initially defined in the first vector of the procedure. Unused signals default to their DefaultState.	Only the delta changes need to be specified in first vector of the Macro. Prior Signal WaveformChars are used when the Macro is expanded.

**Table 13—Comparison of procedures and MacroDefs (continued)**

Procedures	MacroDefs
The current WaveformTable shall be defined prior to any vectors (via W statement).	The last WaveformTable specified is used. Note that the last WaveformTable may have been specified in a previous Macro.
Procedures return to the caller upon reaching the end of the procedure block (there is no “Return” statement).	Macro terminates upon reaching the end of the macro block.
Each Signal’s WaveformChar prior to calling the procedure is reinstated upon exiting the procedure.	The current Signals’ WaveformChars remain in effect upon completion of the macro.
The WaveformTable in effect prior to calling the procedure is reinstated upon exiting the procedure.	The current WaveformTable remains in effect upon completion of the macro.
Cannot branch outside of the Procedure block.	Branch label shall be within the Pattern block.

## 24.1 Procedures block

The Procedures block defines named sets of procedures that may be called from inside a Pattern block, a Procedure block, or a Macro block via the Call statement. A Procedures block may have an associated domain name, allowing specific reference from a PatternBurst block or PatList block. A single global Procedures block (no domain name specified) may also be defined. Only one global Procedures block shall be defined in a set of STIL information.

All procedures shall be defined in either the Global Procedures block or a referenced Domain-named Procedures block prior to being invoked by a Call statement.

A procedure may invoke another procedure or macro, but the invoked item shall be defined before invocation. This requirement prevents recursion in procedures or macros.

The syntax for a Procedures block is:

```

Procedures ( PROCEDURE_DOMAIN_NAME ) {
    ( PROCEDURE_NAME {
        ( PATTERN-STATEMENT )*
    } )*
}

```

**Procedures:** Identifies the start of the block defining procedures.

**PROCEDURE\_DOMAIN\_NAME:** Identifies an optional domain name. If omitted, then the procedures block is globally accessible. Contains 0 or more procedure definitions.

**PROCEDURE\_NAME:** Identifies the name of a procedure. This is the name that is to be used in the Call statement that references the procedure. PROCEDURE\_NAME shall be unique within a Procedures block.

**PATTERN-STATEMENT:** Any pattern statement may be used in a Procedure (see Clause 22). However, identifying a BreakPoint or using Stop in a procedure may be confusing and lead to unexpected results.

## 24.2 Procedures example

```

Procedures {
  procx {
    W waves;
    V { sigs1=11010; }
    V { sigs1=00110; sigx=0; }
  }
}

```

## 24.3 MacroDefs block

The MacroDefs block defines named sets of macros that may be instantiated from inside a Pattern block, a Procedure block, or a Macro block via the Macro statement. A MacroDefs block may have an associated domain name, allowing specific reference from a PatternBurst block or PatList block. A single global MacroDefs block (no domain name specified) may also be defined. Only one global MacroDefs block shall be defined in a set of STIL information.

All macros shall be defined in either the global MacroDefs block or a referenced domain-named MacroDefs block prior to being invoked by a Macro statement. However, expansion of macros shall occur only after a macro is instantiated, because the complete pattern context is necessary to properly process macros.

A macro may invoke another macro or procedure, but the invoked item shall be defined before the invocation. This requirement prevents recursion in procedures or macros.

The syntax for a MacroDefs block is:

```

MacroDefs ( MACRO_DOMAIN_NAME ) {
  ( MACRO_NAME {
    ( PATTERN-STATEMENT )*
  } )*
}

```

**MacroDefs:** Identifies the start of the block defining macros.

**MACRO\_DOMAIN\_NAME:** Identifies an optional domain name. If omitted, then the MacroDefs block is globally accessible. Contains 0 or more macro definitions.

**MACRO\_NAME:** Identifies the name of a macro. This is the name that is to be used in the Macro statement that instantiates the macro. **MACRO\_NAME** shall be unique within a MacroDefs block.

**PATTERN-STATEMENT:** Any pattern statement may be used in a Macro (see Clause 22). However, identifying a BreakPoint or using Stop in a macro may be confusing and lead to unexpected results.

## 24.4 Scan testing

Scan Testing is a design for test methodology which incorporates interconnected shift register latches within a product. Data may then be “scanned” into the device to pre-condition internal nets, and results may be “scanned” out of the device for observation. Scan tests are commonly implemented as Procedures or Macros, depending on desired disposition at the end of the scan operation. (Procedure = states return to values prior to scan operation; Macro = states retain value after the scan operation.)

Scan Testing utilizes a special Shift block within a Procedure or Macro. The Shift block contains the vector(s) required to shift one value into/out of the scan chains. Scan testing may require pattern statements prior to the Shift block to precondition the scan operation. Similarly, pattern statements may follow the Shift block to post-condition the scan operation. Only one Shift statement is defined in a scan-oriented block. The general format of a scan Procedure or Macro is:

```
PROCEDURE_OR_MACRO_NAME {
  ( PATTERN-STATEMENT )*           // SETUP FOR SCAN
  ( Shift { (PATTERN-STATEMENT)+ } // PERFORM SCAN
  ( PATTERN-STATEMENT )*           // RELEASE AFTER SCAN
  )+
}
```

## 24.5 Procedure and Macro Data substitution

The processing of the Shift block has an implied repeat count contingent on the scan data defined in the procedure or macro invocation. This repeat count is the normalized length of all of the scan data. The rules for determining this repeat count are explained in this subclause. The Shift block lacks an explicit count, since this would impede incomplete scan. Also, STIL doesn't define the normalized length for each scan invocation, since it may be inferred from the data.

Two special WaveformChar characters are used within the body of Procedures or Macros to identify where data substitution is to occur. These WaveformChars are “#” and “%.” The data to substitute is provided either from the Procedure or Macro invocation, or through a default value mechanism.

“%” defines where a constant-value parameter is substituted. Any reference to a signal that has a % WaveformChar will apply the WaveformChar specified in the invocation (or the default value defined later) to that signal. All occurrences of % for a signal, in a Procedure or Macro body, applies the same WaveformChar value to that signal.

“#” defines where incremental data substitution is to occur. As such, # is used to pass values into the scan signals of a Shift block. Each iteration of the Shift block substitutes the next normalized scan state into any signal that has a # for a WaveformChar. Only a single value is substituted for each iteration of the Shift block. Therefore, if a Signal defines the # WaveformChar in multiple vectors within the Shift block, then the same value is substituted in each vector for the iteration.

