



IEEE Standard for Signal and Test Definition

IEEE Standards and Standards Innovations (S&SI)
Strategic Management and Delivery Committee (SMDC)

Developed by the
Test and Diagnosis for Electronic Systems Standards Committee (SCC20)

IEEE Std 1641™-2022
(Revision of IEEE Std 1641-2010)

STANDARDS

IEEE Standard for Signal and Test Definition

Developed by the

Test and Diagnosis for Electronic Systems Standards Committee (SCC20)

on

IEEE Standards and Standards Innovations (S&SI)

Strategic Management and Delivery Committee (SMDC)

Approved 21 September 2022

IEEE SA Standards Board

Abstract: The means to define and describe signals used in testing are provided in this standard. It also provides a set of common basic signals, built upon formal mathematical specifications so that signals can be combined to form complex signals usable across all test platforms.

Keywords: ATE, ATLAS, automatic test equipment, IEEE 1641™, signal definitions, test definitions, test requirements, test signals, unit under test, UUT

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

Copyright © 2023 by The Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 10 February 2023. 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 978-1-5044-9279-9 STD25838
Print: ISBN 978-1-5044-9280-5 STDPD25838

IEEE prohibits discrimination, harassment, and bullying.

For more information, visit <https://www.ieee.org/web/aboutus/whatis/policies/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 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 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 this standard, including but not limited to the warranties of merchantability, fitness for a particular purpose and non-infringement. 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 his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

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

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, nor be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that 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).

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 in revisions to an IEEE standard is welcome to join the relevant IEEE 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](#).¹ An IEEE Account is needed to access the application.

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

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.

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

² Available at: <https://standards.ieee.org/content/ieee-standards/en/about/contact/index.html>.

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 US 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.

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

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

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

IEEE Standards do not guarantee or ensure safety, security, health, or environmental protection, or ensure against interference with or from other devices or networks. 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, and interference protection practices and all applicable laws and regulations.

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

Participants

At the time this IEEE standard was completed, the P1641 Working Group had the following membership:

Ion Neag, *Chair*
Chris Gorringer, *Vice Chair*

Larry Adams
Malcolm Brown
Jonathan Castleberry
Mark Dunn
Rick Freeman

Joe Headrick
Jean-Christophe Hertzog
Anand Jain
Steve Kelly
Teresa Lopes

Ben Matthews
Mike Seavey
Joe Stanco
Ron Taylor
Damien Willemet

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

Larry Adams
Malcom Brown
Koti Reddy Butukuri
Chris Gorringer
Werner Hoelzl
Anand Jain
Osamu Karatsu

Adam Ley
Teresa Lopes
Ion Neag
Leslie Orlidge
Howard Penrose
Bartien Sayogo
Mike Seavey
Mark Siira

Joe Stanco
Walter Struppler
Ron Taylor
John Vergis
Lisa Ward
Stephen Webb
Oren Yuen

When the IEEE SA Standards Board approved this standard on 21 September 2022, it had the following membership:

David J. Law, *Chair*
Ted Burse, *Vice Chair*
Gary Hoffman, *Past Chair*
Konstantinos Karachalios, *Secretary*

Edward A. Addy
Ramy Ahmed Fathy
J. Travis Griffith
Guido R. Hiertz
Yousef Kimiagar
Joseph L. Koepfinger*
Thomas Koshy
John D. Kulick

Johnny Daozhuang Lin
Kevin Lu
Daleep C. Mohla
Andrew Myles
Damir Novosel
Annette D. Reilly
Robby Robson
Jon Walter Rosdahl

Mark Siira
Dorothy V. Stanley
Lei Wang
F. Keith Waters
Karl Weber
Sha Wei
Philip B. Winston
Daidi Zhong

*Member Emeritus

Introduction

This introduction is not part of IEEE Std 1641™-2022, IEEE Standard for Signal and Test Definition.

This signal and test definition (STD) standard provides the ability to unambiguously define test signals. It includes a rigorous mathematical and definitive foundation for all of its signal components. Any signal defined using this standard shall be the same regardless of the equipment that is used to create it. The standard supports the implementation of new technologies by providing users with the ability to describe their own signals by combining existing signals. Thus, any desired signal may be described, and there is no limit on the extensibility of signals supported by this standard.

Signals defined using this standard can be used in a programming environment of the user's choice provided that that environment fulfills the minimum requirements defined in this standard. This universality enables the user to take full advantage of modern program structures and development environments, including graphical programming environments.

This standard was developed by the P1641 Working Group (of the IEEE Standards Coordinating Committee 20 (SCC20) on Test and Diagnosis for Electronic Systems), which has prepared a companion guide, IEEE Std 1641.1™, to explain how to implement signal definitions and test requirements in conformance with STD.

Contents

1. Overview	12
1.1 Scope	12
1.2 Purpose	12
1.3 Word Usage	12
1.4 Application	13
1.5 Annexes	13
2. Definitions, abbreviations, and acronyms	13
2.1 Definitions	13
2.2 Abbreviations and acronyms	15
3. Structure of this standard	16
3.1 Layers	16
3.2 Signal Modeling Language (SML) layer	17
3.3 BSC layer	17
3.4 TSF layer	18
3.5 Test requirement layer	18
3.6 Using the layers	18
4. Signals and SignalFunctions	18
4.1 Introduction	18
4.2 Physical signal states	19
4.3 Event states	20
4.4 Digital stream states	21
5. SML layer	22
6. BSC layer	23
6.1 BSC layer base classes	23
6.2 General description of BSCs	23
6.3 SignalFunction template	24
7. TSF layer	25
7.1 TSF classes	25
7.2 TSF signals defined by a model	26
7.3 TSF signals defined by an external reference	28
8. Test procedure language (TPL)	28
8.1 Goals of the TPL	28
8.2 Elements of the TPL	29
8.3 Use of the TPL	29
9. Maximizing test platform independence	29
Annex A (normative) Signal modeling language (SML)	30
A.1 Use of the SML	30
A.2 Introduction	30
A.3 Physical types	31
A.4 Signal definitions	34
A.5 Pure signals	36
A.6 Pure signal-combining mechanisms	38

A.7 Pure function transformations.....	44
A.8 Measure, limiting, and sampling signals.....	44
A.9 Digital signals.....	46
A.10 Basic component SML.....	50
A.11 Fast Fourier analysis support.....	76
Annex B (normative) Basic signal components (BSC) layer	78
B.1 BSC layer base classes.....	78
B.2 BSC subclasses	78
B.3 Description of a BSC	83
B.4 Physical class	91
B.5 PulseDefns class.....	103
B.6 SignalFunction class	104
Annex C (normative) Dynamic signal descriptions.....	163
C.1 Introduction.....	163
C.2 Basic classes	164
C.3 Dynamic signal goals and use cases.....	172
Annex D (normative) Interface definition language (IDL) basic components	174
D.1 Introduction	174
D.2 IDL BSC library	174
Annex E (informative) Test signal framework (TSF) for C/ATLAS	175
E.1 Introduction.....	175
E.2 TSF library definition in extensible markup language (XML).....	175
E.3 Interface definition language (IDL) for the TSF for C/ATLAS	175
E.4 AC_SIGNAL<type: Current Power Voltage>	176
E.5 AM_SIGNAL.....	178
E.6 DC_SIGNAL<type: Voltage Current Power>	180
E.7 DIGITAL_PARALLEL	182
E.8 DIGITAL_SERIAL.....	184
E.9 DIGITAL_TEST	186
E.10 DME_INTERROGATION	189
E.11 DME_RESPONSE.....	192
E.12 FM_SIGNAL<type: Voltage Power Current>	195
E.13 ILS_GLIDE_SLOPE<type: Voltage Power>	198
E.14 ILS_LOCALIZER<type: Power Voltage>	201
E.15 ILS_MARKER.....	204
E.16 PM_SIGNAL	207
E.17 PULSED_AC_SIGNAL<type: Current Power Voltage>	209
E.18 PULSED_AC_TRAIN<type: Voltage Current Power>.....	211
E.19 PULSED_DC_SIGNAL<type: Voltage Current Power>	213
E.20 PULSED_DC_TRAIN<type: Voltage Current Power>.....	216
E.21 RADAR_RX_SIGNAL.....	218
E.22 RADAR_TX_SIGNAL<type: Current Voltage Power>	220
E.23 RAMP_SIGNAL<type: Voltage Current Power>	222
E.24 RANDOM_NOISE	224
E.25 RESOLVER	226
E.26 RS_232.....	229
E.27 SQUARE_WAVE<type: Current Voltage Power>	230
E.28 SSR_INTERROGATION<type: Voltage Current Power>.....	232
E.29 SSR_RESPONSE<type: Voltage Current Power>	235
E.30 STEP_SIGNAL.....	239
E.31 SUP_CAR_SIGNAL.....	241
E.32 SYNCHRO.....	243

E.33 TACAN	247
E.34 TRIANGULAR_WAVE_SIGNAL<type: Voltage Current Power>	251
E.35 VOR	253
Annex F (informative) Test signal framework (TSF) library for digital pulse classes	257
F.1 Introduction	257
F.2 TSF library definition in extensible markup language (XML)	257
F.3 Graphical models of TSFs	257
F.4 Pulse class family of TSFs	257
F.5 DTIF	274
Annex G (normative) Carrier language requirements	276
G.1 Carrier language requirements	276
G.2 Interface definition language (IDL)	276
G.3 Datatypes	276
Annex H (normative) Test procedure language (TPL)	282
H.1 TPL layer	282
H.2 Elements of the TPL	282
H.3 Structure of test requirements	282
H.4 Carrier language	282
H.5 Signal statements	282
H.6 Mapping of test statements to carrier language	284
H.7 Test statement definitions	285
H.8 Elements used in test statement definitions	303
H.9 Attributes with multiple properties	306
H.10 Transferring data in digital signals	310
H.11 Creating test requirements	314
H.12 Delimiting TPL statements	316
Annex I (normative) Extensible markup language (XML) signal descriptions	318
I.1 Introduction	318
I.2 XSD for BSCs	319
I.3 XSD for TSFs	325
Annex J (informative) Support for ATLAS nouns and modifiers	333
J.1 Signal and test definition (STD) support for ATLAS signals	333
J.2 STD support for ATLAS nouns	333
J.3 STD support for C/ATLAS noun modifiers	336
J.4 Support for C/ATLAS extensions	344
Annex K (informative) Guide for maximizing test platform independence	345
K.1 Introduction	345
K.2 Guiding principles	345
K.3 Best practice rules	345
Annex L (informative) IEEE download web-site material associated with this document	349
Annex M (informative) Bibliography	350

IEEE Standard for Signal and Test Definition

1. Overview

1.1 Scope

This standard provides the means to define and describe signals used in testing. It provides a set of common basic signal definitions, built upon formal mathematical specifications, so that signals can be combined to form complex signals usable across all test platforms. The standard provides support for structural textual languages and programming language interfaces for interoperability.

1.2 Purpose

This standard provides a common reference for signal definitions, which may be used throughout the life cycle of a unit under test (UUT) or test system. Such a reference shall in turn facilitate information transfer, test reuse, and broader application of test information—accessible through commercially available development tools.

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*).

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*).^{6, 7}

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*).

⁶ The use of the word *must* is deprecated and cannot be used when stating mandatory requirements, *must* is used only to describe unavoidable situations

⁷ The use of *will* is deprecated and cannot be used when stating mandatory requirements, *will* is only used in statements of fact.

1.4 Application

This signal and test definition (STD) standard provides the capability to model, describe and control signals, while permitting a choice of operating environment, including the choice of carrier language. STD permits signal operations to be embedded in any object-oriented environment and thus to be used by the architecture standards of various automatic test systems (ATSS). STD may be used to create truly portable test requirements. It facilitates test information to pass freely between the design, test, evaluation and maintenance phases of a project and enables the same models and information to be used directly across project phases. This more efficient re-use of models and information directly leads to reduced life-cycle costs.

