

IEEE Standard for Extensions to Standard Test Interface Language (STIL) for Semiconductor Design Environments

IEEE Computer Society

Developed by the
Test Technology Standards Committee

IEEE Std 1450.1™-2025
(Revision of IEEE Std 1450.1-2005)

STANDARDS

IEEE Standard for Extensions to Standard Test Interface Language (STIL) for Semiconductor Design Environments

Developed by the

Test Technology Standards Committee
of the
IEEE Computer Society

Approved 12 February 2025

IEEE SA Standards Board

Abstract: An interface between digital test generation tools and test equipment is provided by Standard Test Interface Language (STIL). Extensions to the test interface language (contained in this standard) are defined that facilitate the use of the language in the design environment and facilitate the use of the language for large designs encompassing subdesigns with reusable patterns.

Keywords: advanced scan architecture, core, environment, fail feedback, IEEE 1450.1™, lockstep, parallel patterns, parameterized data, pattern tiling, pragma, signal variable, SoC, system on chip, test protocol

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

Copyright © 2025 by The Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 26 June 2025. Printed in the United States of America.

IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by The Institute of Electrical and Electronics Engineers, Incorporated.

PDF: ISBN 979-8-8557-2140-9 STD27889
Print: ISBN 979-8-8557-2141-6 STDPD27889

IEEE prohibits discrimination, harassment, and bullying.

For more information, visit <https://www.ieee.org/about/corporate/governance/p9-26.html>.

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

Important Notices and Disclaimers Concerning IEEE Standards Documents

IEEE Standards documents are made available for use subject to important notices and legal disclaimers. These notices and disclaimers, or a reference to this page (<https://standards.ieee.org/ipr/disclaimers.html>), appear in all IEEE standards and may be found under the heading “Important Notices and Disclaimers Concerning IEEE Standards Documents.”

Notice and Disclaimer of Liability Concerning the Use of IEEE Standards Documents

IEEE Standards documents are developed within IEEE Societies and subcommittees of IEEE Standards Association (IEEE SA) Board of Governors. IEEE develops its standards through an accredited consensus development process, which brings together volunteers representing varied viewpoints and interests to achieve the final product. IEEE standards are documents developed by volunteers with scientific, academic, and industry-based expertise in technical working groups. Volunteers involved in technical working groups are not necessarily members of IEEE or IEEE SA and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

IEEE makes no warranties or representations concerning its standards, and expressly disclaims all warranties, express or implied, concerning all standards, including but not limited to the warranties of merchantability, fitness for a particular purpose and non-infringement. IEEE Standards documents do not guarantee safety, security, health, or environmental protection, or compliance with law, or guarantee against interference with or from other devices or networks. In addition, IEEE does not warrant or represent that the use of the material contained in its standards is free from patent infringement. IEEE Standards documents are supplied “AS IS” and “WITH ALL FAULTS.”

Use of an IEEE standard is wholly voluntary. The existence of an IEEE standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

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

IN NO EVENT SHALL IEEE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO: THE NEED TO PROCURE SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE PUBLICATION, USE OF, OR RELIANCE UPON ANY STANDARD, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND REGARDLESS OF WHETHER SUCH DAMAGE WAS FORESEEABLE.

Translations

The IEEE consensus balloting process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English language version published by IEEE is the approved IEEE standard.

Use by artificial intelligence systems

In no event shall material in any IEEE Standards documents be used for the purpose of creating, training, enhancing, developing, maintaining, or contributing to any artificial intelligence systems without the express, written consent of IEEE SA in advance. “Artificial intelligence” refers to any software, application, or other system that uses artificial intelligence, machine learning, or similar technologies, to analyze, train, process, or generate content. Requests for consent can be submitted using the Contact Us form.

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE SA Standards Board Operations Manual is not, and shall not be considered or inferred to be, the official position of IEEE or any of its committees and shall not be considered to be, or be relied upon as, a formal position of IEEE or IEEE SA. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that the presenter’s views should be considered the personal views of that individual rather than the formal position of IEEE, IEEE SA, the Standards Committee, or the Working Group. Statements made by volunteers may not represent the formal position of their employer(s) or affiliation(s). News releases about IEEE standards issued by entities other than IEEE SA should be considered the view of the entity issuing the release rather than the formal position of IEEE or IEEE SA.

Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE or IEEE SA. However, **IEEE does not provide interpretations, consulting information, or advice pertaining to IEEE Standards documents.**

Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its Societies and subcommittees of the IEEE SA Board of Governors are not able to provide an instant response to comments or questions, except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in evaluating comments or revisions to an IEEE standard is welcome to join the relevant IEEE SA working group. You can indicate interest in a working group using the Interests tab in the Manage Profile & Interests area of the [IEEE SA myProject system](https://development.standards.ieee.org/myproject-web/public/view.html#landing).¹ An IEEE Account is needed to access the application.

Comments on standards should be submitted using the [Contact Us](#) form.²

¹ Available at: <https://development.standards.ieee.org/myproject-web/public/view.html#landing>.

² Available at: <https://standards.ieee.org/about/contact/>.

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not constitute compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Data privacy

Users of IEEE Standards documents should evaluate the standards for considerations of data privacy and data ownership in the context of assessing and using the standards in compliance with applicable laws and regulations.

Copyrights

IEEE draft and approved standards are copyrighted by IEEE under U.S. and international copyright laws. They are made available by IEEE and are adopted for a wide variety of both public and private uses. These include both use by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making these documents available for use and adoption by public authorities and private users, neither IEEE nor its licensors waive any rights in copyright to the documents.

Photocopies

Subject to payment of the appropriate licensing fees, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400; <https://www.copyright.com/>. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Updating of IEEE Standards documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect.

Every IEEE standard is subjected to review at least every 10 years. When a document is more than 10 years old and has not undergone a revision process, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE standard.

In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit [IEEE Xplore](#) or [contact IEEE](#).³ For more information about the IEEE SA or IEEE's standards development process, visit the IEEE SA Website.

Errata

Errata, if any, for all IEEE standards can be accessed on the [IEEE SA Website](#).⁴ Search for standard number and year of approval to access the web page of the published standard. Errata links are located under the Additional Resources Details section. Errata are also available in [IEEE Xplore](#). Users are encouraged to periodically check for errata.

Patents

IEEE standards are developed in compliance with the [IEEE SA Patent Policy](#).⁵

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 by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE SA Website at <https://standards.ieee.org/about/sasb/patcom/patents.html>. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

IMPORTANT NOTICE

Technologies, application of technologies, and recommended procedures in various industries evolve over time. The IEEE standards development process allows participants to review developments in industries, technologies, and practices, and to determine what, if any, updates should be made to the IEEE standard. During this evolution, the technologies and recommendations in IEEE standards may be implemented in ways not foreseen during the standard's development. IEEE standards development activities consider research and information presented to the standards development group in developing any safety recommendations. Other information about safety practices, changes in technology or technology implementation, or impact by peripheral systems also may be pertinent to safety considerations during implementation of the standard. Implementers and users of IEEE Standards documents are responsible for determining and complying with all appropriate safety, security, environmental, health, data privacy, and interference protection practices and all applicable laws and regulations.

³ Available at: <https://ieeexplore.ieee.org/browse/standards/collection/ieee>.

⁴ Available at: <https://standards.ieee.org/standard/index.html>.

⁵ Available at: <https://standards.ieee.org/about/sasb/patcom/materials.html>.

Participants

At the time this standard was completed, the 1450.1 Working Group had the following membership:

Ric Dokken, *Chair*
Boon Chong Ang, *Vice Chair*
Cahyo Primawidodo, *Secretary*

Saif Aldeen Saad Alkadhim
Peter Bauwens
Gerald Chan

Anthony Coyette
Chris Green

Greg Maston
Kurt Neugebauer
Paul Reuter

The following members of the individual Standards Association balloting group voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Boon Chong Ang
Peter Bauwens
Paul Berndt
Kevin Cameron
John Cosley
Anthony Coyette
Ric Dokken

Heiko Ehrenberg
Werner Hoelzl
Greg Luri
Greg Maston
Ian McIntosh
Bansi Patel

Cahyo Primawidodo
Jhony Sembiring
Walter Struppler
Bambang Suparjo
David Thompson
Oren Yuen
Vladimir Zivkovic

When the IEEE SA Standards Board approved this standard on 12 February 2025, it had the following membership:

Lei Wang, *Chair*
Vacant Position, *Vice Chair*
David J. Law, *Past Chair*
Alpesh Shah, *Secretary*

Edward Au
Ted Burse
Xiaofeng (Alfred) Chen
Doug Edwards
Nehad El-Sherif
J. Travis Griffith
Deborah R. Hagar
Guido R. Hiertz

Ronald W. Hotchkiss
Tyler L. Jaynes
Thomas Koshy
Howard Li
Xiaohui Liu
Kevin W. Lu
Hiroshi Mano

Daleep C. Mohla
Annette D. Reilly
Robby Robson
Jon Walter Rosdahl
Daniel Sabin
F. Keith Waters
Sha Wei
Luyang (Eric) Zhang

Introduction

This introduction is not part of IEEE Std 1450.1-2025, IEEE Standard for Extensions to Standard Test Interface Language (STIL) for Semiconductor Design Environments.