The # and % operators may be applied as a single character to a *sigref\_expr* that consists of a group of signals, or they may be expanded to represent the individual signals of that group. If they are expanded to map individual signals, then there shall be a one-to-one correspondence of WaveformChars to signals in the group. If they are expanded, # and % may appear in a WaveformChar list in combination with other defined (constant) WaveformChar values for signals in the specified group.

Hybrid scan testing has arisen which requires signals to change while applying the final scan state. STIL supports this by allowing the # WaveformChar to be specified in vector(s) preceding the Shift block, and/or vector(s) following the Shift block. The following illustrates this concept:

```
complex_scan { // could be a procedure or macro
  C {ins=0000; outs=xxxx; } // define all pins initially
  V {s1=#; } // apply the first scan state
  Shift { V { s1=#; clk=P; } } // apply the second to next
  // to last scan state
  V {s1=#; cnt1=1; } // apply the last scan state
} // complex_scan
```

Data substitution using the # and % WaveformChars has the following rules:

- Data specified in a Procedure or Macro invocation, assigned to a specific *sigref\_expr*, shall be applied first to the matching *sigref\_expr* in the Procedure or Macro body. If *sigref\_expr* does not appear in the body of the Procedure or Macro, then the data shall be applied to the individual components of a *sigref\_expr* in the Procedure or Macro that contains a # or % for each signal that is a component of the *sigref\_expr* specified in the invocation. For example:

```
SignalGroups { sg_in='si1+si2+si3' {ScanIn;} sg_ou='so1+so2+so3'
{ScanOut;} }

// the scan BODY could define:
all_scans { // procedure or macro
  Shift { v1: V {sg_in=#; sg_ou=###; } } // '#' and '###' are
// equivalent here
}
// the scan INVOCATION could refer to:
all_scans { si1=111; si2=111; si3=111; so1=HHHHH; so2=HHH; so3=HHH;
}
```

- It is an error if the same *sigref\_expr* in the body of a Procedure or Macro uses both # and % references. If a signal requires scan and constant values in different statements, these values should be passed through different *sigref\_exprs* into that signal.
- It is an error if the same signal is referenced by multiple *sigref\_exprs* in the invocation of a Procedure or Macro, and multiple references to that signal in the body of a Procedure or Macro rely on component-name-matching. Note that this is not an issue if the parameters are resolved by explicit name matching. An example of the error situation is:

```
SignalGroups { sg1='si1+si2+si3'; sg2='si1'; sgA='si1'; sgB='si1';
sgC='si2+si3'; }

// the Procedure/Macro BODY could contain:
  V {sg1=%; }
  V {sg2=%; }
}
// An incorrect call (ambiguous component-matching):
fill_ { sgA=0; sgB=1; sgC=00; }
//cannot determine whether sgA or sgB go with sg1 or
//sg2
// A correct call (relying on explicit name matching):
fill_ { sg1=100; sg2=0; }
```

- If data is defined for a signal in a Procedure or Macro invocation, and the Procedure/Macro body does not specify # or % for that signal in any vectors, then the invocation data is ignored.
- If data is not defined for a signal in a Procedure or Macro invocation, and the Procedure/Macro body specifies # or % on that signal in any vector, then the last defined WaveformChar that precedes the first # or % is used. This situation is a derivative of incomplete scan, where individual chains may be inactive for certain scan operations. The last defined WaveformChar is found in the previous V or C statement in the Macro or Procedure body.

- If more vectors containing #’s exist in a Procedure/Macro body than there is data defined in the Procedure or Macro invocation, then substitution is performed with the available data, and the last explicitly defined WaveformChar that precedes the first # or % is used to complete any missing data. For example:

```

    complicated { // procedure or macro
        v1: V { pin1=0; }
        v2: V { pin1=#; }
        v3: V { pin1=1; }
        v4: V { pin1=#; }
        Shift { v5: V {pin1=#; } }
        v6: V { pin1=#; }
    }

```

- If the data passed was “pin1=HL,” then the value on pin1 in vector v1 is 0, as specified in that vector; the value in vector v2 is H (first data substitution); and the value in v4 is L (second substitution). Because no data is left, the Shift block is not executed and vector v5 is suppressed. Finally, vector v6 is 0 because of padding (i.e., the last explicitly defined WaveformChar before the first #’ was 0 in vector v1).
- A particular *sigref\_expr* shall appear only once in a Shift block. Consecutive shift blocks require unique *sigref\_exprs* for each Shift statement.
- Data across multiple signals referenced with a # is normalized as follows. For each signal with a # WaveformChar:
  - 1) Count the number of Vectors preceding the Shift block containing # for that signal.
  - 2) Count the number of Vectors after the Shift block containing # for that signal.
  - 3) If this signal has a # in the Shift block, then determine the number of possible scan shifts cycles for this signal by subtracting the counts in steps (a) and (b) from the number of WaveformChars specified in the invocation argument for this signal (the actual length of the data specified for this signal). This number may be negative (implying insufficient data for the # references outside of the Shift block). For purposes of step (d), the number of scan shifts is zero; but for purposes of calculating the number of pad WaveformChars needed in step (e), this negative value is applied.
  - 4) If this signal does not have a # in the Shift block, then the required number of pad WaveformChars is the result of steps (a) + (b), subtracted from the data-length passed in on the invocation.
  - 5) Select the maximum value of step (c) across all signals in the shift. This is the number of scan cycles to be executed.
  - 6) For each signal, the data is padded by the amount of shift data for this signal [from step (c)], subtracted from the number of scan cycles [in step (d)]. Scan input signals are pre-padded and scan output signals are post-padded. The padding state is always the last explicitly defined vector WaveformChar (from either a previous V or C statement). For example:

```

SignalGroups {sg_in='si1+si2+si3' {ScanIn;}
              sg_ou='so1+so2+so3' {ScanOut;}
              sg   ='sg_in+sg_ou'; }
// the scan BODY:
    all_scans { // procedure or macro
        C { sg = PPPPPP; }
            // this defines the default pad state for all signals
            // unless overridden by another state before a '#’

```

```

    V { sg = #00##X; }
    Shift { v1: V {sg_in=#; sg_ou=#; } }
    V { sg = ##0#XX; }
  }
// the scan INVOCATION:
  all_scans { si1=111; si2=111; si3=111; so1=HHHHH; so2=HHH; so3=HHH;
}

```

The calculation to determine both the shift-count and the padding per signal, given the algorithm above, is shown in Table 14.