1.5 Annexes

This standard also contains annexes that describe various elements of the standard in detail. The normative annexes include definitions of the basic signals (in words and with reference to an extensible markup language (XML) format), supporting mathematical definitions for these signals, dynamic model information, interface definition descriptions, and a definition of the requirements of a supporting computer language.

Informative annexes are provided to present examples of signal libraries together with their associated XML definition.

2. Definitions, abbreviations, and acronyms

2.1 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.⁸

Abbreviated Test Language for All Systems (ATLAS): A stylized, abbreviated English language used in the preparation and documentation of test requirements and test programs, which can be implemented either manually or with automatic or semi-automatic test equipment.⁹

argument: Input values that can be passed to a function.

attribute: A property value that is used to define signal characteristics or behavior.

automatic test system (ATS): A system that includes the automatic test equipment (ATE) and all support equipment, support software, test programs, and interface adapters.

base class: A class from which another class inherits attributes or properties.

basic signal component (BSC): The lowest level of building block used to define signals.

class: A generic set of predefined abstract test objects.

⁸*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.

⁹ In this standard, the term “ATLAS” refers to any version or subset whether it is a formal standardized version or a project specific modified subset.

component: A part of a system, which may be hardware or software and which may be subdivided into other components. Components communicate their functionality through their interface definitions.

connection: The application of a signal to a unit under test (UUT).

data bus: A signal line or set of signal lines used by a data communication system to interconnect a number of devices and to communicate information.

dynamic signal: A signal whose definition changes over time, by use of the control interface. These changes should be initiated with one of the signal method calls or by changing the interconnections of a signal model.

function: A construct that is a logically separated block of code that operates upon test values (i.e., arguments). Another name for a function is method. Syn: method.

interface definition language (IDL): A machine-compilable language that is used to describe the interfaces that software objects call and object implementations provide. The language provides a neutral way to define software interfaces[B5].

method: Syn: **function**.

model: A mathematical or physical representation (i.e., simulation) of system relationships for a process, device, or concept.

physical: Pertaining to the natural characteristics of the universe according to the natural laws of science.

procedural: The part of a signal and test definition (STD) test requirement that defines the tests in the manner and order required for testing.

property: The special form of method (or function) that supports the semantics of assignment (l-value) and reading (r-value).

reserved word: A keyword whose meaning and use are fixed by the semantics of a language. In certain or all contexts, a reserved word cannot be used for any purpose other than as defined for that language.

semantics: A branch of linguistics concerned with meaning. For the test procedure language (TPL), semantics is the connotative meaning of words in a TPL statement. For software, semantics is the relationships of symbols and their meaning, independent of the manner of their interpretation and use. For meta-languages, semantics is the discipline for expressing the meanings of computer-language constructs in a meta-language.

sensor: A transducer that converts a test parameter to a form suitable for measurement.

SignalFunction: The generic name for all classes that provide signals.

Subclass: A class that inherits attributes or properties from a base class.

static signal: A signal whose definition does not change over time. All basic signal components (BSCs) and test signal framework (TSF) models are static signals.

syntax: The grammatical arrangement of words in a language statement.

system: A set of interconnected hardware and/or software components that achieves a defined objective by performing specified functions.

system architecture: The structure of and relationship between the components of a system. A system architecture may include the system interface with its operational environment.

template: A pattern or design that establishes the outline, dimensions, or process for subsequent users or implementers.

test: (A) An action or group of actions that are performed on a unit under test (UUT) to evaluate its parameter(s) or characteristic(s). (Derived from IEEE Std 771-1998). (B) An observed activity that may be caused to occur (e.g., stimulus-response) in order to obtain information about the behavior of a test subject. (Derived from IEEE Std 1671-2010). (C) A set of stimuli, either applied or known, combined with a set of observed responses and criteria for comparing these responses to a known standard. (Derived from IEEE Std 1232-2002).

test requirement: A definition of the tests and test conditions required to be performed on a unit under test (UUT) to verify conformance with its performance specification.

test specification: A definition of the tests to be performed on a unit under test (UUT) to verify conformance with its performance specification, with or without fault diagnostics, and without reference to any specific test equipment.

test procedure: A description of the tests, test methods, and test sequences to be performed on a unit under test (UUT) to verify conformance with its test specification, with or without fault diagnostics, and without reference to specific test equipment.

test program: An implementation of the tests, test methods, and test sequences to be performed on a unit under test (UUT) to verify conformance with its test specification, with or without fault diagnosis. A test program is configured for execution on a specific test system.

transducer: A device that converts a physical magnitude of one form of energy into another form, generally on a one-to-one correspondence or according to a specified mathematical formula.

unit under test (UUT): An entity that can be tested and that may range from a single component to a complete system.

value: The quantitative size of a signal attribute.

2.2 Abbreviations and acronyms

ARB	auxiliary reference burst
ARINC	Aeronautical Radio, Inc.
ASCII	American Standard Code for Information Interchange
ATC	air traffic control
ATE	automatic test equipment
ATLAS	Abbreviated Test Language for All Systems
ATS	automatic test system
BSC	basic signal component
C/ATLAS	Common/Abbreviated Test Language for All Systems (IEEE Std 716™-1995 [B13] ¹⁰)
DME	distance measuring equipment
DTIF	Digital Test Interchange Format

¹⁰ The numbers in brackets correspond to the numbers of the bibliography in Annex M.

IDL	interface definition language
IFF	identification, friend, or foe
ILS	instrument landing system
MRB	main reference burst
PRF	pulse repetition frequency
RF	radio frequency
rms	root mean square
SML	signal modeling language
SSR	secondary surveillance radar
STANAG	NATO Standardization Agreements
STD	signal and test definition
TACAN	tactical air navigation
trms	true root mean square
TPL	test procedure language
TSF	test signal framework
UHF	ultrahigh frequency
UUT	unit under test
VHF	very high frequency
VOR	VHF omnidirectional range
XSD	XML Schema Definition
XML	extensible markup language

3. Structure of this standard

3.1 Layers

This standard has a layered format depicted in Figure 1. Each layer builds on items defined in previous layers. This format does not require that each layer use only items in its immediate lower level, but does imply that each layer has to be fully defined in terms of its lower level layers.

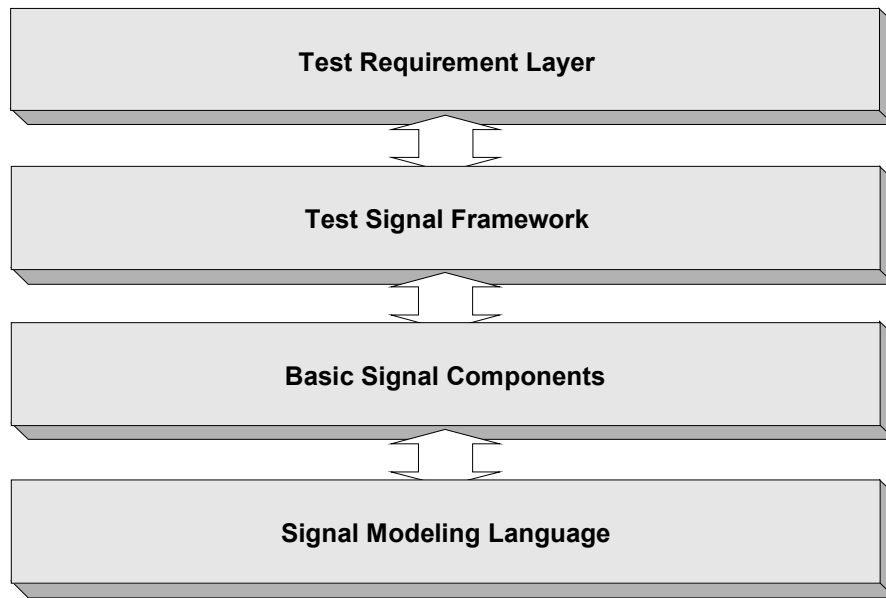


Figure 1—Standard Layers

This standard provides the capability to describe and control signals and allows the user to choose the operating environment, including the choice of programming language. The STD test requirements take the form of signal model definitions, which can be written in any programming or declarative language, including XML or object-oriented environment.

The link to Abbreviated Test Language for All Systems (ATLAS) and Common/Abbreviated Test Language for All Systems (C/ATLAS) standards is preserved as part of the test requirements layer and are mapped to signals that have formal definitions tied to the test signal framework (TSF) and basic signal component (BSC) layers. This link comprises an example TSF library for ATLAS (see Annex E) and an annex showing how ATLAS NOUNS and NOUN MODIFIERS may be supported (see Annex J).

3.2 Signal Modeling Language (SML) layer

The SML layer provides the mathematical definitions that support the description of BSCs. This mathematical underpinning provides evidence that the signals defined by BSCs can be functionally compared and simulated. The SML signal definitions form the basis for reuse that is essential to the extension of STD capabilities without a corresponding explosion in nomenclature and complexity.

3.3 BSC layer

The BSC layer provides reusable, formally described, fundamental signal classes. These classes define the lowest level of signal building blocks available to the STD environment. Each BSC is described by its class name, class type, properties and default values, XML Schema definitions (XSDs), interface definition language (IDL) description, and SML signal definition.

3.4 TSF layer

The TSF layer identifies how libraries of reusable, formally described signal classes are defined. The content of a TSF library is a collection of domain-specific signal model definitions made up from other TSF signals and/or BSCs.

The TSF layer provides for TSF libraries, which are the extendibility mechanism that allows the creation of additional signal class definitions. Each TSF class within a TSF library is described by its class name, class type, properties and default values, XSD schema description containing an interface definition, a static signal model definition, and a textual description. The IDL interface can be created by taking each TSF property in turn and creating an equivalent runtime attribute of the same type to define the TSF programming interface. The TSF Library XSD can be derived to validate XML signals using TSF signals, by concatenating each of the XSD schema interface definitions into a single file.

A TSF library contains a collection of TSF classes, which themselves may refer to other TSF classes. In order that the library definition is complete, any TSF classes so referenced shall be provided.

3.5 Test requirement layer

The test requirement layer allows test descriptions (e.g., test requirements, test procedures, test programs) to be formally described by combining STD signals with features that satisfy the carrier language requirements (see Annex G).

Test specifications and signal libraries conforming to the requirements of this layer may be ported between different ATSS with the same functional capability and carrier language. Minimal translation would be required to convert between different carrier languages.

Tests and signals may be described using XML, carrier language procedural calls to the IDL interface, or the STD-specific test procedure language (TPL). The TPL allows test actions to be formally described in a stylized textual format suitable for preprocessing into a target carrier language.

3.6 Using the layers

A user need only refer to the layers that are required to directly support a signal or test that needs to be defined. In many cases, only BSCs and TSFs are required to describe a signal using, for example, XML descriptions. Signal descriptions may be reused as required with different attribute values. They are created when required and destroyed when their function is complete, e.g., a signal used in a measurement is destroyed when the measurement is complete (unless otherwise determined in the test).

All names used for entities defined in this standard are case sensitive, e.g., signal names, attribute names. However, names of elements within the same scope shall not be differentiated solely by case.

4. Signals and SignalFunctions

4.1 Introduction

Throughout all the layers, there exists the concept of signals. Signals are the outputs from SignalFunctions, where SignalFunctions can be interconnected and built up as required to provide the required signal

characteristics. A signal model is regarded as the collection of these built up SignalFunctions used to generate a signal.

The signal is considered “physical” only when it is applied to a UUT pin interface via a connection. By definition all BSCs and TSFs are SignalFunctions. SignalFunctions are characterized by their type, which may be typeless, generic, abstract or more commonly map onto a physical type e.g. Constant(Voltage). Examples of typeless SignalFunctions are events. Generic SignalFunctions (such as Sum) inherit their type from their inputs. Abstract SignalFunctions provide typeless value information (such as RMS sensor). An example of a type that maps onto a physical type is Voltage, such as in a Sinusoidal signal of type Voltage. The type of a SignalFunction also includes the reference type; most signals encountered in this standard have the reference type Time. Therefore, a Sinusoidal(Voltage, Time) signal describes how the signal’s voltage changes with respect to time, in a sinusoidal manner.