The Standard Test Interface Language (STIL) was initially developed by an ad hoc consortium of automatic test equipment vendors (ATE), electronic design automation vendors (EDA), 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 scope of the initial STIL standard was limited to satisfy the basic needs of pattern definition. Additional capabilities are developed as separate projects resulting in separate (dot) extensions to the initial STIL standard. The scope of this extension is defined in 1.1 and is primarily to address design needs.

Whereas the initial STIL standard was developed by reviewing many languages already in existence in the industry, this standard has been developed by inventing new capabilities in support of new device designs. The new language constructs have been added such that they do not alter in any way the initial definition of STIL yet are syntactically compatible with the initial STIL language.

Much of the work to develop and validate these extensions has been done by prototyping on the part of the contributing companies.

Contents

1. Overview	13
1.1 Scope	13
1.2 Purpose	13
1.3 Word usage	14
1.4 Introduction	14
2. Normative references	16
3. Definitions, acronyms, and abbreviations	16
3.1 Definitions	16
3.2 Acronyms and abbreviations	17
4. Structure of this standard	17
5. STIL syntax description	18
5.1 Reserved words	18
5.2 Reserved characters	19
5.3 Reserved UserFunctions	20
5.4 Signal and group name characters	20
5.5 STIL name spaces and name resolution	21
6. Expressions	22
6.1 Constant and variable expressions	22
6.2 Expression delimiters—Single quotes and parentheses	22
6.3 Arithmetic expressions—Integer, real, time, boolean	24
6.4 Pattern data expressions	26
6.5 Expression processing	28
6.6 Boolean— <i>boolean_expr</i>	33
6.7 Integers— <i>integer_expr</i>	33
6.8 Logic expressions— <i>logic_expr</i>	34
6.9 Real expressions— <i>real_expr</i>	35
6.10 Addition to timing expressions— <i>time_expr</i>	36
6.11 SignalVariables— <i>sigvar_expr</i>	37
6.12 Formal parameters in procedures and macros	39
6.13 Integer lists— <i>integer_list</i>	39
7. Statement structure and organization of STIL information	40
8. STIL statement	40
8.1 STIL syntax	41
8.2 STIL example	41
9. UserKeywords statement	41
9.1 UserKeywords syntax	41
9.2 UserKeywords example	41
10. Variables block	42
10.1 Variables block syntax	42
10.2 Variables example	45
10.3 Variables scoping	46
10.4 Variables synchronizing	47

11. Signals block	49
11.1 Signals block syntax	49
11.2 Signals example	50
11.3 Bracketed signal notation enhancement	50
12. SignalGroups block	51
12.1 SignalGroups syntax	51
12.2 SignalGroups, WFCMap, and Variables example	51
12.3 Default WFCMap attribute value	53
12.4 Defining indexed signal groups	53
13. PatternBurst block	53
13.1 PatternBurst syntax	53
13.2 PatternBurst example	56
13.3 Tiling and synchronization of patterns	57
13.4 If and While statements	59
14. Timing block and WaveformTable block	60
14.1 Additional domain specification	60
14.2 CompareSubstitute operation—s, S	60
15. ScanStructures block	61
15.1 ScanStructures syntax	61
15.2 Scan cell naming— <i>cell_ref</i> , <i>chain_ref</i> , <i>cell_group</i> , <i>chain_group</i>	64
15.3 Scoping rules for ScanStructures block	66
15.4 Example indexed list of scan cells	66
15.5 Example of ScanChainGroups and ActiveScanChain	67
15.6 Scan chain segments and cell groups	68
16. Functions block	69
16.1 Functions syntax	69
16.2 Functions example	70
17. Pattern data	71
17.1 Data content read back—\C, \D, \E, \S, \U, \W	72
17.2 Vector data mapping and joining—\m, \j	75
17.3 Specifying event data in a pattern—\e	76
17.4 Using expressions within pattern data	77
18. Pattern statements	78
18.1 Additional Pattern syntax	78
18.2 Vector data constraints—F, E	80
18.3 Shift and LoopData statements	82
18.4 Loop statement using an integer expression	84
18.5 MergedScan function	85
19. Procedure and macro data substitution	85
19.1 Nested procedure and macro cells	85
19.2 Passing parameters to variables	86
19.3 Default value of formal parameters	87
19.4 Data substitution using WFCConstant and SignalVariable	87
20. Environment block	88
20.1 Environment syntax	89
20.2 MAP_STRING syntax	91

20.3 NameMaps example	92
20.4 Compact scan-cell mapping using InheritNameMap	93
21. Pragma block	94
21.1 Pragma syntax	94
22. PatternFailReport	95
22.1 PatternFailReport syntax	95
22.2 PatternFailReport example	97
Annex A (informative) Glossary	98
Annex B (informative) Signal mapping using SignalVariables	99
B.1 Example of parameter passing to Macros (or Procedures)	99
B.2 Parameter passing with signal mapping	99
B.3 A more complete example of parameters and signal variable expressions	100
Annex C (informative) Using logic expression with signals	103
Annex D (informative) Using boolean expressions in patterns	104
Annex E (informative) Variables and expressions in algorithmic patterns	106
Annex F (informative) Using AllowInterleave	108
Annex G (informative) Vector data mapping using \m	111
G.1 Use case #1	111
G.2 Use case #2	112
Annex H (informative) Vector data joining using \j	114
H.1 Example	114
H.2 Usage example	115
Annex I (informative) Block data collection	117
I.1 Step 1—Identifying signals	117
I.2 Step 2—Identifying response waveforms	118
I.3 Step 3—Pattern reference	118
I.4 Step 4—Tester response return	118
Annex J (informative) Using Fixed and Equivalent statements	120
J.1 Example—Signal constraints	120
J.2 Example—Constraints in a scan design	121
Annex K (informative) Applications using new ScanStructures syntax	122
K.1 Example—Inversion inside a scan cell	122
K.2 Example—Scan cells with multiple state elements	123
Annex L (informative) BreakPoints using MergedScan() function	126
L.1 Example of merged and unmerged STIL scan patterns using MergedScan()	126
L.2 Processing of STIL scan patterns that use the MergedScan() function	128
Annex M (informative) Labels and X statements for diagnostic feedback	129
Annex N (informative) Function Examples	131
N.1 General	131

N.2 Use Case 1	131
N.3 Use Case 2, IDDQ test.....	132
N.4 Use Case 3	133
N.5 Use Case 4	135
Annex O (informative) Use of STIL.1 for specific applications	136
Annex P (informative) 2025 Revision Updates.....	138
P.1 New capabilities and clauses	138
P.2 New capabilities	138
Annex Q (informative) Bibliography	139

IEEE Standard for Extensions to Standard Test Interface Language (STIL) for Semiconductor Design Environments

1. Overview

1.1 Scope

Structures are defined in STIL to support usage as semiconductor simulation stimulus, including the following:

- a) Mapping signal names to equivalent design references
- b) Interface between scan and built-in self test (BIST) and the logic simulation
- c) Data types to represent unresolved states in a pattern
- d) Parallel or asynchronous pattern execution on different design blocks
- e) Expression-based conditional execution of pattern constructs

Structures are defined in STIL to support the definition of test patterns for sub-blocks of a design (i.e., embedded cores) such that these tests can be incorporated into a complete higher level device test.⁶

Structures are defined in STIL to relate fail information from device testing environments back to original stimulus and design data elements.

1.2 Purpose

The STIL language definition is enhanced to support the usage of STIL in the design environment, which includes extending the execution concept to support STIL as a stimulus language, to allow STIL to be used as an intermediate form of data and to allow STIL to capture design information needed to port simulation data to device test environments.

⁶ Syntax in this document that is used in the definition of patterns for sub-blocks is summarized in Annex O.

In addition, extensions are defined to support the definition of subelement tests and to define the mechanisms to integrate those tests into a complete device test. This effort is to be performed in conjunction with IEEE Std 1500™-2005 [B4] and IEEE Std 1450.6™ [B3], which are defining standards for the definition and integration of embedded cores.⁷

Finally, the constructs necessary to correlate test failure information back to the design environment, to allow debug and diagnosis operations to be performed based on failure information in STIL format, are defined.

1.3 Word usage

The word *shall* indicates mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (*shall* equals *is required to*).^{8,9}

The word *should* indicates that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required (*should* equals *is recommended that*).

The word *may* is used to indicate a course of action permissible within the limits of the standard (*may* equals *is permitted to*).

The word *can* is used for statements of possibility and capability, whether material, physical, or causal (*can* equals *is able to*).

1.4 Introduction

STIL is an evolving standard being developed in support of various needs for interfacing between test generation tools and test equipment. IEEE Std 1450™ (STIL.0) forms the basis for this evolution.¹⁰ New “dot” standards (like this one) are being developed to address specific needs beyond STIL.0.

This (STIL.1) standard addresses design-related aspects of digital test data. This standard can also be viewed as the addition of advanced features to the STIL.0 baseline to allow for the usage of STIL in more complex applications, while leaving the basic standard unchanged as a vehicle for transmitting basic test data. The following is a brief overview of the new features in STIL.1 to support advanced applications such as embedded cores;¹¹ families of test patterns; mapping to automated test equipment (ATE) systems;¹² mapping to simulation; and devices with advanced design for test (DFT).

Please see Annex O for a list of specific statements for each of these features.

⁷ The numbers in brackets correspond to those of the bibliography in Annex Q.

⁸ The use of the word *must* is deprecated and cannot be used when stating mandatory requirements; *must* is used only to describe unavoidable situations.

