



IEC 62525

Edition 1.0 2007-11

INTERNATIONAL STANDARD

IEEE 1450™

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





THIS PUBLICATION IS COPYRIGHT PROTECTED

Copyright © 2007 IEEE

All rights reserved. IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by the Institute of Electrical and Electronics Engineers, Inc.

Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the IEC Central Office.

Any questions about IEEE copyright should be addressed to the IEEE. Enquiries about obtaining additional rights to this publication and other information requests should be addressed to the IEC or your local IEC member National Committee.

IEC Central Office
3, rue de Varembe
CH-1211 Geneva 20
Switzerland
Email: inmail@iec.ch
Web: www.iec.ch

The Institute of Electrical and Electronics Engineers, Inc
3 Park Avenue
US-New York, NY10016-5997
USA
Email: stds-info@ieee.org
Web: www.ieee.org

About the IEC

The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications

The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigenda or an amendment might have been published.

- Catalogue of IEC publications: www.iec.ch/searchpub

The IEC on-line Catalogue enables you to search by a variety of criteria (reference number, text, technical committee,...). It also gives information on projects, withdrawn and replaced publications.

- IEC Just Published: www.iec.ch/online_news/justpub

Stay up to date on all new IEC publications. Just Published details twice a month all new publications released. Available on-line and also by email.

- Electropedia: www.electropedia.org

The world's leading online dictionary of electronic and electrical terms containing more than 20 000 terms and definitions in English and French, with equivalent terms in additional languages. Also known as the International Electrotechnical Vocabulary online.

- Customer Service Centre: www.iec.ch/webstore/custserv

If you wish to give us your feedback on this publication or need further assistance, please visit the Customer Service Centre FAQ or contact us:

Email: csc@iec.ch

Tel.: +41 22 919 02 11

Fax: +41 22 919 03 00



IEC 62525

Edition 1.0 2007-11

INTERNATIONAL STANDARD

IEEE 1450™

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

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

PRICE CODE **XG**

ICS 25.040;19.080

ISBN 2-8318-9337-2

CONTENTS

FOREWORD	6
IEEE Introduction	9
1. Overview	10
1.1 Scope	12
1.2 Purpose	13
2. References	13
3. Definitions, acronyms, and abbreviations	13
3.1 Definitions	13
3.2 Acronyms and abbreviations	16
4. Structure of this standard	16
5. STIL orientation and capabilities tutorial (informative)	17
5.1 Hello Tester	17
5.2 Basic LS245	22
5.3 STIL timing expressions/"Spec" information	26
5.4 Structural test (scan)	31
5.5 Advanced scan	35
5.6 IEEE Std 1149.1-1990 scan	41
5.7 Multiple data elements per test cycle	46
5.8 Pattern reuse/direct access test	50
5.9 Event data/non-cyclized STIL information	54
6. STIL syntax description	64
6.1 Case sensitivity	64
6.2 Whitespace	64
6.3 Reserved words	64
6.4 Reserved characters	66
6.5 Comments	67
6.6 Token length	67
6.7 Character strings	67
6.8 User-defined name characteristics	68
6.9 Domain names	68
6.10 Signal and group name characteristics	69
6.11 Timing name constructs	69
6.12 Number characteristics	69
6.13 Timing expressions and units (time_expr)	70
6.14 Signal expressions (sigref_expr)	72
6.15 WaveformChar characteristics	73
6.16 STIL name spaces and name resolution	74
7. Statement structure and organization of STIL information	76
7.1 Top-level statements and required ordering	68
7.2 Optional top-level statements	70
7.3 STIL files	70

