



**IEEE**

**IEC 62529**

Edition 3.0 2024-06

# INTERNATIONAL STANDARD

**IEEE Std 1641™**



**Standard for Signal and Test Definition**

IECNORM.COM : Click to view the full PDF of IEC 62529:2024





**THIS PUBLICATION IS COPYRIGHT PROTECTED**  
**Copyright © 2022 IEEE**

All rights reserved. IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by the Institute of Electrical and Electronics Engineers, Inc. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the IEC Central Office. Any questions about IEEE copyright should be addressed to the IEEE. Enquiries about obtaining additional rights to this publication and other information requests should be addressed to the IEC or your local IEC member National Committee.

IEC Secretariat  
3, rue de Varembe  
CH-1211 Geneva 20  
Switzerland  
Tel.: +41 22 919 02 11  
[info@iec.ch](mailto:info@iec.ch)  
[www.iec.ch](http://www.iec.ch)

Institute of Electrical and Electronics Engineers, Inc.  
3 Park Avenue  
New York, NY 10016-5997  
United States of America  
[stds.info@ieee.org](mailto:stds.info@ieee.org)  
[www.ieee.org](http://www.ieee.org)

**About the IEC**

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

**About IEC publications**

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

**IEC publications search - [webstore.iec.ch/advsearchform](http://webstore.iec.ch/advsearchform)**

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 It also gives information on projects, replaced and withdrawn publications.

**IEC Just Published - [webstore.iec.ch/justpublished](http://webstore.iec.ch/justpublished)**

Stay up to date on all new IEC publications. Just Published details all new publications released. Available online and once a month by email.

**IEC Customer Service Centre - [webstore.iec.ch/csc](http://webstore.iec.ch/csc)**

If you wish to give us your feedback on this publication or need further assistance, please contact the Customer Service Centre: [sales@iec.ch](mailto:sales@iec.ch).

**IEC Products & Services Portal - [products.iec.ch](http://products.iec.ch)**

Discover our powerful search engine and read freely all the publications previews, graphical symbols and the glossary. With a subscription you will always have access to up to date content tailored to your needs.

**Electropedia - [www.electropedia.org](http://www.electropedia.org)**

The world's leading online dictionary on electrotechnology, containing more than 22 500 terminological entries in English and French, with equivalent terms in 25 additional languages. Also known as the International Electrotechnical Vocabulary (IEV) online.

IECNORM.COM : Click to view the full text of IEC 62529:2024



**IEEE**

**IEC 62529**

Edition 3.0 2024-06

# INTERNATIONAL STANDARD

**IEEE Std 1641™**



**Standard for Signal and Test Definition**

IECNORM.COM : Click to view the full PDF of IEC 62529:2024

INTERNATIONAL  
ELECTROTECHNICAL  
COMMISSION

ICS 25.040.01; 35.060

ISBN 978-2-8322-8703-3

**Warning! Make sure that you obtained this publication from an authorized distributor.**

## 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
Annex N (informative) Participants .....	352

## Standard for Signal and Test Definition

### FOREWORD

- 1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC document(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation.

IEEE Standards documents are developed within IEEE Societies and Standards Coordinating Committees of the IEEE Standards Association (IEEE SA) Standards Board. IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of IEEE and serve without compensation. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards. Use of IEEE Standards documents is wholly voluntary. *IEEE documents are made available for use subject to important notices and legal disclaimers (see <https://standards.ieee.org/ipr/disclaimers.html> for more information).*

IEC collaborates closely with IEEE in accordance with conditions determined by agreement between the two organizations. This Dual Logo International Standard was jointly developed by the IEC and IEEE under the terms of that agreement.

- 2) The formal decisions of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees. The formal decisions of IEEE on technical matters, once consensus within IEEE Societies and Standards Coordinating Committees has been reached, is determined by a balanced ballot of materially interested parties who indicate interest in reviewing the proposed standard. Final approval of the IEEE standards document is given by the IEEE Standards Association (IEEE SA) Standards Board.
- 3) IEC/IEEE Publications have the form of recommendations for international use and are accepted by IEC National Committees/IEEE Societies in that sense. While all reasonable efforts are made to ensure that the technical content of IEC/IEEE Publications is accurate, IEC or IEEE cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
- 4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications (including IEC/IEEE Publications) transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC/IEEE Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
- 5) IEC and IEEE do not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC and IEEE are not responsible for any services carried out by independent certification bodies.
- 6) All users should ensure that they have the latest edition of this publication.
- 7) No liability shall attach to IEC or IEEE or their directors, employees, servants or agents including individual experts and members of technical committees and IEC National Committees, or volunteers of IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE SA) Standards Board, for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC/IEEE Publication or any other IEC or IEEE Publications.
- 8) Attention is drawn to the normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
- 9) Attention is drawn to the possibility that implementation of this IEC/IEEE Publication may require use of material covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. IEC or IEEE shall not be held responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patent 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.

IEC 62529/IEEE Std 1641™-2022 was processed through IEC technical committee 91: Electronics assembly technology, under the IEC/IEEE Dual Logo Agreement. It is an International Standard.

The text of this International Standard is based on the following documents:

IEEE Std	FDIS	Report on voting
1641 (2022)	91/1933/FDIS	91/1945/RVD

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

The language used for the development of this is English.

The IEC Technical Committee and IEEE Technical Committee have decided that the contents of this document will remain unchanged until the stability date indicated on the IEC website under [webstore.iec.ch](http://webstore.iec.ch) in the data related to the specific document. At this date, the document will be

- reconfirmed,
- withdrawn, or
- revised.

**IMPORTANT – The "colour inside" logo on the cover page of this document indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this document using a colour printer.**

IECNORM.COM : Click to view the full PDF of IEC 62529:2024

[IECNORM.COM](https://www.iecnorm.com) : Click to view the full PDF of IEC 62529:2024

# 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**

IECNORM.COM : Click to view the full PDF of IEC 62529:2024

**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

IECNORM.COM : Click to view the full PDF of IEC 62529:2024

## 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](#).<sup>1</sup> An IEEE Account is needed to access the application.

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

## 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.

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

<sup>2</sup> 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](#).<sup>3</sup> 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](#).<sup>4</sup> 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.

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

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

## Patents

IEEE Standards are developed in compliance with the [IEEE SA Patent Policy](#).<sup>5</sup>

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.

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

## IEEE 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.

IECNORM.COM : Click to view the full PDF of IEC 62529:2024

[IECNORM.COM](https://www.iecnorm.com) : Click to view the full PDF of IEC 62529:2024

# 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*).<sup>6,7</sup>

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*).

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

<sup>7</sup> 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.<sup>8</sup>

**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.<sup>9</sup>

**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.

<sup>8</sup>*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.

<sup>9</sup> 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-compileable 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] <sup>10</sup> )
DME	distance measuring equipment
DTIF	Digital Test Interchange Format

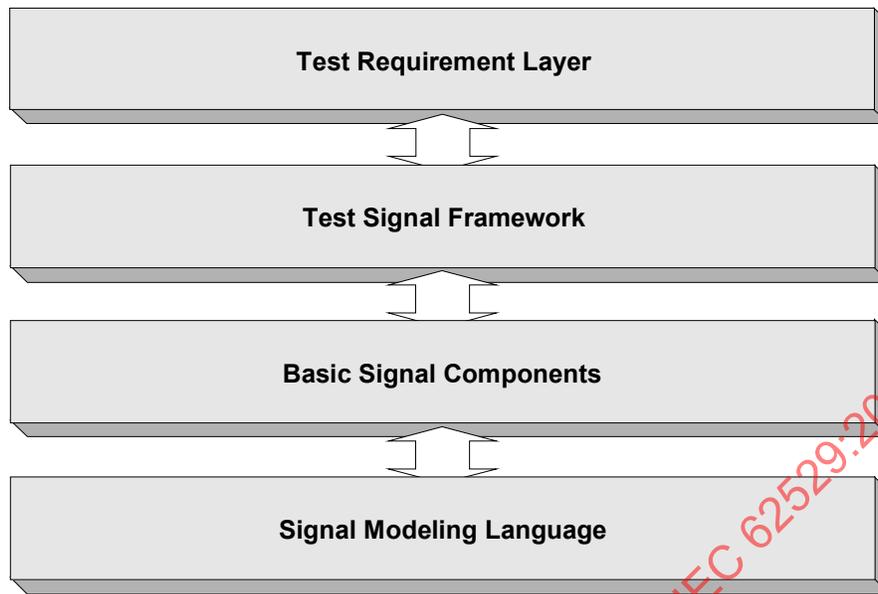
<sup>10</sup> The numbers in brackets correspond to the numbers of the bibliography in Annex M.

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.<sup>11</sup>

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.

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

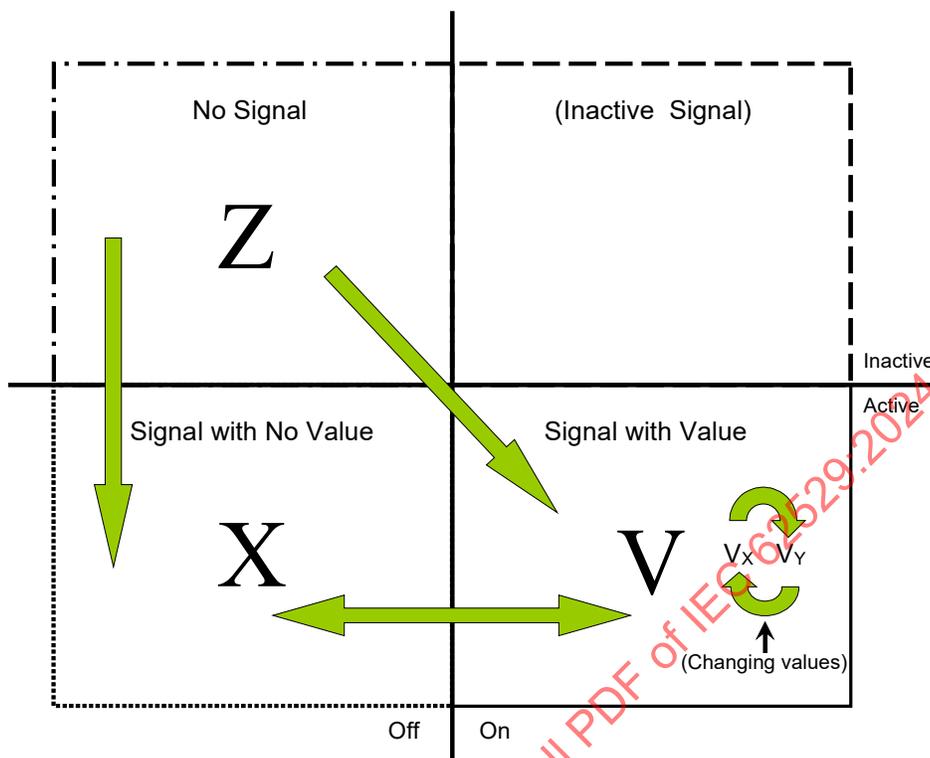


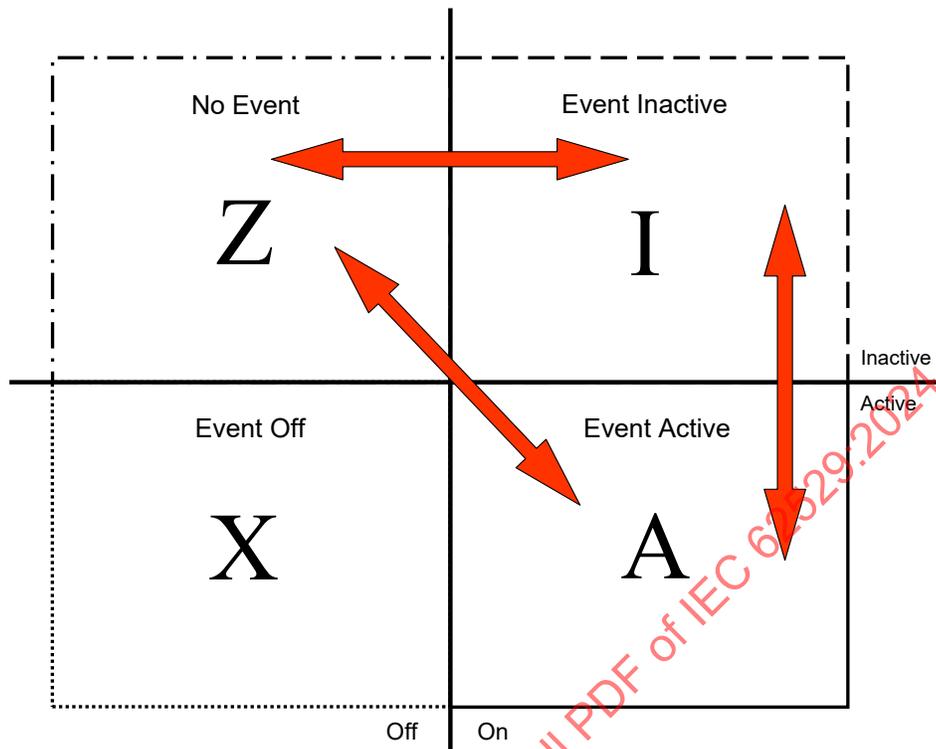
Figure 2 —Signal states

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.



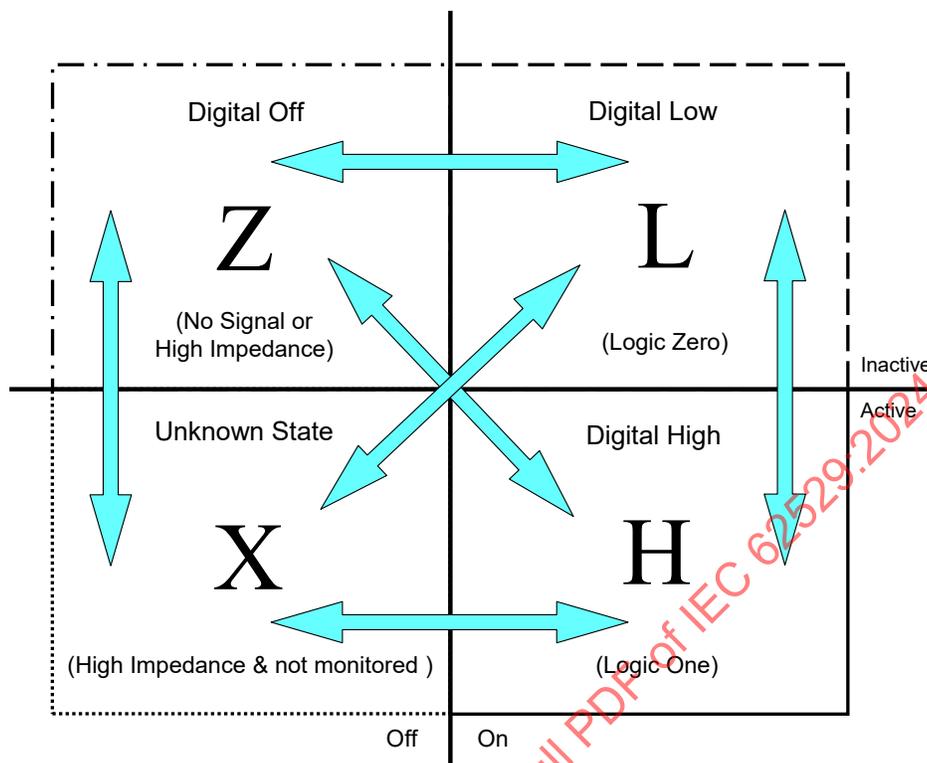
**Figure 3 —Event states**

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) A class derived from SignalFunction base class (or subclass)
- b) Class type and reference type description
- c) Attributes and default values
- d) A control interface (defined using an IDL description)
- e) 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 extensibility 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 $\pi$ 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., “group: 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 kΩ 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 <sup>0</sup> SML	10 <sup>3</sup> SML	10 <sup>6</sup> SML	10 <sup>9</sup> SML	10 <sup>-3</sup> SML	10 <sup>-6</sup> SML	10 <sup>-9</sup> SML	10 <sup>-12</sup> 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 <sup>2</sup>	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 <sup>0</sup> SML	10 <sup>3</sup> SML	10 <sup>6</sup> SML	10 <sup>9</sup> SML	10 <sup>-3</sup> SML	10 <sup>-6</sup> SML	10 <sup>-9</sup> SML	10 <sup>-12</sup> 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
- $\tau$  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
- $\omega$  is the frequency (multiplied by  $2\pi$ )
- $\theta$  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 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 = foldll sumSig (map toSig ls)
>diffSigList [] = ZRep
>diffSigList ls = foldll 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 = 2ceiling(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 DigitalOpers a where
> (&*), (|*), (!*), (=*), (+*), (**):: a -> a -> a
> notd:: a -> a

>instance DigitalOpers 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 DigitalOpers DigitalSignal where
> dl@(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))
> dl@(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)
> dl@(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))
> dl@(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))
> dl@(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))
> dl@(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=='') 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))

```

### A.9.4 Patterns

Digital signals are often used in sets that represent related signals; for example, a set of 32 digital signals can represent a single, changing integer. Specifying sets of such digital signals is inconvenient using the form given above. In addition, such sets of digital signals usually share a common clock so that repeated specification of the time parameter is redundant.

A pattern is a convenient mechanism for specifying a number of parallel digital signals. It specifies the clock time once and then gives the digital values as a number of strings. An example of a parallel digital string utilizing whitespace is as follows:

```
Pattern (fromPhysical (USEC 1)) "HLHL LLLL, LHLH HHHH, HHHH LLLL"
```

The definition for a pattern is as follows:

```
>data Pattern = Pattern Float String deriving (Eq, Show)
```

Given a pattern, it may be converted into a list of digital signals:

```
>pat2diglist:: Pattern -> [ DigitalSignal ]
>pat2diglist (Pattern t xs) = map (Dig t) (digitalList xs)
```

### A.10 Basic component SML

The BSC SML is defined in the module BSC

```
>module BSC where
>import Data.Complex
>import FFT
>import Data.Bits
>import Data.List
```

making use of the previously defined types Physical, Pure, and Digital (see A.3, A.5, A.6, A.7, and A.9, respectively)

```
>import Pure
>import Digital
>import Physical
```

by the following SML BSC definitions: Source, Conditioner, EventFunction, Sensor, Control, Digital, and Connection, and using the following type aliases.