The use of types is such that SignalFunctions can be combined to build signals only when they have compatible types such as when one or more of the types are typeless, abstract, or generic or where the different signal types can be provided independently. Types that appear different and that represent a transform are allowed, as they represent a different method of specifying the same signal, e.g., Pulse (Voltage, Frequency) with Sinusoid (Voltage, Time). An example of an illegal signal definition is one in which both current and voltage are specified because both cannot be controlled independently. Extensibility is served by providing the capability to formally describe new signals by creating them from the existing signals from either the TSF or BSC layer. SignalFunctions are described in detail in Annex B.

NOTE—The standard does allow a voltage to be specified together with a current limit (or vice versa) because limit signals are generic types.¹¹

A physical signal, event, or digital stream element may be active or inactive and at the same time it may be gated on or off. Thus, there are four possible states that may be adopted: active & gated on, active & gated off, inactive & gated on, or inactive & gated off. Not all four states are applicable to all signals, events, and digital elements. The states that can be adopted by each may be illustrated diagrammatically (see Figure 2, Figure 3, and Figure 4). These figures provide a simplified indication of the states that can be attained by a sources. The states attained by conditioners shall also depend upon the signals on their inputs.

4.2 Physical signal states

Figure 2 shows the states available to a signal. Before a signal exists, it may be considered to be in the “No Signal” (Z) state. When it is invoked, it passes into the active state and, in the simplest case, appears with its specified value (V), i.e., the “Signal with Value” state. The value may be changed while the signal exists, by the value changing from V_X to V_Y and so on. If the signal is provided with a gate, the signal may be gated on and off by external events. When the signal is gated on, the signal is output with its specified value. When gated off, the signal has no value, i.e., the signal is active but nothing is known about its value. When subsequently gated on again, the signal appears with its current value. In other words, the value may have changed while it was gated off, but cannot be determined until it is gated on again.

¹¹ Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement this standard.

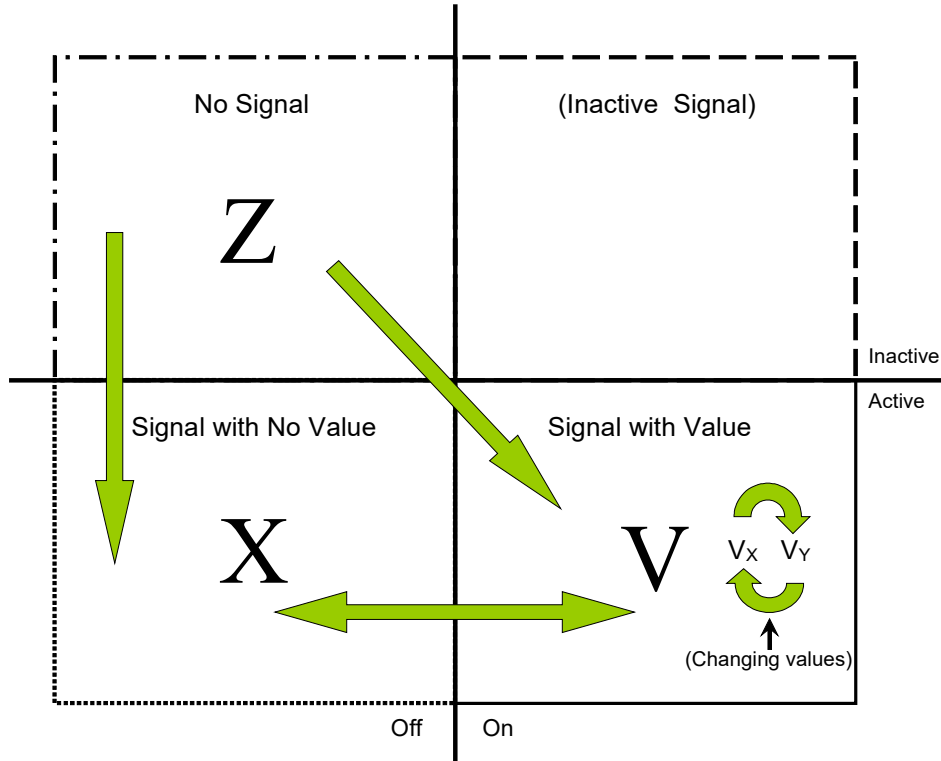


Figure 2 —Signal states

Generally, the signal cannot be switched into the “Inactive Signal” or “No Signal” states by any action on the signal. The signal is returned to the “No Signal” state only when it is destroyed, e.g., when it is no longer required. At that time, the signal may be considered to have passed momentarily through the “Inactive Signal” state.

The output of a signal carries its own event information: an active signal is equivalent to “Event Active,” and “No Signal” is equivalent to “Event Inactive.”

4.3 Event states

Figure 3 shows the states available to an event. Before an event exists, it may be considered to be in the “No Event” (Z) state. When it is invoked, it passes into the gated on state, but may be active or inactive. An event may change between active and inactive while gated on. Signals may be gated on and off by an event as explained in 4.2. Other events may be gated on and off by an event, and as shown in Figure 3, these other events are switched between the “No Event” state and either the “Event Inactive” or “Event Active” state, as appropriate.

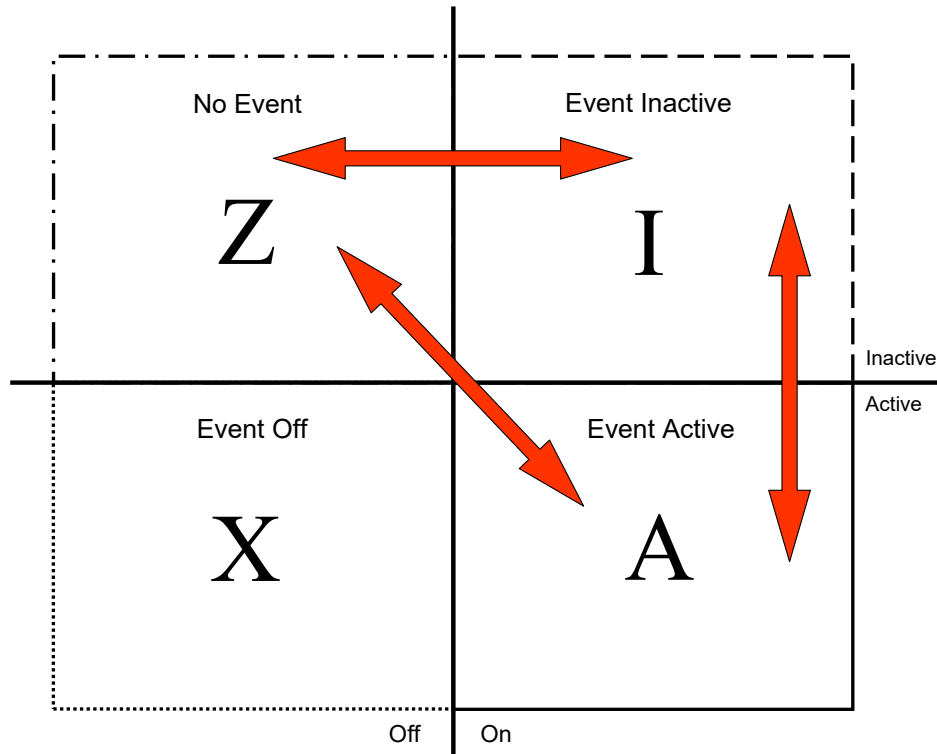


Figure 3 —Event states

When used to gate a signal, there is very little difference between an event in the “No Event” state and the “Event Inactive” state; either of these shall effectively gate the signal off. Only the “Event Active” state shall gate the signal on. The difference is apparent when gating an event such as a NotEvent. A NotEvent changes an “Event Active” state at its input to an “Event Inactive” state at its output, and vice versa. Gating a NotEvent shall change its output between “No Event” and the appropriate “Event Active” or “Event Inactive” state.

The fourth state, “Event Off,” may be considered not to exist for an event source.

4.4 Digital stream states

Figure 4 shows the states available to elements of a digital stream. When there is no digital signal, it may be considered to be in the “Digital Off” state. A digital stream, whether serial or parallel, may comprise a series of H, L, X, and Z characters, which represent the usual digital states as shown in Figure 4.

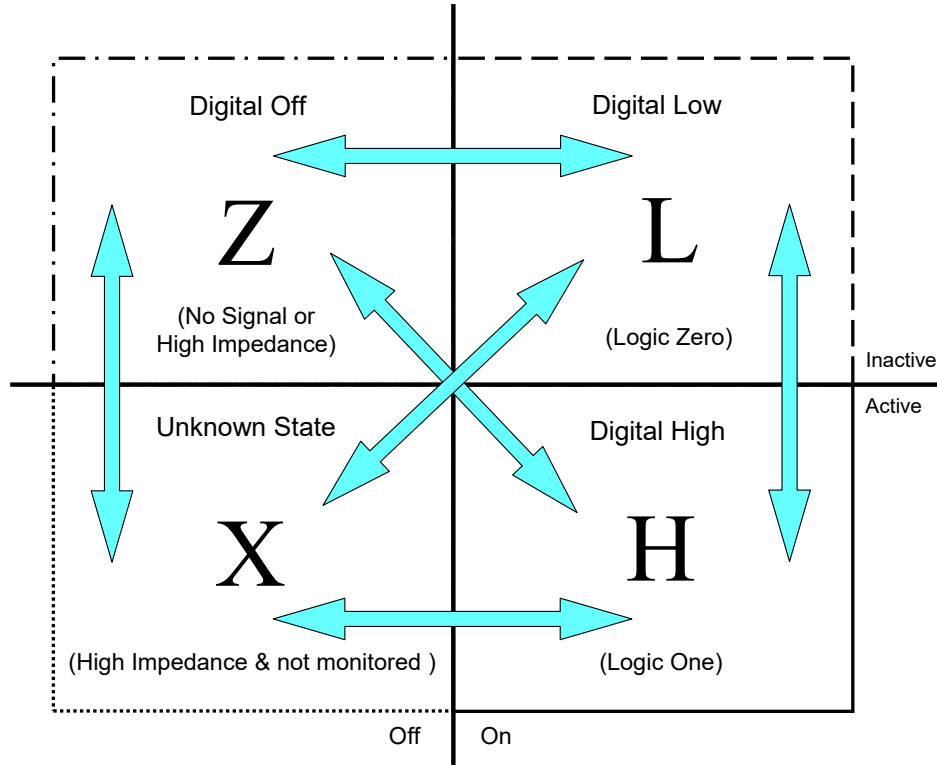


Figure 4 —Digital stream states

A “Digital Off” (Z) state indicates that there is no signal. The “Digital High” (H) and “Digital Low” (L) states represent the normal two digital states (0 and 1). The “Unknown State” (X) represents the fact that a digital signal is present, but it is not possible to know what it is. It is not driving a 1 or 0 when used as a stimulus, and it is not monitored when used as response information (a “do not care”). Both the X and Z states shall be represented by an X (Signal with No Value) state when converted into a physical signal, usually in the form of a high impedance output.

It is important to note that Figure 4 applies only to digital stream information. After a digital stream has been converted into a physical signal, the description in 4.2 applies.

5. SML layer

The SML layer provides the definition of the behaviors of signals, both analog and digital, as well as their functions in any number of domains. It provides this capability by giving a number of predefined signal behaviors that can be combined as necessary to produce the desired BSC behavior definition. Annex A describes the use of SML to define signals, the measurement of signal parameters, and the conditions that a signal shall meet.

The SML provides an exact mathematical definition for each BSC, in terms of dependent and independent variables, by using the de-facto functional programming concepts of Haskell [B7]. Each definition represents the functioning of a component, which is a requirement for use and reuse. This representation is accomplished by giving a formal definition of the syntax and predefined signals. The SML Layer provides for simulating the modeled signal, plotting against its definition domain, and measuring its various properties. Within the SML layer, the type of a signal is known as the dependent variable, and the reference type is known as the independent variable.