**Table 14—Example scan data normalization**

Step	si1	si2	si3	so1	so2	so3
Data-length	3	3	3	5	3	3
a) Pre-shift V's by signal	1	0	0	1	1	0
b) Post-shift V's by signal	1	1	0	1	0	0
c) Data-length [(a) + (b)]	1	2	3	3	2	3
d) Maximum scan shift			3	3		3
e) Padding length [(d) - (c)]	2	1	0	0	1	0

This process generates the following test cycles (equivalent Vector statements after data substitution). WaveformChars in bold are values specified in the Procedure body. “P” is the WaveformChar specified in the initial C statement that defines the default pad state for each signal:

```

V { sg = P00HHX; } // pre-shift V statement
V { sg = P01HHH; } // shift cycle 1
V { sg = 111HHH; } // shift cycle 2
V { sg = 111HPH; } // shift cycle 3
V { sg = 110HXX; } // post-shift V statement

```

The state for si2 in shift cycle 1 is a 0, because the last defined state before the first substitution was ‘0’ for this signal, which became the pad state for this signal.

It is illegal to pass data through nested procedures or macros.

## Annex A

(informative)

### Glossary

The following terms and definitions are used within the context of this standard:

**A.1 Domain name:** A STIL block statement may contain a string before the opening brace. This string becomes the “domain name” for all statements enclosed within those braces.

**A.2 Event:** A prescribed operation to be performed on a signal. An example of an event is “U,” which specifies “drive to a high value” (“Up”). Events are listed in Table 9, Table 10, Table 11, and Table 12.

**A.3 Pad state:** A logic state used to extend scan data to a uniform length.

**A.4 Serial\_data:** **(A)** A metatype used to present language construct. **(B)** A list of WaveformChars or alternate notations.

**A.5 Sigref\_expr:** **(A)** A metatype used to present language constructs. **(B)** A reference to a signal, group, or signal expression.

**A.6 Spec category:** A subindexing of values defined in a Spec Table, which allows variables to be defined once and assigned multiple values. During timing expression evaluation, the specific category must be selected to define the set of values to be used for the variables defined.

**A.7 Spec table:** A collection of variables referenced in timing expressions (inside Waveforms), with values associated with each variable to be used to resolve timing expressions.

**A.8 Timed event:** An event at a specified time. This may or may not cause a change from a previous value.

**A.9 Time\_expr:** **(A)** A metatype used to present language construct. **(B)** A reference to a timing expression.

**A.10 Vec\_data:** **(A)** A metatype used to present language constructs. **(B)** A list of WaveformChars or alternate notations.

**A.11 WFC:** *See:* **WaveformChar.**

**A.12 WGL:** A test interchange language defined and supported by Test System Strategies, Inc.

**A.13 WaveformChar (WFC):** A symbol used to reference a waveform. A single ASCII character in STIL. This character is used to reference a waveform definition defined in a Timing block.

**A.14 WaveformTable:** A collection of waveforms defined to a set of Signals. Waveform Tables are referenced in the Patterns block to define the set of waveforms that may be used by each Signal in the Vectors. Each signal may have its own set of waveforms associated with it in the table.

## Annex B

(informative)

### STIL data model

This annex presents a Data Model representation of the STIL language. Figure B.1, Figure B.2, and Figure B.3 present the main objects of the STIL language and their relationships. Figure B.4 shows the Pattern constructs, and Figure B.5 shows the Timing constructs.<sup>7</sup>

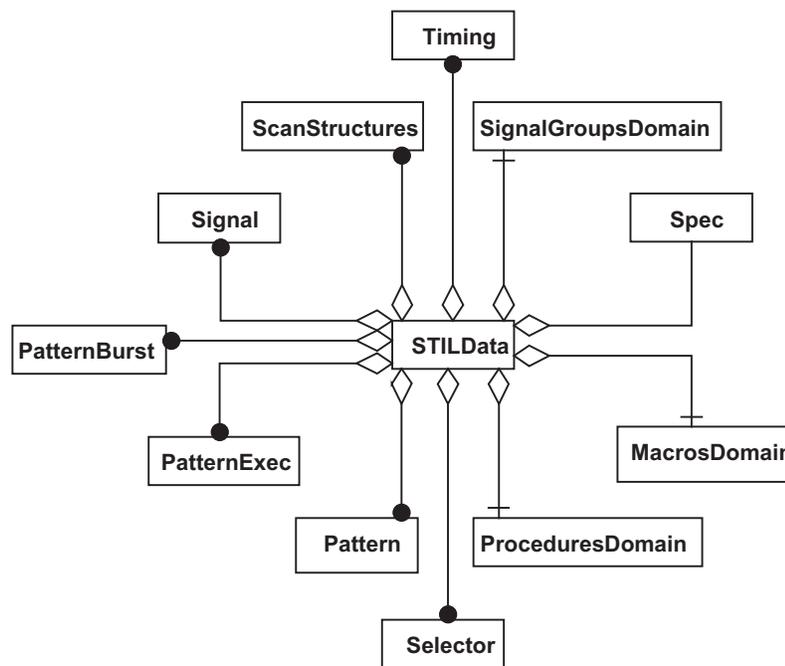


Figure B.1—Top level of STIL data model

<sup>7</sup>For an understanding of the notation used in these diagrams, see *Object-Oriented Modeling and Design*, James Rumbaugh, et al, Prentice Hall, ISBN 0-13-629841-9.