```
>type PulseDefn = (Float, Float, Float)
>type PulseDefns = [PulseDefn]
>instance Show (a->b) where
>  showsPrec _ _ = showString "<function>"
>
>data BSC a b =
>  Null |
>  ---Source
>  ---Non Periodic
```

```
> Constant {amplitude::b} |
> Step {amplitude::b, startTime::a} |
> SingleTrapezoid {amplitude::b,
>                 startTime::a, riseTime::a, pulseWidth::a, fallTime::a} |
> Noise {amplitude::b, seed::Integer, frequency::Frequency} |
> SingleRamp {amplitude::b, riseTime::a, startTime::a} |
>---Periodic
> Sinusoid {amplitude::b, frequency::Frequency, phase::PlaneAngle} |
> Trapezoid {amplitude::b,
>            period::a, riseTime::a, pulseWidth::a, fallTime::a} |
> Ramp {amplitude::b, period::a, riseTime::a} |
> Triangle {amplitude::b, period::a, dutyCycle::Ratio} |
> SquareWave {amplitude::b, period::a, dutyCycle::Ratio} |
> WaveformRamp {amplitude::b, period::a,
>               samplingInterval::a, points::[Float]} |
> WaveformStep {amplitude::b, period::a,
>               samplingInterval::a, points::[Float]} |
>---Conditioner
>---Filter
> BandPass {centerFrequency::Frequency, frequencyBand::Frequency,
>            gain::Ratio, rollOff::Ratio, passBandRipple::Ratio, stopBandRipple::Ratio,
>            fInterval::Float, samples::Int, signal::(BSC a b)} |
> LowPass {cutoff::Frequency, gain::Ratio, rollOff::Ratio,
>           passBandRipple::Ratio, stopBandRipple::Ratio,
>           fInterval::Float, samples::Int, signal::(BSC a b)} |
> HighPass {cutoff::Frequency, gain::Ratio, rollOff::Ratio,
>            passBandRipple::Ratio, stopBandRipple::Ratio,
>            fInterval::Float, samples::Int, signal::(BSC a b)} |
> Notch {centerFrequency::Frequency, frequencyBand::Frequency,
>         gain::Ratio, rollOff::Ratio, passBandRipple::Ratio,
>         stopBandRipple::Ratio,
>         fInterval::Float, samples::Int, signal::(BSC a b)} |
>---Combiner
> Sum {signals::[(BSC a b)]} |
> Product {signals::[(BSC a b)]} |
> Diff {signals::[(BSC a b)]} |
>---Modulator
> FM {amplitude::b, carrierFrequency::Frequency,
>      frequencyDeviation::Frequency, signal::(BSC a b)} |
> AM {modIndex::Ratio, carrier::(BSC a b), signal::(BSC a b)} |
> PM {amplitude::b, carrierFrequency::Frequency,
>      phaseDeviation::PlaneAngle, signal::(BSC a b)} |
>---Transformation
> SignalDelay {acceleration::Frequency,
>              delay::a, rate::Ratio, signal::(BSC a b)} |
> Exponential {dampingFactor::Float, signal::(BSC a b)} |
> E {signal::(BSC a b)} |
> Ln {signal::(BSC a b)} |
> Negate {signal::(BSC a b)} |
> Inverse {signal::(BSC a b)} |
> PulseTrain {pulses::PulseDefns, repetition::Int, signal::(BSC a b)} |
> Attenuator {gain::Ratio, signal::(BSC a b)} |
> Load {resistance::Resistance, reactance::Resistance, signal::(BSC a b)} |
> Limit {limit::b, signal::(BSC a b)} |
> FFT {samples::Int, interval::a, signal::(BSC a b)} |
>---Event Function
>---Event Source
> Clock {clockRate::Frequency} |
> TimedEvent {delay::a, duration::a, period::a, repetition::Int} |
> PulsedEvent {pulses::PulseDefns, repetition::Int} |
>---EventConditioner
> EventedEvent {events::[(BSC a b)]} |
> EventCount {count::Int, event::(BSC a b)} |
> ProbabilityEvent {seed::Integer, propability::Ratio, event::(BSC a b)} |
> NotEvent {event::(BSC a b)} |
>---Logical
> OrEvent {events::[(BSC a b)]} |
> XOrEvent {events::[(BSC a b)]} |
> AndEvent {events::[(BSC a b)]} |
>---Sensor
> Counter {readings::[(a,b)]} |
```

```
> Interval {readings::[(a,b)]} |
> Instantaneous {readings::[(a,b)]} |
> RMS {readings::[(a,b)]} |
> Average {readings::[(a,b)]} |
> PeakToPeak {readings::[(a,b)]} |
> Peak {readings::[(a,b)]} |
> PeakPos {readings::[(a,b)]} |
> PeakNeg {readings::[(a,b)]} |
> MaxInstantaneous {readings::[(a,b)]} |
> MinInstantaneous {readings::[(a,b)]} |
>---Measure
> Decode {datatype::String, encoding::String} |
>---Control
> SelectIf {selector::(BSC a b), signals::[(BSC a b)]} |
> SelectCase {mask::Int, selectors::[(BSC a b)], signals::[(BSC a b)]} |
> Encode {data_::String,
>         width::Int, repetition::Int, datatype::String, encoding::String,
>         signal::(BSC a b), channel::Int} |
> Transform {mapping::String, as::(Maybe (BSC a b->BSC a b)),
>           signal::(BSC a b)} |
> InverseTransform {mapping::String, as'::(BSC a b), signal::(BSC a b)} |
> Channels {channelNames::String, ass::(Maybe ([BSC a b]->BSC a b)),
>          signals::[(BSC a b)], channel::Int} |
>---Digital
> SerialDigital {data_::String, period::a,
>               logic_H_value::b, logic_L_value::b, pulseClass::String} |
> ParallelDigital {data_::String, period::a,
>                 logic_H_value::b, logic_L_value::b, pulseClass::String,
>                 channel::Int} |
>---Connection {channelWidth::Int, signals::[(BSC a b)], channel::Int} |
> TwoWire {hi::String, lo::String, channelWidth::Int,
>          signals::[(BSC a b)], channel::Int} |
> TwoWireComp {true::String, comp::String, channelWidth::Int,
>              signals::[(BSC a b)], channel::Int} |
> ThreeWireComp {true::String, comp::String, lo::String, channelWidth::Int,
>                signals::[(BSC a b)], channel::Int} |
> SinglePhase {a::String, n::String, channelWidth::Int,
>              signals::[(BSC a b)], channel::Int} |
> TwoPhase {a::String, b::String, n::String, channelWidth::Int,
>            signals::[(BSC a b)], channel::Int} |
> ThreePhaseDelta {a::String, b::String, c::String, channelWidth::Int,
>                  signals::[(BSC a b)], channel::Int} |
> ThreePhaseWye {a::String, b::String, c::String, n::String,
>                channelWidth::Int, signals::[(BSC a b)], channel::Int} |
> ThreePhaseSynchro {x::String, y::String, z::String, channelWidth::Int,
>                    signals::[(BSC a b)], channel::Int} |
> FourWireResolver {s1::String, s2::String, s3::String, s4::String,
>                   channelWidth::Int, signals::[(BSC a b)], channel::Int} |
> SynchroResolver {r1::String, r2::String, r3::String, r4::String,
>                  channelWidth::Int, signals::[(BSC a b)], channel::Int} |
> Series {via::String, channelWidth::Int,
>         signals::[(BSC a b)], channel::Int} |
> FourWire {hi::String, lo::String, hiRef::String, loRef::String,
>           channelWidth::Int, signals::[(BSC a b)], channel::Int} |
> NonElectrical {to::String, from::String, channelWidth::Int,
>                signals::[(BSC a b)], channel::Int} |
> DigitalBus {pins::String, channelWidth::Int,
>             signals::[(BSC a b)], channel::Int} |
> deriving Show

>instance Signal BSC where
> toSig Null = ZRep
>---Source
>---Non Periodic
> toSig (Constant amplitude) = bscConstant amplitude
> toSig (Step amplitude startTime) = bscStep amplitude startTime
> toSig (SingleTrapezoid amplitude startTime riseTime pulseWidth fallTime) =
>     bscSingleTrapezoid amplitude startTime riseTime pulseWidth fallTime
> toSig (Noise amplitude seed frequency) =
>     bscNoise amplitude seed frequency
> toSig (SingleRamp amplitude riseTime startTime) =
```

```
>                                     bscSingleRamp amplitude riseTime startTime
>---Periodic
> toSig (Sinusoid amplitude frequency phase) =
>                                     bscSinusoid amplitude frequency phase
> toSig (Trapezoid amplitude period riseTime pulseWidth fallTime) =
>                                     bscTrapezoid amplitude period riseTime pulseWidth fallTime
> toSig (Ramp amplitude period riseTime) =
>                                     bscRamp amplitude period riseTime
> toSig (Triangle amplitude period dutyCycle) =
>                                     bscTriangle amplitude period dutyCycle
> toSig (SquareWave amplitude period dutyCycle) =
>                                     bscSquareWave amplitude period dutyCycle
> toSig (WaveformRamp amplitude _ samplingInterval points) =
>                                     bscWaveformRamp amplitude samplingInterval points
> toSig (WaveformStep amplitude _ samplingInterval points) =
>                                     bscWaveformStep amplitude samplingInterval points
>---Conditioner
>---Filter
> toSig (BandPass centerFrequency frequencyBand gain rolloff passBandRipple
>                                     stopBandRipple fInterval samples signal) =
>     bscBandPass centerFrequency frequencyBand gain rolloff passBandRipple
>                                     stopBandRipple fInterval samples signal
> toSig (LowPass passband gain rolloff passBandRipple stopBandRipple
>                                     fInterval samples signal) =
>     bscLowPass passband gain rolloff passBandRipple stopBandRipple
>                                     fInterval samples signal
> toSig (HighPass passband gain rolloff passBandRipple stopBandRipple
>                                     fInterval samples signal) =
>     bscHighPass passband gain rolloff passBandRipple stopBandRipple
>                                     fInterval samples signal
> toSig (Notch centerFrequency frequencyBand gain rolloff
>                                     passBandRipple stopBandRipple fInterval samples signal) =
>     bscNotch centerFrequency frequencyBand gain rolloff
>                                     passBandRipple stopBandRipple fInterval samples signal
>---Combiner
> toSig (Sum signals) = bscSum signals
> toSig (Product signals) = bscProduct signals
> toSig (Diff signals) = bscDiff signals
>---Modulator
> toSig (FM amplitude carrierFrequency frequencyDeviation signal) =
>     bscFM amplitude carrierFrequency frequencyDeviation signal
> toSig (AM modIndex carrier signal) = bscAM modIndex carrier signal
> toSig (PM amplitude carrierFrequency phaseDeviation signal) =
>     bscPM amplitude carrierFrequency phaseDeviation signal
>---Transformation
> toSig (SignalDelay acceleration delay rate signal) =
>     bscSignalDelay acceleration delay rate signal
> toSig (Exponential dampingFactor signal) =
>     bscExponential dampingFactor signal
> toSig (E signal) = bscE signal
> toSig (Ln signal) = bscLn signal
> toSig (Negate signal) = bscNegate signal
> toSig (Inverse signal) = bscInverse signal
> toSig (PulseTrain pulses repetition signal) =
>     bscPulseTrain pulses repetition signal
> toSig (Attenuator gain signal) = bscAttenuator gain signal
> toSig (Load resistance reactance signal) =
>     bscLoad resistance reactance signal
> toSig (Limit limit signal) = bscLimit limit signal
> toSig (FFT samples sampleInterval signal) =
>     bscFFT samples sampleInterval (toPhysical 0) signal
>---Event Function
>---EventSource
> toSig (Clock clockRate) = bscClock clockRate
> toSig (TimedEvent delay duration period repetition) =
>     bscTimedEvent delay duration period repetition
> toSig (PulsedEvent pulses repetition) = bscPulsedEvent pulses repetition
>---EventConditioner
> toSig (EventedEvent events) = bscEventedEvent events
> toSig (EventCount count event) = bscEventCount count event
> toSig (ProbabilityEvent seed propability event) =
```

```
>
> toSig (NotEvent event) = bscNotEvent event
>----Logical
> toSig (OrEvent events) = bscOrEvent events
> toSig (XOrEvent events) = bscXOrEvent events
> toSig (AndEvent events) = bscAndEvent events
>--Sensors
>--Control
> toSig (SelectIf selector signals) = bscSelectIf selector signals
> toSig (SelectCase mask selectors signals) =
>     bscSelectCase mask selectors signals
> toSig (Transform mapping as signal) = bscTransform mapping as signal
> toSig (InverseTransform mapping as' signal) =
>     bscInverseTransform mapping as' signal
> toSig (Encode data_width repetition datatype encoding signal channel) =
>     (bscEncode data_width repetition datatype encoding signal) !! channel
> toSig (Channels channelNames ass signals channel) =
>     (bscChannels channelNames ass signals) !! channel
>--Digital
> toSig (SerialDigital data_period logic_H_value logic_L_value pulseClass) =
>     bscSerialDigital data_period logic_H_value logic_L_value pulseClass
> toSig (ParallelDigital
>     data_period logic_H_value logic_L_value pulseClass channel) =
>     (bscParallelDigital data_period logic_H_value logic_L_value
>     pulseClass) !! channel
>--Connections
> toSig (TwoWire hi lo channelWidth signals channel) =
>     (bscTwoWire hi lo channelWidth signals) !! channel
> toSig (TwoWireComp true comp channelWidth signals channel) =
>     (bscTwoWireComp true comp channelWidth signals) !! channel
> toSig (ThreeWireComp true comp lo channelWidth signals channel) =
>     (bscThreeWireComp true comp lo channelWidth signals) !! channel
> toSig (SinglePhase a n channelWidth signals channel) =
>     (bscSinglePhase a n channelWidth signals) !! channel
> toSig (TwoPhase a b n channelWidth signals channel) =
>     (bscTwoPhase a b n channelWidth signals) !! channel
> toSig (ThreePhaseDelta a b c channelWidth signals channel) =
>     (bscThreePhaseDelta a b c channelWidth signals) !! channel
> toSig (ThreePhaseWye a b c n channelWidth signals channel) =
>     (bscThreePhaseWye a b c n channelWidth signals) !! channel
> toSig (ThreePhaseSynchro x y z channelWidth signals channel) =
>     (bscThreePhaseSynchro x y z channelWidth signals) !! channel
> toSig (FourWireResolver s1 s2 s3 s4 channelWidth signals channel) =
>     (bscFourWireResolver s1 s2 s3 s4 channelWidth signals) !! channel
> toSig (SynchroResolver s1 s2 s3 s4 channelWidth signals channel) =
>     (bscSynchroResolver s1 s2 s3 s4 channelWidth signals) !! channel
> toSig (Series via channelWidth signals channel) =
>     (bscSeries via channelWidth signals) !! channel
> toSig (FourWire hi lo hiRef loRef channelWidth signals channel) =
>     (bscFourWire hi lo hiRef loRef channelWidth signals) !! channel
> toSig (NonElectrical to from channelWidth signals channel) =
>     (bscNonElectrical to from channelWidth signals) !! channel
> toSig (DigitalBus pins channelWidth signals channel) =
>     (bscDigitalBus pins channelWidth signals) !! channel
>--Section Ends
```

## A.10.1 Source ::SignalFunction

### A.10.1.1 NonPeriodic ::Source

#### A.10.1.1.1 Constant ::NonPeriodic

```
>bscConstant::(Physical a, Physical b) => b -> SignalRep a b
```

```
>bscConstant =
> (\amplitude -> constant amplitude)
```

#### A.10.1.1.2 Step ::NonPeriodic

```
>bscStep::(Physical a, Physical b) => b -> a -> SignalRep a b
>bscStep =
> (\amplitude startTime -> let
>   st = fromPhysical startTime
>   zero = constant (toPhysical 0.0)
>   lvl = constant amplitude
>   wins = Window LocalZero (TimeEvent st) zero |>
>         Window LocalZero (TimeEvent inf) lvl |>
>         nullWindow
>   in pieceRep wins
> )
```

#### A.10.1.1.3 SingleTrapezoid ::NonPeriodic