6. BSC layer

This Clause describes the methodology adopted to define signals with BSCs and the mechanisms by which they may be combined and synchronized.

6.1 BSC layer base classes

All BSC classes used to define signals are derived from the SignalFunction class. BSC classes define their properties using one of the base classes shown in Table 1 or using traditional programming types. This class approach is useful for categorizing BSCs according to their characteristics, behavior, and interfaces.

Table 1—BSC base classes

Base class	Description
SignalFunction	The base class of all BSCs
Signal	Allows BSCs to exchange information
PulseDefns	Defines a group of pulses
Physical	Real, dimensioned signal values

6.2 General description of BSCs

BSCs are the fundamental components of this standard. The BSCs are the building blocks used to define more complex signals and cannot be decomposed into simpler components.

BSCs are used to build signal models, which define the required signal. A signal model can contain a single BSC to define a simple signal or combined BSCs to define a more complex signal.

BSCs can be used to either define static signal models or perform dynamic signal programming by programmatically changing signal models through a programming language.

Signal models represent static signal descriptions, where the signal model does not change over time. The BSC control interface (i.e., the IDL description) can also be used to define dynamic signal definitions, where the value of the attributes or the signal model changes while the signal is being used.

The signal described by a signal model can be used to create a source signal or to measure a signal characteristic attribute.

Unless otherwise stated, the default reference type for a BSC is Time; and where required, the default signal type is Voltage.

Each BSC is described using object orientation terminology as follows:

- A class derived from SignalFunction base class (or subclass)
- Class type and reference type description
- Attributes and default values
- A control interface (defined using an IDL description)
- A formal SML description

- f) An XSD entry

The following annexes describe BSC features:

- Annex A gives details of the BSC formal SML descriptions.
- Annex B gives details of the BSC classes and attributes.
- Annex C gives details of the dynamic signal model behavior.
- Annex D gives a list of the IDL descriptions for each BSCs.
- Annex I gives a list of the XML descriptions for mapping signal models.

In the event of conflict between descriptions within these annexes of the standard or their associated files, The later the annex the higher the precedence. As such the XML descriptions described in Annex I and provided in the XML schema definitions shall take precedence.

NOTE—Annex D and Annex I are normative annexes in that they provide the normative descriptions for the BSCs in IDL and XML, respectively. BSCs may also be described in other interface languages.

6.3 SignalFunction template

A template is used as document shorthand to define types of derived classes of SignalFunction, where each class has similar behavior and supports the same IDL description, but describes signals of different types. Note that the class name shall remain the same; only the class type changes.

The use of the template defines alternative class types, where each derived class name could equally have been written using “cut & paste” and replacing the keyword “type” with any physical class defined within this standard. Substituting `<type:...[,ref:...]>` with each alternative creates the class type that the template defines. The specific type alternatives provided in the template identify the more commonly used signal classes; once the template is defined, any type or reference can be used.

The format for the template header is as follows:

ClassName<type:typeName[||typeName]*[,ref:typeName[||typeName]*]>

Where no explicit types are provided, the default type is Voltage, and the default reference type (ref) is Time.

For example, **Sinusoid**<type:**Voltage**||**Current**||**Power**> defines the following classes, where each class supports the **Sinusoid** interface, through the IDL definition:

- a) Sinusoid (Voltage) full type: Sinusoid (Voltage, Time)
- b) Sinusoid (Current) full type: Sinusoid (Current, Time)
- c) Sinusoid (Power) full type: Sinusoid (Power, Time)

NOTE 1—Alternatives are separated by the double-bar characters (||).

NOTE 2—This template convention is adopted in the remainder of this standard.

7. TSF layer

The TSF layer describes how BSCs are combined into more complex signals and packaged for reuse in TSF libraries.

The TSF layer also provides a packaging mechanism for grouping signal models into library elements. These libraries are constructed from individual TSF classes where each class defines an interface, a signal model, and textual description. A TSF library shall generally be a collection of domain-specific signal definitions.

All TSF classes shall exist in an XML TSF library conforming to Annex I. Additional support documentation and interface files may also be provided.

7.1 TSF classes

A TSF IDL library shall be defined in terms of an IDL library module, shall contain an entry for each TSF class within the library domain, and shall be derived from the XML TSF library.

A TSF class shall be defined with a valid XML TSF element utilizing the attribute model and description elements.

A TSF class model shall be described only as a static signal model and may contain elements from the following:

- BSCs
- Other TSF classes
- Reference to the TSF class being defined

A TSF class's signal description utilizing a TSF component is identical to a signal description incorporating the complete TSF static signal model. As such, the following statements are generally true:

- A TSF component output is available only when its Gate (see Annex B) event is on.
- A TSF component restarts its operation when its Sync (see Annex B) event arrives.

TSFs provide the extendibility mechanism that allows the user community to create additional signal class definitions.

There is no difference between a TSF class built from BSCs only and a TSF class built from BSCs and/or other existing TSF classes. The resultant signal from a TSF class is determined by the operation of all its components down to the lowest level (i.e., BSCs).

When a TSF signal model makes references to itself, it creates a recursive signal model definition. In this case, the TSF is regarded as a signal transform, where each iteration is providing a more accurate signal. The actual number of iterations required is reflected by the accuracy of the signal required or uncertainty of the measurement. To be a proper signal model definition, the TSF signal transform should be convergent.

Use of self-referencing models that contain a reference to themselves and are not convergent (e.g., RS232) is deprecated and maintained for support of the earlier edition of this standard. Such TSF definitions should now be defined by an external reference (see 7.3).

7.2 TSF signals defined by a model

Each signal described in a TSF may have the following information:

- Title, which indicates the syntax of the TSF class name.
- Definition of the signal.
- Model diagram, which shows the component parts of the signal and their relationship. This diagram is provided to give a convenient pictorial representation of the signal model. In the event of any conflict between the model diagram and the model description table, the model description table takes precedence.
- Interface properties table listing the TSF interface properties.
- Notes as needed for any additional explanations.
- Model description table, which lists the component parts of the signal and their relationship. Each element in the model is supported by a definition in the BSC.
- Equations as needed to define the operation of the model.
- Rules as needed relating to the operation of the model.

7.2.1 Interface properties table

The interface properties table shall comprise the following five columns:

- *Description*. The descriptive name of the attribute.
- *Name*. The syntactical name of the attribute as defined in the BSC.
- *Type*. The attribute type as defined in the BSC. If the type is given as physical, the actual type shall be chosen from the list provided with the signal title. If more than one attribute has the type given as physical, then all attributes shall be of the same type.
- *Default*. If a value is provided, the default is the value that the attribute shall take if the attribute is not specifically defined. If no default value is given, then the user shall provide a value.
- *Range*. If a range is given, it indicates the valid range for the attribute. The attribute value shall fall within this range.

Table 2 shows an interface property table for a sample signal (in this example, AC_SIGNAL). It indicates that the user should provide the physical type, amplitude, and frequency as a minimum. The physical type is usually determined from the units given with the value; for example, an amplitude of 10 V indicates that the physical type is Voltage. If no DC Offset or initial phase angle is defined, each attribute shall assume the default values given in the table. Note that the DC Offset is of the same type as the AC Signal amplitude.

Table 2—Example of TSF interface property table

Description	Name	Type	Default	Range
AC Signal amplitude	ac_ampl	Physical	—	—
DC Offset	dc_offset	Physical	0	—
AC Signal frequency	freq	Frequency	—	—
AC Signal phase angle	phase	PlaneAngle	0 rad	0 – 2 π rad

An attribute of a TSF class that is subsequently not used in the signal model description cannot be used to change or control the signal. It may be used to describe the capabilities required from the signal; for example, if an attribute Distortion Max 3% were added to the interface, it would mean that distortion needs to be less than (or equal to) 3%. As such, all such capability attributes are optional.

7.2.2 Model description table

The model description table describes the signal model using a network list format. The model description table shall comprise the following six columns:

- *Name*. The given name of the BSC or TSF within the model.
- *Type*. The type of the BSC or TSF.
- *Terminal*. The terminal names of the BSC or TSF, usually in the order of outputs followed by inputs.
- *Inputs*. The signal or attribute that is connected to the terminal listed in the “Terminal” column.
- *Output*. The output(s) of the BSC or TSF. Some BSC or TSF signal outputs shall be inputs to other BSCs or TSF signals within the model, and at least one shall be the output from the model.
- *Formula*. An optional mathematical definition (or constant value) of the function or input. Where the formula references another attribute from within the TSF model, the formula value should be evaluated every time the reference value changes. The implicit translation from one type to another shall be the natural translation between the types supported by the expression handler defined.

Table 3 shows a signal model for AC_SIGNAL. A BSC of type Sinusoid (named AC Component) is summed with a BCS of type Constant (named DC Offset) to create the AC_SIGNAL.

Table 3—Example of TSF model description table

Name	Type	Terminal	Inputs	Output	Formula
AC Signal	Sum	Signal [Out]	—	AC_SIGNAL	—
		Signal [In]	DC Offset	—	—
		Signal [In]	AC Component	—	—
AC Component	Sinusoid	Signal [Out]	—	AC Signal	—
		amplitude	ac_ampl	—	—
		frequency	freq	—	—
		phase	phase	—	—
DC Offset	Constant	Signal [Out]	—	AC Signal	—
		amplitude	dc_offset	—	—

7.2.3 TSF figures

Each TSF figure provides a pictorial description of the model description and interface properties. These figures give an intuitive representation of a signal model, which is based on the BSC diagram (see Figure B.1). They do not infer the use of any specific signal resources. If the TSF figure is not consistent with the model description table or the interface properties table, then the tables take precedence. The external interfaces and properties are illustrated in the same manner as for an individual BSC (see Figure B.1). The internal structure shows how BSCs and other TSF components are combined to provide the required signal. See Annex E and Annex F for examples of TSF figures.

7.2.4 Other properties

A TSF may also have additional properties that are not used directly in the TSF model. They may be used to describe qualitative properties such as capability attributes, which provide additional information about a signal (to help select an appropriate instrument) but do not modify the signal.

TSFs may have two attributes that have significance only within a TSF library:

- Hidden, which indicates that the TSF is available for use only by other TSFs within the library, i.e., it is hidden from use and cannot be used directly outside the library. To hide a TSF, set the Hidden attribute to true.
- Group, which provides the facility to indicate that a set of TSFs is part of a single group. TSFs are considered to be in the same group if their Group attributes have the same name. Subgroups can be introduced by using concatenated group names separated by a colon (:), e.g., “group1:subgroupA.”

All attributes used within a TSF shall have their Type defined. In the event that the Type of an attribute is left undefined, it shall be assumed to of Type xs:string.

7.3 TSF signals defined by an external reference

A TSF may be defined by an external reference such as an internationally recognized standard. If this method is used to define a TSF, the reference shall be complete and explicit with a full source reference.

Any attribute names used shall be defined in the external reference or fully described in terms of elements in the external reference.

8. Test procedure language (TPL)

The TPL (see Annex H) provides a mechanism for users who want to document test requirements in a textual format. The use of the TPL to write test requirements is analogous to using ATLAS inasmuch as the TPL uses stylized English signal statements to describe tests and to manipulate signals. It differs from using ATLAS in that it does not provide a fully defined programming language. Instead, STD allows users to adopt their own preferred programming language in which the signal statements and the underlying semantics of tests can be written.

8.1 Goals of the TPL

The goals of the TPL are as follows:

- a) Its keywords have meanings that are normally accepted by the worldwide testing community.
- b) It is an effective means for communicating test information relating to the testing of a UUT between an originator of a test requirement and an implementer of a test requirement.
- c) Test requirements written according to the TPL rules shall be portable to implementations on different designs of test equipment that have the same testing capability no matter how it is controlled.

8.2 Elements of the TPL

The TPL requires two elements:

- a) Signal statements, which are used to configure, manipulate, control, and measure signals.
- b) Carrier language, which is a programming language in which the signal statements can be written, sequenced, observed, and generally supported.