⁹ The use of *will* is deprecated and cannot be used when stating mandatory requirements; *will* is only used in statements of fact.

¹⁰ Information on references can be found in Clause 2.

¹¹ This standard contains syntax in support of embedded cores. See IEEE Std 1450.6™-2005 (Core Test Language) [B3] for the complete specification.

¹² This standard contains syntax in support of ATE systems. See IEEE Std 1450.3™ (Test Resource Constraints) [B2] for the complete specification.

Environment mapping: Data for a device exists in many forms and in many other software environments. Examples include the following:

- a) Simulation environment
- b) Static analysis environment
- c) Specific ATE system environment

The STIL Environment block is a new mechanism to relate STIL data to these other environments. No assumptions, expectations, or limitations are imposed on the other environments. It is just a way of relating one to the other.

Parameterized data: Much of STIL data is declarative in nature (i.e., it defines various static attributes of a device or pattern set). The addition of constant declarations, IntegerConstant and WFCConstant, allows a data set to be created that applies to a family of devices.

Complex test protocol definition: Test protocol definitions are usually contained in STIL procedures or MacroDefs and are used to specify the application of a series of data values to a device. STIL.0 supports scan chain data passing and simple WaveformCharacter (WFC) data passing via the # and % characters. STIL.1 enhances this capability by allowing the use of data substitution from SignalVariables and integer expressions. STIL.1 also enhances the functionality of Loops and Vectors and adds If/While decisions on pattern statements. These capabilities are needed for BIST, embedded cores, and various test access mechanisms.

Advanced scan architecture: Advanced DFT techniques require additional capabilities beyond what is defined in STIL.0, which includes multistate scan cells, reconfigurable scan-chains, and scan-chain indexing.

Run-time pattern decisions: The If, Else, While, and LoopData are new STIL.1 constructs that have been added for specification of pattern activity. These statements are needed in the specification of patterns to be run in the simulation environment. Although there is no standardization among ATE systems on run-time instructions for pattern execution, it is anticipated that restricted versions of these statements will be incorporated into ATE test patterns.

Pattern burst options: New variations on the PatternBurst have been added to allow for patterns running in parallel, patterns running in LockStep, and patterns that can be reordered. For parallel pattern execution, the specification for how the patterns fit together can be specified with the Fixed and Extend statements.

Enhanced user extensibility: The UserKeyword extensibility defined in STIL.0 has been extended to allow keywords to be defined on a per-block-type basis.

Signal relationships: Additional syntax is provided to allow the specification of relationships between signals. This process is preformed via \m to map WFCs to another WFC, \j to join WFCs, Extend to define behavior of signals beyond the bounds of a given pattern, and Fixed to restrict any further changes to signals within a pattern.

Fail feedback: Three new features are added to facilitate the processing of failure data from an ATE system back to design tools. The first is the X or cross-reference statement that allows the specification of where in a pattern/vector sequence a failure occurs. The second is the FailFeedback block for reporting fails. The third is the S/s timing event that allows for the specification of a data capture protocol for the purpose of capturing bulk fail data for processing.

2. Normative references

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEEE Std 1450™, IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data.^{13, 14}

3. Definitions, acronyms, and abbreviations

3.1 Definitions

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary Online* should be consulted for terms not defined in this clause.¹⁵

automated test equipment (ATE): It refers to a tester that is capable of interfacing to a semiconductor device and executing test pattern data that is imported from a Standard Test Interface Language (STIL) file/stream.

built-in self-test (BIST): A design practice in which test logic is incorporated into the circuitry of a semiconductor device. This circuitry may provide completely autonomous testing of a device (i.e., without any requirement of a tester). It may be such that stimulation by an external tester is required; however, the Standard Test Interface Language (STIL) file may be substantially different from a device without this circuitry incorporated.

core: A component or module that contains separately developed functionality and that is integrated into a chip to provide additional overall functionality. *See also:* **System on Chip (SoC)**.

electronic design automation (EDA): The set of software tools that are used for the design and creation of semiconductor chips. It includes the software tools that create the test patterns for the chips, which are often referred to as automated test pattern generators (ATPGs).

Standard Test Interface Language (STIL): The set of IEEE standards, including IEEE Std 1450™-1999, and all dotted extensions, including this one.

System on Chip (SoC): An integrated circuit containing modules that are designed/integrated such that they can be tested independently and have associated test patterns for each module.

WaveformCharacter (WFC): A symbol used for referencing waveforms.

NOTE—See Annex A¹⁶

WaveformGenerationLanguage (WGL): A proprietary standard that was, in part, used as the basis for STIL.0.

¹³ The IEEE standards or products referred to in Clause 2 are trademarks owned by The Institute of Electrical and Electronics Engineers, Incorporated.

¹⁴ IEEE publications are available from The Institute of Electrical and Electronics Engineers (<https://standards.ieee.org/>).

¹⁵ *IEEE Standards Dictionary Online* is available at: <http://dictionary.ieee.org>. An IEEE account is required for access to the dictionary, and one can be created at no charge on the dictionary sign-in page.

¹⁶ Notes in text, tables, and figures are given for information only, and do not contain requirements needed to implement the standard.

WaveformTable (WFT): A Standard Test Interface Language (STIL) block statement used to define waveforms across multiple signals and WaveformCharacters (WFCs).

NOTE—See Annex A.

3.2 Acronyms and abbreviations

ATE	automated test equipment
BIST	built-in self-test
DFT	design for test
DUT	device under test
EDA	electronic design automation
STIL	Standard Test Interface Language
STIL.0	IEEE Std 1450
STIL.1	IEEE Std 1450.1 (this standard)
WFC	WaveformCharacter
WFT	WaveformTable
WGL	WaveformGenerationLanguage

4. Structure of this standard

This standard is an adjunct to STIL.0 (IEEE Std 1450). The conventions established and defined in STIL.0 are used in this standard and are included verbatim.

Many clauses in this standard add additional constructs to existing clauses in STIL.0 (IEEE Std 1450) and are so identified in the title. The Environment block is a new construct introduced in this standard. All clauses in this standard are normative. Example code is provided within each clause. More complete examples are provided in the annexes, which are informative.

The following is a copy of the conventions as defined in STIL.0 and adhered to in this standard.

Different fonts are used as follows:

- SMALL CAP TEXT indicates user data.
- `Courier text` indicates code examples.

In the syntax definitions, the following applies:

- a) SMALL CAP TEXT indicates user data.
- b) **Bold text** indicates keywords.

- c) *Italic text* references metatypes.
- d) () indicates optional syntax that may be used zero or one time.
- e) ()+ indicates syntax that may be used one or more times.
- f) ()* indicates optional syntax that may be used zero or more times.
- g) <> indicates multiple-choice arguments or syntax.

In the syntax explanations, the verb “shall” indicates 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 (writers), “shall” denotes mandatory characteristics of the language. The resulting output shall 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 as identified here in the dashed list.

5. STIL syntax description

This clause defines extensions to STIL.0 (Clause 6).

All constructs and restrictions for STIL.0 (Clause 6) are in effect here, with the following additions:

- Additional STIL reserved words are defined (see Table 1) for the top-level blocks specified within this standard.
- Additional STIL reserved characters are defined (see Table 2) for the new characters specified within this standard.
- Additional definition of signal and group naming, as well as name space resolution, is provided in this clause.
- Extensions to the expression environment are defined in this standard. Expression definitions [which are part of STIL.0 (Clause 5)] are now specified in a separate clause (Clause 7 in this standard).

5.1 Reserved words

Table 1 lists all STIL reserved words defined by this standard. Only top-level block names that are defined in this standard are added to the STIL reserved word list. New keywords that appear inside of top-level blocks are not restricted from usage in other contexts outside of the definition of that keyword. No change to reserved words as defined in STIL.0 is made by this standard.

Table 1—Additions to STIL reserved words

Environment
Functions
PatternFailReport, Pragma
Variables

5.2 Reserved characters

Several reserved characters identified in STIL.0 are applied in additional contexts for this standard. Table 2 lists additional STIL reserved characters defined in this standard as well as additional contexts of previously identified reserved characters. No change to reserved characters as defined in STIL.0 is made by this standard. New reserved characters are so indicated in column two. All other characters in this table have extended capability beyond that defined in STIL.0.

Table 2—Additions to STIL reserved characters

Char	New in STIL.1	Usage
!	—	exclamation (NOT sign): An inversion operator in expressions
%	—	percent sign: Is used as the modulus in expressions and for parameter passing to procedures and macros
()	—	parentheses: Are used in expressions
*	—	multiply: Is used in expressions
+	—	add: Is used in expressions
-	—	subtract: Is used in expressions
&	YES	logical and: Is used in expressions
	YES	logical or: Is used in expressions
^	YES	logical xor: Is used in expressions
~	YES	logical negation: Is used in logic expressions error return value: Is used when \W has no WFC chars to return
<	—	less than: Is used in integer and real expressions
>	—	greater than: Is used in integer and real expressions
<=	—	less than or equal to: Is used in integer and real expressions
>=	—	greater than or equal to: Is used in integer and real expressions
==	—	equal to: Is used with pattern data expressions (i.e., to compare strings of WFCs)
!=	—	not equal to: Is used with pattern data expressions (i.e., to compare strings of WFCs)

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

Char	New in STIL.1	Usage
:==	YES	equal to: Is used with integer and real expressions (i.e., to compare integer or real numbers)
<>	YES	not equal to: Is used with integer and real expressions (i.e., to compare integer or real numbers)
?:	—	conditional: Is used in expressions
=	—	assignment: Is used in signal variable expressions, vector data, group name definitions, spec category expressions, and spec variable expressions
:=	YES	assignment: Is used in integer and real expressions
_	YES	underscore: Is used as a separator in integer values
.	—	dot: Is used to designate macro and procedure calls in FailData block
:	—	colon: Is used to designate subcomponents of a design (i.e., embedded cores)
::	YES	double colon: Is used to designate a domain reference in expressions
{* }	—	brace, asterisk: Is used to designate a block of data in the case of Pragma (same as delimiters used for annotation blocks)

5.3 Reserved UserFunctions

The user function names in Table 3 are reserved for use in expressions.