```
>bscSingleTrapezoid::(Physical a, Physical b) => b->a->a->a->SignalRep a b
>bscSingleTrapezoid =
> (\amplitude startTime riseTime pulseWidth fallTime ->let
>   startTime' = fromPhysical startTime
>   riseTime' = fromPhysical riseTime
>   pulseWidth' = fromPhysical pulseWidth
>   fallTime' = fromPhysical fallTime
>   wins = Window LocalZero (TimeEvent startTime') (constant (toPhysical 0)) |>
>         Window LocalZero (TimeEvent riseTime')
>         (linear ((fromPhysical amplitude)/riseTime') (toPhysical 0)) |>
>         Window LocalZero (TimeEvent pulseWidth') (constant amplitude) |>
>         Window LocalZero (TimeEvent fallTime')
>         (linear (-(fromPhysical amplitude)/fallTime') amplitude) |>
>         Window LocalZero (TimeEvent inf) (constant (toPhysical 0)) |>
>         nullWindow
>   in pieceRep wins
> )
```

#### A.10.1.1.4 Noise ::NonPeriodic

```
>bscNoise::(Physical a, Physical b) => b->Integer->Frequency->SignalRep a b
>bscNoise =
> (\amplitude seed frequency -> let
>   pfive = constant (toPhysical (- 0.5))
>   amp = constant (toPhysical (2.0 * (fromPhysical amplitude)))
>   per = toPhysical ( 1.0 / (fromPhysical frequency))
>   in mulSig amp (sumSig pfive (random seed per))
> )
```

#### A.10.1.1.5 SingleRamp ::NonPeriodic

```
>bscSingleRamp::(Physical a, Physical b) => b->a->a->SignalRep a b
>bscSingleRamp =
> (\amplitude riseTime startTime -> let
>   amplitude' = fromPhysical amplitude
```

```

>   startTime' = fromPhysical startTime
>   riseTime' = fromPhysical riseTime
>   wins = Window LocalZero (TimeEvent startTime') (constant (toPhysical 0)) |>
>         Window LocalZero (TimeEvent riseTime')
>         (linear (amplitude'/riseTime') (toPhysical 0)) |>
>         Window LocalZero (TimeEvent inf) (constant amplitude) |>
>         nullWindow
>   in pieceRep wins
> )

```

## A.10.1.2 Periodic ::Source

### A.10.1.2.1 Sinusoid ::Periodic

```

>bscSinusoid::(Physical a, Physical b) => b->Frequency->PlaneAngle->SignalRep a b
>bscSinusoid =
> (\amplitude frequency phase ->
>   sine amplitude frequency phase
> )

```

### A.10.1.2.2 Trapezoid ::Periodic

```

>bscTrapezoid::(Physical a, Physical b) => b->a->a->a->a->SignalRep a b
>bscTrapezoid =
> (\amplitude period riseTime pulseWidth fallTime -> let
>   period' = fromPhysical period
>   riseTime' = fromPhysical riseTime
>   pulseWidth' = fromPhysical pulseWidth
>   fallTime' = fromPhysical fallTime
>   trapezoid = pieceRep $
>   Window LocalZero (TimeEvent 0.0) (ZRep) |>
>   Window LocalZero (TimeEvent riseTime')
>   (linear ((fromPhysical amplitude)/riseTime') (toPhysical 0)) |>
>   Window LocalZero (TimeEvent pulseWidth') (constant amplitude) |>
>   Window LocalZero (TimeEvent fallTime')
>   (linear ((fromPhysical amplitude)/fallTime') amplitude) |>
>   Window LocalZero (TimeEvent (period'-(riseTime'+pulseWidth'+fallTime')))
>   (constant (toPhysical 0)) |>
>   nullWindow
>   in pieceRep $ cycleWindows $
>   Window LocalZero (TimeEvent period') trapezoid |> nullWindow
> )

```

### A.10.1.2.3 Ramp ::Periodic

```

>bscRamp::(Physical a, Physical b) => b->a->a->SignalRep a b
>bscRamp =
> (\amplitude period riseTime -> let
>   per = fromPhysical period
>   rt = fromPhysical riseTime
>   v = fromPhysical amplitude
>   rsl = (v / rt)
>   fsl = (- v / (per - rt))
>   wins = Window LocalZero (TimeEvent rt) (linear rsl (toPhysical 0.0)) |>
>         Window LocalZero (TimeEvent (per - rt)) (linear fsl amplitude) |>
>         nullWindow

```

```

> ramp = pieceRep wins
> in pieceRep $ cycleWindows $
>     Window LocalZero (TimeEvent per) ramp |> nullWindow
> )

```

#### A.10.1.2.4 Triangle ::Periodic

```

>bscTriangle::(Physical a, Physical b) => b->a->Ratio->SignalRep a b
>bscTriangle =
> (\amplitude period dutyCycle -> let
>   per = fromPhysical period
>   v = fromPhysical amplitude
>   qper = per *(fromPhysical dutyCycle) / 2.0
>   sl = (v / qper)
>   nsl = 2.0* (-v) / (per - 2.0*qper)
>   wins = Window LocalZero (TimeEvent qper) (linear sl (toPhysical 0.0)) |>
>         Window LocalZero (TimeEvent (per - 2.0*qper)) (linear nsl amplitude) |>
>         Window LocalZero (TimeEvent qper) (linear sl (toPhysical (- v))) |>
>         nullWindow
>   in pieceRep (cycleWindows wins)
> )

```

#### A.10.1.2.5 SquareWave ::Periodic

```

>bscSquareWave::(Physical a, Physical b) => b->a->Ratio->SignalRep a b
>bscSquareWave =
> (\amplitude period dutyCycle -> let
>   per = fromPhysical period
>   lvl = fromPhysical amplitude
>   trans = per * (fromPhysical dutyCycle)
>   slvl = constant amplitude
>   nslvl = constant (toPhysical (- lvl))
>   wins = Window LocalZero (TimeEvent trans) slvl |>
>         Window LocalZero (TimeEvent (per-trans)) nslvl |>
>         nullWindow
>   in pieceRep (cycleWindows wins)
> )

```

#### A.10.1.2.6 WaveformRamp ::Periodic

```

>bscWaveformRamp::(Physical a, Physical b) => b->a->[Float]->SignalRep a b
>bscWaveformRamp =
> (\amplitude samplingInterval points -> let
>   pts = map ((* (fromPhysical amplitude)) points)
>   in waveform samplingInterval (map toPhysical pts)
> )

```

#### A.10.1.2.7 WaveformStep ::Periodic

```

>bscWaveformStep::(Physical a, Physical b) => b->a->[Float]->SignalRep a b
>bscWaveformStep =
> (\amplitude samplingInterval points -> let
>   pts = map ((* (fromPhysical amplitude)) points)
>   in FunctionRep (\t->cycle (map toPhysical pts) !!

```

```
> floor (fromPhysical t / (fromPhysical samplingInterval))
> )
```

## A.10.2 Conditioner ::SignalFunction

### A.10.2.1 Filter ::Conditioner

#### A.10.2.1.1 BandPass ::Filter

```
>bscBandPass::(Physical a, Physical b, Signal s) =>
>   Frequency->Frequency->Ratio->Ratio->Ratio->Ratio->
>   Float->Int->s a b->SignalRep a b
>bscBandPass =
> (\centerFrequency frequencyBand gain
>   rolloff passBandRipple stopBandRipple fInterval samples signal -> let
>   t = toPhysical (fInterval);s=2^ceiling(logBase 2 $ fromIntegral samples)
>   cf = fromPhysical centerFrequency; fb = fromPhysical frequencyBand
>   x = truncate((cf-fb/2)*(fromPhysical t))
>   y = truncate((cf+fb/2)*(fromPhysical t))
>   z = cycle [0:+0]
>   o = cycle [1:+0]
>   mask = take (x) z ++ take (y-x+1) o ++ take (s-2*y-1) z
>         ++ take (y-x+1) o ++ take (x) z
>---Remember only good for 0.5 sample freq
>---The top freq half is need to get good response. add padding to middle
>   times = zipWith (*)
>   til = sampleCount (toPhysical 0.0) t s signal
>   fil = fft $ map (\x->(fromPhysical x):+0.0) til
>   frl = map realPart $ fftinv $ if x*2<s then mask`times` fil
>         else take s z
>   in waveform (toPhysical ((fromPhysical t)/(fromIntegral s)))
>               (map toPhysical frl)
> )
```

#### A.10.2.1.2 LowPass ::Filter

```
>bscLowPass::(Physical a, Physical b, Signal s) =>
>   Frequency-> Ratio-> Ratio-> Ratio->Ratio-> Float->Int->s a b->SignalRep a b
>bscLowPass =
> (\passband gain rolloff passBandRipple stopBandRipple
>   fInterval samples signal -> let
>   t = toPhysical (fInterval);s=2^ceiling(logBase 2 $ fromIntegral samples)
>   x = truncate((fromPhysical passband)*(fromPhysical t))
>   z = cycle [0:+0]
>---Remember only good for 0.5 sample freq
>---The top freq half is need to get good response. add padding to middle
>   til = sampleCount (toPhysical 0.0) t s signal
>   fil = fft $ map (\x->(fromPhysical x):+0.0) til
>   frl = map realPart $ fftinv $ if x*2<s
>         then (take (1+x) fil) ++ (take (s-x*2-1) z) ++ (drop (s-x) fil)
>         else fil
>   in waveform (toPhysical ((fromPhysical t)/(fromIntegral s)))
>               (map toPhysical frl)
> )
```

### A.10.2.1.3 HighPass ::Filter

```

>bscHighPass::(Physical a, Physical b, Signal s) =>
>   Frequency-> Ratio-> Ratio-> Ratio->Ratio->Float->Int->s a b->SignalRep a b
>bscHighPass =
> (\passband gain rollOff passBandRipple stopBandRipple
>   fInterval samples signal -> let
>   t = toPhysical fInterval;s=2^ceiling(logBase 2 $ fromIntegral samples)
>   x = truncate((fromPhysical passband)*(fromPhysical t))
>   z = cycle [0:+0]
>--Remember only good for 0.5 sample freq
>--The top freq half is need to get good response. add padding to middle
>   til = sampleCount (toPhysical 0.0) t s signal
>   fil = fft $ map (\x->(fromPhysical x):+0.0) til
>   frl = map realPart $ fftinv $ if x*2<s
>         then (take (1+x) z) ++ drop (1+x) (take (s-x) fil) ++ (take (x) z)
>         else take s z
>   in waveform (toPhysical ((fromPhysical t)/(fromIntegral s)))
>               (map toPhysical frl)
> )

```

### A.10.2.1.4 Notch ::Filter

```

>bscNotch::(Physical a, Physical b, Signal s) =>
>   Frequency->Frequency->Ratio->Ratio->Ratio->Ratio->
>   Float->Int->s a b->SignalRep a b
>bscNotch =
> (\centerFrequency frequencyBand gain rollOff passBandRipple stopBandRipple
>   fInterval samples signal -> let
>   t = toPhysical (fInterval);s=2^ceiling(logBase 2 $ fromIntegral samples)
>   cf = fromPhysical centerFrequency; fb = fromPhysical frequencyBand
>   x = truncate((cf-fb/2)*(fromPhysical t))
>   y = truncate((cf+fb/2)*(fromPhysical t))
>   z = cycle [0:+0]
>   o = cycle [1:+0]
>   mask = take (x) o ++ take (y-x+1) z ++ take (s-2*y-1) o
>         ++ take (y-x+1) z ++ take (x) o
>--Remember only good for 0.5 sample freq
>--The top freq half is need to get good response. add padding to middle
>   times = zipWith (*)
>   til = sampleCount (toPhysical 0.0) t s signal
>   fil = fft $ map (\x->(fromPhysical x):+0.0) til
>   frl = map realPart $ fftinv $ if x*2<s then mask`times` fil
>         else take s z
>   in waveform (toPhysical ((fromPhysical t)/(fromIntegral s)))
>               (map toPhysical frl)
> )

```

### A.10.2.2 Combiner ::Conditioner

#### A.10.2.2.1 Sum ::Combiner

```

>bscSum::(Signal s, Physical a, Physical b) => [s a b]->SignalRep a b
>bscSum =
> (\signals -> foldl1 sumSig ((map toSig signals)++[ZRep]))

```

### A.10.2.2.2 Product ::Combiner

```
>bscProduct::(Signal s, Physical a, Physical b) => [s a b]->SignalRep a b
>bscProduct =
> (\signals -> foldl1 mulSig ((map toSig signals)++[ZRep]))
```

### A.10.2.2.3 Diff ::Combiner

```
>bscDiff::(Signal s, Physical a, Physical b) => [s a b]->SignalRep a b
>bscDiff =
> (\signals -> foldl1 diffSig ((map toSig signals)++[ZRep]))
```

### A.10.2.3 Modulator ::Conditioner

#### A.10.2.3.1 FM ::Modulator

```
>bscFM::(Signal s, Physical a, Physical b) =>
> b->Frequency->Frequency-> (s a b)->SignalRep a b
>bscFM =
> (\amplitude carrierFrequency frequencyDeviation signal -> let
>   phsfnc = mulSig (constant (toPhysical (fromPhysical frequencyDeviation*2*pi)))
>   (i_dx (1/(16*fromPhysical carrierFrequency)) $ toSig signal)
>   freqFn = constant (toPhysical (fromPhysical carrierFrequency))
>   in sineFunc (constant amplitude) freqFn phsfnc
> )
```

#### A.10.2.3.2 AM ::Modulator

```
>bscAM::(Physical a, Physical b, Signal s, Signal s') => Ratio->(s a b)->(s' a b)-
>SignalRep a b
>bscAM =
> (\modIndex carrier signal -> let
>   one = constant (toPhysical 1.0)
>   modsig = mulSig (constant (toPhysical (fromPhysical modIndex))) signal
>   in mulSig carrier (sumSig one modsig)
> )
```

#### A.10.2.3.3 PM ::Modulator

```
>bscPM::(Physical a, Physical b, Signal s) =>
> b->Frequency->PlaneAngle-> (s a b)->SignalRep a b
>bscPM =
> (\amplitude carrierFrequency phaseDeviation signal -> let
>   phsfnc = mulSig (constant (toPhysical (fromPhysical phaseDeviation))) signal
>   freqFn = constant (toPhysical (fromPhysical carrierFrequency))
>   in sineFunc (constant amplitude) freqFn phsfnc
> )
```

## A.10.2.4 Transformation ::Conditioner

### A.10.2.4.1 SignalDelay ::Transformation

```

>bscSignalDelay::(Signal s, Physical a, Physical b) =>
>      Frequency->a->Ratio->s a b->SignalRep a b
>bscSignalDelay =
> (\acceleration delay rate signal -> let
>   acceleration' = fromPhysical acceleration
>   delay' = fromPhysical delay
>   rate' = fromPhysical rate
>   dt t = delay' + rate'*t + acceleration'*t*t/2
>   t' t = max 0 (t-(dt t))
>   delaySigInit s =
>     FunctionRep (\t ->mapSignal s (toPhysical (t' (fromPhysical t))))
>
>-- The above function creates a functional signal delay but does not maintain the
>-- states. The code below maintains states for models where time does not go
>-- negative. Currently user needs to select simulation model best suited to their
>-- Use case
>
>   delaySigInit_alt s = pieceRep $ Windows $
>     (Window LocalZero (TimeEvent (t' 0)) ZRep) :
>     delayWin 0.0 (functionWindows s)
>
>   delaySig _ ZRep = ZRep
>   delaySig _ XRep = XRep
>   delaySig _ NullRep = NullRep
>   delaySig gt (FunctionRep fn) =
>     FunctionRep (\t ->fn (toPhysical (kt gt (fromPhysical t))))
>   delaySig _ (PieceContRep (Windows xs)) =PieceContRep$Windows $ delayWin 0.0 xs
>   st t = t - delay' - rate'*t - acceleration'*t*t/2
>   kt gt t = st ((t' gt)+t) - gt
>
>   delayWin gt ((Window z e s):xs) =
>     (Window z (TimeEvent (t' (ec e gt))-(t' gt)) (delaySig gt s)):
>     (delayWin (ec e gt) xs)
>   delayWin gt [] = []
>
>   sqe c 0 0 t = -c
>   sqe c b 0 t = -c/b
>   sqe c b a t = (-b + (sqrt ((b*b-4*a*c))))/(2*a)
>   t'' t = max 0 (sqe (-delay'-t) (1.0-rate') (-acceleration'/2.0) t)
>-- t'' inf = inf
>   ec e gt = eventOccurs e gt
>   in delaySigInit $ toSig signal
> )

```

### A.10.2.4.2 Exponential ::Transformation

```

>bscExponential::(Physical a, Physical b, Signal s) => Float ->
>      s a b->SignalRep a b
>bscExponential =
> (\dampingFactor signal-> mulSig (expc (dampingFactor)) signal)

```

### A.10.2.4.3 E ::Transformation

```

>bscE::(Physical a, Physical b, Signal s) => s a b->SignalRep a b
>bscE =

```

```
> (\f ->
>   let f1 t = fromPhysical (mapSignal f t)
>   in FunctionRep (\t -> toPhysical (exp (f1 t)))
> )
```

#### A.10.2.4.4 Ln ::Transformation