8.3 Use of the TPL

To produce test requirements using the TPL, users shall embed the test statements in their preferred carrier language.

Users need to recognize that there shall be some translation mechanism to convert from this neutral format of the signal statements into their preferred carrier language format before the test statements can be compiled and executed.

Use of a translator to convert the neutral representation of the test statements into the carrier language format offers certain benefits in that parameter type checking and semantics checks can be conducted prior to test execution.

9. Maximizing test platform independence

The use of this standard allows signal definitions to be created without reference to their final application. This approach in turn facilitates the definition of test requirements that provide the ability to achieve test application interchangeability across different test platforms. Annex K, “Guide for maximizing test platform independence” provides guidance in how to write test requirements suitable for test platform independence.

Annex A

(normative)

Signal modeling language (SML)

A.1 Use of the SML

In general, unless a user needs to generate new keywords for the test methodology, there is no requirement for a user to refer to the SML in order to generate test requirements for a unit under test (UUT). Occasionally, a user may need to refer to the SML either for the mathematical justification or simulation of some signal construct or for the introduction of a new keyword to cover some new test application that the test methodology does not embrace. In the latter instance, a user shall need to refer to the SML in order to help ensure that any new keyword that is introduced into the test methodology is coherent with the existing predefined keywords. The purpose of this step would be to submit a proposal for future releases to the standard.

For the convenience of using processing software suites that are freely available without any restrictions to generate the signal diagrams, a derived version of the functional programming language Haskell [B7] has been adopted.

NOTE—The blocks of SML code defined in this annex are preceded and followed by a blank line. Each line is continuous starting at the “>” symbol. Code often appears to flow onto a second or subsequent line due to the restricted line length of a published standard. When the code is used with a Haskell compiler, the lines should be reinstated as continuous starting at the “>” symbol. Care should be taken to make sure that the “>” (start of line) symbol is not confused with the “->” symbol.

A.2 Introduction

This annex describes how the SML can define the signal characteristics, signal measurements, and signal conditions to meet particular applications. A SML signal model is a mathematical definition of the signal and its properties. The definition represents the functioning of a signal entity for both its use and reuse. It provides a formal definition of the syntax and semantics of predefined signal types. An execution mechanism may be provided for simulating a modeled signal by plotting it against its definition domain and measuring its various properties.

This annex provides and builds upon the following:

- Definitions for the basic signal components (BSCs)
- Definitions for the combining mechanism for piecewise continuous signals and others

As a convention, the reserved words for the SML entities are written in *italics* in the format descriptions. The SML uses a consistent naming convention, however within SML type names shall consistently start with a capital letter, function and property names shall consistently start with a lowercase letter, and names which clash with reserved language words shall have an underscore character (“_”) appended to the name.

Functional programming consists of building definitions that are subsequently used to evaluate expressions. Expressions represent questions that evaluate to answers or values, generally through symbolic substitution, using rules or functions that represent the definitions, all of which obey normal mathematical principles.

Values are partitioned into organized collections called types. Examples of predefined types are *Integer* and *Float*. All type names shall start with a capital letter in the format descriptions. The double colon symbol (::) is used to define the type of a function or expression.

Example:

```
pi :: Float — means pi is of type Float
pi = 3.14159
succ :: Int -> Int — function succ that takes an Int value and returns an Int
value
succ n = n+1
```

User types can be defined using the keyword *data* that introduces the name of the type and the type values by using a type constructor. All type constructor names shall start with a capital letter in the format descriptions.

Example:

```
data Resistance = OHM Float | KOHM Float | MOHM Float
```

In other words, a value of type *Resistance* is written using one of the three type constructor keywords, *OHM*, *KOHM*, *MOHM*, followed by a value of type *Float*, e.g., 5 k Ω is written *KOHM 5.0*.

Type classes can be defined using the keyword *class* that introduces the name of the type class and the allowed functions that operate on any type belonging to this type class. All type class names shall start with a capital letter in the format descriptions. The type class definition can also constrain the types that are allowed to belong to the type class by ensuring that they belong to other type classes.

Example:

```
class (Ord a, Show a) => Physical a where ...
```

In other words, a type class *Physical* is defined so that only types that belong to the type classes *Ord* and *Show* may belong to the new type class *Physical*. The language word “where” starts the scope of the remaining definition.

A type definition uses the keyword “instance” to declare itself a member of a particular type class.

Example:

```
instance Physical Resistance where ...
```

A.3 Physical types

All physical types are held in the module *Physical*:

```
>module Physical where
```

Table A.1 defines the physical types used within the SML and the units involved. The column headers of the units indicate the exponent of 10 that is used for the unit; in other words, if that unit is used, the basic degree is multiplied by 10 raised to that exponent. The first unit in each list is the basic unit, i.e., the unit in terms of which other units are defined.

All of the type names in Table A.1 may appear in type signatures that give the type of a signal. Each of the unit names may be used with a floating-point number or expression to create a physical constant. The form in this case is (<unit_name> <expression>), where the expression is given in normal mathematical notation.

The specification in this clause (i.e., A.3) does not include a complete definition of operations on values of a physical type. Operations are conducted on normal floating-point types and then converted into physical quantities as necessary. Conversion of a physical value into a floating-point value is accomplished using the following form:

```
>class (Ord a, Show a) => Physical a where
>    fromPhysical:: a -> Float
```

Floating-point values may be converted to physical values by prefixing them with one of the units in Table A.1. In some cases, the appropriate unit is not clear (e.g., in a very general signal creation method). In these cases, a general routine is used to convert the floating-point value into a physical value whose type is determined from the context. This conversion is accomplished by the following form:

```
>    toPhysical:: Float -> a
```

All physical types are defined with a data declaration and the instance mapping fromPhysical and toPhysical and shall also be instances of classes Eq and Ord implementing (==) and (<=) functions, using the fromPhysical function, e.g., OHM 1000 == KOHM 1 and OHM 1100 >= KOHM 1.

In Table A.1 and Table A.2, the physical types are declared in the “SML type” column, and the alternative constructor names are provided in the other SML columns. Standard physical conversions may need to be used when multiple units are used.

Examples:

```
>data PlaneAngle = RAD Float | MRAD Float | URAD Float |
>    DEG Float |
>    REV Float deriving (Show)
>instance Physical PlaneAngle where
>    fromPhysical (RAD x) = x
>    fromPhysical (MRAD x) = x * 1.0e-3
>    fromPhysical (URAD x) = x * 1.0e-6
>    fromPhysical (DEG x) = x * (pi/180)
>    fromPhysical (REV x) = x * (2*pi)
>    toPhysical x = RAD x

>instance Eq PlaneAngle where
> x == y = (fromPhysical x) == (fromPhysical y)
>instance Ord PlaneAngle where
> x <= y = (fromPhysical x) <= (fromPhysical y)

>data Resistance = OHM Float | KOHM Float | MOHM Float deriving (Show)
>instance Physical Resistance where
>    fromPhysical (OHM x) = x
>    fromPhysical (KOHM x) = x * 1000
>    fromPhysical (MOHM x) = x * 1000000
>    toPhysical x = OHM x

>instance Eq Resistance where
> x == y = (fromPhysical x) == (fromPhysical y)
>instance Ord Resistance where
> x <= y = (fromPhysical x) <= (fromPhysical y)
```

Table A.1—SML physical types and their units

SML type	Unit	Symbol	10 ⁰ SML	10 ³ SML	10 ⁶ SML	10 ⁹ SML	10 ⁻³ SML	10 ⁻⁶ SML	10 ⁻⁹ SML	10 ⁻¹² SML
PlaneAngle	radian	rad	RAD	—	—	—	MRAD	URAD	—	—
	degree	°	DEG	—	—	—	—	—	—	—
	revolution	—	REV	—	—	—	—	—	—	—
SolidAngle	steradian	sr	SR	—	—	—	MSR	—	—	—
Capacitance	farad	F	FD	—	—	—	—	UFD	NFD	PFD
Charge	coulomb	C	C	KC	—	—	—	UC	NC	—
Conductance	siemens	S	S	—	—	—	—	—	—	—
Current	ampere	A	A	KA	—	—	MA	UA	NA	—
Distance	meter	m	M	KM	—	—	MM	UM	NM	—
	inch	in	IN	—	—	—	—	—	—	—
	foot	ft	FT	—	—	—	—	—	—	—
	stat. mile	mi	SMI	—	—	—	—	—	—	—
	naut.mile	nmi	NMI	—	—	—	—	—	—	—
Energy	joule	J	J	KJ	—	—	MJ	—	—	—
	electronvolt	eV	EV	KEV	MEV	—	—	—	—	—
MagneticFlux	weber	Wb	WB	—	—	—	MWB	—	—	—
MagneticFluxDensity	tesla	T	T	—	—	—	MT	UT	—	—
Force	newton	N	N	KN	—	—	MN	UN	—	—
Frequency	hertz	Hz	HZ	KHZ	MHZ	GHZ	—	—	—	—
Illuminance	lux	lx	LX	—	—	—	—	—	—	—
Inductance	henry	H	HEN	—	—	—	MH	UH	NH	PH
Luminance	candela per square meter	cd/m ²	NT	—	—	—	—	—	—	—
LuminousFlux	lumen	lm	LM	—	—	—	—	—	—	—
LuminousIntensity	candela	cd	CD	—	—	—	—	—	—	—
Mass	kilogram	kg	KG	—	—	—	G	MG	UG	—
Power	watt	W	W	KW	—	—	MW	UW	—	—
Pressure	pascal	Pa	PA	KPA	—	—	MPA	UPA	—	—
	millibar	mbar	MB	—	—	—	—	—	—	—
Resistance	ohm	Ω	OHM	KOHM	MOHM	—	—	—	—	—
Temperature	kelvin	K	KEL	—	—	—	—	—	—	—
	deg. Celsius	°C	DEGC	—	—	—	—	—	—	—
	deg. Fahrenheit	°F	DEGF	—	—	—	—	—	—	—
Time	second	s	SEC	—	—	—	MSEC	USEC	NSEC	—
	minute	min	MIN	—	—	—	—	—	—	—
	hour	h	HR	—	—	—	—	—	—	—
Voltage	volt	V	V	KV	—	—	MV	UV	—	—
Volume	liter	L	LITER	—	—	—	ML	—	—	—

Table A.2 provides two further special physical types that are provided to support the SML.

Table A.2—Special physical types and their units

SML type	Unit	Symbol	10 ⁰ SML	10 ³ SML	10 ⁶ SML	10 ⁹ SML	10 ⁻³ SML	10 ⁻⁶ SML	10 ⁻⁹ SML	10 ⁻¹² SML
BurstLength	cycle	—	CYCLE	—	—	—	—	—	—	—
	pulse	—	PULSE	—	—	—	—	—	—	—
Ratio	decibel	dB	DB	—	—	—	—	—	—	—
	percent	%	PC	—	—	—	—	—	—	—
	scalar	—	Float	—	—	—	—	—	—	—

A.4 Signal definitions

All SML primitive signal definitions are held in the module `Pure` and make use of the previous module `Physical` and the Haskell system modules `Complex` and `FFT`.

```
>module Pure where
>import FFT
>import Data.Complex
>import System.Random
>import Physical
>infixr 7 |>
```

Any datatype can declare itself as a signal by declaring itself an instance of the class `Signal`.

```
>class Signal s where
> mapSignal:: (Physical a, Physical b) => (s a b) -> a -> b
> mapSigList:: (Physical a, Physical b) => (s a b) -> [a] -> [b]
> toSig:: (Physical a, Physical b) => (s a b) -> SignalRep a b
> isInactive:: (Physical a, Physical b) => (s a b) -> a -> Bool
> isOff:: (Physical a, Physical b) => (s a b) -> a -> Bool
> isZ:: (Physical a, Physical b) => (s a b) -> a -> Bool
> isX:: (Physical a, Physical b) => (s a b) -> a -> Bool
> isL:: (Physical a, Physical b) => (s a b) -> a -> Bool
> isH:: (Physical a, Physical b) => (s a b) -> a -> Bool
> mapSignal = mapSignal . toSig
> mapSigList = map . mapSignal
> toSig = FunctionRep . mapSignal
> isInactive = isInactive . toSig
> isOff = isOff . toSig
> isZ s t = (isInactive s t) && (isOff s t)
> isX s t = not (isInactive s t) && (isOff s t)
> isL s t = (isInactive s t) && not (isOff s t)
> isH s t = not (isInactive s t) && not (isOff s t)
```

An instance of a signal can be observed at a specific point in its domain; in other words, the instance effectively calls the function associated with the signal by providing an argument to the function. This capability is referred to as mapping the signal onto a specific point and has the following form:

`mapSignal <signal_name> <independent_value>`

A common signal representation is used to define all signals. A signal representation can be represented any of the following:

- By a function.
- As piecewise continuous windows made up of other signal representations.