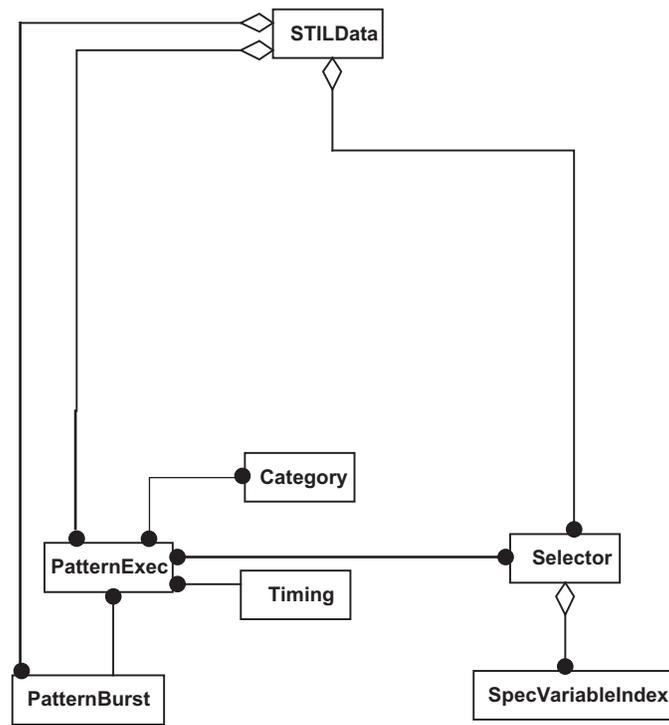


Figure B.2—PatternBurst, PatternExec, and Selector relationships

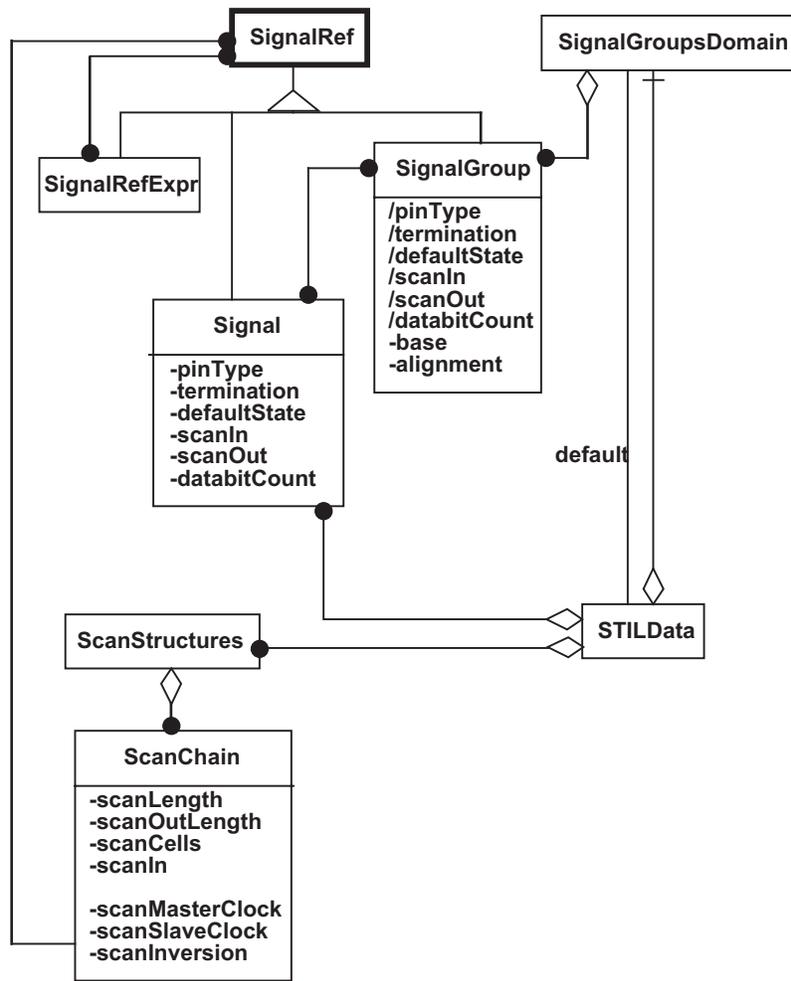


Figure B.3—Signal, SignalGroup, and ScanStructures group relationships

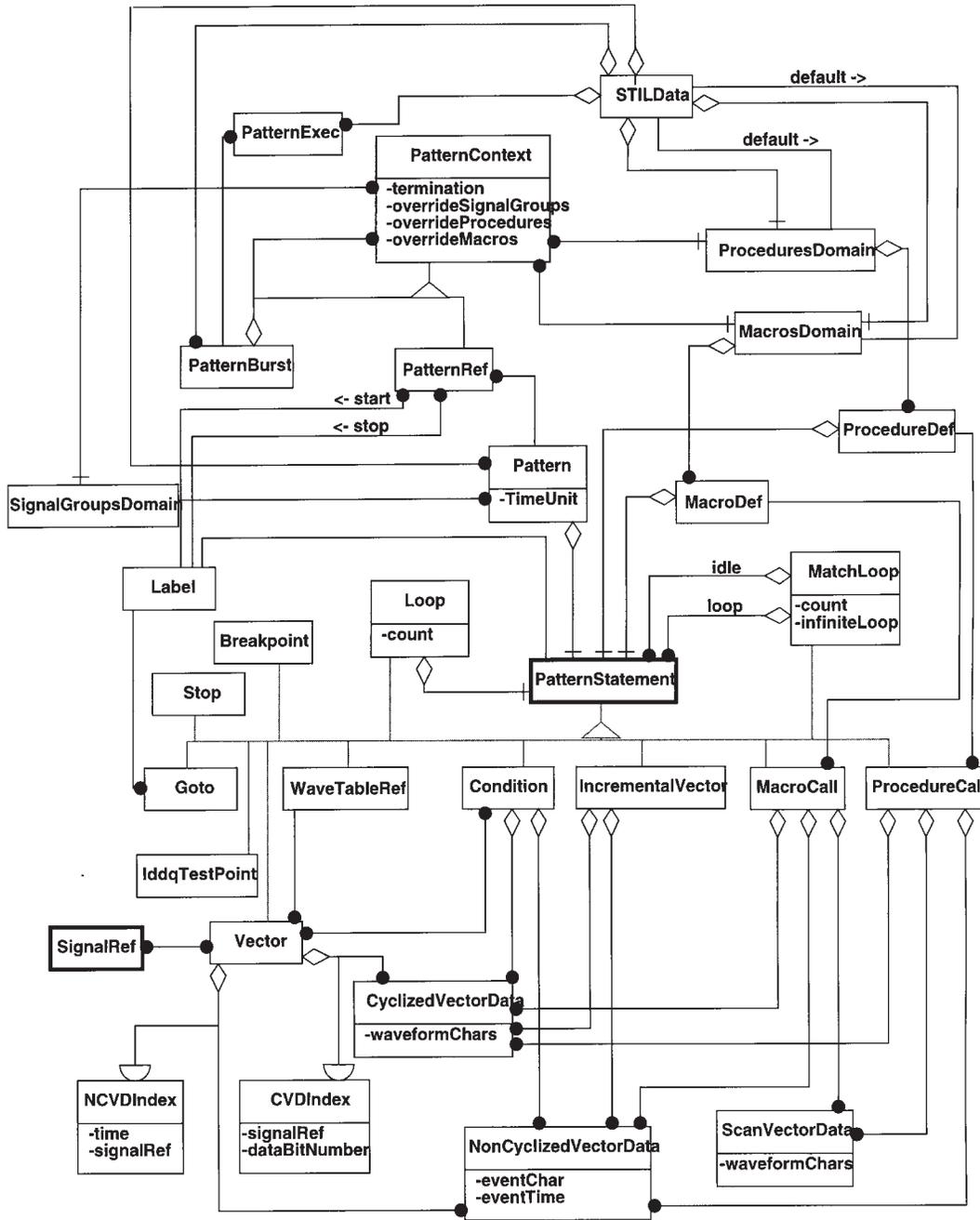


Figure B.4—Pattern statement relationships

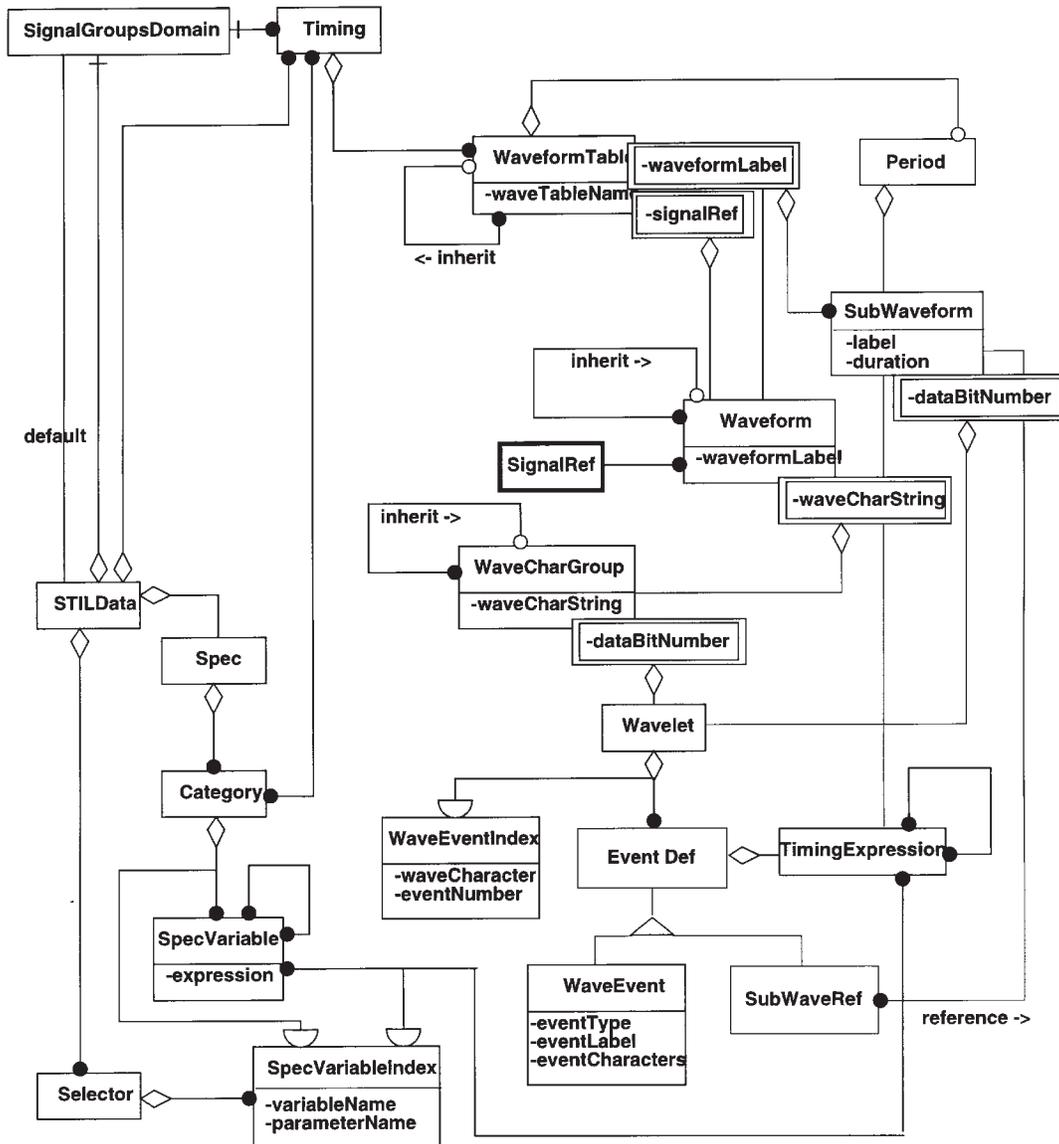


Figure B.5—Timing statement relationships

## Annex C

(informative)

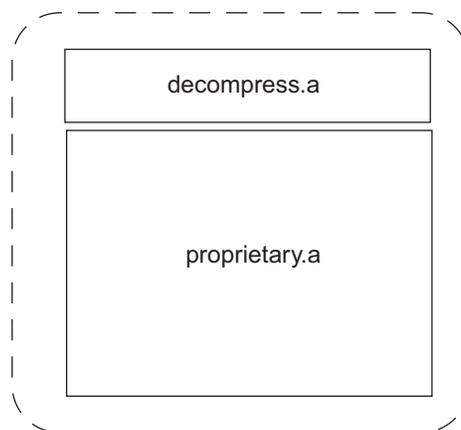
### GNU GZIP reference

STIL files may be compressed using the GNU GZIP (deflate) program.<sup>8</sup> Compressing files addresses the concerns with transferring and storing huge amounts data associated with the design and testing of complex digital VLSI circuits (e.g., “the Gigabit problem”).

The GZIP package is available from various locations, including:

- prep.ai.mit.edu:/pub/gnu/gzip-1.2.4.tar (or.shar or .tar.gz : source);
- prep.ai.mit.edu:/pub/gnu/gzip-msdos-1.2.4.exe (MSDOS, lha self-extract);
- oak.oakland.edu:/simtel/msdos/compress/gzip124.zip (MSDOS exe);
- garbo.uwasa.fi:/unix/arcers/gzip-1.2.4.tar.Z (source);
- garbo.uwasa.fi:/pc/unix/gzip124.zip (MSDOS exe);
- ftp.uu.net:/pub/archiving/zip/VMS/gzip124x.vax\_exe (VMS exe);
- mac.archive.umich.edu:/mac/util/compression/macgzip0.3b2.sit.hqx (Mac);
- src.doc.ic.ac.uk:/computing/systems/mac/info-mac/cmp/mac-gzip-022.hqx;
- mac.archive.umich.edu:/mac/development/source/macgzip0.2src.cpt.hqx.

It is suggested that any decompression code that is derived from GZIP be contained in a separate library to protect proprietary code in the remainder of each reader (see Figure C.1).