Table 3—UserFunctions

Name	New in STIL.1	Usage
min	—	function: Is used in integer or real expressions to return the minimum value from a list of values (as defined in STIL.0).
max	—	function: Is used in integer or real expressions to return the maximum value from a list of values (as defined in STIL.0).
MergedScan	YES	function: Is used in patterns to test whether scan in/out data have been merged. See 18.5.

5.4 Signal and group name characters

This clause defines extensions to STIL.0 (6.10 in 1999; 6.11 in 2023).

All constructs and definitions in STIL.0 (6.10 in 1999; 6.11 in 2023) are in effect. In addition, signals expressed with a bracketed construct shall allow the use of a previously declared integer-constant, in which integer values are allowed. See 6.1 for the definition of IntegerConstant.

5.5 STIL name spaces and name resolution

Additions to STIL.0 (6.16 in 1999; 6.17 in 2023).

The Environment block augments the STIL name space as defined in Table 4. This table is incremental to STIL.0 (Table 6 in 1999; Table 24 in 2023); all definitions present in that table remain unchanged.

Variable names and constant names are used by expressions. This name space also contains signal names, signal group names, and spec variables defined for the context of that expression. Referenced variable names shall be unique against all signal names, signal groups names, and spec variable names defined for a PatternBurst or application of a Pattern referenced by that PatternBurst.

Table 4—Additions to STIL name spaces

STIL block	Type of name	Domain restrictions
Environment	Environment domain names	Supports a single unnamed block and domain named blocks. Domain names shall be unique across all Environment blocks.
Variables	Variables domain names	Supports a single unnamed block and domain named blocks. Domain names shall be unique across all Variables blocks.
Functions	Functions domain names	Supports a single unnamed block and domain named blocks. Domain names shall be unique across all Functions blocks.
Variables, Signals, SignalGroups, Spec	Signal names SignalGroup names SignalVariable names Spec variable names Integer names IntegerConstant names WFCConstant names	Names present in this name space are dependent on the domain reference statements in the PatternBurst (SignalGroups domains, Variables domains) and the PatternExec (Category domains). It is an error for defined Variable or Constant names to conflict with Signals, SignalGroups, or Spec variable names accessible in the same context.
ScanStructures	ScanCell names ScanChain names (also used for scan segments and cell groups)	Names present in this space are dependent on the domain reference statements for ScanStructures in the PatternBurst. Scan cell names, scan chain names, and cell group names in this common name space are unique except if an unnamed ScanStructures block exists. Names present in an unnamed block are available unless hidden by a definition of the same name in a named and referenced ScanStructure block.

6. Expressions

This clause defines extensions to STIL.0, Clause 6.

STIL.0 defines a limited usage of expressions; see STIL.0 (6.13 and 6.14 in 1999; 6.14 and 6.15 in 2023). This standard extends these expression capabilities with additional variable types (SignalVariable, Integer, IntegerConstant, WFCConstant) and additional expression constructs. The use of operators and the expression constructs as defined for STIL.0 is unchanged. Table 6 includes implicit numeric casting rules. The full set of STIL.0 and STIL.1 expression operations are defined in Table 7. The detail of syntax and usage of the new expressions are defined in 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 6.10, 6.11, 6.12, and 6.13.

6.1 Constant and variable expressions

Expression constructs (which are defined in 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 6.10, 6.11, 6.12, and 6.13) can be divided into two classes: those that can be completely resolved at parse time (i.e., constant expressions) and those that can only be resolved at run time (i.e., variable expressions).

Constant expressions are those that contain only literal values (e.g., ‘5ns’ or ‘10ns’), or named constants (e.g., Spec-Variable, IntegerConstant, and WFCConstant). As in STIL.0, statements that contain constant expressions can be fully resolved once the STIL file/stream has been fully parsed and the domain references have been resolved. Constant expressions can be used anywhere that the syntax definitions allow a literal value. The use of constant expressions provides for the following:

- a) Improved readability
- b) Parameterized STIL data
- c) Enhanced reuse
- d) Improved maintainability

Variable expressions are those that contain named variables, such as SignalVariable and Integer. The use of variable expressions is intended primarily to support “design” applications. The ability of an ATE system to support variable expressions may be limited, if possible at all.

Both constant and variable expressions can be further classified into “arithmetic expressions” and “pattern data expressions.” The allowed constructs of each of these expression types are defined in 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 6.10, 6.11, 6.12, and 6.13.

6.2 Expression delimiters—Single quotes and parentheses

Single quotes are defined in STIL.0 as the delimiter to be used around expressions. This usage is unchanged with the introduction of new expression capabilities in STIL.1, and a STIL.0 file is completely compatible with all STIL.1 rules. However, the rules governing the use of single quotes and parentheses are relaxed.

6.2.1 Single quotes

The following rules apply to the use of single quotes in STIL.1:

- a) Single quotes, when used, are always the outermost symbols of an expression.

```
'T2+5ns' // timing expression
```

```
'SIG1+SIG2+SIG3' // signal group expression
```
- b) Assignment and boolean operators shall not be used inside a single-quoted expression.

```
V { 'INT := INT+5'; } // illegal
```

```
If 'INT == 5' { } // illegal
```
- c) Single quotes shall not be nested.

```
V { 'BUS[1..'K+1']' = XXX; } // illegal - nesting of quotes
```
- d) Wherever expressions are allowed, a single token may be used without delimiters. A single token may be a literal value, a named constant, or a named variable.

```
V { INT := '5'; } // single token with quotes
```

```
V { INT := 5; } // single token without quotes
```

```
V { INT1 := INT2; } // single token without quotes
```
- e) Wherever expressions are allowed, they may be represented without delimiters, with the exception that if a multiterm expression is within a complex data stream, then the delimiters are required. Examples of cases in which delimiters are required are shown in the later examples, using parentheses.

```
V { INT := 'INT+1'; }
```

```
V { INT := INT+1; }
```
- f) Parentheses may be used inside a single-quoted expression.

```
V { INT := '(INT+1)*2'; }
```

```
V { '(GRP-SIG)+SIG' = 00001; } // make SIG the last WFC in the list
```

6.2.2 Parentheses

Parentheses are the preferred method of delimiting expressions in STIL.1. Parentheses can be used around the entire expression or inside the expression to specify the order of processing. The following rules apply to the use of parentheses:

- a) Parentheses are allowed anywhere that single quotes are allowed in STIL.0.

```
V { (SIG1+SIG2) = XX; }
```

```
01 { (T2+5ns) D/U; }
```
- b) Assignment expressions may be constructed with or without the use of delimiters.

```
V { INT := INT+1; }
```

```
V { INT := (INT+1); }
```

```
V { (INT := INT+1); }
```

```
V { INT := (X+2)*2; } // parens required to define order
```

```
V { SIG1+SIG2 = XX; }
```

```
V { (SIG1+SIG2) = XX; }
```

```
V { (SIG1+SIG2 = XX); }
```

- c) Parentheses are not required to delimit boolean expressions; however, in practice, they are commonly used.

```
If (\W(A+B) == \wxx) { }
If \W(A+B) == \wxx { }
If (\WGRP != \w\r(K+12) X) { }
If \WGRP != \w\r(K+12) X { }
```

- d) Parentheses may be nested.

```
V {SIG = \r(K+12) X; }
If (\WBUS[1..(K-1) != \wXXX) { }
```

- e) Parentheses are used to delimit parameters to functions within expressions.

```
V { INT := Min(X,Y); }
```

- f) If a multiterm expression is within a complex data stream, then delimiters around the expression are required.

```
SIG1 + SIG2 = XX;
\W(SIG1 + SIG2) == XX;           //parens required
AB { 2ns D/U; }
AB { TIM D/U; }
AB { (TIM + 2ns) D/U; }           //parens required
V { GRP = \r2 X YZ; }
V { GRP = \rTWO X YZ; }
V { GRP = \r(TWO+1) X YZ; }       //parens required
V { GRP[5 9] = XX; }
V { GRP[K5 K9] = XX; }
V { GRP[(KK+5) (KK+9)] = XX; }    //parens required
```

6.3 Arithmetic expressions—Integer, real, time, boolean

Arithmetic expressions are used in various STIL statements. In the statement syntax definition within this standard, the reference identifiers, *integer_expr*, *real_expr*, *boolean_expr*, and *time_expr*, indicate that an arithmetic expression is expected, and they signify the expected output from the expression evaluation. It is allowed to use an arithmetic expression anywhere a literal arithmetic value is allowed as long as it contains only constants and literals. This subclause defines the common rules for all arithmetic expressions.