- By the value not present, as in an `inActive` event, and detected through the use of the `isInactive` method. This represents a signal whose event state is inactive.
- By the signal not being present and detected through the use of the `isOff` method. This represents a signal in the tri-state or Z state.

```
>data SignalRep a b =
>  ZRep      |
>  XRep      |
>  NullRep   |
>  FunctionRep (a -> b) |
>  PieceContRep (PieceCont a b)
```

A signal representation is an instance of the class `Signal` and an instance of class `Calculus`, which provides function integration and differentiation methods.

```
>instance Signal SignalRep where
>  mapSignal ZRep = \t -> toPhysical (-0.0)
>  mapSignal XRep = \t -> toPhysical 0.0
>  mapSignal NullRep = \t -> toPhysical (-0.0)
>  mapSignal (FunctionRep f) = f
>  mapSignal (PieceContRep f) = mapSignal f
>  mapSigList (FunctionRep f) = map f
>  mapSigList (PieceContRep f) = mapSigList f
>  toSig = id
>  isInactive ZRep _ = True
>  isInactive NullRep _ = True
>  isInactive (PieceContRep f) t = isInactive f t
>  isInactive _ _ = False
>  isOff ZRep _ = True
>  isOff XRep _ = True
>  isOff (PieceContRep f) t = isOff f t
>  isOff _ _ = False

>class Calculus a where
>  i_dx :: Float -> a -> a
>  d_dx :: Float -> a -> a
>
>instance (Physical a, Physical b) => Calculus (SignalRep a b) where
>  i_dx dx ZRep = ZRep
>  i_dx dx XRep = XRep
>  i_dx dx NullRep = NullRep
>  i_dx dx s = FunctionRep (\x -> toPhysical $ dx * (sum $ map fromPhysical $
>                                     mapSigList s $ map toPhysical [0,dx..fromPhysical x]))
>  d_dx dx ZRep = ZRep
>  d_dx dx XRep = XRep
>  d_dx dx NullRep = NullRep
>  d_dx dx s = FunctionRep (\x -> toPhysical $
>                                     (fromPhysical (mapSignal s x) - fromPhysical (mapSignal s
>                                     (toPhysical((fromPhysical x)+dx)))) / dx)
```

A new signal definition is created by either defining a function that returns a signal representation (e.g., `SignalRep a b`) or creating a new data class that supports the signal class interface. When defining a new signal, the name may be used twice in the definition:

- First, to give the type of the independent and dependent variables of the signal. This use is optional and can often be inferred from the use of the parameters in the signal definition.
- Second, to define the signal itself, using the mechanisms provided for basic signal definitions.

SML signals represent single channel, or scalar, signals. Multichannel signals, or vector signals, can be constructed by using the Haskell array `[]` or Tuple `()` collection classes.

A.5 Pure signals

The SML provides a number of mathematically pure signal definitions, which represent the behavior of *Active* and *On* signals without any representation of noise, distortion, or spurious phenomena. The user can take these signals and build more complex signals with them using the construction techniques. These pure signals are divided and presented as signals that are nonperiodic (i.e., do not have a given period or frequency) and signals that are periodic (i.e., have a given period or frequency).

A.5.1 Nonperiodic signals

The signals presented in this subclause have no implicit period. They identify specific, one-time events or functions that do not repeat themselves.

A.5.1.1 Constant

A constant signal retains its given level for all values of its independent variable. It has the following form:

```
>constant:: (Physical a, Physical b) => b -> SignalRep a b
>constant level = FunctionRep (\t -> level)
```

A.5.1.2 Linear

A linear signal forms a line within a plane. The line is defined by its slope and intercept. The equation is the standard $y = mx + b$. It has the following form:

```
>linear:: (Physical a, Physical b) => Float -> b -> SignalRep a b
>linear m b =
> FunctionRep (\x -> toPhysical (m*(fromPhysical x) + (fromPhysical b)))
```

A.5.1.3 Random

A random signal consists of an unbounded number of random levels between zero and one. It takes two parameters: an integer seed and a sampling interval, which is of the same type as the independent variable. The same random signal is given for the same seed; the seed enables deterministic testing. It has the following form:

```
>rand:: Integer -> [Float]
>rand i = randoms (mkStdGen (fromInteger i))
>
>random:: (Physical a, Physical b) => Integer -> a -> SignalRep a b
>random seed sample_interval = let
> waveform:: (Physical a, Physical b) => a -> [b] -> SignalRep a b
> waveform samp ampls = let
>   stepSlope y y' = ((fromPhysical y') - (fromPhysical y))/(fromPhysical samp)
>   makeWin (v,v') = Window LocalZero (TimeEvent (fromPhysical samp))
>                                     (linear (stepSlope v v') v)
>   points = cycle ampls
>   in pieceRep (Windows (map makeWin (zip points (tail points))))
> in waveform sample_interval (map toPhysical (rand seed))
```

A.5.1.4 Exponential

An exponential is a damping factor, which is equivalent to the following function:

$$e^{-t/\tau}$$

where

t is the time interval
 τ is the damping factor.

An exponential allows any signal to be damped over a given time, according to a floating-point damping factor:

```
>expc:: (Physical a, Physical b) => Float -> SignalRep a b
>expc damp = FunctionRep (\t->toPhysical (exp (-((fromPhysical t)*damp))))
```

A.5.2 Periodic signals

The signals defined in A.5.2.1 and A.5.2.2 have either a period or a frequency assigned to them. In other words, they repeat their values for some fixed value of their independent variable.

A.5.2.1 Sinusoid

A sinusoid is the familiar sine relationship. It takes an amplitude, a frequency, and a phase angle. The amplitude has the type of the dependent variable, the frequency is of type Frequency, and the phase angle is a PlaneAngle. The result is given as follows:

$$A * \sin(\omega t + \theta)$$

where

A is the amplitude
 ω is the frequency (multiplied by 2π)
 θ is the phase angle

It has the following form:

```
>sine:: (Physical a, Physical b) =>
>      b -> Frequency -> PlaneAngle -> SignalRep a b
>sine mag omeg phase =
>      FunctionRep (\x -> toPhysical ((fromPhysical mag)*
>      (sin(2*pi*(fromPhysical omeg)*(fromPhysical x) +
>      (fromPhysical phase)))))
```

A sinusoid is a simpler form of a more complex function, whose amplitude, phase angle, and frequency are functions rather than scalars. This has the same format as that above:

```
>sineFunc::(Physical a, Physical b)=>
>   SignalRep a b->SignalRep a b->SignalRep a b
>sineFunc mag omeg phase =
> FunctionRep (\x-> toPhysical((fromPhysical (mapSignal mag x))*
>   (sin(2*pi*(fromPhysical (mapSignal omeg x))*(fromPhysical x)
>   + (fromPhysical (mapSignal phase x))))))
```

where

the type of all three signals is from the independent to the dependent type

A.5.2.2 Waveform

A waveform is defined by a sampling interval and a list of values. The waveform cycles through those values sequentially and infinitely, starting from zero. The width of each window is the same, and each window consists of a line segment.

A waveform has the following form:

```
>waveform:: (Physical a, Physical b) => a -> [b] -> SignalRep a b
>waveform samp lev =
>   let s = fromPhysical samp
>   stepSlope y y' = ((fromPhysical y') - (fromPhysical y)) / s
>   makeWin (v,v') = Window LocalZero (TimeEvent s)(linear(stepSlope v v') v)
>   points = cycle lev
>   in pieceRep (Windows (map makeWin (zip points (tail points))))
```

A.6 Pure signal-combining mechanisms

Clause A.5 established a number of ways to create signals, and this clause (i.e., A.6) presents some mechanisms of combining signals together. Each mechanism shall be handled separately.

A.6.1 Piecewise continuous signals

A piecewise continuous signal is made up of a number of windows, each with its own signal defined within the window. Window boundaries are defined by events.

A.6.1.1 Window events

An event marks the transition from one window to another. An event can be an amount of time either relative or absolute (i.e., a time event or fixed event), a function of another signal (i.e., function event), the moment when another signal becomes active (i.e., active event), or a fixed number of event occurrences (i.e., a burst event). Each type of event has a distinct form.

```
>data Event a b=
```

- a) A time event is either a specified relative period of time or a fixed absolute point in time. The window lasts for the duration of time given within the immediate scope of the outer signal. It has the following form:

```
> TimeEvent Float |
> FixedEvent Float |
```

Although the type of event is called a time event, it may be of any physical type, thus, the use of the floating-point expression.

- b) A function event is when the argument of this function is a function that, in turn, takes a signal and produces a boolean result. It takes the following form:

```
> FunctionEvent (Float -> Bool) |
```

- c) An active event is when a signal representation transitions from a ZRep or NullRep representation to a non-ZRep or non-NullRep representation, i.e., isInactive transitions to False. It takes the following form:

```
> ActiveEvent (SignalRep a b) |
```

- d) A burst event is triggered by a given number of triggers of another defined event. It takes the following form:

```
> BurstEvent Int (Event a b)
```

A user may use the following inf expression with TimeEvent to specify an infinite period:

```
>inf = (1/0)::Float
```

A user may determine the absolute time (or the value of the independent variable) when a given event occurs by the following expressions:

```
>timeOccurs:: (Physical a, Physical b) => (Event a b) -> a
>timeOccurs e = toPhysical (eventOccurs e 0.0)

>eventOccurs:: (Physical a, Physical b) => (Event a b) -> Float -> Float
>eventOccurs (TimeEvent t) x = x+t
>eventOccurs (FixedEvent t) x = if (t>x) then t
>                                else error ((show x) ++ ">" ++ (show t))
>eventOccurs (FunctionEvent f) x = stepEval f x
>eventOccurs (BurstEvent i e) x =
>    if i == 1 then
>        eventOccurs e x
>    else
>        eventOccurs (BurstEvent (i-1) e) st
>    where st = eventOccurs e x
>eventOccurs (ActiveEvent ges) x = let {
>    ;isInactive NullRep = True
>    ;isInactive ZRep = True
>    ;isInactive XRep = False
>    ;isInactive _ = False
>    ;active True st x w@((Window z e NullRep):ws) = active False st x w
>    ;active True st x w@((Window z e ZRep):ws) = active False st x w
>    ;active True 0.0 x _ = 1.0e-38 -- must not be zero (0.0)
>    ;active True st x _ = st
>    ;active False st x [] = inf
>    ;active False st x ((Window z e s):ws) = let {
>    ;et = eventOccurs e st
>    } in active ((isInactive s)&&(et>x)) et x ws
```

```
>      }in active (0.0>=x) 0.0 x (functionWindows ges)

>stepEval:: (Float -> Bool) -> Float -> Float
>stepEval f x = if not (f x) && f x' then x' else stepEval f x'
>      where x' = x + epsilon x

>epsilon:: Float -> Float
>epsilon x | inf==x = x
>      | 0==x = epsilon 1
>      | otherwise = encodeFloat (a+1) b - x where (a,b)=decodeFloat x
>---epsilon x = let eps = abs (x/1000) in if (1.0e-5 < eps || eps == 0.0)
>      then 1.0e-5 else eps
```

The boolean function above may be any sort of function that takes physical value and produces a boolean value. As an example, the transition functions are given below. The transition functions take a transition point, a signal, and a time (or value of the type of the independent variable of the signal) at which to perform the test; and they produce the value true if the signal has crossed that value since the last sample and the value false otherwise. The function hilo produces true on a falling edge, and the function lohi produces true on a rising edge. The transition cannot be detected prior to time zero (e.g., 0.0). The forms of these functions are as follows:

```
hilo <transition_point> <signal_name> <test_point>
lohi <transition_point> <signal_name> <test_point>
```

A.6.1.2 Windows

A window is specified by an event, which gives the width of the window, and a function, which specifies the value of the signal within the window.