**Figure C.1—Recommended code partitioning for incorporating GZIP**

<sup>8</sup>The author and copyright owner of the GZIP program is Jean-loup Gailly (jloup@chorus.fr). The GZIP program is free software; it can be redistributed and/or modified under the terms of the GNU General Public License. The GNU General Public License is available from the Free Software Foundation, Inc., 675 Massachusetts Ave., Cambridge, MA 02139, USA.

## Annex D

(informative)

### Binary STIL data format

#### D.1 Binary STIL considerations

The original mandate of the STIL working group was to:

- a) Solve the high-density vector transportation problems to testers (e.g., the “gigabit problem”);
- b) Optimize the “Tools to Testers” binary and ASCII test vector formats.

These mandates address the transfer, storage, and processing of huge amounts of “pattern” information associated with the design and testing of complex digital VLSI circuits. Primarily, this means minimizing the time it takes for a consumer of this information to read/process the file. Secondly, the file size should be minimized.

The binary representation was perceived as only required for the pattern/vector data. This is the primary data volume contributor to the “gigabit problem.” All other test constructs (e.g., signals, timings, etc.) would have an ASCII only representation. In addition, the pattern/vector data would not be limited to being binary only; it still has an ASCII representation.

In seeking a binary representation, the following issues had to be considered/met:

- a) Read/translation time efficiency. The time required to read and translate the file must be minimized.
- b) Storage size efficiency. The file size must be minimized to ease disk space requirements and network throughput.
- c) Interchangeability. Writers and readers must operate in a heterogeneous environment.
- d) Flexibility and extensibility. The format must be flexible to accommodate user extensions and extendable to accommodate future revisions.
- e) No information lost. All pertinent information must be preserved. Formatting may be lost from an ASCII equivalent.
- f) Provide direct access (to be considered but not required). Can a format allow for direct or pseudo-direct access of information, or must the complete file be read?
- g) Load time (to be considered but not required). Can a common binary be devised which allows for direct loading into testers, effectively eliminating the translation step?

Possible binary representations include:

- **Compaction.** A compact binary could be devised which is similar to the ASCII definition, only represented in the minimum number of bits.
- **Compression.** A compressed binary could be devised using run-length encoding and/or Huffman codes (like GNU’s GZIP and UNIX Compress). Run-length encoding reduces serial redundancies, and is generally most effective in vertical (per signal) vs. horizontal (per vector) representations. Huffman codes replace common sequences with a small number of bits. Less common sequences use either more bits or are represented without compression. Huffman codes have the following characteristics: decompression time is independent of the compression, and it is generally possible for the writer to achieve higher levels of compression by spending more time identifying the “common sequences.”

Run-length encoding would be custom defined. Huffman codes could be custom tailored for the binary data, or generic implementations (GNU's GZIP or UNIX Compress) could be used.

- Compaction and Compression. A binary could be devised which encodes the data into the minimum number of bits, and then compresses common byte sequences. Again, the compression could be custom implementations or could utilize generic implementations.

The conclusion of the STIL binary subgroup was to utilize only data compression rather a combination of compaction and compression. Compression alone achieves:

- A minimized file size;
- Minimized read time (decompression time offset by reduced read time, which is negligible relative to overall translation process);
- Cross-platform portability;
- Flexibility and extensibility;
- Total persistence of all ASCII information;
- Applicability to all STIL data, not just Pattern data.

The GZIP compression format (deflate) was selected over the UNIX Compress format or a custom Huffman codes implementation. The GZIP compression format is:

- Very efficient;
- Available in cross-platform implementations;
- Copyright protected with provisions for freely copying and distributing (see Annex C).

## D.2 Binary STIL conclusions

The results of the binary issue evaluations which led to the GZIP compression format conclusion were:

- a) Read/translation time. This requirement was subjective, with no specific time per data quantified. Read time is inversely proportional to translation time. The smaller the file, the faster the read time. However, a compressed file adds to the translation time when performing the decompression. This decompression time may be offset by transmission times for a smaller file when operating in a network environment.

Direct mapping into binary (compaction) is impacted by the indirection within STIL (which provides its flexibility). Each signal would require a separate symbol table to associate its WaveformChars to binary codes. Symbol tables would also be required for the Signals, SignalGroups, and WaveformTable names.

The only benchmark in this area was Teradyne's translation of their ASCII source vectors to their binary format (LVMDB), compared to translating IBM "compact" binary source vectors to LVMDB. The binary translation is approximately one-third faster. However, when considering STIL, this ratio would be reduced by the additional tasks (also applicable to ASCII) of:

- 1) WaveformChar association and resolution to the active WaveformTable;
- 2) Potential recycling of vectors;
- 3) Scan data normalization and potential merging of unload/load data;
- 4) Pattern splitting due to possible resource constraints (timing, buffer sizes, etc.).

- b) Storage size efficiency. This requirement was subjective, with no specific data sizes quantified. Compression using the GZIP format provides good file size reduction, providing two to three times the reduction over compaction alone. Compression of STIL ASCII vs. compression of “compact” binary results in similar file sizes (see Table D.1). By utilizing pipes or incorporating the decompression into STIL readers, the large uncompressed file is not stored on disk or transmitted across networks. A custom compression implementation may yield further reduction, but was outside of the group’s resources (prototype developers, availability of proprietary “real” test data, and testing time).
- c) Interchange. Data compression in the GZIP format operates at the byte level and is, therefore, platform independent. Binary compaction requires integers which have byte ordering implications. Possible sceneries include: (A) defining a canonical ordering requiring incompatible machines to perform byte manipulations, or (B) allowing any byte ordering in the file and requiring all readers to perform byte manipulations, if necessary.  
Binary compaction requires alignment. Each platform and/or compiler may pack data structures differently. This includes “when is byte-alignment used,” as well as word-alignment and double-word alignment.
- d) Flexibility and extensibility. A compact binary could be devised to allow for user extensions and future additions. Vertical run-length encoding compression may be impacted by non-signal-oriented extensions. New extensions will be transparent to the GZIP compression format.
- e) No information lost. Compact binary has the potential to lose information, such as formatting (base, whitespace, etc.) and comments. The GZIP compression format loses no information.
- f) Direct access. Requires storing the data into known block sizes/structures. Predefined block sizes may not be advantageous to the final file size. Delta changes and variable length scan data may impact savings. Non-vector information (WaveformTable references, comments, extensions, etc.) may impede block definitions. Compression inhibits direct access.
- g) Load time. A proposal arose to have the binary directly loadable by ATE testers. However, this may hinder individual ATE companies from developing new features/architectures. Also, load time could be impacted by on-the-fly translation tasks such as:
  - 1) Recycled vectors for complex timings;
  - 2) Scan normalization and merging;
  - 3) Resource constraints (timing and vectors).