Arithmetic expressions use the following operators:

- := assignment operator
- ?: conditional operator
- ::= compare for equality; return integer 1 if equal; else return 0
- <> compare for inequality; return integer 1 if not equal; else return 0
- < compare less than; return integer 1 if less; else return 0
- > compare greater than; return integer 1 if greater; else return 0
- <= compare less than or equal to; return integer 1 if less or equal; else return 0
- >= compare greater than or equal to; return integer 1 if greater or equal; else return 0

The one exception to the use of := for integer assignments is in the Spec block. For compatibility with STIL.0, it is allowed to use either a bare '=' or the ':=' in this context.

Arithmetic expressions use the following operands:

- Integer, IntegerConstant that are in scope according to the Variables context
- Spec variables that are in scope according to the Category selection
- Timing event labels when used inside a Timing block as defined in STIL.0
- Literal integers, real numbers, and engineering numbers

Pairs of single quotes are used as delimiters in arithmetic expressions according to the rules as defined in STIL.0. It is not allowed to nest single quotes, and the usage should be limited to simple expressions (e.g., 'T2+5ns'). Some statements require that delimiters always be used around expressions, even when the expression is a single literal value or name. If not so stated in the statement definition, then single operands may be used without any delimiters.

Paired parentheses are the preferred method of delimiting expressions in STIL.1. Parentheses can be used around the entire expression or inside the expression to specify the order of execution.

Multiple assignment operators are not allowed in an arithmetic expression:

```
V { INT1 := INT2 := 0; }           // illegal use of multiple assignment operators
```

The following examples are of keyword identified expressions:

```
Loop 50 {}  
ScanLength 100;  
Loop K+5 {}  
Loop 'K+5' {}  
Loop (K+5) {}
```

The following examples are of assignment expressions:

```
Condition { INT := 5; }  
V { INT := 5; }  
C { INT := 5; }  
C { INT := INT + 1; }  
C { INT := K; }  
C { INT := (INT==0) ? 99 : INT-1; }
```

The following examples are of improper assignment expressions:

Signals { CLK In; }

Variables{ INT1 := 0; }

Pattern pat1 {

```
V { CLK := 1; } // CLK requires = for WFC assignment  
V { INT1 = 5; } // INT requires := for variable assignment
```

The following examples are of boolean expressions:

```
If (INT == 6) { }  
If (INT >= K*2) { }  
If (INT > SPEC) { }  
If (SPEC < 250ns) { }  
While (INT <> 13) { }
```

The following example is of an arithmetic expression within a pattern data expression:

```
V { GRP[1..(K+1)] = \r(K+1) X; }
```

6.4 Pattern data expressions

Whereas arithmetic expressions operate on numeric data, pattern data expressions operate on lists of WFCs. To differentiate the two, a different set of operators is used as follows:

- = assignment operator
- == compare for equality; return integer 1 if equal; else return 0
- != compare for inequality; return integer 1 if not equal; else return 0
- ?: conditional operator
- The pattern data operators that are preceded with a backslash (see Clause 17)

Pattern data expressions use the following operands:

- Signals
- SignalGroups that are in scope according to the PatternBurst context
- Literal lists of WFCs
- SignalVariables and WFCConstants that are in scope according to the Variables context
- Formal parameters from Procedures and Macros

Single quotes are used as delimiters in pattern data expressions according to the rules as defined in STIL.0. It is not allowed to nest single quotes, and the usage should be limited to identifying signal groups (e.g., 'SIGA+SIGB,' 'BUS[1..10]'). Some statements require that delimiters always be used around expressions, even when the expression is a single literal value or name. If not so stated in the statement definition, then single operands may be used without any delimiters.

Unlike arithmetic expressions, parentheses are not to be used on pattern data assignment expression. Assignments should be of the form: TWOSIGS=XX; or 'SIGA+SIGB'=XX;. Parentheses may be used when needed to control order of evaluation as in the following: '(XBUS-XBUS[5])+YBUS'=11110000;.

Multiple assignment operators are not allowed in pattern data expressions, such as follows:

```
V { SIG1 = SIG2 = X; }           // illegal - use of multiple assignment operators
```

The following examples are of pattern data assignment expressions:

```
V { SIG = X; }           // assign a WFC to a signal
V { GRP = XXXXX; }       // assign WFCs to a signal group
V { SIGVAR = XXX; }       // assign WFCs to a signal variable
V { GRP[5..6] = XX; }     // selective assign of WFCs
V { 'A+B+C' = XXX; }      // assign WFCs to multiple signals
V { SIGVAR = \W SIG; }    // assign WFC from a signal to a variable
V { SIG = \W SIGVAR[3]; } // assign WFC from a variable to a signal
V { GRP = \W HALT; }      // assign a WFC-const to a signal group
V { GRP[1..12] = \W SIGVAR[1..4] XXXX \W SIGVAR[5..8]; }
```

The following examples are of pattern data boolean expressions. Note that the \W may be used on both sides of the expression. The \W operator is required to indicate that WFC data are being compared. The \W operator is used to extract WFC data from a named entity. The \w operator is used to indicate that literal WFC data follow. Also note that when literal WFC data are used, it may appear on either side of the expression.

```
If (\WSIGVAR[6..5] == \wXX) { }
If (\wXX == \WSIGVAR[6..5]) { }
If (\WSIGVAR1 != \WSIGVAR2) { }
```

The following example is of a pattern data expression inside an arithmetic expression:

```
V { INT := (\WGRP==\w010) ? 1 : 0; }
```

The following complete example of statements contains both arithmetic and pattern data expressions:

```
1:  STIL 1.0 { Design 2025; }
2:  Header {
3:    Source "STD 1450.1-2025";
4:    Ann { * sub-clause 6.4 * }
5:  }
6:  Signals { S[1..4] In; }
7:  SignalGroups { SIGGRP = 'S[1..4]'; }
8:  Variables {
9:    IntegerConstant RUN := 0;
10:   IntegerConstant EXIT := 1;
11:   IntegerConstant LOAD := 2;
12:   IntegerConstant UNLOAD := 3;
13:   Integer CMD;           // use values RUN, EXIT, LOAD, UNLOAD
14:   Integer INT;
15:   Integer HEX {InitialValue 0x1000;}
16:   Integer CC;
17:   Integer DD;
18:   SignalVariable SIGVAR1[1..4];
19:   WFCConstant STOP=00;
20:   WFCConstant GO=01;
21:   WFCConstant RESET=10;
22:   SignalVariable SIGVAR2[1..2]; // use values STOP, GO, RESET
23: }
24: Spec {
25:   Category CAT {
26:     TIME = '25ns';
27:   }
28: }
29: Pattern PAT {
30:   C {INT := 1;}
```

```

31:      If (INT) {} Else {}           // Execute the If block, INT is True
32:      C { INT := 0; }
33:      If (INT) {} Else {}           // Execute the Else block, INT is False
34:      If (INT == 956) {}            // a boolean expression
35:      If (TIME >= 20ns) {}           // a real number in engr units
36:      If (CC <= DD) {}               // test for CC less than or equal to DD
37:      If (INT < min(CC,DD)) {}       // use of the min function
38:      C {INT := 5;}                  // set variable INT equal to 5
39:      C {CMD = \W RUN;}              // set using constant definition
40:      C {HEX := 0xFF;}               // set integer to 255
41:      If (CMD <> LOAD) {}             // integer compare to a constant
42:      C {INT := INT+1;}              // integer expression
43:      C {SIGVAR1 = LH01;}            // set signal variable to the string LH01
44:      V {SIGGRP = \WSIGVAR1;}        // set signals equal to a variable
45:      If (\WSIGVAR2 == \WSTOP)       // compare with a constant
46:      If (\WSIGVAR2 == HH) {}        // compare with a WFC list
47:  } // end Pattern

```

6.5 Expression processing

This subclause defines the rules for evaluating expressions. Please refer to 6.6, 6.7, 6.8, 6.9, 6.10, 6.11, and 6.12 for the definition of the expression types.

Expressions are built up in a hierarchical manner; internal subexpressions are processed, and they, in turn, are used in the next level of the hierarchy. A subexpression is legal expression syntax that is used within another expression. The only distinction between an expression and a subexpression is that a subexpression is a part of another expression.

At the lowest level are the primitive terms (Table 5). A primitive term used by itself represents a legal expression. Each primitive term represents a value and a type. In some cases, the value may not be determinable until run time. In all cases, the type is determinable when the STIL file/stream is parsed.

Subexpressions may be combined with operators to form subexpressions at the next higher level in the expression hierarchy. Like the primitive terms, each subexpression represents a value and a type. And, as with the primitive terms, the value may not be determinable until run time, but the type is determinable when the STIL file/stream is parsed.