There is an additional complication when determining the signal value within a window. In some cases (e.g., the normal use of a time window), the beginning of the window is to be regarded as time 0.0 for the purposes of evaluating the signal. In other cases (e.g., selecting between two different signals at given points on a time line), it may be required to evaluate the signal against the global time zero (i.e., the time zero of the outer piecewise continuous signal). Therefore, a flag is included that determines the zero against the event defining the value of the signal within that window.

The form of a window definition is, therefore, as follows:

```
>data FunctionWindow a b = Window ZeroIndicator (Event a b) (SignalRep a b)
```

A ZeroIndicator flag is one of two identifiers, LocalZero or GlobalZero, as follows:

```
>data ZeroIndicator = LocalZero | GlobalZero deriving (Eq, Show)
```

A helper functionWindows is provided to extract any FunctionWindow component from a SignalRep:

```
>functionWindows::(Physical a, Physical b) =>
>      (SignalRep a b) -> [FunctionWindow a b]
>functionWindows (PieceContRep (Windows ws)) = ws
>functionWindows s = [Window LocalZero (TimeEvent inf) s]
```

The splice function combines multiple FunctionWindows by splicing them into a single FunctionWindow with each segment bounded by the next event and having their SignalRep determined by the first parameter:

```
>splice :: (Physical a, Physical b) =>
> (SignalRep a b -> SignalRep a b -> SignalRep a b) ->
> [FunctionWindow a b] -> [FunctionWindow a b] -> Float -> [FunctionWindow a b]
>splice f [] [] x = []
>splice f ((w@(Window z e s)):wl) [] x =
> (Window z e (f s ZRep)): (splice f wl [] x)
>splice f [] ((w@(Window z e s)):wl) x =
> (Window z e (f ZRep s)): (splice f [] wl x)
>splice f ((w@(Window z e s)):wl) ((w'@(Window z' e' s')):wl') x = let {
>   ;et = eventOccurs e x
>   ;nt = et-et'
>   ;win = Window z e (f s s')
>   ;dw = Window z (TimeEvent nt) s
>   ;et' = eventOccurs e' x
>   ;nt' = et'-et
>   ;win' = Window z' e' (f s s')
>   ;dw' = Window z' (TimeEvent nt') s'
> } in if abs(nt)<epsilon et then win : (splice f wl wl' et) else
>      if et<et' then win : (splice f wl (dw':wl') et)
>      else win': (splice f (dw:wl) wl' et')
```

The default behavior of conditioners is described by the helper function stdConditioner. When the input of the conditioner is not active, no operation occurs.

```
>stdConditioner::(Physical a, Physical b) =>
> (SignalRep a b -> SignalRep a b) ->
> [FunctionWindow a b] -> [FunctionWindow a b]
>stdConditioner fn (fw@(Window z e ZRep):fws) =
> (Window z e ZRep): stdConditioner fn fws
>stdConditioner fn (fw@(Window z e NullRep):fws)= fw: stdConditioner fn fws
>stdConditioner fn (fw@(Window z e XRep):fws)= fw: stdConditioner fn fws
>stdConditioner fn ((Window z e (PieceContRep(Windows fws'))):fws) =
> (Window z e (PieceContRep (Windows (stdConditioner fn fws')))):
> stdConditioner fn fws
>stdConditioner fn ((Window z e s):fws) =
> (Window z e (fn s)): stdConditioner fn fws
```

A.6.1.3 Piecewise continuous functions

A piecewise continuous function is represented by the PieceCont datatype, is built from windows, and is an instance of a signal. The function getWindow is available to retrieve the local time and first window where the window event happens after the required time.

```
>getWindow:: (Physical a, Physical b) =>
> Float -> Float -> [ FunctionWindow a b ] ->
> (Float, FunctionWindow a b, [ FunctionWindow a b ])
>getWindow st t [] = (t, Window LocalZero (TimeEvent (2*t)) ZRep, [])
>getWindow st t (w:wl) = if t < et then (t', w, wl)
>                        else getWindow et t wl
>   where et = eventOccurs e st
>         (Window z e s) = w
>         t' = if z == LocalZero then t-st else t
>
>data PieceCont a b = Windows [FunctionWindow a b]
>instance Signal PieceCont where
> mapSignal (Windows []) t = mapSignal ZRep t
```

```
> mapSignal (Windows wl) t = (mapSignal s) (toPhysical t')
>   where (t', (Window z e s), wl') = getWindow 0.0 (fromPhysical t) wl
> toSig = pieceRep
> isInactive (Windows []) t = True
> isInactive (Windows wl) t = (isInactive s) (toPhysical t')
>   where (t', (Window z e s), wl') = getWindow 0.0 (fromPhysical t) wl
```

The operator for combining Windows is `|>`. A window that has no duration and is completely empty is called `nullWindow`.

```
>nullWindow = Windows []
>(|>)::(Physical a, Physical b) =>
>   FunctionWindow a b->PieceCont a b->PieceCont a b
>(|>) w (Windows wl) = Windows (w:wl)
```

This operator, as well as the `nullWindow`, allows the specification of the piecewise continuous signal. The form of a piecewise continuous signal is, therefore, as follows:

```
[ cycleWindows ( ) <window> <<|><window>>>|> nullWindow [ ] ]
```

The preceding piecewise continuous signal, `cycleWindows`, is used when the piecewise continuous function is intended to repeat infinitely for all time; otherwise, after the last window, the signal value returns to zero. Of course, the closing parenthesis is needed only if the signal specification is preceded by `cycleWindows` and the opening parenthesis.

```
>cycleWindows:: (Physical a, Physical b) => PieceCont a b -> PieceCont a b
>cycleWindows (Windows wl) = Windows (cycle wl)
```

A similar form is used when a set of windows is to be repeated N times:

```
[ repNWindows ( ) <count> <window> <<|><window>>>|> nullWindow [ ] ]
```

where `count` represents the number of times the windows are to be replicated.

```
>repNWindows:: (Physical a, Physical b) => Int -> PieceCont a b -> PieceCont a b
>repNWindows i (Windows wl) = let
>   repN::(Physical a, Physical b)=>
>   Int -> [FunctionWindow a b] -> [FunctionWindow a b]
>   repN 0 _ = []
>   repN x ls = ls ++ (repN (x-1) ls)
>   in Windows (repN i wl)
```

The function `pieceRep` is used as a generator for normalized, or flattened, windows within piecewise continuous signals and is used in preference to the constructor `PieceContRep` to allow for optimization of windows behavior

```
>pieceRep:: (Physical a, Physical b) => PieceCont a b -> SignalRep a b
>pieceRep (Windows wl) = PieceContRep (Windows (flattenWindows 0.0 wl))
```

or an optimized version

```
pieceRep (Windows wl) = PieceContRep (Windows (flattenWindows 0.0 wl))
```

A.6.2 Sum and Diff

Another mechanism of making signals from other signals is to sum or diff them together. This mechanism is identified by simply naming the signals to be summed, in the following form:

```
>sumSig, diffSig:: (Physical a, Physical b, Signal s, Signal s') =>
>   (s a b) -> (s' a b) -> SignalRep a b
>sumSig f f' =
>   pieceRep $ Windows $ splice sigState
>   (functionWindows (toSig f)) (functionWindows (toSig f')) 0.0
>   where
>     sigState s ZRep = s
>     sigState ZRep s = s
>     sigState s NullRep = s
>     sigState NullRep s = s
>     sigState s XRep = s
>     sigState XRep s = s
>     sigState s s' =
>       let s1 t = fromPhysical (mapSignal s t)
>           s2 t = fromPhysical (mapSignal s' t)
>       in FunctionRep (\t -> toPhysical ((s1 t) + (s2 t)))
>diffSig f f' =
>   pieceRep $ Windows $ splice sigState
>   (functionWindows (toSig f)) (functionWindows (toSig f')) 0.0
>   where
>     sigState s ZRep = s
>     sigState ZRep s = s
>     sigState s NullRep = s
>     sigState NullRep s = s
>     sigState s XRep = s
>     sigState XRep s = s
>     sigState s s' =
>       let s1 t = fromPhysical (mapSignal s t)
>           s2 t = fromPhysical (mapSignal s' t)
>       in FunctionRep (\t -> toPhysical ((s1 t) - (s2 t)))
```

An entire list of signals may be summed with a function of the following form:

```
>sumSigList, diffSigList:: (Physical a, Physical b, Signal s) =>
>   [ s a b ] -> SignalRep a b
>sumSigList [] = ZRep
>sumSigList ls = foldl1 sumSig (map toSig ls)
>diffSigList [] = ZRep
>diffSigList ls = foldl1 diffSig (map toSig ls)
```

A.6.3 Product

Two signals may be multiplied together via an operation of the following form:

```
>mulSig:: (Physical a, Physical b, Signal s, Signal s') =>
>   (s a b) -> (s' a b) -> SignalRep a b
>mulSig f f' =
>   pieceRep $ Windows $ splice sigState
>   (functionWindows (toSig f)) (functionWindows (toSig f')) 0.0
>   where
>     sigState s ZRep = s
>     sigState ZRep s = s
>     sigState s@NullRep _ = s
>     sigState _ s@NullRep = s
>     sigState s@XRep _ = s
>     sigState _ s@XRep = s
```

```
> sigState s s' =
>   let f1 t = fromPhysical (mapSignal s t)
>       f2 t = fromPhysical (mapSignal s' t)
>   in FunctionRep (\t -> toPhysical ((f1 t) * (f2 t)))
```

A.7 Pure function transformations

Transformations take a signal and transform it, e.g., converting it from the time domain to the frequency domain. These transformations are pure in the sense that they are defined here in English.

A.7.1 Fourier transform

The Fourier transform converts time domain signals to frequency domain signals using a discrete Fourier Transform conversion. It is, therefore, more restricted than other signal combination mechanisms. It takes a number of samples (which are rounded up to the nearest power of two), the amount of time over which the signal shall be sampled, a frequency zero-offset and a function that can be *magnitude*, *realPart* or *imagPart* plus the signal to be converted. It has the following form:

```
>fourTrans:: (Physical a, Physical a', Physical b)=>
>   Int -> a -> a' -> (ComplexF->Float) -> SignalRep a b -> SignalRep a' b
>fourTrans sam t o cfn f =
>  let
>    waveform:: (Physical a, Physical b) => a -> [b] -> SignalRep a b
>    waveform samp ampls =
>      let stepSlope y y' = (/) ((fromPhysical y') - (fromPhysical y))
>          (fromPhysical samp)
>          makeWin (v,v') = Window LocalZero (TimeEvent (fromPhysical samp))
>                          (linear (stepSlope v v') v)
>          points = ampls ++ (cycle [(toPhysical 0.0)])
>      in pieceRep (Windows (map makeWin (zip points (tail points))))
>      s = 2^ceiling(logBase 2 $ fromIntegral sam)
>      si = toPhysical (1.0 / (fromPhysical t))
>      trl = pointsCount (toPhysical 0.0) t s f
>      z t = cos(2*pi*t*fromPhysical o) :+ sin(-2*pi*t*fromPhysical o)
>      mc x = ((fromPhysical $ fst x) :+ 0.0) * (z (fromPhysical $ snd x))
>      til = map mc trl
>      fil = fft til
>      frl = map cfn fil
>  in waveform si (map toPhysical frl)
```

where the functions `fft` and `fftnv` are imported from the module `FFT`. The function `fft` provides the complex coefficients of a Fourier transform of a sample, and the function `fftnv` provides the complex coefficients of the original sample, as follows:

```
fft, fftnv:: [Complex Float] -> [Complex Float]
```

NOTE—The way by which SML defines the Fourier transform inherently utilizes a sampling technique. This technique is not rigorously identical to the Fourier transform, but tends towards the true transform as the number of samples is increased and when the time over which the samples are taken is the period of the signal.