Table D.1 illustrates some metrics of test cases created in a “compact” binary vs. STIL ASCII, with GZIP compression and GUNZIP decompression time.

**Table D.1—Comparison of formats**

	STIL.asc STIL.asc.gz (Mb)	GUNZIP time	compact.bin <sup>a</sup> compact.bin.gz (Mb)	GUNZIP time
Micro1 Scan based 174 Mb ASCII data	56.6 17.4	49.26 s	43.3 17.1	43.28 s
Micro2 Scan based 359 Mb ASCII data	115.9 39.6	3 m 33 s	82.2 38	3 m 5.38 s
Micro3 Functional based 278.7 Mb source data	35.0 2.1	15.72 s	30.4 3.1	18.31 s
Asic1 Scan based <sup>b</sup>	21.7 8.8	22.68 s	33.2 8.5	58.36 s
Asic2 Scan based <sup>b</sup>	90.1 35.0	1 m 30.92 s	72.6 25.27	1 m 9.94 s

<sup>a</sup>Internal IBM format; names replaced with integers, delta changes, normalized scans, and minimum grouping capability.

<sup>b</sup>Only processed a subset of the patterns from a binary database; file size for just the subset of patterns is unknown.

## Annex E

(informative)

### LS245 design description

The LS245 is an octal bus transceiver. Figure E.1 shows one structural representation for this model; a Verilog<sup>TM</sup> representation<sup>9</sup> is shown in Figure E.2. This information is provided solely to assist comprehension of the examples in the tutorial (Clause 5) that are based on this design.

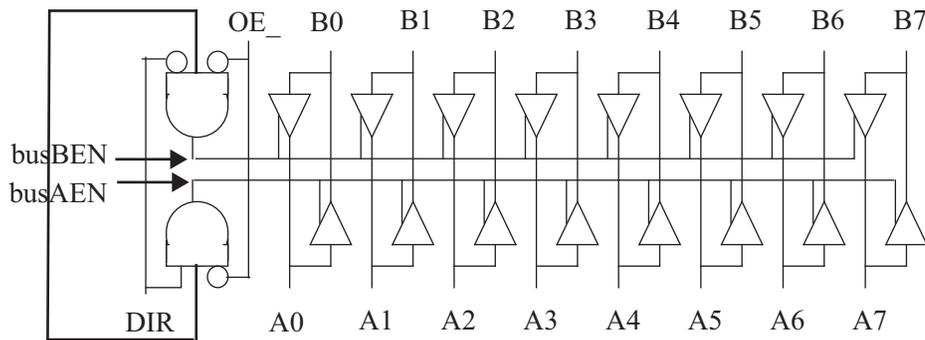


Figure E.1—LS245 structural model

<sup>9</sup>This refers to IEEE Std 1364-1995, IEEE Standard Description Language Based on the Verilog<sup>TM</sup> Hardware Description Language. IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (<http://www.standards.ieee.org/>).

```
/* this model makes use of the devices:
AND2 (output, in1, in2)      -- 2-input AND gate
INV  (output, input)        -- single-input inverter
TBUF (output, input, enable) -- float-state driver
*/
module ls245
(DIR, OE_, A0, A1, A2, A3, A4, A5, A6, A7, B0, B1, B2, B3, B4, B5, B6, B7);
  input DIR, OE_;
  inout A0, A1, A2, A3, A4, A5, A6, A7, B0, B1, B2, B3, B4, B5, B6, B7;

  AND2 ibusA (busAEN, DIR, notOE_);
  INV  inotOE_ (notOE_, OE_);
  INV  inotDIR (notDIR, DIR);
  AND2 ibusB (busBEN, notDIR, notOE_);
  TBUF iA0 (A0, B0, busBEN); TBUF iA1 (A1, B1, busBEN);
  TBUF iA2 (A2, B2, busBEN); TBUF iA3 (A3, B3, busBEN);
  TBUF iA4 (A4, B4, busBEN); TBUF iA5 (A5, B5, busBEN);
  TBUF iA6 (A6, B6, busBEN); TBUF iA7 (A7, B7, busBEN);

  TBUF iB0 (B0, A0, busAEN); TBUF iB1 (B1, A1, busAEN);
  TBUF iB2 (B2, A2, busAEN); TBUF iB3 (B3, A3, busAEN);
  TBUF iB4 (B4, A4, busAEN); TBUF iB5 (B5, A5, busAEN);
  TBUF iB6 (B6, A6, busAEN); TBUF iB7 (B7, A7, busAEN);
endmodule
```

**Figure E.2—LS245 Verilog™ representation**

## Annex F

(informative)

### STIL FAQs and language design decisions

Some common questions arise as new individuals come to understand STIL. This annex covers some of the most frequently asked questions about STIL and STIL structures.

- 1) STIL allows some keywords and statement fragments to be reused in different contexts. Why?

During STIL development, the Working Group felt that the usage of common terms was often better than attempting to define a new term. This has resulted in the reuse of some terms in slightly different contexts. The Group felt this to be less confusing than the alternative process of creating names.

Also, certain statement fragments, noticeably block-definition statements and block-reference statements, do have much the same form. For example, “timing one {}” declares a timing block, and “timing one;” references or uses that timing block in a PatternExec. This causes the keyword “timing” to be applied differently, depending on the type of statement in which it occurs. Again, the Working Group’s decision was to accept this issue rather than attempt to create names.

- 2) STIL uses some of the WGL states, but not the “data” and “inverted data” states of WGL (S and Q). Why not? And what about surround-by-complement definitions?

WGL “data-driven” characters are restricted to “0” and “1” values only. During development of STIL this restriction was questioned, and an alternative implementation was defined that supports the mapping of more-than-two states into the waveform. This solution was seen to be more general and, therefore, preferred. Also, by not using “S” and “Q” environments, a very uniform WaveformChar-mapping environment is defined; all Vector data maps back to waveforms through WaveformChar resolution only, and is not dependent on event characters used in the waveform to define how to interpret the data. Finally, “inverted-data” relationships, which are necessary to define surround-by-complement waveforms, are supported in the STIL strategy by reversing the order of events presented in the waveform.

- 3) In waveform definitions, STIL allows a signal to be potentially referenced in several waveform definitions; for instance, “SIG1 { 01 {} } SIG1 { HL {} }”. It would seem that the declaration “SIG1 { 01 {} HL {} }” would be a more direct way to accomplish the same thing (and is also currently supported). Why not require signals to be referenced once in the Waveform block (this question from Intel review of 0.15)?