```
>bscLn::(Physical a, Physical b, Signal s) => s a b->SignalRep a b
>bscLn =
> (\f ->
>   let f1 t = fromPhysical (mapSignal f t)
>   in FunctionRep (\t -> toPhysical (log (f1 t)))
> )
```

#### A.10.2.4.5 Negate ::Transformation

```
>bscNegate::(Physical a, Physical b, Signal s) => s a b->SignalRep a b
>bscNegate =
> (\signal -> diffSigList [signal, signal, signal])
```

#### A.10.2.4.6 Inverse ::Transformation

```
>bscInverse::(Physical a, Physical b, Signal s) => s a b->SignalRep a b
>bscInverse =
> (\f ->
>   let f1 t = fromPhysical (mapSignal f t)
>   in FunctionRep (\t -> toPhysical (1.0 / (f1 t)))
> )
```

#### A.10.2.4.7 PulseTrain ::Transformation

```
>bscPulseTrain::(Physical a, Physical b, Signal s) =>
>   PulseDefns -> Int -> s a b->SignalRep a b
>bscPulseTrain =
> (\pulses repetition ->let
> {
>   rpt = let
>   {
>     pt ps= let
>     {
>       pulse (a, b, c) = let
>       {
>         zero = constant (toPhysical 0.0)
>         ;level = constant(toPhysical c)
>         ;wins = Window LocalZero (TimeEvent a) zero |>
>           Window LocalZero (TimeEvent (b)) level |>
>           nullWindow
>       }
>     in pieceRep (wins)
>   }
>   in sumSigList(map pulse ps)
> }
>   ;width (a, b, _) = a + b
>   ;maxWidth ps = foldl (\v p->max v (width p)) 0 ps
```

```

>
>     ;win2 ps= Window LocalZero (TimeEvent (maxWidth ps)) (pt ps) |>
>         nullWindow
>     ;repN 0 [] = nullWindow
>     ;repN 0 pts = cycleWindows (win2 pts)
>     ;repN rep pts = repNWindows rep (win2 pts)
> }
> in pieceRep(repN repetition pulses)
> }
> in mulSig rpt )

```

#### A.10.2.4.8 Attenuator ::Transformation

```

>bscAttenuator::(Physical a, Physical b, Signal s) => Ratio ->
>                                                     s a b -> SignalRep a b
>bscAttenuator =
> (\gain -> mulSig (constant (toPhysical (fromPhysical gain))))

```

#### A.10.2.4.9 Load ::Transformation

```

>bscLoad::(Physical a, Physical b, Signal s) =>
> Resistance -> Resistance -> s a b -> SignalRep a b
>bscLoad _ _ =
> id.toSig

```

#### A.10.2.4.10 Limit ::Transformation

```

>bscLimit::(Physical a, Physical b, Signal s) => b -> s a b -> SignalRep a b
>bscLimit =
> (\limit_ signal ->
>   FunctionRep (\t->Pure.limit (toPhysical (- fromPhysical limit_))
>     (toPhysical (fromPhysical limit_)) (mapSignal signal t)))

```

#### A.10.2.4.11 FFT ::Transformation

```

>bscFFT::(Physical a, Physical a', Physical b, Signal s) =>
> Int -> a -> a' -> s a b -> SignalRep a' b
>bscFFT =
> (\samples interval zoom signal ->
>   fourTrans samples interval zoom (\z->2*magnitude z) (toSig signal))

```

## A.10.3 EventFunction ::SignalFunction

### A.10.3.1 EventSource ::EventFunction

#### A.10.3.1.1 Clock ::EventSource

```
>bscClock::(Physical a, Physical b) => Frequency -> SignalRep a b
>bscClock =
> (\clock_rate->
>   let {
>     ;per = 0.5 / (fromPhysical clock_rate)
>     ;one = constant (toPhysical 1.0)
>     ;zer = NullRep
>     ;wins = Window GlobalZero (TimeEvent per) one |>
>             Window GlobalZero (TimeEvent per) zer |>
>             nullWindow
>   }in pieceRep (cycleWindows wins))
```

#### A.10.3.1.2 TimedEvent ::EventSource

```
>bscTimedEvent::(Physical a, Physical b) =>
>   a -> a-> a -> Int -> SignalRep a b
>bscTimedEvent =
> (\delay duration period repetition -> let {
>   ;delay' = fromPhysical delay
>   ;duration' = fromPhysical duration
>   ;period' = fromPhysical period
>   ;one = constant (toPhysical 1)
>   ;zero = NullRep
>   ;repNX 0 0 1s = cycleWindows 1s
>   ;repNX delay 0 1s = Window GlobalZero (TimeEvent delay) zero |>
>                       cycleWindows 1s
>   ;repNX 0 x 1s = repNWindows x 1s
>   ;repNX delay x 1s (= Window GlobalZero (TimeEvent delay) zero |>
>                       repNWindows x 1s
>   } in pieceRep (repNX delay' repetition
>                 (Window GlobalZero (TimeEvent duration') one |>
>                 Window GlobalZero (TimeEvent (period'-duration')) zero |>
>                 nullWindow))
> )
```

#### A.10.3.1.3 PulsedEvent ::EventSource

```
>bscPulsedEvent::(Physical a, Physical b) =>
>   [(Float, Float, Float)]-> Int -> SignalRep a b
>bscPulsedEvent =
> (\pulses repetition -> let {
>   pt ps = let
>     {
>       pulse (a, b, _) = let
>         {
>           zero = NullRep
>           ;sig = constant (toPhysical 1)
>           ;funcwins = Window GlobalZero (TimeEvent (a)) zero :
>                       Window GlobalZero (TimeEvent (b)) sig :
>                       []
>         } in funcwins
```

```

>         ;xOr ZRep s = s
>         ;xOr s ZRep = s
>         ;xOr NullRep s = s
>         ;xOr s _ = s
>         ;sumEvnts a b = splice xOr a b 0.0
>     }
>     in Windows $ foldll sumEvnts (map pulse ps)
>     ;repN 0 [] = nullWindow
>     ;repN 0 ps = cycleWindows ( pt ps)
>     ;repN rep ps = repNWindows rep ( pt ps)
> } in pieceRep(repN repetition pulses)

```

### A.10.3.2 EventConditioner ::EventFunction

#### A.10.3.2.1 EventedEvent ::EventConditioner

```

>bscEventedEvent::(Signal s, Physical a, Physical b) => [s a b] -> SignalRep a b
>bscEventedEvent =
> (\events -> let{
>     ;one = constant (toPhysical 1)
>     ;ebe e d = let
>     {
>         ;enable = e
>         ;disable = d
>         ;wins = Window GlobalZero (ActiveEvent enable) NullRep |>
>             Window GlobalZero (ActiveEvent disable) one |>
>             nullWindow
>     }
>     in pieceRep $ cycleWindows wins
> ;sglEvt e = let
> {
>     ;wins = Window GlobalZero (ActiveEvent e) NullRep |>
>         Window GlobalZero (TimeEvent inf) one |> nullWindow
> }
>     in pieceRep $ wins
> } in case (map toSig events) of
> (e:[]) -> sglEvt e
> (es) -> foldll ebe es)

```

#### A.10.3.2.2 EventCount ::EventConditioner

```

>bscEventCount::(Physical a, Physical b, Signal s) =>
> Int -> s a b -> SignalRep a b
>bscEventCount =
> (\count event -> let {
>     ;ec [] _ = [Window GlobalZero (TimeEvent inf) ZRep]
>     ;ec ((w@(Window z e ZRep)):wl) x = w:ec wl x
>     ;ec ((w@(Window z e NullRep)):wl) x = w:ec wl x
>     ;ec ((w@(Window z e s)):wl) x = if (x<=0) then w:ec wl (x+count)
>         else (Window z e NullRep):ec wl (x-1)
>     } in pieceRep $ Windows $ ec (functionWindows (toSig event)) count
> )

```

#### A.10.3.2.3 ProbabilityEvent ::EventConditioner

```

>bscProbabilityEvent::(Physical a,

```

```

> Physical b, Signal s) => Integer -> Ratio -> s a b -> SignalRep a b
>bscProbabilityEvent =
> (\seed propability event -> let {
> ;pbe [] _ _ _ = [Window GlobalZero (TimeEvent inf) ZRep]
> ;pbe ((w@(Window z e ZRep)):wl) xs _ _ = w:pbe wl xs True True
> ;pbe ((w@(Window z e NullRep)):wl) xs _ _ = w:pbe wl xs True True
> ;pbe ws (x:xl) True _ = pbe ws xl False (x<=fromPhysical propability)
> ;pbe ((w@(Window z e s)):wl) xs False notNull =
> (if notNull then w else (Window z e NullRep)):pbe wl xs False notNull
> } in pieceRep $ Windows $
> pbe (functionWindows (toSig event)) (rand seed) True True
> )

```

#### A.10.3.2.4 NotEvent ::EventConditioner

```

>bscNotEvent::(Physical a, Physical b, Signal s) => s a b -> SignalRep a b
>bscNotEvent =
> (\event ->
> let {
> ;xNot ZRep _ = ZRep
> ;xNot XRep _ = XRep
> ;xNot NullRep _ = constant (toPhysical 1.0)
> ;xNot _ _ = NullRep
> } in pieceRep $ Windows $
> (\a b->splice xNot a b 0.0) (functionWindows (toSig event)) []
> )

```

#### A.10.3.2.5 Logical ::EventConditioner

##### A.10.3.2.5.1 OrEvent ::Logical

```

>bscOrEvent::(Signal s, Physical a, Physical b) => [s a b] -> SignalRep a b
>bscOrEvent =
> (\events ->
> let {
> ;xOr s ZRep = s
> ;xOr ZRep s = s
> ;xOr s XRep = s
> ;xOr XRep s = s
> ;xOr NullRep s = s
> ;xOr s _ = s
> } in pieceRep $ Windows $
> foldll (\a b->splice xOr a b 0.0) (map (functionWindows.toSig) events)
> )

```

##### A.10.3.2.5.2 XOrEvent ::Logical

```

>bscXOrEvent::(Signal s, Physical a, Physical b) => [s a b] -> SignalRep a b
>bscXOrEvent =
> (\events ->
> let {
> ;xXOr s ZRep = s
> ;xXOr ZRep s = s
> ;xXOr s@XRep XRep = s
> ;xXOr XRep _ = NullRep
> ;xXOr _ XRep = NullRep

```

```

> ;xXOr s NullRep = s
> ;xXOr NullRep s = s
> ;xXOr _ _ = NullRep
> } in pieceRep $ Windows $
> foldl1 (\a b-> splice xXOr a b 0.0) (map (functionWindows.toSig) events)
> )

```

### A.10.3.2.5.3 AndEvent ::Logical

```

>bscAndEvent::(Signal s, Physical a, Physical b) => [s a b] -> SignalRep a b
>bscAndEvent =
> (\events ->
> let {
> ;xAnd s ZRep = s
> ;xAnd ZRep s = s
> ;xAnd s@NullRep _ = s
> ;xAnd _ s@NullRep = s
> ;xAnd s XRep = s
> ;xAnd XRep s = s
> ;xAnd s _ = s
> } in pieceRep $ Windows $
> foldl1 (\a b->splice xAnd a b 0.0) (map (functionWindows.toSig) events)
> )

```

### A.10.4 Sensor ::SignalFunction

#### A.10.4.1 Counter ::Sensor

```

>bscCounter:: (Physical a)=> [a] -> Int
>bscCounter =
> \points -> length points
>bscCounter':: (Physical a)=> [a] -> Int
>bscCounter' =
> \points -> length points

```

#### A.10.4.2 Interval ::Sensor

```

>bscInterval:: (Physical a, Physical b)=> [(a, b)] -> b
>bscInterval =
> \points->snd (head points)
>bscInterval':: (Physical a, Physical b)=> [(a, b)] -> b
>bscInterval' =
> \points->toPhysical $
> (fromPhysical(snd (last points))) - (fromPhysical(snd (head points)))

```

#### A.10.4.3 Instantaneous ::Sensor

```

>bscInstantaneous:: (Physical a, Physical b)=> [(a, b)] -> a
>bscInstantaneous =
> \points -> fst (head points)
>bscInstantaneous':: (Physical a, Physical b)=> [(a, b)] -> b
>bscInstantaneous' =

```

```
> \points -> snd (head points)
```

#### A.10.4.4 RMS ::Sensor

```
>bscRMS:: (Physical a, Physical b)=> [(a, b)] -> a  
>bscRMS =  
> \points -> toPhysical $ sqrt $ (foldl (+) 0 $ map  
> ((\x->x*x).fromPhysical.fst) points) / fromIntegral (length points)  
>bscRMS':: (Physical a, Physical b)=> [(a, b)] -> b  
>bscRMS' =  
> \points -> toPhysical $ sqrt $ (foldl (+) 0 $ map  
> ((\x->x*x).fromPhysical.snd) points) / fromIntegral (length points)
```

#### A.10.4.5 Average ::Sensor

```
>bscAverage:: (Physical a, Physical b)=> [(a, b)] -> a  
>bscAverage =  
> \points -> toPhysical $  
> (foldl (+) 0 (map (fromPhysical.fst) points)) / fromIntegral (length points)  
>bscAverage':: (Physical a, Physical b)=> [(a, b)] -> b  
>bscAverage' =  
> \points -> toPhysical $  
> (foldl (+) 0 (map (fromPhysical.snd) points)) / fromIntegral (length points)
```

#### A.10.4.6 PeakToPeak ::Sensor

```
>bscPeakToPeak:: (Physical a, Physical b)=> [(a, b)] -> a  
>bscPeakToPeak =  
> \points -> let {  
> ;h = fromPhysical.fst $ maximum points  
> ;l = fromPhysical.fst $ minimum points  
> } in toPhysical $ h - l  
>bscPeakToPeak':: (Physical a, Physical b)=> [(a, b)] -> b  
>bscPeakToPeak' =  
> \points -> let {  
> ;h = fromPhysical.snd $ maximum points  
> ;l = fromPhysical.snd $ minimum points  
> } in toPhysical $ h - l
```

#### A.10.4.7 Peak ::Sensor

```
>bscPeak:: (Physical a, Physical b)=> [(a, b)] -> a  
>bscPeak =  
> \points -> let {  
> ;peakNeg = fromPhysical (bscPeakNeg points)  
> ;peakPos = fromPhysical (bscPeakPos points)  
> } in if abs peakNeg < abs peakPos  
> then bscPeakPos points  
> else bscPeakNeg points  
>bscPeak':: (Physical a, Physical b)=> [(a, b)] -> b  
>bscPeak' =  
> \points -> let {  
> ;peakNeg = fromPhysical (bscPeakNeg points)  
> ;peakPos = fromPhysical (bscPeakPos points)
```

```

> } in if abs peakNeg < abs peakPos
>     then bscPeakPos' points
>     else bscPeakNeg' points

```

#### A.10.4.8 PeakPos ::Sensor

```

>bscPeakPos:: (Physical a, Physical b)=> [(a, b)] -> a
>bscPeakPos =
> \points -> toPhysical $
> (fromPhysical (bscMaxInstantaneous points)) - (fromPhysical (bscAverage points))
>bscPeakPos':: (Physical a, Physical b)=> [(a, b)] -> b
>bscPeakPos' =
> \points -> toPhysical $ (fromPhysical
> (bscMaxInstantaneous' points)) - (fromPhysical (bscAverage' points))

```

#### A.10.4.9 PeakNeg ::Sensor

```

>bscPeakNeg:: (Physical a, Physical b)=> [(a, b)] -> a
>bscPeakNeg =
> \points -> toPhysical $
> (fromPhysical (bscMinInstantaneous points)) - (fromPhysical (bscAverage points))
>bscPeakNeg':: (Physical a, Physical b)=> [(a, b)] -> b
>bscPeakNeg' =
> \points -> toPhysical $ (fromPhysical
> (bscMinInstantaneous' points)) - (fromPhysical (bscAverage' points))

```

#### A.10.4.10 MaxInstantaneous ::Sensor

```

>bscMaxInstantaneous:: (Physical a, Physical b)=> [(a, b)] -> a
>bscMaxInstantaneous =
> \points -> fst $ maximum points
>bscMaxInstantaneous':: (Physical a, Physical b)=> [(a, b)] -> b
>bscMaxInstantaneous' =
> \points -> snd $ maximum points

```

#### A.10.4.11 MinInstantaneous ::Sensor

```

>bscMinInstantaneous:: (Physical a, Physical b)=> [(a, b)] -> a
>bscMinInstantaneous =
> \points -> fst $ minimum points
>bscMinInstantaneous':: (Physical a, Physical b)=> [(a, b)] -> b
>bscMinInstantaneous' =
> \points -> snd $ minimum points

```

#### A.10.4.12 Measure ::Sensor

```

>-- Implements ONLY specific Physical BSC Generic Measurement attributes
>bscMeasure::(Physical a, Physical b, Physical c)=>
> String -> c -> (BSC a b) -> c
>bscMeasure "amplitude" nominal signal =

```