Table 5—Primitive terms

Term	Type	Example—Comment
Integer	numeric	5, 1066
Hex integer	numeric	0xFFFF, 0x123
SI units	numeric	2mV, 25ns —SI units are defined in Table 3 and Table 4 of STIL.0
Real	numeric	2e+3, 4.5e6, 25e-9
Time event reference	numeric	@, @1, @@, @T1, @T1.1 —used in the context of a Timing->Waveform block
Time label	numeric	TXX: —used in the context of a Timing->Waveform block
Signal Signal group	sigref	NAME, “NAME”
Signal Signal group Signal variable WFC constant	WFC list	\WNAME, \W NAME, \W “NAME” —use \W to extract the WFC list from the named entity
String	WFC list	10HLXT, \w 10HLXT, \h01 FF, \d01 255 —the \ is required with the boolean operators (== <>) —the dot notation for concatenation is neither required or allowed in WFC data; a whitespace character shall serve as separator for WFC lists of greater than 1024 characters
Integer list	integer list	1359 —an ordered set of integers, typically used in the formation of signal groups and scan chains
Bare string	string	STRING, “STRING”, “STRING WITH SPACES”, “STRING WITH SPECIAL CHARS @\$%^” —used to reference named entities
Concatenated string	string	XXX.YYY yields XXXYYY —used when a string is greater than 1024 characters
Domain reference	string	DOM::NAME —used to reference an item within a named block, that is, within a named SignalGroups, Timing, or ScanStructures block
Subcomponent reference	string	COMP:NAME, COMP:DOM::NAME —reserved for referencing subcomponents of a design (i.e., embedded cores)
—	boolean	—a boolean type is the result of a boolean operation, that is, ==, :=, ! (see Table 7) —integer 0 is returned for a boolean false result, and integer 1 for a boolean true result —the values TRUE and FALSE are not defined —there is no variable type for boolean

Table 6 defines implicit casting rules that are allowed on an expression to change a number from one type of primitive term to another. The casting rule shall be used when necessary to do the following:

- Conform to the requirements of the next level of hierarchy
- Conform to the requirements of the statement containing the expression
- Compare a real to an integer number

Table 6—Implicit numeric casting rules^a

Term	Cast to	Comment
SI units	real	—Used when expression requires a real number
real	integer	—Used when expression requires an integer value —The fractional part is truncated; that is, 3.7 yields 3
integer	real	—Used when comparing an integer to a real number
integer	boolean	—Used when expression requires a boolean value —Value of 1 or greater yields TRUE —Value of 0 or negative yields FALSE

^aIf the required behavior is different from the implicit type casting rules, then explicit casting should be performed via user functions.

Table 7 is expanded from Table 5 in STIL.0 (1999) and from Table 23 in STIL.0 (2023) and includes the new STIL.1 operators. The behavior of the logic operators is defined in IEEE Std 1364™-2001 (Verilog) [B1]. This table is ordered by precedence. Operators within each double-line separated group are of equal precedence and are processed from left to right. Each double-line separated group of operators has higher precedence than the groups below them and shall be processed first.

Table 7—Operators and functions allowed in expressions

Operator	New	Definition	Number operands	Requirements of operands	Type of the result
Min ()	—	minimum value function	≥ 1	all of same numeric type	same as operands
Max ()	—	maximum value function	≥ 1	all of same numeric type	same as operands
MergedScan()	STIL.1	See 18.5	—	—	—
()	—	parentheses	1	any	same as operand
[]	—	square brackets	1	integer or integer list	integer list
Table 3 and Table 4 in STIL.0 (1999) and Table 21 and Table 22 in STIL.0 (2023)	—	SI units and prefixes	—	—	—
,	—	comma—Used as an argument separator in functions	2	any	same as operands
!	STIL.1	negation (unary)	1	boolean or sigref	same as operand
~	STIL.1	bit-wise negation (unary)	1	integer	same as operand
+	—	plus (unary)	1	numeric	same as operand
–	—	minus (unary)	1	numeric	same as operand
/	—	divide	2	numeric	numeric, SI units adjusted per rules
*	—	multiply	2	numeric	numeric, SI units adjusted per rules
%	STIL.1	modulus	2	integer	integer

Table 7—Operators and functions allowed in expressions (*continued*)

Operator	New	Definition	Number operands	Requirements of operands	Type of the result
+	—	add	2	same numeric type	same as operands
−	—	subtract	2	same numeric type	same as operand
+	—	concatenate signal list	2	signal or signal group	signal group
-	—	remove from signal list	2	signal or signal group	signal group
..	—	integer range (two consecutive dots)	2	integer	integer list
	STIL.1	(whitespace) used as a token separator			does not return a value
<	STIL.1	less than	2	same numeric type	boolean
>	STIL.1	greater than	2	same numeric type	boolean
<=	STIL.1	less or equal	2	same numeric type	boolean
>=	STIL.1	greater or equal	2	same numeric type	boolean
= =	STIL.1	equal (WFC)	2	WFC list	boolean
!=	STIL.1	not equal (WFC)	2	WFC list	boolean
:==	STIL.1	equal (numeric)	2	same numeric type	boolean
<>	STIL.1	not equal (numeric)	2	same numeric type	boolean
&	STIL.1	bit-wise and	2	integer or sigref	same as operands
^	STIL.1	bit-wise exclusive or	2	integer or sigref	same as operands
^~, ~^	STIL.1	bit-wise equivalence	2	integer or sigref	same as operands
	STIL.1	bit-wise inclusive or	2	integer or sigref	same as operands
&&	STIL.1	and	2	boolean	boolean
	STIL.1	or	2	boolean	boolean
?:	—	conditional expression	3 (1?2:3)	1: boolean 2,3: same type— numeric or WFC list	same as operands 2, 3
=	—	assignment (WFC)	2	—left: shall be a name of a symbol that can be assigned WFCs —right: a WFC list	WFC list
:=	STIL.1	assignment (numeric)	2	—left: shall be a name of a symbol that can be assigned to same type —right: any numeric type	same as second operand

6.6 Boolean—*boolean_expr*

An expression is interpreted as a boolean if the context in which it is used requires a boolean result. A boolean result can be achieved by means of the boolean operators. For pattern data expressions, the boolean operator shall be either `==` or `!=`, and both sides shall be of type WFC (i.e., a list of WFCs preceded by `\w`, else a signal, signal group, signal variable, or WFC-constant preceded by `\W`). For arithmetic expressions, the boolean operator shall be either `==`, `<`, `<=`, or `>=`, and both sides shall be of type integer or real.

An expression that results in an integer or real value shall be interpreted as a boolean if the usage context so requires; in which case, a nonzero positive value is interpreted as true, whereas a value of zero or negative is interpreted as false. Expressions that evaluate to WFCs shall not be used as booleans.

Although it may be good practice to use parentheses around boolean expressions, this is not a requirement of the language.

The following examples are of boolean expressions:

```
48:   STIL 1.0 { Design 2025; }
49:   Header {
50:     Source "STD 1450.1-2025";
51:     Ann { * sub-clause 6.6 *}
52:   }
53:
54:   Variables {
55:     Integer INT;
56:     Integer FLAG;
57:     IntegerConstant FALSE := 0;      // FLAG value
58:     IntegerConstant TRUE := 1;      // FLAG value
59:     SignalVariable OP_CODE[3..0];
60:     WFCConstant RUN = 1011;          // OP_CODE value
61:     WFCConstant STOP = 0111;         // OP_CODE value
62:     WFCConstant RESET = 0000;        // OP_CODE value
63:   } // end Variables
64:
65:   Pattern MY_PAT {
66:     If (INT == 13) {}
67:     If (FLAG == TRUE) {}
68:     If (\WOP_CODE == \WRESET) {}
69:     If (\w1111 == \WOP_CODE) {}
70:   } // end Pattern
```

6.7 Integers—*integer_expr*

An integer expression is an expression that evaluates to integer. Integer expressions are allowed anywhere that the statement syntax calls for an “*integer_expr*”. See the list of allowed operators in Table 7. The following rules of interpretation apply to integers and integer expressions:

- A bare integer may be declared either with or without single quotes, for example, 5 or ‘5’.
- The underscore character may be used as a separator within an integer declaration, for example, 65_535.
- If an integer result is called for and the expression results in a number with a fractional part, the fraction is truncated to produce an integer value; for example, the values [6.1, 6.8, -3.1, -3.8] become [6, 6, -3, -3], respectively. The truncation is performed at the conclusion of the evaluation; that is, $(3/2) \times 2$ results in a value of 3.

- d) The following formats are not allowed for integers: 56E3, 56K.
- e) Integer may be expressed in hexadecimal by preceding the number with 0x, for example, 0xFF.

Integer expressions may comprise integer literals, integer variables, integer constants, or multiterm expressions that result in an integer value. The following examples are of integer expressions and their usage:

```

71:  STIL 1.0 { Design 2025; }
72:  Header {
73:    Source "STD 1450.1-2025";
74:    Ann { * sub-clause 6.7 * }
75:  }
76:
77:  Signals { SIGA In; SIGB Out; SIGC InOut; FOO[0x0..0x3F] In; }
78:  Variables {
79:    Integer II;
80:    Integer JJ { InitialValue 56*1024; }
81:    IntegerConstant KK := 2;
82:  }
83:  SignalGroups {
84:    ALL = 'SIGA+SIGB+SIGC+FOO';
85:  }                                     // end SignalGroups
86:
87:  Timing { WaveformTable WFT {
88:    Waveforms {
89:      ALL {
90:        01 { '2ns+KK*0.5ns' U/D; }
91:      }
92:    }                                     // end Waveforms
93:  }}                                     // end Timing
94:
95:  Pattern P {
96:    C { II := 1234; }
97:    C { II := 48000000; }
98:    C { II := 48_000_000; }             // equivalent to 48000000
99:    C { II := II + 2; }                 // expression with a variable
100:   C { II := 13; JJ := 13; }           // expression containing assignment
101:   C { II := JJ/2; }                   // truncate (i.e., 13/2 yields 6)
102:   C { II := JJ*0.5; }                 // truncate (i.e., 13*0.5 yields 6)
103:   C { II := 0x7FF_FFFF; }             // hex representation of integer
104:   C { II := JJ&0x00ff; }              // bitwise and with a hex number
105:   If (II >= 99) {}                   // conditional expression
106:   Loop II {}                          // variable loop count
107:   Loop 0xff { }                      // loop count defined in hex
108: } // end Pattern

```

6.8 Logic expressions—*logic_expr*

A logic expression is used to represent a combination of signal names and signal group names in the context of a design model. Logic expressions are allowed anywhere that the statement syntax calls for a “*logic_expr*”. The rules for interpretation of this expression are outside of the definition of this standard (see IEEE Std 1364™-2001 [B1]).