8.	STIL statement.....	79
8.1	STIL syntax.....	79
8.2	STIL example.....	79
9.	Header block	80
9.1	Header block syntax.....	80
9.2	Header example	80
10.	Include statement	80
10.1	Include statement syntax.....	81
10.2	Include example.....	81
10.3	File path resolution with absolute path notation	81
10.4	File path resolution with relative path notation	81
11.	UserKeywords statement	82
11.1	UserKeywords statement syntax.....	82
11.2	UserKeywords example.....	82
12.	UserFunctions statement.....	82
12.1	UserFunctions statement syntax	83
12.2	UserFunctions example.....	83
13.	Ann statement	83
13.1	Annotations statement syntax	83
13.2	Annotations example	83
14.	Signals block.....	83
14.1	Signals block syntax	84
14.2	Signals block example	86
15.	SignalGroups block.....	86
15.1	SignalGroups block syntax	86
15.2	SignalGroups block example	87
15.3	Default attribute values	87
15.4	Translation of based data into WaveformChar characters.....	88
16.	PatternExec block	89
16.1	PatternExec block syntax.....	90
16.2	PatternExec block example.....	90
17.	PatternBurst block.....	90
17.1	PatternBurst block syntax	91
17.2	PatternBurst block example	92

18.	Timing block and WaveformTable block	92
	18.1 Timing and WaveformTable syntax	93
	18.2 Waveform event definitions	96
	18.3 Timing and WaveformTable example	98
	18.4 Rules for timed event ordering and waveform creation	99
	18.5 Rules for waveform inheritance	102
19.	Spec and Selector blocks	103
	19.1 Spec and Selector block syntax	103
	19.2 Spec and Selector block example	105
20.	ScanStructures block	106
	20.1 ScanStructures block syntax	107
	20.2 ScanStructures block example	108
21.	STIL Pattern data	109
	21.1 Cyclized data	109
	21.2 Multiple-bit cyclized data	110
	21.3 Non-cyclized data	111
	21.4 Scan data	111
	21.5 Pattern labels	112
22.	STIL Pattern statements	112
	22.1 Vector (V) statement	112
	22.2 WaveformTable (W) statement	113
	22.3 Condition (C) statement	113
	22.4 Call statement	114
	22.5 Macro statement	114
	22.6 Loop statement	115
	22.7 MatchLoop statement	115
	22.8 Goto statement	116
	22.9 BreakPoint statements	116
	22.10 IDDQTestPoint statement	116
	22.11 Stop statement	117
	22.12 ScanChain statement	117
23.	Pattern block	117
	23.1 Pattern block syntax	117
	23.2 Pattern initialization	118
	23.3 Pattern examples	118
24.	Procedures and MacroDefs blocks	118
	24.1 Procedures block	119
	24.2 Procedures example	120
	24.3 MacroDefs block	120
	24.4 Scan testing	120
	24.5 Procedure and Macro Data substitution	121

Annex A (informative) Glossary 125

Annex B (informative) STIL data model..... 126

Annex C (informative) GNU GZIP reference 131

Annex D (informative) Binary STIL data format 132

Annex E (informative) LS245 design description 136

Annex F (informative) STIL FAQs and language design decisions 138

Annex G (informative) List of participants..... 142

INTERNATIONAL ELECTROTECHNICAL COMMISSION

STANDARD TEST INTERFACE LANGUAGE (STIL) FOR DIGITAL TEST VECTOR DATA

FOREWORD

- 1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
- 2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
- 3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
- 4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
- 5) IEC provides no marking procedure to indicate its approval and cannot be rendered responsible for any equipment declared to be in conformity with an IEC Publication.
- 6) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC/IEEE 62525 has been processed through Technical Committee 93: Design automation.

The text of this standard is based on the following documents:

IEEE Std	FDIS	Report on voting
1450(1999)	93/247/FDIS	93/258/RVD

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

The committee has decided that the contents of this publication will remain unchanged until the maintenance result date indicated on the IEC web site under "<http://webstore.iec.ch>" in the data related to the specific publication. At this date, the publication will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

IEC/IEEE Dual Logo International Standards

This Dual Logo International Standard is the result of an agreement between the IEC and the Institute of Electrical and Electronics Engineers, Inc. (IEEE). The original IEEE Standard was submitted to the IEC for consideration under the agreement, and the resulting IEC/IEEE Dual Logo International Standard has been published in accordance with the ISO/IEC Directives.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The 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 the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an IEC/IEEE Dual Logo International Standard is wholly voluntary. The IEC and IEEE disclaim 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 this, or any other IEC or IEEE Standard document.

The IEC and IEEE do not warrant or represent the accuracy or content of the material contained herein, and expressly disclaim 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 herein is free from patent infringement. IEC/IEEE Dual Logo International Standards documents are supplied "AS IS".