```

>         toPhysical (fromPhysical (BSC.amplitude signal))
>bscMeasure "fallTime" nominal signal =
>         toPhysical (fromPhysical (BSC.fallTime signal))
>bscMeasure "frequency" nominal signal =
>         toPhysical (fromPhysical (BSC.frequency signal))
>bscMeasure "period" nominal signal =
>         toPhysical (fromPhysical (BSC.period signal))
>bscMeasure "phase" nominal signal = toPhysical (fromPhysical (BSC.phase signal))
>bscMeasure "pulseWidth" nominal signal =
>         toPhysical (fromPhysical (BSC.pulseWidth signal))
>bscMeasure "riseTime" nominal signal =
>         toPhysical (fromPhysical (BSC.riseTime signal))
>bscMeasure "startTime" nominal signal =
>         toPhysical (fromPhysical (BSC.startTime signal))
>-- Users Add as necessary

```

#### A.10.4.13 Decode ::Sensor

```

>-- Implements ONLY specific BSC Encoder / Decoder
>bscDecode::(Physical a, Physical b )=>
>         String -> String -> (BSC a b) -> String
>bscDecode _ _ (Encode data_ width repetition datatype encoding signal channel) =
>         data_

```

#### A.10.5 Control ::SignalFunction

##### A.10.5.1 SelectIf ::Control

```

>bscSelectIf::(Physical a, Physical b, Signal s, Signal s')=>
>         (s a b) -> [(s' a b)] -> SignalRep a b
>bscSelectIf selector inputs = let {
> ; makewin s k ((w@(Window z e _)):wl) [] = (Window z e s): makewin s k wl []
> ; makewin s k [] (i:is) =
>         (Window GlobalZero (TimeEvent inf) s): makewin s k [] (i:is)
> ; makewin s k ((w@(Window z e ZRep)):wl) (i:is) =
>         (Window GlobalZero e s): makewin s k wl (i:is)
> ; makewin s k ((w@(Window z e XRep)):wl) is = w:makewin XRep k wl is
> ; makewin s False ((w@(Window z e NullRep)):wl) (i:is) =
>         (Window GlobalZero e (toSig i)): makewin (toSig i) False wl (i:is)
> ; makewin s True ((w@(Window z e NullRep)):wl) (i':i:is) =
>         (Window GlobalZero e (toSig i)): makewin (toSig i) False wl (i:is)
> ; makewin s False ((w@(Window z e _)):wl) (i':i:is) =
>         (Window GlobalZero e (toSig i)): makewin (toSig i) True wl (i:is)
> ; makewin s True ((w@(Window z e _)):wl) (i:is) =
>         (Window GlobalZero e (toSig i)): makewin (toSig i) True wl (i:is)
> } in pieceRep $ Windows $
>         makewin ZRep False (functionWindows (toSig selector)) $ cycle inputs

```

##### A.10.5.2 SelectCase ::Control

```

>--Simple mapping ignoring input states
>bscSelectCase::(Physical a, Physical b, Signal s', Signal s)=>
>         Int -> [(s' a b)] -> [(s a b)] -> SignalRep a b
>bscSelectCase mask selectors inputs = let {
> ; inputs' = (map toSig inputs) ++ cycle [ZRep]
> ; toNum x y = if y then 2*x+1 else 2*x

```

```

> ; active t = foldl toNum 0 $
>               reverse $ map (\s -> not $ isInactive s t) selectors
> ; mask1 = if mask==0 then (-1) else mask
> } in FunctionRep (\t->mapSignal (inputs'!!(mask1 .&. active t)) t)

```

### A.10.5.3 Encode ::Control

```

>bscEncode::(Physical a, Physical b, Signal s)=>
>           String -> Int -> Int -> String -> String ->
>           (s a b) -> [SignalRep a b]
>bscEncode data_ width repetition datatype encoding signal = [ZRep]
>
>bscEncodeDigital::(Physical a, Physical b, Signal s)=>
>                 [[Digital]] -> Int -> Int -> String ->
>                 (s a b) -> [SignalRep a b]
>bscEncodeDigital data_ width repetition encoding signal = let
>   makeDigSignal Z = ZRep
>   makeDigSignal X = XRep
>   makeDigSignal L = NullRep
>   makeDigSignal H = constant (toPhysical 1)
>   makeSigWindows digit =
>     Window GlobalZero (ActiveEvent (toSig signal)) (makeDigSignal digit)
>   makeSignals digits = pieceRep $ Windows $
>     Window GlobalZero (ActiveEvent (toSig signal)) ZRep:
>     map makeSigWindows digits
>   in map makeSignals data_
>
>bscEncodeBits::(FiniteBits n, Physical a, Physical b, Signal s)=>
>               [n] -> Int -> Int -> String ->
>               (s a b) -> [SignalRep a b]
>bscEncodeBits data_ width repetition encoding signal = let
>   makeSigWindows nBit n = Window GlobalZero (ActiveEvent (toSig signal)) $
>     if (testBit n nBit) then constant (toPhysical 1) else NullRep
>   makeSignals nBit = pieceRep $ Windows $
>     Window GlobalZero (ActiveEvent (toSig signal)) ZRep:
>     map (makeSigWindows nBit) data_
>   in map makeSignals [0..finiteBitSize (head data_)-1]

```

### A.10.5.4 Transform ::Control

```

>bscTransform::(Physical a, Physical b)=>
>             String->Maybe (BSC a b->BSC a b)->BSC a b->SignalRep a b
>bscTransform _ Nothing signal = toSig $ bsc signal
>bscTransform _ (Just as) signal = toSig $ bsc $ as signal
>
>bsc::(Physical a, Physical b)=>BSC a b->BSC a b
>bsc = id

```

### A.10.5.5 InverseTransform ::Control

```

>bscInverseTransform::(Physical a, Physical b)=>
>                   String->BSC a b->BSC a b->SignalRep a b
>bscInverseTransform _ as' signal = toSig $ bsc' as' signal
>
>bsc'::(Physical a, Physical b)=>BSC a b->BSC a b->BSC a b
>bsc' (Null) s = s
>bsc' ((BandPass _ _ _ _ _ as)) (BandPass _ _ _ _ _ s) = bsc' (as) s
>bsc' ((LowPass _ _ _ _ _ as)) (LowPass _ _ _ _ _ s) = bsc' (as) s

```

```

>bsc' ((HighPass _ _ _ _ _ as)) (HighPass _ _ _ _ _ s)= bsc' (as) s
>bsc' ((Notch _ _ _ _ _ as)) (Notch _ _ _ _ _ s)= bsc' (as) s
>bsc' ((FM _ _ _ as)) (FM _ _ _ s)= bsc' (as) s
>bsc' ((AM _ Null as)) (AM _ _ s)= bsc' (as) s
>bsc' ((PM _ _ _ as)) (PM _ _ _ s)= bsc' (as) s
>bsc' ((SignalDelay _ _ _ as)) (SignalDelay _ _ _ s)= bsc' (as) s
>bsc' ((Exponential _ _ as)) (Exponential _ _ s)= bsc' (as) s
>bsc' ((E as)) (E s)= bsc' (as) s
>bsc' ((Ln as)) (Ln s)= bsc' (as) s
>bsc' ((Negate as)) (Negate s)= bsc' (as) s
>bsc' ((Inverse as)) (Inverse s)= bsc' (as) s
>bsc' ((PulseTrain _ _ as)) (PulseTrain _ _ s)= bsc' (as) s
>bsc' ((Attenuator _ _ as)) (Attenuator _ _ s)= bsc' (as) s
>bsc' ((Load _ _ _ as)) (Load _ _ _ s)= bsc' (as) s
>bsc' ((Limit _ _ _ as)) (Limit _ _ _ s)= bsc' (as) s
>bsc' ((FFT _ _ _ as)) (FFT _ _ _ s)= bsc' (as) s
>bsc' ((EventCount _ _ as)) (EventCount _ _ s)= bsc' (as) s
>bsc' ((ProbabilityEvent _ _ as)) (ProbabilityEvent _ _ s)= bsc' (as) s
>bsc' ((NotEvent as)) (NotEvent s)= bsc' (as) s
>bsc' ((Encode _ _ _ _ _ as _ _ )) (Encode _ _ _ _ _ s _ ) = bsc' (as) s
>bsc' ((Transform _ _ _ as)) (Transform _ _ _ s) = bsc' (as) s
>bsc' ((InverseTransform _ _ as' as )) (InverseTransform _ _ as' s) = bsc' (as) s
>bsc' as' (Transform _ (Just as) s) = bsc' as' $ bsc $ as s
>bsc' as' (Transform _ (Nothing) s) = bsc' as' $ bsc s
>
>bscs'::(Physical a, Physical b)=>(BSC a b)->BSC a b->[BSC a b]
>bscs' (Null) s = [s]
>bscs' ((Sum [Null])) (Sum s) = s
>bscs' ((Diff [Null])) (Diff s) = s
>bscs' ((Product [Null])) (Product s) = s
>bscs' ((EventedEvent [Null])) (EventedEvent s) = s
>bscs' ((OrEvent [Null])) (OrEvent s) = s
>bscs' ((XOrEvent [Null])) (XOrEvent s) = s
>bscs' ((AndEvent [Null])) (AndEvent s) = s
>bscs' ((SelectIf Null [Null])) (SelectIf _ s) = s
>bscs' ((SelectCase _ [Null] [Null])) (SelectCase _ _ s) = s
>bscs' ((Channels _ _ [Null] _)) (Channels _ _ s _ ) = s
>bscs' ((TwoWire _ _ _ [Null] _)) (TwoWire _ _ _ s _ ) = s
>bscs' ((TwoWireComp _ _ _ [Null] _)) (TwoWireComp _ _ _ s _ ) = s
>bscs' ((ThreeWireComp _ _ _ [Null] _)) (ThreeWireComp _ _ _ s _ ) = s
>bscs' ((SinglePhase _ _ _ [Null] _)) (SinglePhase _ _ _ s _ ) = s
>bscs' ((TwoPhase _ _ _ [Null] _)) (TwoPhase _ _ _ s _ ) = s
>bscs' ((ThreePhaseDelta _ _ _ [Null] _)) (ThreePhaseDelta _ _ _ s _ ) = s
>bscs' ((ThreePhaseWye _ _ _ [Null] _)) (ThreePhaseWye _ _ _ s _ ) = s
>bscs' ((ThreePhaseSynchro _ _ _ [Null] _)) (ThreePhaseSynchro _ _ _ s _ ) = s
>bscs' ((FourWireResolver _ _ _ [Null] _)) (FourWireResolver _ _ _ s _ ) = s
>bscs' ((SynchroResolver _ _ _ [Null] _)) (SynchroResolver _ _ _ s _ ) = s
>bscs' ((Series _ [Null] _)) (Series _ _ s _ ) = s
>bscs' ((FourWire _ _ _ [Null] _)) (FourWire _ _ _ s _ ) = s
>bscs' ((NonElectrical _ _ _ [Null] _)) (NonElectrical _ _ _ s _ ) = s
>bscs' ((DigitalBus _ _ [Null] _)) (DigitalBus _ _ s _ ) = s
    
```

### A.10.5.6 Channels ::Control

```

>bscChannels::(Physical a, Physical b)=>
>    String->Maybe ([BSC a b]->BSC a b)->[BSC a b]->[SignalRep a b]
>bscChannels channelNames (Nothing) = map toSig
>bscChannels channelNames (Just ass) = map (\x -> toSig (ass [x]))
    
```

## A.10.6 Digital ::SignalFunction

### A.10.6.1 SerialDigital ::Digital

```
>bscSerialDigital::(Physical a, Physical b)=>  
>   String -> a -> b -> b -> String -> SignalRep a b  
>bscSerialDigital =  
> (\datas period logic_H_value logic_L_value pulseClass ->  
>   d2a logic_H_value logic_L_value (str2dig (fromPhysical period) datas))
```

### A.10.6.2 ParallelDigital ::Digital

```
>bscParallelDigital::(Physical a, Physical b)=>  
>   String -> a -> b -> b -> String -> [SignalRep a b]  
>bscParallelDigital =  
> (\datas period logic_H_value logic_L_value pulseClass ->  
>   map (d2a logic_H_value logic_L_value)  
>       (pat2diglist $ Pattern (fromPhysical period) datas))
```

## A.10.7 Connection ::SignalFunction

### A.10.7.1 TwoWire ::Connection

```
>bscTwoWire hi lo channelWidth =  
> (\xs -> let {  
>   ;f s@(s':[]) = s  
>   ;f [] = error "No Channels (hi)defined"  
>   ;f (x:xs) = error "Too many channels"  
>   } in f $ map toSig xs  
> )
```

### A.10.7.2 TwoWireComp ::Connection

```
>bscTwoWireComp true comp channelWidth =  
> (\xs -> let {  
>   ;f s@(s':[]) = s  
>   ;f [] = error "No Channel (true) defined"  
>   ;f (x:xs) = error "Too many channels"  
>   } in f $ map toSig xs  
> )
```

### A.10.7.3 ThreeWireComp ::Connection

```
>bscThreeWireComp true comp lo channelWidth =  
> (\xs -> let {  
>   ;f s@(s':s'':[]) = s  
>   ;f [] = error "No Channels (true,comp) defined"  
>   ;f (s:[]) = error "No channel (comp) defined"  
>   ;f (x:xs) = error "Too many channels"  
> )
```

```
>   } in f $ map toSig xs  
> )
```

#### A.10.7.4 SinglePhase ::Connection

```
>bscSinglePhase a n channelWidth =  
> (\xs -> let {  
>   ;f s@(s':s'':[]) = s  
>   ;f [] = error "No Channel (n) defined"  
>   ;f (x:xs) = error "Too many channels"  
>   } in f $ map toSig xs  
> )
```

#### A.10.7.5 TwoPhase ::Connection

```
>bscTwoPhase a b n channelWidth =  
> (\xs -> let {  
>   ;f s@(s':s'':s''':[]) = s  
>   ;f [] = error "No Channels (a,b) defined"  
>   ;f (s:[]) = error "No channel (b) defined"  
>   ;f (x:xs) = error "Too many channels"  
>   } in f $ map toSig xs  
> )
```

#### A.10.7.6 ThreePhaseDelta ::Connection

```
>bscThreePhaseDelta a b c channelWidth =  
> (\xs -> let {  
>   ;f s@(s':s'':s''':s''':[]) = s  
>   ;f [] = error "No Channels (a,b,c) defined"  
>   ;f (s:[]) = error "No channels (b,c) defined"  
>   ;f (s:s':[]) = error "No channel (c) defined"  
>   ;f (x:xs) = error "Too many channels"  
>   } in f $ map toSig xs  
> )
```

#### A.10.7.7 ThreePhaseWye ::Connection

```
>bscThreePhaseWye a b c n channelWidth =  
> (\xs -> let {  
>   ;f s@(s':s'':s''':s''':[]) = s  
>   ;f [] = error "No Channels (a,b,c) defined"  
>   ;f (s:[]) = error "No channels (b,c) defined"  
>   ;f (s:s':[]) = error "No channels (c) defined"  
>   ;f (x:xs) = error "Too many channels"  
>   } in f $ map toSig xs  
> )
```

#### A.10.7.8 ThreePhaseSynchro ::Connection

```

>bscThreePhaseSynchro x y z channelWidth =
> (\xs -> let {
>   ;f s@(s':s'':s''':[]) = s
>   ;f [] = error "No Channels (x,y,x) defined"
>   ;f (s:[]) = error "No channels (y,z)defined"
>   ;f (s:s':[]) = error "No channel (z) defined"
>   ;f (x:xs) = error "Too many channels"
>   } in f $ map toSig xs
> )

```

#### A.10.7.9 FourWireResolver ::Connection

```

>bscFourWireResolver s1 s2 s3 s4 channelWidth =
> (\xs -> let {
>   ;f s@(s':s'':s''':s''':[]) = s
>   ;f [] = error "No Channels (s1,s2,s3,s4) defined"
>   ;f (s:[]) = error "No channels (s2,s3,s4)defined"
>   ;f (s:s':[]) = error "No channels (s3,s4) defined"
>   ;f (s:s':s'':[]) = error "No channels (s4) defined"
>   ;f (x:xs) = error "Too many channels"
>   } in f $ map toSig xs
> )

```

#### A.10.7.10 SynchroResolver ::Connection

```

>bscSynchroResolver r1 r2 r3 r4 channelWidth_ =
> (\xs -> let {
>   ;f s@(s':s'':s''':s''':[]) = s
>   ;f [] = error "No Channels (r1,r2,r3,r4) defined"
>   ;f (s:[]) = error "No channels (r2,r3,r4)defined"
>   ;f (s:s':[]) = error "No channels (r3,r4) defined"
>   ;f (s:s':s'':[]) = error "No channels (r4) defined"
>   ;f (x:xs) = error "Too many channels"
>   } in f $ map toSig xs
> )

```

#### A.10.7.11 Series ::Connection

```

>bscSeries via channelWidth =
> (\xs -> let {
>   ;f s@(s':[]) = s
>   ;f [] = error "No Channels (via) defined"
>   ;f (x:xs) = error "Too many channels"
>   } in f $ map toSig xs
> )

```

#### A.10.7.12 FourWire ::Connection

```

>bscFourWire hi lo hiRef loRef channelWidth =
> (\xs -> let {
>   ;f s@(s':s'':s''':s''':[]) = s
>   ;f [] = error "No Channels (hi,lo,hiRef,loRef) defined"
>   ;f (s:[]) = error "No channels (lo,hiRef,loRef)defined"
>   ;f (s:s':[]) = error "No channels (hiRef,loRef) defined"

```

```

> ;f (s:s':s'':[]) = error "No channels (loRef) defined"
> ;f (x:xs) = error "Too many channels"
> } in f $ map toSig xs
> )

```

### A.10.7.13 NonElectrical ::Connection

```

>bscNonElectrical to from channelWidth =
> (\xs -> let {
> ;f s@(s':[]): s
> ;f [] = error "No Channel (to) defined"
> ;f (x:xs) = error "Too many channels"
> } in f $ map toSig xs
> )

```

### A.10.7.14 DigitalBus ::Connection

```

>bscDigitalBus pins channelWidth =
> id.map toSig

```

## A.11 Fast Fourier analysis support

In A.7.1, the need is discussed for a FFT module exporting type Complex Float and functions fft and fftinv. The following is provided as a default implementation, but may be substituted by more efficient alternatives.