The logic relationship represented by this expression is independent of any timing or waveform considerations. It is solely indicating that the intended function of the design is to combine the named signals and signal groups according to the specified logic relationship.

The allowed operators in a logic expression are as follows:

- () parentheses
- ~ bit-wise negation
- & bit-wise and
- | bit-wise inclusive or
- ^ bit-wise exclusive or
- ^~, ~^ bit-wise equivalence

The following example contains a logic expression in the ScanEnable statement. The scan enable condition for the scan chain CHAIN1 is active if signals AA and BB are both true, thereby satisfying the logic expression 'AA&BB'.

```
109: STIL 1.0 { Design 2025; }
110: Header {
111:     Source "STD 1450.1-2025";
112:     Ann { * sub-clause 6.8 * }
113: }
114:
115: Signals { AA In; BB In; SI1 In; SO1 Out; }
116:
117: ScanStructures S1 {
118:     ScanChain CHAIN1 {
119:         ScanLength 100;
120:         ScanCells CC[1..100];
121:         ScanIn SI1;
122:         ScanOut SO1;
123:         ScanEnable AA&BB;
124:     }
125: }
```

6.9 Real expressions—*real_expr*

A real expression is an expression that evaluates to a real number as determined by the context (i.e., key word) or the variable type on the left-hand side of an expression. Real expressions are allowed anywhere that the statement syntax calls for a “*time_expr*” or “*real_expr*”; that is, anywhere that an engineering unit value is allowed, then a real expression is allowed. Real variables and expressions are defined in a Spec block in STIL.0. A real number can be expressed in one of two formats as follows:

- a) A real number can be of the form: <number>e<+|-><number>. A real number can be used to represent values that are not standard SI units, for example, a slew rate in volts/nanoseconds.
- b) A real number can also be of the form <number><prefix><SI unit>. For example, ‘23ns’ is a time expression (nanoseconds), ‘10uF’ is a capacitance expression (micro-Farads). This generic reference can be used whenever one of the standard unit definitions is allowed.

There are commonly accepted rules with regard to the algebraic combination of values with engineering units. Such rules are not defined in this standard; however, specific tools that implement this standard may require compliance with these rules, and it is good practice to keep units consistent with the usage. For example, if a spec value defines VX to be of type voltage, it can be used as time by expressing it as VX*(1ns/1V).

The following examples are of real number expressions:

```

126: STIL 1.0 { Design 2025; }
127: Header {
128:   Source "STD 1450.1-2025";
129:   Ann { * sub-clause 6.9 * }
130: }
131:
132: Signals { SIG_NAME InOut; }
133:
134: Spec {
135:   Category CAT {
136:     TIME = 25ns;
137:     REALVAR = 5.5;
138:     VOLTAGE = 2.5V;
139:     WATTAGE = 25mW;
140:     SLEWRATE = 1V/1ns;
141:   } // end Category
142: } // end Spec
143:
144: Pattern PAT {
145:   If (TIME >= 23.5ns+1.5ns/2) {} // time expression
146:   If (WATTAGE >= 5V*2uA) {} // where WATTAGE is of type 'Watts'
147:   If (SLEWRATE >= 5V/1ns) {} // where SLEWRATE is of type 'Real'
148:   If (REALVAR >= 5) {} // where REALVAR is of type 'Real'
149:   If (REALVAR*1mA >= 5mA) {} // where REALVAR is of type 'Real'
150: } // end Pattern
151:
152: Timing {
153:   WaveformTable WFT {
154:     Waveforms { SIG_NAME {
155:       01 { 25ns D/U; } // constant time
156:       01 { (REALVAR*1ns) D/U; } // use a real to define time
157:       01 { (VOLTAGE*(1ns/1V)) D/U; } // use a voltage to define time
158:     }}
159:   }
160: } // end Timing

```

6.10 Addition to timing expressions—*time_expr*

Additional symbols are added for referencing timing events in waveforms. These are additions to the @n symbol (as defined in STIL.0: 6.13 in 1999; 6.14 in 2023), which allows reference to the timing edges in a timing expression that are in the current period. The new symbols, as defined here (along with the other time referencing capabilities), allow access to period markers and events in subsequent periods. These new facilities are provided to support application in other STIL extensions, such as STIL.3 and STIL.6.

- a) @ => references the time of the prior event (STIL.0).
- b) @n => references the time of the n-th event, where first event is @1 (STIL.0).
- c) @@ => references the time of the current event (STIL.1).
- d) @Tm => references the time of the start of the m-th period, where the current period is numbered 0. The time value returned is relative to T0 of the current period. @T1 represents the end of the current period (i.e., the time relative to the current period for the start of the next period).
- e) @Tm.n => references the events of subsequent periods. The time value returned is relative to T0 of the current period. @T1.3 represents the third event of the second period. Note that events are numbered starting at 1, because the 0-th event is T0 (STIL.1).

- f) `EVENT_LABEL =>` references a label that is defined by `EVENT_LABEL:` anywhere in the current WaveformTable or WaveformDescriptions block (STIL.0).
- g) `SPEC_VARIABLE =>` references a variable defined in a Spec block (STIL.0).
- h) `VARIABLE =>` references an Integer or IntegerConstant defined in a Variables block (STIL.1).

Figure 1 illustrates events across three periods.

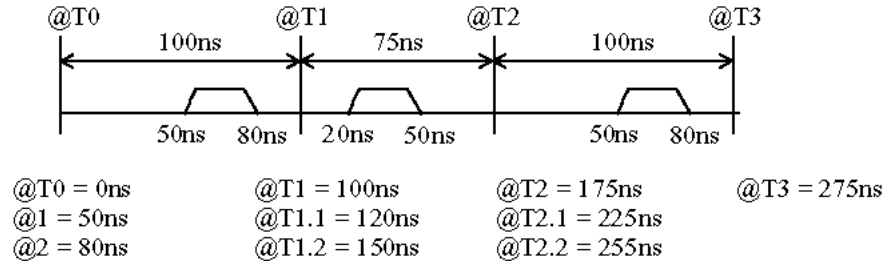


Figure 1—Referencing time edges

6.11 SignalVariables—*sigvar_expr*

A SignalVariable is a variable that is used to hold strings of WFC characters. It can be assigned WFC strings in the same way that signals and signal groups are assigned WFC strings. A *sigvar_expr* is a reference to a signal variable, with the result being a list of WFCs. A SignalVariable is assigned a list of WFC values, which may then be assigned to actual signals or groups. The list of WFC values contained in a variable of type SignalVariable is obtained by preceding it with \W. A SignalVariable name may be used to pass WFC information as a parameter in a macro or a procedure calls.

SignalVariables can be used in-line by setting them to a list of WFCs in a vector or a condition statement. When used in this way, the data are immediately available for use.

NOTE—See also the definition of formal parameters in the next section.

The following example shows in-line use of a signal variable:

```
C { SV = 1111; }
V { GRP = \WSV; }
```

The contents of the SignalVariable are maintained on a call to a procedure and the return; and the content is maintained on the invocation and exit from a macro.

The WFC data in a SignalVariable can be used in much the same way as Signals and SignalGroups. Square brackets are used to indicate the length if there are multiple elements. The following example illustrates typical SignalVariable definitions:

```
Variables {
  SignalVariable SV1;           // define a single element variable
  SignalVariable SV3[0..2];     // define a three element variable
}
```

When used in pattern data, the SignalVariable may be used without the square brackets; in which case, all elements of the SignalVariable are used. If a subset of the elements is desired, then the square bracket notation is used. The length on both sides of the equal sign shall be the same, except when the SignalVariable is being used as a formal parameter (see 6.12). The following example illustrates typical SignalVariable usage:

```
Pattern
  V { SIG = \W SV1; }
  V { GRP_OF_THREE = \W SV3; }
  V { GRP_OF_TWO = \W SV3[0..1]; }
}
```

The following examples are of SignalVariable usage:

```
161: STIL 1.0 { Design 2025; }
162: Header {
163:   Source "STD 1450.1-2025";
164:   Ann { * sub-clause 6.11 * }
165: }
166:
167: Signals {
168:   BUSX[1..5] In;           // defines 5 signals plus a group named BUSX
169: }
170:
171: Variables {
172:   SignalVariable SIG_VAR[1..5];
173: }
174:
175: MacroDefs {
176:   APPLY_VAR {
177:     C { SIG_VAR[1..5] = #; }
178:     V { BUSX = \W SIG_VAR[1..5]; }
179:     // error if above two lines were changed to:
180:     // V { SIG_VAR[1..5] = #; BUSX = \W SIG_VAR[1..5]; }
181:   }
182:   APPLY_TWO_VARS {
183:     C { SIG_VAR[1..5] = #; }
184:     V { BUSX = \W SIG_VAR[1..5]; } // apply 11111
185:     C { SIG_VAR[1..5] = #; }
186:     V { BUSX = \W SIG_VAR[1..5]; } // apply 00000
187:   }
188: }
189:
190: Pattern PAT {
191:   // following macro call use signal variables as parameters
192:   Macro APPLY_VAR { SIG_VAR[1..5] = 11100; }
193:   Macro APPLY_VAR { SIG_VAR[5 4 2 3 1] = 00011; }
194:   Macro APPLY_VAR { SIG_VAR[5..1] = ABBAB; }
195:   Macro APPLY_TWO_VARS { SIG_VAR[5..1] = 11111 00000; }
196:   // following use signal variables in line
197:   C { SIG_VAR[1..5] = 11001; }
198:   V { BUSX = \W SIG_VAR[1..5]; }
199: }
```