The existence of an IEC/IEEE Dual Logo International 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 IEC/IEEE Dual Logo International 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 five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, 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 this document available, the IEC and IEEE are not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Neither the IEC nor IEEE is undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEC/IEEE Dual Logo International Standards or IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations – Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received 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 interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of IEC/IEEE Dual Logo International Standards are welcome from any interested party, regardless of membership affiliation with the IEC or IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA and/or General Secretary, IEC, 3, rue de Varembé, PO Box 131, 1211 Geneva 20, Switzerland.

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.

NOTE – Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

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

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

IEEE Introduction

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.

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.

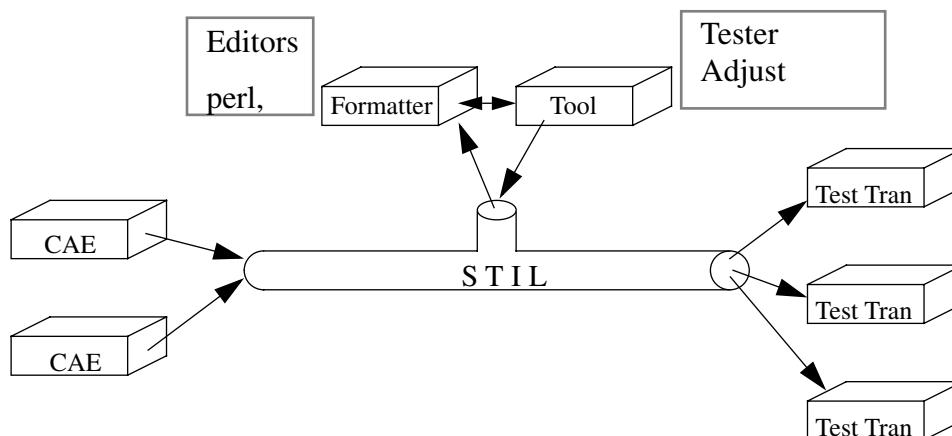


Figure 1—A conduit for transporting data from CAE to ATE

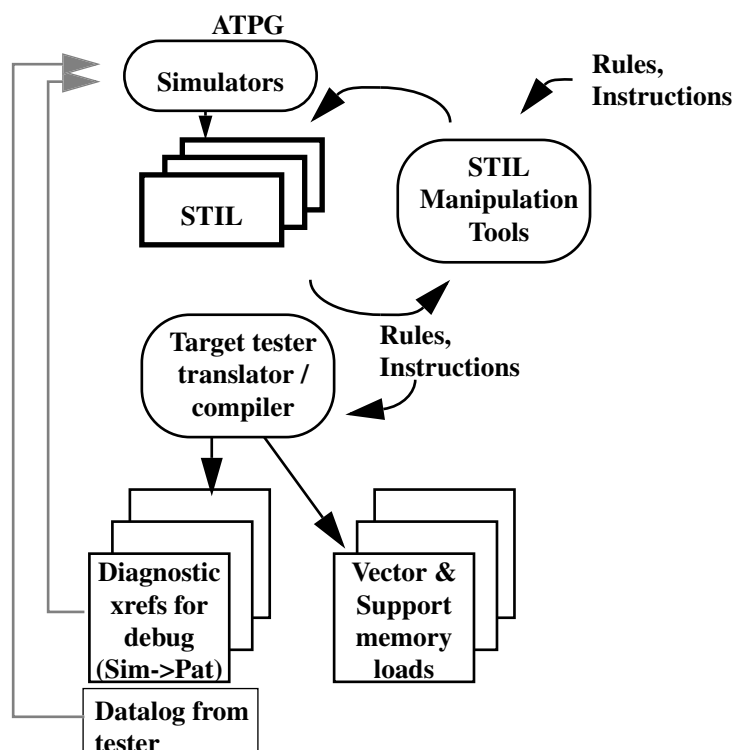


Figure 2—STIL usage model

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)¹ 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.²

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.³

ISO/IEC 9899:1999, Programming languages—C.⁴

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.

¹The 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.

²IEEE 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/>).

³ISO publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, 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/>).