It is true that both forms are allowed in STIL, and the reason is to facilitate the use of groups in the waveform definition. When groups are used, it may be natural to define a set of output characteristics across all signals that have output capability, and to define a set of input characteristics across all signals that have input capability. InOut (bidirectional) signals would need to be defined separately if this requirement were in place, as they share characteristics in both groups. This would require additional, redundant waveform definitions solely because the language disallowed multiple references to a single signal.

- 4) There seems to have been a consideration made for  $I_{DDQ}$  testing, with the  $I_{DDQ}$  TestPoint statement. But there are no other tests defined. Why this one? Why not others?

The Working Group decided that an  $I_{DDQ}$  stop identification was straightforward to define. This test operates on the entire state of the design, whereas most other tests require some reference to specific signals and perhaps even external measurement criteria. In order to define other tests, additional parameters of those tests

have to be accounted for, and this was determined to be outside the scope of this project, as currently defined. It is also the concept of the Working Group that all other test environments, including any additional parameters that must be considered for those tests, can be defined in STIL through the use of Ann and User-Defined Keywords. While this does not lead to a shared common solution, it allows those with critical need to put in place the information, and facilitates requirement discussions for future revisions of the language.

- 5) One version of STIL contained reference statements in the Pattern block, to allow SignalGroups, Procedures and MacroDefs to be specified directly from the Pattern and not solely through the PatternBurst. Why was this removed?

In a Working Group meeting on April 18-19, 1997, constructs supporting referencing SignalGroups, Procedures, and MacroDefs from the Pattern block were removed. There were several motivations behind this change:

- a) As the language stood with those constructs, procedure processing could not occur until Patterns were being read. This means the processing flow would have to maintain this data until all the patterns were read, as procedure behavior could be different if SignalGroups were different. The new environment allows procedure processing to occur once a PatternBurst (or PatternExec as well, depending on issues with timing resolution) is read that references that procedure.
  - b) By localizing the references, it is much easier to identify the context in which something is used; a user doesn't have to check two different locations.
  - c) "Binding" of signals is defined much earlier in the processing, rather than being deferred to the Pattern (which is basically very late binding).
  - d) Because STIL supports multiple and hierarchical PatternBursts, there are no additional features provided by resolution in the Pattern that are not available with PatternBurst support.
- 6) Constantly needing to reference signals in Vector statements seems to be expensive. How about a way to indirectly reference signals?

This was considered in a meeting held on June 6, 1996. The information below was discussed. This construct was removed because of complications concerning consistent handling of hex/decimal options for this statement:

#### Incremental Data Vector (V) Statement

The Incremental Data Vector statement contains only *vec\_data*. The *sigref\_expr* for this data is taken relative to the previous Vector statement. All signals in the previous vector are used as a group reference for the *vec\_data* present in this statement. The syntax of an Incremental Data Vector statement is:

```
( LABEL : ) V(ector) vec_data;
```

For example:

```
w wavedef;
v { special_1 = 1; special_2 = 1; };
v 10; //incremental vector, using special_1 and special_2;
v 11; //next incremental vector.
```

The Incremental Data Vector statement may be used only after a complete Vector {} statement; the signals referenced in that complete Vector are used as the signals (in the same order) for the data in the Incremental Data Vector. In this statement, all data is provided by WaveformChars only to the individual signals; there are no hex or decimal options for the *vec\_data*.

- 7) Scan-in and Scan-out constructs are defined to apply to single signals only. Why?

While the language constructs can be defined to support multiple-signal situations, the complications with multi-bit options, etc., caused the Working Group to decide that scan data should be defined individually.

- 8) The uniform use of the term “sigref\_expr” implies that groups can be defined using expressions anywhere. Is this true?

Yes, the language does allow references to signal expressions any place a multiple-signal reference can be made. This was not always a part of the language; during early discussions, when a binary format was being postulated, the creation of “groups on-the-fly” was considered counter to being able to reference all definitions from a central location, and groups on-the-fly were not allowed. However, when the binary discussion moved to a more general compress option, the ability to define groups on-the-fly was supported. It must be noted that while groups can be defined “on-the-fly,” they can only be assigned names when defined inside the SignalGroups block.

- 9) STIL is defined to be case-sensitive. However, communication between different tool environments may make this decision a difficult thing to enforce, particularly on signal names. For an environment meant to transport data between different tool sets, case-sensitivity may be more of a pain than a value. Why make this restriction?

This is an issue when moving data between different environments, and the concern is valid. Different environments will preserve information—particularly signal names—differently. However, the concern is not limited to case. Special characters may result in tool-dependent interpretation of additional meaning behind names; this meaning doesn’t get transported between systems either. Therefore, it is highly recommended that a single source of signal name definitions in the STIL environment be supported by all tools using STIL data, and that individual tools, if operating from a different database internally, be prepared to map into the STIL names. Name mapping can also be supported through single-signal group names, which would allow tools to generate information using the names defined in their respective databases. However, even with this capability, the creation of the name-aliasing group section still requires a correlation function back to the STIL names. This is most of the effort required to address the name-correlation issue anyway, and the benefits of preserving tool-dependent names in STIL is probably minimal.

- 10) The SignalGroups statement, when used to reference a group, can occur in the PatternBurst and the Timing sections. If it doesn’t occur in the PatternBurst, will groups used in the Patterns be resolved to definitions found from the reference in the Timing?

No. Timing data and Pattern data are considered parallel streams of information; definitions of Timing information must be complete at the Timing block level (albeit final resolution of timing values occurs in the PatternExec), and definition of Pattern information is complete in the PatternBurst. If a group is used that is not defined at either of those points for Timing and Pattern data, respectfully, then that usage is an error.

- 11) What is the order of execution of PatternExecs?

There is no order of execution defined for PatternExecs in STIL today. The actual assembly of a complete test program (including levels, binning, order, and application of PatternExecs, etc.) is left to the test equipment vendors support.

- 12) The Stop statement appears in both the Patterns and PatternBurst. Does it have the same effect in either location?

The Stop statement, when present in the Patterns, causes complete termination of the test procedure. This is because there is no context to link this pattern execution into a sequence at this point. The Stop statement may also be defined in two different locations in the PatternBurst. When the Stop statement is defined

outside the PatList statement in the PatternBurst, then that Stop statement terminates execution of the burst, which will allow a subsequent burst to start executing if one is defined. If the Stop statement is defined internal to the PatList statement, then the Stop is applied only to the Pattern that contains the matching label when that label is executed. This causes that Pattern to stop executing; however, any subsequent Patterns in the PatternBurst will execute after that Stop.