6.12 Formal parameters in procedures and macros

The concept of a formal parameter was used in STIL.0; however, it was not called that. Now with the introduction of more complicated pattern data expressions, it is necessary to keep the concept of variable usage and formal parameter separate. Whereas a signal variable may be set to a value that becomes immediately available, a parameter in a macro or procedure is only available when it is referenced within the referenced macro or procedure by the use of the # or % characters. It is the same behavior as for any Signal or SignalGroup that is used as a parameter as defined in STIL.0. See the following example:

When the name of a SignalVariable is used as a parameter in a macro or a procedure call:

```
Macro MAC { SV = 1111; }  
Call PROC { SV = 1111      0101; } )
```

then the WFC data are not available until a # or % operator is used inside the procedure or macro:

```
MAC {  
  C { SV=#; } V { GRP=\WSV; }  
}  
PROC (  
  C { SV=#; } V { GRP=\WSV; }  
  C { SV=#; } V { GRP=\WSV; }  
)
```

6.13 Integer lists—*integer_list*

An *integer_list* is used to specify an ordered list of integer values. Integer lists are allowed anywhere that the statement syntax calls for an “*integer_list*.” The only allowed operator in an integer list is the integer range operator “..”, which specifies a range of integers. An *integer_list* may contain either single integers, whitespace-separated lists of integers, or integer ranges. IntegerConstants may appear in place of integers. An integer range is represented by two integers (or IntegerConstants) separated by “..”. For example, ‘3..6’ is equivalent to ‘3 4 5 6’ and ‘4..2’ is equivalent to ‘4 3 2’. Whitespace characters are allowed between the integer or IntegerConstant, and the “..” operator. The following code shows examples of integer lists:

```
200: STIL 1.0 { Design 2025; }  
201: Header {  
202:   Source "STD 1450.1-2025";  
203:   Ann { * sub-clause 6.13 * }  
204: }  
205: Signals {  
206:   SIG[1..5] In;  
207:   SI1 In { ScanIn 100; }  
208: }  
209: ScanStructures S1 {  
210:   ScanChain CHAIN1 {  
211:     ScanLength 100;  
212:     ScanCells CC[1..100];  
213:   }  
214: }  
215: Pattern PAT {  
216:   V { SIG[5 4 3 2 1] = 11001; }  
217:   V { SIG[5..1] = 00110; }  
218: }
```

7. Statement structure and organization of STIL information

This standard defines additional top-level STIL blocks: Environment. It is delineated in Table 8, which is provided to be complete with Table 8 in STIL.0 (1999) and Table 27 in STIL.0 (2023).

Table 8—Additions to top-level statements

Statement	Purpose
Variables Clause 10	This Variables block defines variables and constants.
Functions Clause 16	The Functions block is used to extend STIL with the intent of synchronizing environments, for example, running STIL patterns and executing code on an ATE.
Environment Clause 20	The Environment block defines relationships of STIL data to external environments. Environment blocks, if referencing other STIL data, shall be defined after the blocks that define that STIL data, unless those references are to information contained in MacroDefs, Procedures, or Pattern blocks, which are allowed to be forward-references. This is always a top-level block. Environment blocks, if present, do not have a defined order with respect to other STIL blocks defined in STIL.0. Environment blocks may appear any place a top-level STIL statement is allowed.
Pragma Clause 21	The Pragma block allows for embedding application-specific code directly into a STIL file/stream. The block may be used as a top-level block or embedded in another STIL block.
PatternFailReport Clause 22	The PatternFailReport block is typically not part of a STIL file/stream that defines a test program; however, as a top-level block, it would be allowed to exist without conflict with other STIL blocks. This block is to be generated by the execution of a STIL test program to report results of execution.

8. STIL statement

This clause defines extensions to STIL.0, Clause 8.

The STIL statement identifies the primary version of STIL.0 information contained in a STIL file and the presence of one or more standard Extension constructs. The primary version of STIL is defined in STIL.0.

The extension to the STIL statement allows for a block containing extension identifiers that allow for additional constructs in the STIL file. There may be multiple Extension statements present, to identify the presence of multiple extension environments. The extension name and the extension statements are defined in the individual documents for those standards.

“Include” files are required (per STIL.0) to start with the STIL statement. The extension context as defined in an “Include” file shall be a subset of the definition in the base file; that is, it is not permissible to specify an extension in an included file that is not already allowed by the base file. One qualifier to this rule is the “IfNeed” option. If the parser does not actually consume the “Include,” then the extensions in the file are not relevant.

All other constructs and restrictions for STIL.0, Clause 8 are in effect here.

8.1 STIL syntax

```
STIL IEEE_1450_0_IDENTIFIER { (1)
    ( EXT_NAME EXT_VERSION; )+ (2)
} // end STIL
```

(1) **STIL**: A statement at the beginning of each STIL file.
IEEE_1450_0_IDENTIFIER: The primary version of STIL, as identified by 1.0.

(2) **EXT_NAME**: The name of the Extension. This standard is identified by the name **Design**.
EXT_VERSION: The version of the extension. This standard is identified by the value **2025**.

8.2 STIL example

```
219: STIL 1.0 {
220:     Design 2025;
221:     DCLevels 2002;
222: }
223: Header {
224:     Source "STD 1450.1-2025";
225:     Ann {* sub-clause 8.2 *}
226: }
227:
```

9. UserKeywords statement

This clause defines extensions to STIL.0, Clause 11.

This clause defines additional locations in a STIL file/stream where the UserKeywords statement is allowed to appear.

The UserKeywords construct is expanded from STIL.0 to allow the UserKeywords statement to be defined within any STIL block. When a UserKeywords statement is defined within a STIL block, those definitions shall apply only within that block and contained sub-blocks. It allows Userkeywords to be “locally scoped” to a containing STIL block.

NOTE—STIL.0 allows for UserKeywords only at the top level.

All other constructs and requirements for STIL.0, Clause 11 are in effect here.

9.1 UserKeywords syntax

As defined in STIL.0.

9.2 UserKeywords example

```
228: STIL 1.0 { Design 2025; }
229: Header {
230:     Source "STD 1450.1-2025";
231:     Ann {* sub-clause 9.2 *}
232: }
233:
234: Signals { A[1..99] InOut; }
```

```

235: SignalGroups { ALLSIGNALS = 'A[1..99]'; }
236:
237:     /* The UserKeywords construct is the same as defined in STIL.0,
238:        with the additional capability to be defined within another STIL block */
239:
240: Timing ONE {
241:     UserKeywords STARTUP SHUTDOWN;
242:     WaveformTable ONE {
243:         STARTUP { ALLSIGNALS; }
244:     }
245: }

```

10. Variables block

This clause defines variables and named constants. If the block is unnamed, then all definitions shall be globally available to all other blocks in the STIL file/stream. If the block has a name, then that name shall be referenced by the Pattern, PatternBurst, or Timing block for the variables to be available. Variables and constants that are defined in the global Variables block shall not be overridden in a named Variables block.

This block should be considered in light of the Spec block that also can define constants. The purpose and usage of these two blocks is sufficiently different to warrant their separation. The Spec block is intended to contain parametric data used in specifying device/test characteristics. The Spec block has special handling defined for Category, Min/Typ/Max/Meas, and Selector, which are to facilitate a flexible definition of these test parameters.

The Variables block is used to contain control variables and constants. Typical uses for these constants and variables is to control flow of execution, to provide “aliasing” of values to meaningful names with constants, and to provide run-time parameters.

Although complex expressions may be supported by an ATE system, in general, most ATE systems will be able to support only a limited subset of the design capabilities defined herein. The full capabilities are intended for use by simulation applications or pattern translation tools. See Annex O for more information.

10.1 Variables block syntax

```

real_type ::=
    Amperes | Celsius | Decibels | Degrees | Farads | General | Henries | Hertz | Meters |
    Ohms | Real | Seconds | Volts | Watts
>

```

```

Variables (VARIABLES_DOMAIN) {                                     (1)
    ( IntegerConstant CONST_NAME := INTEGER ; ) *                 (2)
    ( Integer VAR_NAME ; ) *                                       (3)
    ( Integer VAR_NAME {
        ( Usage Test ; )
        ( InitialValue integer_expr ; )
    } ) * // end Integer
    ( SignalVariable VAR_NAME ; ) *                                 (4)
    ( SignalVariable VAR_NAME {
        ( Base < Hex | Dec > WFC_LIST ; )
        ( Alignment < MSB | LSB > ; )
        ( InitialValue vec_data ; )
    } ) * // end SignalVariable
    ( WFCConstant CONST_NAME = WFC_LIST ; ) *                     (5)
}

```