⁴IEC publications are available from the Sales Department of the International Electrotechnical Commission, Case Postale 131, 3, rue de Varembé, 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

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

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.

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

```

① STIL 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=0000000000LLLLLLLL; }
   V { ALL=0010000000HLLLLLLLL; }
   V { ALL=0001000000LHLLLLLLLL; }
   V { ALL=0000100000LLHLLLLLLLL; }
   V { ALL=0000010000LLLHLLLLLLLL; }
   V { ALL=0000001000LLLLLHLLLL; }
   V { ALL=0000000100LLLLLLHLL; }
   V { ALL=0000000010LLLLLLHLL; }
   V { ALL=00000000010LLLLLLHL; }
   V { ALL=00000000001LLLLLLHLH; }
} //end Pattern "hello tester pattern"

```

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_ 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:

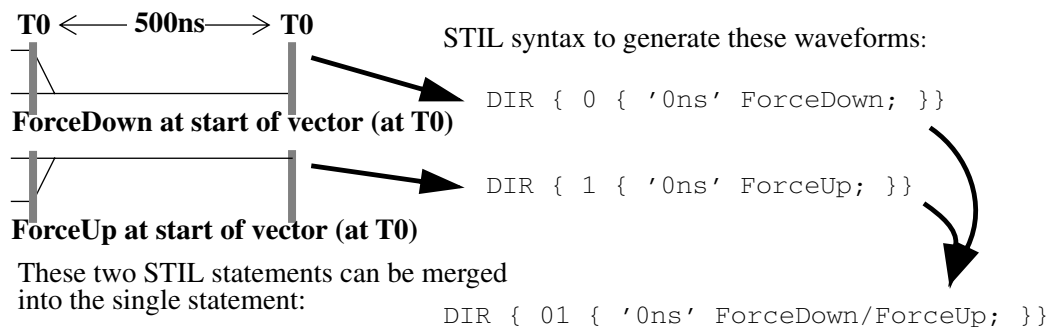


Figure 4—Waveforms associated with signal DIR

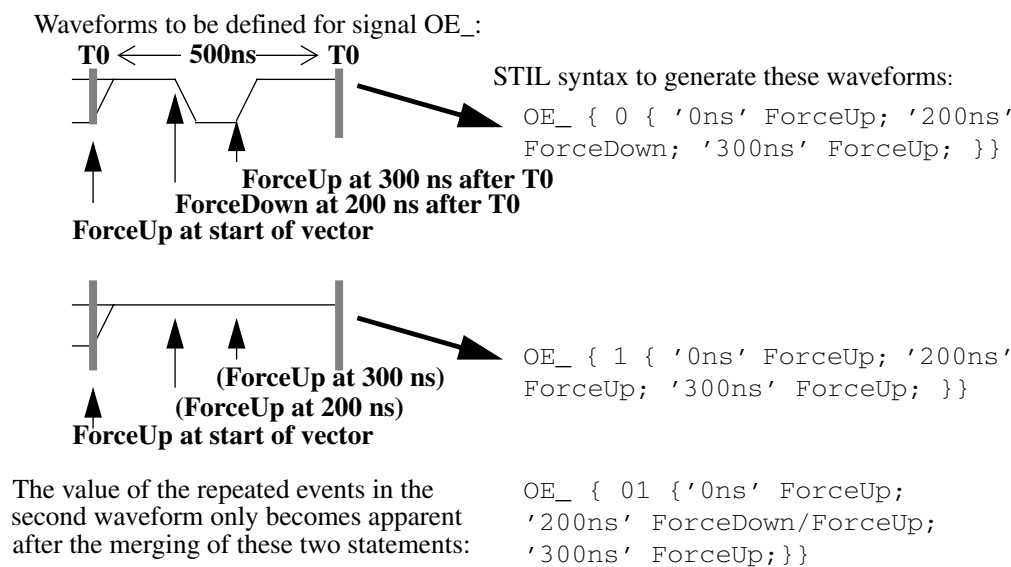


Figure 5—Waveforms associated with the signal OE_

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;