A.8 Measure, limiting, and sampling signals

The signals produced may be checked and their attributes measured. Two levels of checking are provided: a check upon the signal parameters and a check upon the signal itself. In addition to these checks, a number of measurements can be applied to the signals. Signals may also be sampled to return a list of values upon which functions may be defined.

Signal measurements are made on samplings of the signal over values of the independent variable. In other words, a window shall be specified and either an interval or the number of samples shall be given, just as with the sampling functions. The user may use the sampling functions given above as inputs into the measurement functions.

Measurements are performed on samplings of a signal. These samplings return a list of tuples consisting of signal values (extracted using the Haskell function `fst`) and independent values (extracted using the Haskell function `snd`).

A.8.1 Confining parameters to a limit

A parameter of any physical type may be limited to a particular range. In other words, if the given value is lower than the low value of the range, the parameter is made equal to that low value. If the parameter is greater than the high value of the range, the value is made equal to that parameter. No error is signaled.

The form of the limiting function is as follows:

limit <low_value> <high_value> <parameter_value>

```
>limit:: Physical a => a -> a -> a -> a
>limit low high val = let
>   rlow = fromPhysical low
>   rhigh = fromPhysical high
>   rval = fromPhysical val
>   in if rval <= rlow then low
>      else if rhigh <= rval then high
>      else val
```

A.8.2 Sampling signals

Signals are sampled within a window, given as a low and a high value of the same type as the independent variable of the signal. Given this window and the signal, there are two ways to specify how the signals are to be sampled:

- a) The user can specify the number of points (i.e., signal value, independent variable) to be drawn from within the window:

```
>pointsCount:: (Physical a, Physical b, Signal s) =>
>   a -> a -> Int -> s a b -> [ (b, a) ]
>pointsCount low high count sig = let
>   rlow = fromPhysical low
>   rhigh = fromPhysical high
>   toff = (rhigh - rlow) / ((fromIntegral count) - 1)
>   roff = toff - (toff / (2 * (fromIntegral count)))
>   creList low high off = if low <= high then
>       (low : creList (low + off) high off)
>       else []
>   appSig t = (mapSignal sig (toPhysical t), toPhysical t)
>   in map appSig (creList rlow rhigh roff)
```

- b) The user can also specify the number of samples (i.e., signal value) to be drawn from within the window:

```
>sampleCount:: (Physical a, Physical b, Signal s) =>
>   a -> a -> Int -> s a b -> [ b ]
```

```
>sampleCount low high count sig = map fst (pointsCount low high count sig)
```

NOTE—Substitute function `snd` for function `fst` to return the independent value.

A.9 Digital signals

A digital signal is a signal where information is represented in one of two values, which are sometimes called by names such as true and false, low and high, 1 and 0, etc. They also support the additional states where the values represent unknown (X) and tri-state (Z)

A digital signal is an abstract representation of the values that are encountered in engineering design; these enumeration values are defined more precisely in A.9.1 through A.9.4.

Digital signals are distinct from analog signals in that their values do not take on physical values; rather, they take on enumeration values that represent physical values. Definitions shall be provided of what it means to operate on digital signals and what it means to convert a digital signal into an analog signal.

A.9.1 Defining Digital

```
>module Digital where  
  
>import Physical  
>import Data.List  
>import Pure  
>import Data.Char
```

The definitions of the digital values that shall be used are as follows:

- X represents the fact that the signal is in transition and, therefore, cannot be said to be at either value.
- Z represents the fact that the digital signal is providing very little current and shall sink very little current. It represents a signal at high impedance.
- L corresponds to false or 0; it translates to 0 in a control signal (but see below).
- H corresponds to true or 1; it translates to 1 in a control signal (but see below).

```
>data Digital = X | Z | L | H deriving (Eq, Show)
```

Several digital operations are defined for digital values, using an asterisk or the letter d to distinguish them from standard boolean functions:

- `notd` is the digital function not.
- `&*` is the digital function and.
- `|*` is the digital function or.
- `!*` is the digital function xor (equivalent to not equals).
- `=*` is the digital equality (i.e., equals).
- `+` is the equivalent to states provided by `Sum`.

— ****** is the equivalent to states provided by Prod.

```
>class DigitalOps a where
>  (&*), (|*), (!*), (=*), (+*), (**): a -> a -> a
>  notd:: a -> a

>instance DigitalOps Digital where
>  notd x = case x of { L->H; H->L; X->X; Z->Z }
>  x &* Z = x
>  _ &* L = L
>  x &* y = case x of { Z->y; X->y; H->x; L->x }
>  x |* y = notd ((notd x) &* (notd y))
>  Z !* y = y
>  x !* Z = x
>  x !* y = (x|*y) &* notd(x&*y)
>  x =* y = notd(( notd x) !* (notd y))
>  Z +* y = y
>  H +* _ = H
>  x +* y = case y of { Z->x; X->y; H->y; L->x;}
>  x ** y = notd ((notd x) +* (notd y))
```

A.9.2 Defining DigitalSignal

A digital signal is specified as a time (which represents the transition period of the digital signal) and a list of digital values (i.e., Digital). A digital signal also supports the digital operation.

The definition for a digital signal is as follows:

```
>data DigitalSignal = Dig Float [ Digital ] deriving (Eq, Show)
>instance DigitalOps DigitalSignal where
>  d1@(Dig t1 l1) &* d2@(Dig t2 l2) =
>    let doAnd (x,y) = x &* y
>    t = if t1 == t2 then t1 else
>    error "Attempting to AND two signals with different times"
>    in Dig t (map doAnd (zip l1 l2))
>  d1@(Dig t1 l1) |* d2@(Dig t2 l2) =
>    let doOr (x,y) = x |* y
>    t = if t1 == t2 then t1 else
>    error "Attempting to OR two signals with different times"
>    in Dig t (map doOr (zip l1 l2))
>  notd (Dig t dl) = Dig t (map notd dl)
>  d1@(Dig t1 l1) !* d2@(Dig t2 l2) =
>    let doXor (x,y) = x !* y
>    t = if t1 == t2 then t1 else
>    error "Attempting to XOR two signals with different times"
>    in Dig t (map doXor (zip l1 l2))
>  d1@(Dig t1 l1) =* d2@(Dig t2 l2) =
>    let doCompare (x,y) = x =* y
>    t = if t1 == t2 then t1 else
>    error "Attempting to COMPARE two signals with different times"
>    in Dig t (map doCompare (zip l1 l2))
>  d1@(Dig t1 l1) +* d2@(Dig t2 l2) =
>    let doSum (x,y) = x +* y
>    t = if t1 == t2 then t1 else
>    error "Attempting to COMPARE two signals with different times"
>    in Dig t (map doSum (zip l1 l2))
>  d1@(Dig t1 l1) ** d2@(Dig t2 l2) =
>    let doProd (x,y) = x Digital.** y
>    t = if t1 == t2 then t1 else
>    error "Attempting to COMPARE two signals with different times"
>    in Dig t (map doProd (zip l1 l2))
```

The conversion functions `digitalString` and `digitalList` are provided to help turn digital strings into digital arrays suitable for Serial and Parallel data.

Examples:

```
digitalString "HL; LLX, ; H" ==> [H,L,L,L,X,H]
digitalList  "HL; LLX, ; H" ==> [[H,L,Z,H],[L,L,Z,Z],[Z,X,Z,Z]].
```

```
>digitalString:: String -> [Digital]
>digitalString xs = map char2dig (filter (ignoreChars) xs) where
>   ignoreChars c = filter (==c) "LHXZ10" /= []
>   char2dig 'L' = L
>   char2dig 'H' = H
>   char2dig 'X' = X
>   char2dig 'Z' = Z
>   char2dig '0' = L
>   char2dig '1' = H
>
>digitalList :: String -> [[Digital]]
>digitalList xs = transposeZ (map digitalString $ split xs) where
>   split "" = []
>   split s = a : split (drop 1 b) where (a, b) = break (\c->c==';' || c=='\n') s
>   transposeZ xxs = let
>       len = maximum (map length xxs)
>       in take len (transpose (map (\xs->xs++cycle[Z]) xxs) )
>
```

Digital signals can be generated using the function `str2dig` that allows digital strings containing the characters H, L, Z, and X and whitespace to be converted into digital signals.

```
>str2dig:: Float -> String -> DigitalSignal
>str2dig t s = Dig t (digitalString s)
```

A.9.3 Conversion routines

Conversion routines convert from analog control signals to digital signals and vice versa. Digital signals can be combined with other digital signals, but need to be converted in order for them to be used with other SML signals. An analog control signal is an analog signal that uses the threshold low and high values and where the no signal value is used to detect tri-state Z values.

Two conversion routines are defined:

- analog to digital (a2d)
- digital to analog (d2a)

The conversion routines use physical threshold values to convert to and from low and high states.

The format of the conversion from analog signals to digital signals is as follows:

```
a2d <low_threshold> <high_threshold> <sample_rate> <analog_signal> <
```

Digital signals have distinct states, whereas their analog values are arbitrary, depending on the logic family thresholds:

- The Z digital state represents a high impedance signal with little or no current and is converted from the no signal, NullRep, SignalRep.

- The H digital state represents a logic high and is converted from values equal to or greater than the high threshold value.
- The L digital state represents a logic low and is converted from values equal to or less than the low threshold value.
- The X digital state represents all other values, i.e., values within the low-high thresholds.

NOTE—The description assumes that the high threshold is greater than the low threshold.

The signal has two thresholds; between the two thresholds, the X value is used. The representation ZRep signifies the presence of a Z; it is controlling in the sense that if there is no signal, then no current flows and it does not matter what the voltage level is. The sample rate simply gives the rate at which analog samples are taken and digital outputs produced.

```
>a2d::(Physical a, Physical b, Signal s) =>
>  b -> b -> Float -> (s a b) -> DigitalSignal
>a2d lowth highth sampRate s =
>  let lt = fromPhysical lowth
>      ht = fromPhysical highth
>      mt = (ht+lt)/2
>      dt = (ht-lt)/2
>      h = if dt<0 then L else H
>      sr = sampRate
>      val:: Float -> Digital
>      val x = let dv = fromPhysical (mapSignal s (toPhysical x)) - mt
>                in if isInactive s (toPhysical x) then Z
>                    else if abs dv < abs dt then X
>                    else if dv > 0 then h else (notd h)
>      sl x = x : sl (x + sr)
>  in Dig sampRate (map val (sl 0.0))
```

The format of the conversion from digital to analog signal is as follows:

```
d2a <low_threshold> <high_threshold> <digital_signal>
```

The digital-to-analog conversion provides an analog signal as follows:

- A value of Z maintains the previous digital value.
- A value of X produces a tri-state gated Off digital signal.
- A value of L produces a low threshold analog value.
- A value of H produces a high threshold analog value.

In addition, when the signal goes to X, the level of the voltage signal tends to “float” as follows:

```
>d2a::(Physical a, Physical b) =>
>  b -> b -> DigitalSignal -> (SignalRep a b)
>d2a lowth highth s@(Dig clock ds) =
>  let
>    win s = Window LocalZero (TimeEvent clock) s
>    makewin Z = win ZRep
>    makewin X = win XRep
>    makewin H = win (constant highth)
>    makewin L = win (constant lowth)
>    makewin' (d:Z:ds) = (makewin d) : (makewin' (d:ds))
>    makewin' ds@(d:[]) = (makewin d) : (makewin' ds)
>    makewin' [] = makewin' (Z:[])
>    makewin' (d:ds) = (makewin d) : (makewin' ds)
>  in pieceRep (Windows (makewin' ds))
```