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; 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 {
  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
```

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=00ZZZZZZZZZZZZZZZZ; }
    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 “1011”	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.

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;
}

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+tpHz' 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+tpH1' l;
            '@+strobe_width' X; }
        H { '0ns' Z; '0ns' X; 'IN_MARK+tpHh' 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_ttyp,
    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=00DDDDDDDDXXXXXXXXXX; }
    // 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.

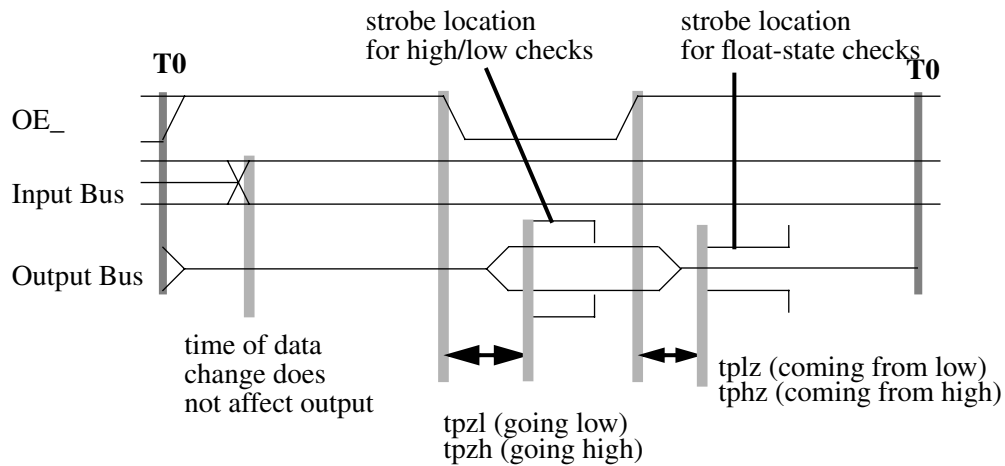


Figure 10—Waveform characteristics of WaveformTable pulsed_oe

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;

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;
}

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

```

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=0011ZZZZZZZZZZZZZZZZ; } // define all signals
        Shift { V { MASTER=P; SLAVE=P; SI1=#; SI2=#; SO1=#; SO2=#; } }
    }
} // end procedures

Pattern "scan" {
    W one;
    V { ALL=00ZZZZZZZZZZZZZZZZ; } // define all signals
    Macro "scan" {
        SI1=00000000000000000000000000000000; ⑥
        SI2=111111111111111111111111111111111111; }
    V { MASTER=1; OE_=1; B0=H; } ⑦
    Call "scan" {
        SO1=LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL; 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 $\backslash$ 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 “ $\backslash$ l,” or it may default to the “ScanIn” or “ScanOut” length if the length information is specified. “ $\backslash$ l10” 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.

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

```

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

```

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 {
    so1=HLHLLLHLHL;
    so2=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 {
    so1=HLHLHLHLHL;
    so2=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 {
    so1=LLLLHLHLLL;
    so2=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 {
    so1=LLHLLLHLLL;
    so2=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 1149.1-1990⁵ scan features, including applying scan data outside of the Shift block.

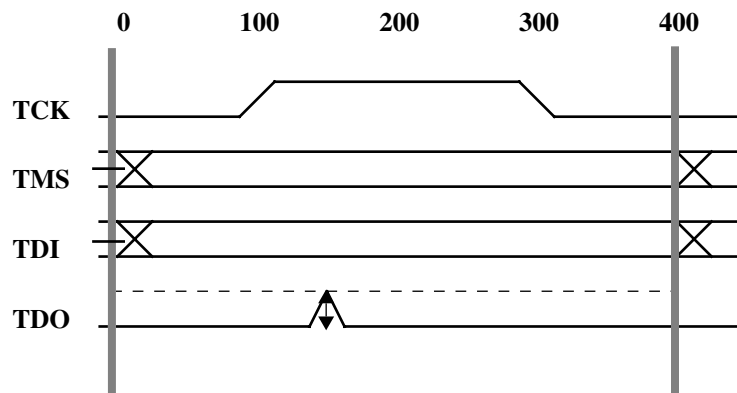


Figure 13—IEEE Std 1149.1-1990 example timings

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).

⁵IEEE 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 {

①

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*)