```

> module FFT
> (ComplexF, fft, fftinv)
> where
> import Data.Complex
> import Data.List (transpose)
> type ComplexF = Complex Float
> rootsOfUnity :: Int -> [ComplexF]
> rootsOfUnity n = zipWith (:+) (map cos (thetas n))
> (map sin (thetas n))
> thetas :: Int -> [Float]
> thetas n = [(2*pi/fromIntegral n)*fromIntegral k | k<-[0 .. n-1]]
> fft :: [ComplexF] ->
> [ComplexF] -- Warning: works only for n=2^km, time=O(n log(n)) algorithm
> fft xs = map((1/(fromIntegral n))* (ffth xs us) where
> us = map conjugate (rootsOfUnity n)
> n = length xs
> fftinv :: [ComplexF] ->
> [ComplexF] -- Warning: works only for n=2^km, time=O(n log(n)) algorithm
> fftinv xs = ffth xs us where
> us = rootsOfUnity n
> n = length xs
> ffth :: [ComplexF] -> [ComplexF] -> [ComplexF]
> ffth xs us
> | n>1 = (cycle fftEvn) `plus`

```

```
>           (us `times` (cycle fftOdd))
> | n==1    = xs
> where
>   fftEvn = ffth (evns xs) uEvns
>   fftOdd = ffth (odds xs) uEvns
>   uEvns  = evns us
>   evns   = everyNth 2
>   odds   = everyNth 2 . tail
>   n      = length xs
>   everyNth n = (map head).(takeWhile (/=[])).(iterate (drop n))
>   plus     = zipWith (+)
>   times    = zipWith (*)
```

IECNORM.COM : Click to view the full PDF of IEC 62529:2024

## Annex B

(normative)

### Basic signal components (BSC) layer

#### B.1 BSC layer base classes

The base classes shown in Table B.1 are used to define BSC class properties (see 6.1).

**Table B.1—Signal function 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

#### B.2 BSC subclasses

The BSC classes are derived from the **SignalFunction** base class as subclasses, where each level is a further derivation from the base class. The hierarchical structure of the base class and subclasses is illustrated in Table B.2, in which a column has been included to indicate the attributes associated with each BSC class.

NOTE—The use of **boldface** denotes a class, subclass, or attribute in the text of this annex.

**Table B.2—BSC subclasses (Derived from SignalFunction base class)**

Subclasses			Attributes
1st level	2nd level	3rd/4th level	
Source	—	—	—
	NonPeriodic	—	—
		Constant	amplitude :: Physical
		Step	amplitude :: Physical startTime :: Time
		SingleTrapezoid	amplitude :: Physical startTime :: Time riseTime :: Time pulseWidth :: Time fallTime :: Time
		Noise	amplitude :: Physical seed :: int frequency :: Frequency
	SingleRamp	amplitude :: Physical riseTime :: Time startTime :: Time	
Periodic	—	—	

Subclasses			Attributes
1st level	2nd level	3rd/4th level	
		Sinusoid	amplitude :: Physical frequency :: Frequency phase :: PlaneAngle

**Table B.2—BSC subclasses (derived from SignalFunction base class)  
(continued)**

Subclasses			Attributes
1st level	2nd level	3rd/4th level	
		Trapezoid	amplitude :: Physical period :: Time riseTime :: Time pulseWidth :: Time fallTime :: Time
		Ramp	amplitude :: Physical period :: Time riseTime :: Time
		Triangle	amplitude :: Physical period :: Time dutyCycle :: Ratio
		SquareWave	amplitude :: Physical period :: Time dutyCycle :: Ratio
		WaveformRamp	amplitude :: Physical period :: Time samplingInterval :: Time points :: list double
		WaveformStep	amplitude :: Physical period :: Time samplingInterval :: Time points :: list double
Conditioner	—	—	—
	Filter	—	—
		BandPass	centerFrequency :: Frequency frequencyBand :: Frequency gain :: Ratio rollOff :: Ratio passBandRipple :: Ratio stopBandRipple :: Ratio
		LowPass	cutoff :: Frequency gain :: Ratio rollOff :: Ratio passBandRipple :: Ratio stopBandRipple :: Ratio
		HighPass	cutoff :: Frequency gain :: Ratio rollOff :: Ratio passBandRipple :: Ratio stopBandRipple :: Ratio
		Notch	centerFrequency :: Frequency frequencyBand :: Frequency gain :: Ratio rollOff :: Ratio passBandRipple :: Ratio stopBandRipple :: Ratio
	Combiner	—	—
		Sum	—
		Product	—

**Table B.2—BSC subclasses (derived from SignalFunction base class)  
 (continued)**

Subclasses			Attributes
1st level	2nd level	3rd/4th level	
	Modulator	Diff	—
		—	—
		FM	amplitude :: Physical carrierFrequency :: Frequency frequencyDeviation :: Frequency
		AM	modIndex :: Ratio Carrier :: Signal
	Transformation	PM	amplitude :: Physical carrierFrequency :: Frequency phaseDeviation :: PlaneAngle
		—	—
		SignalDelay	acceleration :: Frequency delay :: Time rate :: Ratio
		Exponential	dampingFactor :: double
		E	—
		Ln	—
		Negate	—
		Inverse	—
		PulseTrain	pulses :: PulseDefns repetition :: int
		Attenuator	gain :: Ratio
		Load	resistance :: Resistance reactance :: Reactance
		Limit	limit :: Physical
	FFT	samples :: int interval :: Time	
EventFunction	—	—	
	EventSource	—	—
		Clock	clockRate :: Frequency
		TimedEvent	delay :: Time duration :: Time period :: Time repetition :: int

**Table B.2—BSC subclasses (derived from SignalFunction base class)  
(continued)**

Subclasses			Attributes	
1st level	2nd level	3rd/4th level		
	EventConditioner	PulsedEvent	pulses :: PulsesDefns repetition :: int	
		—	—	
		EventedEvent	—	
		EventCount	count :: int	
		ProbabilityEvent	seed :: int probability :: Ratio	
		NotEvent	—	
		Logical	—	—
			OrEvent	—
			XOrEvent	—
			AndEvent	—
Sensor	—	—	measuredVariable::enum measurement::any measurements::list_any samples::int count::int gateTime::Physical nominal::Physical condition::enumCondition GO::int NOGO::int HI::int LO::int UL::Physical LL::Physical As::SignalFunction	

IECNORM.COM : Click to view the full PDF of IEC 62529:2024

**Table B.2—BSC subclasses (derived from SignalFunction base class)  
 (continued)**

Subclasses			Attributes
1st level	2nd level	3rd/4th level	
	Counter	—	—
	Interval	—	—
	Instantaneous	—	—
	RMS	—	—
	Average	—	—
	PeakToPeak	—	—
	Peak	—	—
	PeakPos	—	—
	PeakNeg	—	—
	MaxInstantaneous	—	—
	MinInstantaneous	—	—
	Measure	—	attribute :: string
	Decode	—	datatype :: string encoding :: string stepsize :: Physical offset :: Physical
Control	—	—	—
	SelectIf	—	Selector :: Signal
	SelectCase	—	Selector :: Signal mask :: int
	Encode	—	data :: any width :: int repetition :: int datatype :: string encoding :: string stepsize :: Physical offset :: Physical
	Transform	—	mapping :: string As :: SignalFunction
	InverseTransform	—	mapping :: string As :: SignalFunction
	Channels	—	channelNames :: pinString As :: SignalFunction
Digital	—	—	—
	SerialDigital	—	data :: digitalString period :: Time logic_H_value :: Physical logic_L_value :: Physical pulseClass :: enumPulseClass
	ParallelDigital	—	data :: digitalString period :: Time logic_H_value :: Physical logic_L_value :: Physical pulseClass :: enumPulseClass
Connection	—	—	channelWidth :: int
	TwoWire	—	(channelWidth = 1) hi :: pinString lo :: pinString
	TwoWireComp	—	(channelWidth = 1) true :: pinString comp :: pinString
	ThreeWireComp	—	(channelWidth = 1) true :: pinString comp :: pinString lo :: pinString

**Table B.2—BSC subclasses (derived from SignalFunction base class)  
(continued)**

Subclasses			Attributes
1st level	2nd level	3rd/4th level	
	SinglePhase	—	(channelWidth = 1) a :: pinString n :: pinString
	TwoPhase	—	(channelWidth = 2) a :: pinString b :: pinString n :: pinString
	ThreePhaseDelta	—	(channelWidth = 3) a :: pinString b :: pinString c :: pinString
	ThreePhaseWye	—	(channelWidth = 3) a :: pinString b :: pinString c :: pinString n :: pinString
	ThreePhaseSynchro	—	(channelWidth = 3) x :: pinString y :: pinString z :: pinString
	FourWireResolver	—	(channelWidth = 2) s1 :: pinString s2 :: pinString s3 :: pinString s4 :: pinString
	SynchroResolver	—	(channelWidth = 2) r1 :: pinString r2 :: pinString r3 :: pinString r4 :: pinString
	Series	—	(channelWidth = 1) via :: pinString
	FourWire	—	(channelWidth = 1) hi :: pinString lo :: pinString hiRef :: pinString loRef :: pinString
	NonElectrical	—	(channelWidth = 1) to :: pinString from :: pinString
	DigitalBus	—	(channelWidth = 0) pins :: pinString

### B.3 Description of a BSC

Clause B.3 describes the generic characteristics of BSCs without stating the detail of the physical characteristics of any particular signal. This approach provides the prototype for all signal building blocks without supplying details or methods.

### B.3.1 Diagrammatic representation of a BSC

Figure B.1 represents a generalized form of a BSC and shows all possible interfaces and properties.

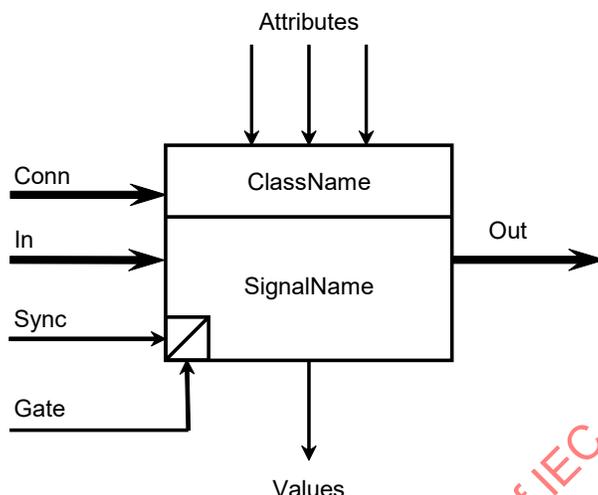


Figure B.1—BSC diagram

In Figure B.1, the following naming conventions are used.

- ClassName is the name of the class that the template represents, e.g., **Constant**.
- SignalName is the name of the specific signal being modeled, e.g., dcSignal.

### B.3.2 BSC attributes

A BSC describes specific signal characteristics described through their attributes. BSCs can be grouped together into models, called signal models, to describe complex signals. A signal model comprises a group of interconnected BSCs that describes one or more signals. BSCs are interconnected through their signal properties of **In**, **Out**, **Sync**, **Gate**, or **Conn**. Control of a signal, defined by such a signal model, is achieved through the use of the **Signal** interface obtained through the **Out** property, commonly called the **Out Signal** interface.

The BSCs describe their behavior in term of their **Signal** properties and attributes. For testing purposes, what happens with signals at the UUT or test interface is of ultimate interest. When a signal is defined by a BSC model and passes through a **Connection** subclass or when a signal is associated with a specific pin name (e.g., pinsIn), it becomes such a signal. It may be described in terms such as physical signal or real signal; however, these items are nothing more than the signals used to perform the testing of the test subject. The corollary of this statement is that the items described as **Signal** are in some way virtual and become a physical entity that can be used for testing only when they are connected to something. The effect of this distinction in the following text is that it describes what would happen if the **Signal** was connected to the test subject as well as what the internal signals need to do. As a convention, the term **Signal** (in boldface) refers to the BSC **Signal** whereas the term signal (in normal typeface) refers to the physical entity that is used to interact with the test subject.

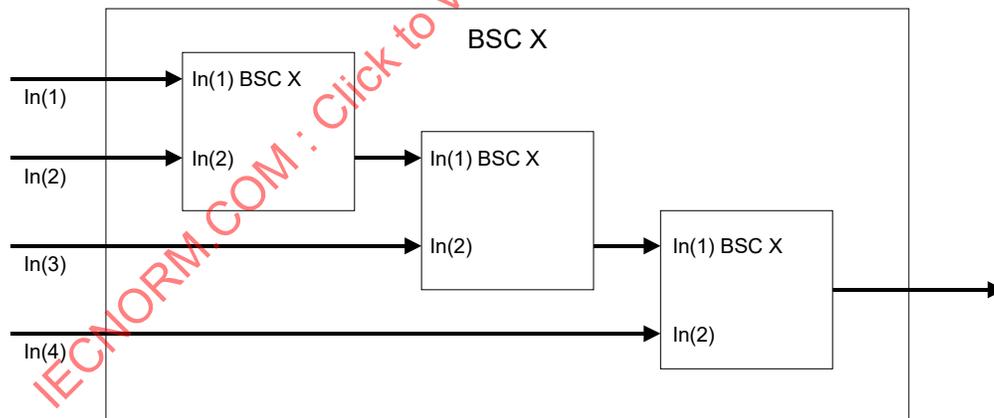
A BSC has the following common properties for **Signal** interfaces:

- a) **Out**—of type **Signal** and represents the signal interface(s) of the BSC, through which the signal can be controlled
- b) **In**—of type reference(s) to **Signal(s)**
- c) **Sync**—of type reference to **Signal**
- d) **Gate**—of type reference to **Signal**
- e) **Conn**—of type reference(s) to **Signal(s)**
- f) **pinsIn**—of type pinString
- g) **pinsOut**—of type pinString
- h) **pinsSync**—of type pinString
- i) **pinsGate**—of type pinString

In addition to these common properties, a BSC may also have attributes, which are used to define the signal characteristics of the output signal, and values, which represent measurable characteristics of any input signals.

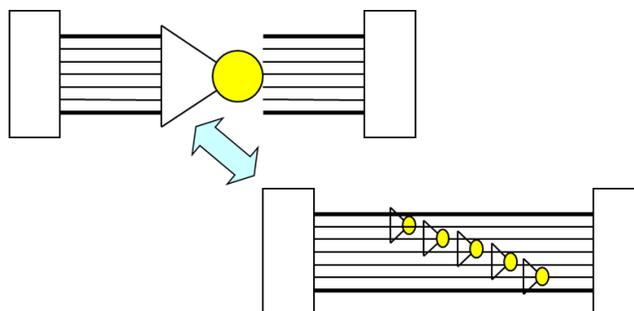
The state of any BSC is affected by both the state of its inputs and by its **Out Signal** interface. This state affects any **Out** signal that the BSC defines, and this scope in turn can affect other connected BSCs. This state control is designed to allow a collection of BSCs in a signal model to be controlled together as a single entity (see Annex C for further description).

The BSC's **In** property consists of signal inputs that the BSC uses. Where multiple **In** signals are defined for BSCs such as **Sum**, **Diff**, **Product**, **Or**, **And**, **Xor**, and **EventedEvent**, the behavior of a BSC with multiple **In** signals shall be the same as multiple, dual-input BSCs chained together with their output being the first input of the next BSC and with their second input being the next input signal as shown in Figure B.2.



**Figure B.2—Multiple inputs semantics**

Signals may comprise one or more signal channels. A BSC's behavior is described in terms of single channel inputs. A BSC's behavior for multichannel input is defined as applying the single channel behavior to each input channel.

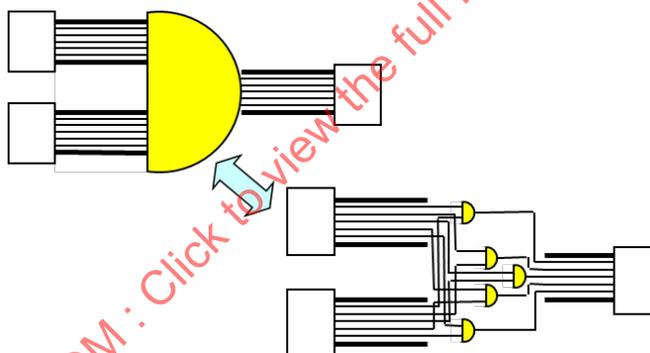


**Figure B.3—Multiple Channel semantics**

Where signals contain multiple inputs, all inputs are coerced into a multiple channel signal by the following rules:

- a) If the input signal has a single channel, this channel is repeated multiple times, equal to the number of channels in the signal with the most channels.
- b) If the input signal has multiple channels, additional tri-state signals channels may be added to the end so that the total number of channels for that signal equal the number of channels in the signal with the most channels.

The action of the BSC is then applied to each corresponding set of single channel inputs.



**Figure B.4—Multiple Channel, Multiple Inputs semantics**

*Examples:*

Attenuating a ThreePhaseWye signal attenuates each channel of the ThreePhaseWye signal.

Summing a three-phase signal (A) with a two-phase signal (B) would result in the following:

- First phase of A and B being summed and output
- Second phase of A and B being summed and output
- Third phase of A being summed with tri-state, i.e., third phase of A is output

Where a BSC is defined for use with a single **In** signal, e.g., **Not**, **Filter**. The use of multiple inputs is considered to be a single input containing multiple channels.

The BSC uses the **Sync** property to initiate its operation. When the **Sync**'s **Signal** first becomes active, the BSC starts its operation. When the **Sync**'s **Signal** subsequently becomes active again, the BSC restarts its operation. As an example for a **Source**, the signal becomes phase-locked on the event; for a **Sensor**, it rearms the measurement. (See subclass descriptions in B.3.5 for more examples.)

The BSC uses the **Gate** property to control its operation. When the **Gate**'s **Signal** is active, the BSC is operating; and when the **Gate**'s **Signal** is not active, the BSC is not operating. As an example for a **Source**, this action would be outputting a signal or not outputting a signal value (tri-state). For a **Sensor**, this action would be taking a measurement or not taking a measurement. For a **Connection**, the action would be becoming connected or disconnected (open circuit). (See subclass descriptions in B.3.5 for more examples.)

A multichannelled signal used as an input to either **Sync** or **Gate** is considered inactive when all its channels are in the No Signal (Z) state.

The **Conn** property allows a user to specify connectivity of BSCs without any implied activation, which is implicit with the **In** property. **Conn** is used for a dynamic model, where the user wants to show connectivity of signals without any implied activation. All BSCs connected solely through the **Conn** property exist in separate time frames and have no implicit synchronization between them. The **Conn** reference is an alternative to using the **In** reference. Unlike a connection made through a **Conn** reference, a BSC's behavior is affected by any **In** connection. All **In** signal's states are monitored, and the **In** signal is controlled by the BSC.

Properties that relate directly to the independent variable take the name appropriate to the usecase, e.g., **riseTime** for a **Ramp** is the time interval taken for the signal to rise. In addition, a property of type **Time** is regarded as the independent variable and as such should consistently match the type of the independent variable as provided by the RefType.

*Example:*

- Step (Voltage, Frequency) startTime="100 kHz" (white noise with a low pass filter of 100 kHz)

### B.3.3 Types of BSC's

The type of a BSC can be one of the following:

- a) **Typeless**—A typeless BSC represents signals, such as events, and can be combined with any other signal of any type.
- b) **Abstract**—An abstract BSC can have a different type from its inputs and can represent either an event stream or an abstract signal. For example, the type of the sensor **Average** may be **Voltage**; however, its values do not represent any real signal, but rather a stream of measured values.
- c) **Generic**—A generic BSC is one that inherits its type from its inputs. For example, the type of the conditioner **Sum** is the type of the signals being summed together. Generic types can generally be used only when their input types are all of the same physical type.
- d) **Physical**—A physical BSC defines the type of a signal, e.g., a voltage signal, defined with respect to time.

The complete type of a physical BSC is expressed by post-fixing the BSC name with its physical type and reference type, separated by commas and set within parentheses ().

*Example:*

- Constant (Voltage, Time)

Where no types are provided in the signal definition, either explicitly or through a unit of a Physical attribute, the default type is Voltage and the default reference type is Time. Where a Physical attribute mapping in Table B.4 exists (such as between Time and Frequency or between Power and Voltage), the type shall be the default type after the mapping has been applied, as illustrated by the following examples:

- `<Sinusoidal amplitude="2 mV" .../>` is a (Voltage, Time) signal
- `<Constant amplitude="1 MOhm" .../>` is a (Resistance, Time) signal
- `<SquareWave amplitude="2 mV" period="1 MHz" .../>` is a (Voltage, Time) signal
- `<Step refType="Frequency" startTime="1 MHz" .../>` is a (Voltage, Frequency) signal
- Constant (Power) is equivalent to Constant (Power, Time).
- Constant is equivalent to Constant(Voltage) and Constant (Voltage, Time).
- Constant amplitude="1 kOhm" is equivalent to type Constant (Resistance, Time)

To express a signal whose reference type is not the default type, the complete type definition shall be used.

*Example:*

- `WaveFormStep (Voltage, Frequency) sampleInterval="1 kHz"`

NOTE—It is possible to express a signal using any BSC with any specified reference type. Care needs to be taken to make sure that any defined signal is valid and realizable in the context of the environment in which it is used.

### B.3.4 BSC attribute default values

Every BSC described in this annex is provided with a default value for each of its attributes. In some cases, the default value provided is meaningful in that it may be a reasonable value in many situations where the BSC is used. For example, a sinusoid has a default value for phase of zero, and this value shall be correct in many instances.

This situation shall not be true for many of the default values. Using the same example of a sinusoid, it may be seen that the default value for frequency is 1 Hz. In most cases, this value is unlikely to be appropriate. The user shall check that the correct value is provided for each use of a BSC.

NOTE—Although the default value is specified, the uncertainty is not. Therefore, although a value is defined, the uncertainty or “how accurate the required signal needs to be” is not defined. In the case of sinusoid phase, any phase shall do.

### B.3.5 SignalFunction subclass descriptions

#### B.3.5.1 Subclass description: Source

A **Source** is used to produce a signal that is based on the value of its attributes. A **Source** shall create an **Out Signal** interface and generally possess at least one attribute; **Source** supports both **Gate** and **Sync** events. A **Source** does not possess values. **Sources** possess, but do not use, any **In** signals.

The type of a **Source** shall be specified; and unless otherwise stated, the default type of a source is voltage with respect to time (**Voltage, Time**).

### B.3.5.2 Subclass description: **Conditioner**

A **Conditioner** combines and conditions one or more input signals into an associated output signal, based on its attribute values. A **Conditioner** shall have at least one **In** signal and shall create an **Out Signal** interface. It supports **Gate** and **Sync** events, may possess attributes, but does not possess values.

The type of a **Conditioner**, unless otherwise stated, is generic.

Where a **Conditioner** contains multiple inputs, the **Conditioner** is operational when the first input signal becomes active (see Annex C) and remains operational while any input signal is active.

### B.3.5.3 Subclass description: **EventFunction**

The **EventFunction** class is the base class for **EventSources** and **EventConditioners**. **EventSources** define events while **EventConditioners** allow event definitions to be modified based on the action of other events and signals. An **EventFunction** shall create an **Out Signal** interface and may possess attributes. It may have **In** signals, shall support **Gate** and **Sync** events, but does not possess values.

The type of an **EventFunction** is typeless unless otherwise stated.

NOTE—The output of an **EventFunction** that uses a **Gate** event has the semantics of a tri-state signal (digital Z value). This has the same effect as an Inactive signal when applied to a **Gate** or **Sync**, or no signal when applied as an input.

Events can originate from either signals or other events.

### B.3.5.4 Subclass description: **Sensor**

A **Sensor** observes the **In** signal and generates values for the specified characteristic of that signal. A **Sensor** shall have an **In** signal and provide a measurement value. It supports **Gate** and **Sync** events and creates an **Out** event signal when the measurement is made or the condition is met.

The type of a **Sensor**, unless otherwise stated, is abstract.

**Sensors** are used to monitor signals and to take measurement values. In all cases, their resultant signals are streamed as abstract signals that contain both event and value information.

### B.3.5.5 Subclasses description: **Control**

A **Control** class allows various signals to be combined or sequenced together and controlled by the signal state. This capability allows a signal model to describe the different signals required in responses to events or digital signals.

The type of a **Control**, unless otherwise stated, is generic.

A **Control** may have **Out** properties and may possess attributes. It may have **In** properties, shall support **Gate** and **Sync** events, but does not possess values.

### B.3.5.6 Subclass description: Digital

A **Digital** is used to produce an analog control signal that represents digital information that is based on the value of its attributes. A **Digital** shall create an **Out Signal** interface and generally possess at least one attribute; **Digital** supports both **Gate** and **Sync** events. A **Digital** does not possess values. **Digital** uses the **In** signals as an external clock overriding any internal digital clock.

The type of a **Digital**, unless otherwise stated, is **Voltage**.

### B.3.5.7 Subclass description: Connection

A **Connection** represents a collection of pins, through which the **In** signals pass or from which any **Out** signals flow through channels. A **Connection** has both **In** and **Out** signals and supports **Gate**, **Sync**, and attributes identifying the names of the pins through which the signals pass. A **Connection** does not possess any values.

An implementer may wish to indicate that a signal is hot-switched or cold-switched by gating a connection BSC before or after the signal appears on its input. However, the implementer should be aware that the standard does not define how the signal is created or connected. A valid implementation may not use switching at all. All that is required is that the signal described is applied to the UUT at the appropriate time.

The type of a **Connection**, unless otherwise stated, is typeless.

The principal purpose of a **Connection** is to identify the names of the pins, such as PL1-1, associated with the channels. A signal is considered an external quantity only where it passes through a **Connection**. Any signal phase information is lost between different **Connections** but maintained within channels of a single **Connection**.

The use of **pinsIn**, **pinsOut**, **pinsSync**, and **pinsGate** is incompatible with any **Connection** and is, therefore, strongly deprecated.

Table B.3 shows an overview of the other derived **Connection** classes and their associated pin attribute names.

**Table B.3—Connection classes**

Connection class	Description	Pin attribute names
TwoWire	Two wire	hi, lo
TwoWireComp	Two-wire complement	true, comp
ThreeWireComp	Three-wire complement	true, comp, lo
SinglePhase	Single phase	a, n
TwoPhase	Two phase	a, b, n
ThreePhaseDelta	Three-phase delta	a, b, c
ThreePhaseWye	Three-phase wye	a, b, c, n
ThreePhaseSynchro	Three-phase synchro	x, y, z
FourWireResolver	Four wire resolver	s1, s2, s3, s4
SynchroResolver	Synchro-resolver	r1, r2, r3, r4
Series	Series	Via
FourWire	Four wire	hi, lo, hiRef, loRef
NonElectrical	Nonelectrical	to, from
DigitalBus	Data, address, or control bus	Pins

### B.3.5.8 Connection subclass attributes

The attribute type for most connection class attributes is <pinString>. This character string may represent one or more UUT pins. A UUT pin name shall not contain a whitespace character; thus the whitespace characters represents a delimiter between multiple UUT pin names.

A single pin attribute containing more than one UUT pin name indicates multiple instances of the connection class where each connection has one set of pin attributes with a single UUT pin for each attribute. This has the effect of simplifying the process of listing connections and pins for any given signal.

## B.4 Physical class

### B.4.1 General

The **Physical** base class (see Table B.1) is used to describe real physical values. It has a value, an associated dimension described by its units, and an uncertainty. Its value may be constrained. All physical types, e.g., time, voltage, are derived from the **Physical** class. These derived **Physical** classes can also offer other interfaces, e.g., a **Period** may be expressed as both **Frequency** and **Time**.

### B.4.2 Properties

A Physical value comprises of a list the following elements:

- Qualifier e.g. *trms*
- Quantities e.g. 10 mV
- Errlmt or uncertainty e.g. *errlmt* +/- 5% *conf* 99.96%
- Ranges of expected values e.g. *range* 0V to 25 V

- e) Resolution e.g. *res* 1 uV
- f) Load e.g. *load* 50Ω

A quantity represents dimensioned and dimensionless values containing a **magnitude** and an associated **unit** with prefix, e.g., "10 mV". Physical uncertainty quantity extends the basic quantity and associates a **qualifier**, uncertainty (**errlmt**s), resolution (**res**), and confidence (**conf**) values with the quantity, e.g., "pk\_pk 10 mV errlmt +- 5% conf 99.96% res 1 uV". Multiple **errlmt** and **res** values can be provided, in which case they are added together, e.g., "errlmt 5% errlmt 1 uV". If multiple confidence values are provided for a quantity, the least confidence value is assumed.

Ranges can be either single values (**range**) or bounded pairs using (**range, to**). The value of a range is defined using a quantity description e.g., "range 1 V to 5 V". The range defines the expected values of the signal attributes and the default characteristics of the physical value when it is within that range, e.g., "range 1 V to 5 V errlmt 5% res 1 uV" implies that a value between 1 V and 5 V shall have an uncertainty and resolution given by "errlmt 5% res 1 uV".

Load (**load**) represents a basic quantity value to be applied when converting between different units, e.g., voltage and power. The load is a single value and is associated with the Physical class.

#### B.4.2.1 Format

The string format of the physical value is described as follows:

```
physical:= [qualifiedQuantity]* [rangingInformation]* [loading]
qualifiedQuantity:= [qualifier] uncertainQuantity
qualifier:= trms|pk_pk|pk|pk_pos|pk_neg|av|inst|inst_max|inst_min
quantity:= <numeric expression> <unit>
uncertainQuantity:= quantity (errorlimit [confidence] [resolution])*
errorlimit:= (errlmt +quantity -quantity) | ([errlmt] (±|+-) quantity)
resolution:= res quantity
confidence:= conf quantity
rangingInformation:=
    (range uncertainQuantity [(to|:) uncertainQuantity]*) |
    ([range] (MAX|MIN) uncertainQuantity )
loading = load quantity
```

where

- All the occurrences of <unit> shall belong to the same quantity (see Table B.4) or, where specified, may be expressed as a **ratio** quantity.
- The <unit> is made up of the <Unit Symbols> and optionally one of any associated <Metric Prefixes> or <Binary Prefixes>. The unit shall not be omitted unless the quantity is dimensionless.
- The **errlmt** shall be expressed as a relative range to the value either as a ratio or as plus and/or minus a fixed amount (e.g., "10 V ± 10 mV" or "10 V ± 0.1%") and represents the uncertainty of the value.
- The **res** property shall be held as an absolute value, identifying the granularity of the value.
- The **conf** property shall be held as an absolute value, as a ratio, and represents the level of confidence associated with the uncertainty (errlmt).
- The **load** property shall be held as an absolute value.

- The **range** property shall be held as absolute values, identifying the range of values that may be used, e.g., “10 V range 11 V to 9 V.” The **value** syntax allows for relative range values to be specified (e.g., “10 V range  $\pm 1$  V” or “10 V range 1%”) where these values are converted to absolute values.
- The asterisk symbol (\*) indicates that there may be zero or more occurrences of the preceding element.

Multiple properties such as **errlmt**, **res**, and **conf** are associated with the current quantity or range. The first occurrence of **errlmt**, **res**, and **conf** defines the default value that shall be used if not explicitly specified.

The syntax allows for a string containing a keyword followed by a value, any number of times. Using a keyword without a value implies the default value for that item should be used. The format defined provides the expected or recommended structure for a physical value. Although the standard defines the meaning of any physical value string, adhering to the specified format provides clarity in the meaning and precludes the possibility of unintentional errors caused by unusual string formats.

A **Physical** class object value does not have to be written with any dimensional quantity. The units are derived from the **unit** property that shall be initialized to the default attribute type.

The **value** property assigns the complete physical value as a whole. Any missing property values imply that the property is not of interest, and any resource selection shall not consider the missing property. Changing property values does not affect other property values, except where there is a need to confirm consistency of dimensions. Specifying an uncertainty of zero implies that any resource selection shall choose the best resource available.

For relative values, a single-ended, positive or negative, uncertainty value is interpreted as a double-ended value unless both positive and negative values are specified. For example, “300 mV errlmt 10 mV” is interpreted as “300 mV  $\pm 10$  mV,” which is the same as “300 mV errlmt +10 mV –10 mV.” Similarly, “300 mV errlmt +10 mV” is also interpreted as “300 mV  $\pm 10$  mV,” which is the same as “300 mV errlmt +10 mV –10 mV” and “300 mV errlmt –10 mV.” If different positive and negative values are required, they shall be specifically stated, even if one of the values is zero, e.g., “300 mV errlmt +10 mV –0 mV.”

The qualified quantity of the physical quantity also carries any attribute **qualifier**, such as **pk\_pk** (e.g., **pk\_pk** 5V).

The **errlmt** and **range** properties refer to the value including the **qualifier**. For example, in a **value** such as “av 65 mV range MAX 100 mV,” the MAX 100 mV refers to the maximum average value, not the maximum instantaneous value.

When assigning physical types to each other, the complete value of the referenced physical type is transferred, e.g., “pk\_pos –8 V errlmt  $\pm 5\%$  res 0.1 uV range –10 V to –5 V.”

#### B.4.2.2 Use of qualifier

A qualifier is defined to be one of the following:

- a) **trms** (true root mean square)
- b) **pk\_pk** (peak-peak)
- c) **pk** (peak)
- d) **pk\_pos** (positive peak)
- e) **pk\_neg** (negative peak)
- f) **av** (average)

- g) **inst** (instantaneous)
- h) **inst\_max** (instantaneous maximum value)
- i) **inst\_min** (instantaneous minimum value)

The way in which the default qualifier may be determined from the signal attribute is illustrated in the following example:

```
<Average nominal="100 V" />
```

This is equivalent to the following:

```
<Average nominal="av 100 V" />  
<Measure nominal="av 100 V" />
```

In a case where the qualifier cannot be determined by the context, it is assumed to be **inst**, as in the following:

```
<amplitude="45 V" />
```

#### B.4.2.3 Use of resolution

The inclusion of a resolution property (**res**) describes the expected granularity of the value, but does not change or constrain the model. The use of the resolution property is intended for, but not restricted to, the simplification of the process of resource selection for signals and measurements, where the resolution value identifies the smallest amount of signal change that can reliably be obtained.

#### B.4.2.4 Use of **errlmt** and level of confidence

The uncertainty property (**errlmt**) describes the permissible uncertainty of the value. For measurements, this is often referred to as expanded uncertainty, which is obtained by the combined standard uncertainty by a coverage factor. The inclusion of a confidence property (**conf**) describes the expected level of confidence associated with the expanded uncertainty of the value, but does not change or constrain the model.<sup>12</sup>

#### B.4.2.5 Use of range

The inclusion of a range does not change or constrain the model. The range property (**range**) describes the range of values that the attribute is expected to take over the life of the signal. For example, the value “10 V range MAX 12 V” indicates that the amplitude is 10 V and that during the life of this signal, the amplitude may vary and is expected to take values up to 12 V.

If the specified magnitude is outside of the given range, then that magnitude takes precedence and is not constrained within the range. It is equivalent to providing two ranges, one at the actual value and one at the given range. For example, the value “12 V range MAX 10 V” indicates that the amplitude is 12 V and that during the life of this signal, the amplitude may vary and may be expected to take other values up to 10 V.

A complete Physical value may include multiple ranges, and these ranges may have different error limits. In this case, the order of **range**, **errlmt**, and **res** properties does have significance. An **errlmt** that precedes any **range** or **res** properties is global in scope and applies to any amplitude value unless otherwise defined. An **errlmt** or **res** following a **range** applies only to amplitude values in that range.

<sup>12</sup> For additional information on level of confidence of expanded uncertainty, refer to NIST Technical Note 1297.

These multiple ranges can be extracted by enumerating through the Physical class **range** property and extracting each entry's Physical components.

*Example:*

```
range 1V to 10V errlmt 0.1% range 15V to 30V errlmt 0.2V
```

This example describes two ranges: 1 V to 10 V with **errlmt**  $\pm 0.1\%$  and 15 V to 30 V with **errlmt**  $\pm 0.2\%$ .

Where overlapping ranges are specified, the smaller uncertainty applies to amplitude values in the overlapping region.

*Example:*

```
errlmt +-1V range 0V to 12V errlmt +-0.1% range MAX 30V errlmt +-0.2V
```

This example indicates that, in the range 0 V to 12 V, the uncertainty is  $\pm 0.1\%$ ; between 12 V and 30 V, the uncertainty is  $\pm 0.2\%$ ; and for all other values (less than 0 V and greater than 30 V), the uncertainty is  $\pm 1\%$ .

Where the overlapping region has the same uncertainty but different **errlmt** values, the range with the lower magnitude is assumed.

*Example:*

```
range 1 V to 10 V errlmt +0.1 V -0.2 V range 5 V to 20 V errlmt +0.2 V  
0.1 V
```

In this example, the range between 0 V and 10 V has an uncertainty of +0.1 V -0.2 V, the range between 10 V and 20 V has an uncertainty of +0.2 V 0:1 V, and no uncertainty is specified for values outside of those ranges.

### B.4.3 Permissible physical types and their units

The physical type can represent a dimensionless value or dimensioned quantity.

Dimensionless physical types can be any computer type whose representation can be held in a VARIANT and whose value can be represented as a string. Where the dimensionless quantity represents a numeric type, e.g. double, byte or Integer, the **Resolution** and **Range** shall be defaulted according to the numeric type. The value of the numeric type shall be coerced to be the closest numeric value. E.g. for scalar types the conversion would be <real>  $\rightarrow$  <numeric type> as a worked example assigning real value (-1.61) to an unsigned byte (VT\_UI1) the closest value is (0). Assigning the same value to a signed byte (VT\_I1) value would be -2 (or 0xFE). The **magnitude** of the physical numeric type shall return a valid value of the numeric type.

Dimensioned values in Table B.4 lists the allowed quantities, physical types, unit symbols, and their units.

**Table B.4—Physical types**

Quantity	Physical type	Unit	SI unit	Unit symbol (See NOTES 1, 2)	Other mappings and notes
Acceleration	Acceleration	meter per second squared	Derived	m/s <sup>2</sup>	—
Admittance	Admittance	—	—	—	See NOTE 3
Amount of information	AmountOfInformation	bit <sup>a</sup>	—	<i>b</i>	See NOTES 4, 5
		byte <sup>a</sup>	—	<i>B</i>	
Amount of substance	AmountOfSubstance	mole	Base	mol	—
Angular acceleration	AngularAcceleration	radian per second squared	Derived	rad/s <sup>2</sup>	—
Angular velocity	AngularSpeed	radian per second	Derived	rad/s	Frequency
Area	Area	square meter	Derived	m <sup>2</sup>	—
Capacitance	Capacitance	farad	Derived	F	—
Concentration	Concentration	mole per cubic meter	Derived	mol/m <sup>3</sup>	—
Current density	CurrentDensity	ampere per square meter	Derived	A/m <sup>2</sup>	—
Dynamic viscosity	DynamicViscosity	pascal second	Derived	Pa•s	—
Electric charge	Charge	coulomb	Derived	C	—
Electric charge density	ElectricChargeDensity	coulomb per cubic meter	Derived	C/m <sup>3</sup>	—
Electric conductance	Conductance	siemens	Derived	S	Resistance See NOTE 3
Electric current	Current	ampère	Base	A	—
Electric field strength	ElectricFieldStrength	volt per meter,	Derived	V/m	See NOTE 6
		newton per coulomb	Derived	N/C	
Electric flux density	ElectricFluxDensity	coulomb per square meter	Derived	C/m <sup>2</sup>	—
Electric potential difference	Voltage	volt	Derived	V	Power (where load is specified by the load property)
Electric resistance	Resistance	ohm	Derived	Ohm	Admittance See NOTES 3, 7
Electromotive force	Voltage	volt	Derived	V	Power (where load is specified by the load property)
Energy	Energy	joule	Derived	J	See NOTE 4
		electronvolt	In use	<i>eV</i>	
Energy density	EnergyDensity	joule per cubic meter	Derived	J/m <sup>3</sup>	—
Entropy	Entropy	joule per kelvin	Derived	J/K	—
Exposure	Exposure	coulomb per kilogram	Derived	C/kg	—
Force	Force	newton	Derived	N	—

(Table continues)

Table B.4—Physical types (continued)

Quantity	Physical type	Unit	SI unit	Unit symbol (See NOTES 1, 2)	Other mappings and notes
Frequency	Frequency	hertz	Derived	Hz	Time
Heat	Heat	joule	Derived	J	—
Heat capacity	HeatCapacity	joule per kelvin	Derived	J/K	—
Heat flux density	HeatFluxDensity	watt per square meter	Derived	W/m <sup>2</sup>	—
Illuminance	Illuminance	lux	Derived	lx	—
Impedance	Impedance	—	—	—	See NOTES 3, 7
Inductance	Inductance	henry	Derived	H	—
Irradiance	Irradiance	watt per square meter	Derived	W/m <sup>2</sup>	—
Kinematic viscosity	KinematicViscosity	square meter per second	Derived	m <sup>2</sup> /s	—
Length	Distance	meter	Base	m	See NOTES 4, 8
		inch	—	<i>in</i>	
		foot	—	<i>ft</i>	
		mile (statute)	—	<i>mi</i>	
		nautical mile	In use	<i>nmi</i>	
Luminance	Luminance	candela per square meter	Derived	cd/m <sup>2</sup>	The use of nit is deprecated
Luminous flux	LuminousFlux	lumen	Derived	lm	—
Luminous intensity	LuminousIntensity	candela	Base	cd	—
Magnetic field strength	MagneticFieldStrength	ampere per meter	Derived	A/m	—
Magnetic flux	MagneticFlux	weber	Derived	Wb	—
Magnetic flux density	MagneticFluxDensity	tesla	Derived	T	—
Mass	Mass	kilogram	kg (Base)	g	See NOTE 9
Mass density	MassDensity	kilogram per square meter	Derived	kg/m <sup>2</sup>	—
Mass Flow	MassFlow	kilogram per second	Derived	kg/s	—
Molar energy	MolarEnergy	joule per mole	Derived	J/mol	—
Molar entropy	MolarEntropy	joule per mole kelvin	Derived	J/(mol•K)	—
Molar heat capacity	MolarHeatCapacity	joule per mole kelvin	Derived	J/(mol•K)	—
Moment of force	MomentOfForce	newton meter	Derived	N•m	—
Moment of inertia	MomentOfInertia	kilogram meter squared	Derived	kg•m <sup>2</sup>	—
Momentum	Momentum	kilogram meter per second	Derived	kg•m/s	—
Permeability	Permeability	henry per meter	Derived	H/m	—
Permittivity	Permittivity	farad per meter	Derived	F/m	—
Plane angle	PlaneAngle	radian	Derived	rad	See NOTES 4, 10
		degree	In use	°, <i>deg</i>	

(Table continues)

**Table B.4—Physical types (continued)**

Quantity	Physical type	Unit	SI unit	Unit symbol (See NOTES 1, 2)	Other mappings and notes
Power	Power	watt	Derived	W	Voltage (is specified by the load property) Also see NOTE 11
		decibel watt <sup>b</sup>	—	<i>dBW, dB(1 W)</i>	
		decibel milliwatt <sup>b</sup>	—	<i>dBm, dB(1 mW)</i>	
Power density	PowerDensity	watt per square meter	Derived	W/m <sup>2</sup>	—
Pressure	Pressure	pascal	Derived	Pa	See NOTES 4, 12
		millibar	In use	<i>mbar</i>	
Radiance	Radiance	watt per square meter steradian	Derived	W/(m <sup>2</sup> •sr)	—
Radiant intensity	RadiantIntensity	watt per steradian	Derived	W/sr	—
Ratio	Ratio	decibel <sup>a</sup>	—	dB	See NOTES 13, 14
		percent <sup>a</sup>	—	%, pc	
		octave	—	octave	
		decade	—	decade	
Reactance	Reactance	ohm	Derived	Ohm	Susceptance Also see NOTES 3, 7
Solid angle	SolidAngle	steradian	Derived	sr	—
Specific energy	SpecificEnergy	joule per kilogram	Derived	J/kg	—
Specific entropy	SpecificEntropy	joule per kilogram kelvin	Derived	J/(kg•K)	—
Specific heat capacity	SpecificHeatCapacity	joule per kilogram kelvin	Derived	J/(kg•K)	—
Specific volume	SpecificVolume	cubic meter per kilogram	Derived	m <sup>3</sup> /kg	—
Speed	Speed	meter per second	Derived	m/s	See NOTES 4, 8
		mile per hour	—	<i>mi/h</i>	
		knot	In use	<i>nmi/h, kn</i>	
		kilometer per hour	—	<i>km/h</i>	
Surface tension	SurfaceTension	newton per meter	Derived	N/m	—
Susceptance	Susceptance	siemens	Derived	S	Reactance Also see NOTE 3
Thermal conductivity	ThermalConductivity	watt per meter kelvin	Derived	W/(m•K)	
Thermodynamic temperature	Temperature	kelvin <sup>b</sup>	Base	K	See NOTES 8, 15
		degree Celsius	Derived	°C, degC	
		degree Fahrenheit	—	°F, degF	
Time	Time	second <sup>c</sup>	Base	s	See NOTE 4
		minute <sup>b</sup>	In use	<i>min</i>	
		hour <sup>b</sup>	In use	<i>h</i>	
		day <sup>b</sup>	In use	<i>d</i>	
		year <sup>b</sup>	In use	<i>y</i>	

(Table continues)

**Table B.4—Physical types (continued)**

Quantity	Physical type	Unit	SI unit	Unit symbol (See NOTES 1, 2)	Other mappings and notes
Volume	Volume	cubic meter	Derived	m <sup>3</sup>	See NOTE 4
		liter	In use	L	
Volume flow	VolumeFlow	liter per second	Derived	L/s	—

NOTE 1—It is preferred practice to leave one space between the numeric value and the unit symbol when defining a value. (See NOTE 7 below).

NOTE 2—The preferred symbol for the power of 2 (i.e., <sup>2</sup>) may be replaced by the symbol 2 where the character set does not allow the <sup>2</sup> symbol, e.g., W/m<sup>2</sup> may be written as W/m2. The symbol for the power of 2 (<sup>2</sup>) is an ASCII character. Similarly, the preferred symbol for the power of 3 (i.e., <sup>3</sup>) may be replaced by the symbol 3. The symbol for the power of 3 (<sup>3</sup>) is not an ASCII character.

NOTE 3—Following electrical engineering convention, the term *resistance* is used to mean the real part of impedance, and the term *reactance* is used to mean the imaginary part of impedance. Similarly, conductance and susceptance are the real and imaginary parts of admittance. Impedance and admittance are not currently supported as complex types. The terms *impedance* and *admittance* are reserved for future use as physical type names.

NOTE 4—For convenience, certain non-SI units are acceptable for use with SI.

NOTE 5—The units for amount of information, b and B, may be used with both metric and binary prefixes. Care should be exercised to make sure that any prefix used is the correct one, e.g., 10 MiB is not the same as 10 MB.

NOTE 6—Certain derived units have special names and symbols. For convenience, derived units are often expressed in terms of other derived units. There are frequently alternative ways to express a derived unit using other derived units (e.g., electric field strength).

NOTE 7—The preferred symbol for the ohm (i.e., Ω) is normally replaced by the term Ohm where the character set does not allow Greek letters. By custom, this convention is normally used in test requirements. The ohm symbol (Ω) is not an ASCII character.

NOTE 8—A limited number of other (non-SI) units have been included. These units were in customary use in some test requirements and have been included for purposes of compatibility.

NOTE 9—For historical reasons, although the SI unit of mass is the kilogram (kg), the SI prefixes are attached to the gram (g).

NOTE 10—When the degree symbol (°) is used for degrees of plane angle, the symbol is normally placed adjacent to the number, e.g., 45°. When a degree of plane angle symbol is required to follow a variable, it is recommended that the symbol deg be used, e.g., bearing deg. Using the degree symbol (°) with a variable may give rise to confusion if it were placed adjacent to the variable name.

NOTE 11—These units of power are equivalent to a level (in decibels) above a reference power of 1 W (in the case of dBW) or 1 mW (in the case of dBm). These equivalents are included to support legacy test requirements. New test requirements should be written with the reference level in parentheses following the ratio unit. For example, 7 dBm should be written as 7 dB (1 mW).

NOTE 12—The use of the bar as a unit of pressure is strongly discouraged. The use of the millibar (mbar) is retained for limited use in meteorology (i.e., for barometric pressure). The value 1 mbar is equal to 100 Pa.

NOTE 13—Ratio is not a quantity. It has been included in this table due to its customary use in test requirements, e.g., as in the case of the specification of amplifier gain. In addition to the unit symbols (dB, %, and pc) shown, ratio values may be dimensionless. The unit symbol pc is included for carrier language implementations that do not support the percent symbol (%).

(Table continues)

**Table B.4—Physical types (continued)**

NOTE 14—Following customary use in electrical engineering, the terms octave and decade are used as units of ratio for the particular cases of 2:1 and 10:1, respectively. The unit to which the ratio refers is determined by the context, e.g., when used with filters, the term *octave* refers to a ratio in frequencies of 2:1.

NOTE 15—The preferred symbols for the degree Celsius (i.e., °C) and the degree Fahrenheit (i.e., °F) may be replaced by the symbols degC and degF where the character set does not allow the degree symbol (°). The degree symbol (°) is an ASCII character.

<sup>a</sup>In this standard, a bit (b) is a basic unit of measurement of information storage in computer science. It represents a binary unit, which can denote a value of 1 or 0. The byte (B) is an ordered, contiguous collection of 8 bits.

<sup>b</sup>These units are not used with the SI prefixes.

<sup>c</sup>This unit is not used with the SI prefixes representing positive powers.

#### B.4.4 Unit prefixes

A unit symbol may be prefixed by one of the metric prefixes from Table B.5.

The  $\mu$  symbol is not supported by some carrier languages. In those cases, it is permissible to use u.

**Table B.5—Metric prefixes**

Prefix	Name	Value	Comments
y	yocto	10 <sup>-24</sup>	—
z	zepto	10 <sup>-21</sup>	—
a	atto	10 <sup>-18</sup>	—
f	femto	10 <sup>-15</sup>	—
p	pico	10 <sup>-12</sup>	—
n	nano	10 <sup>-9</sup>	—
$\mu$ , u	micro	10 <sup>-6</sup>	See NOTE 1
m	milli	10 <sup>-3</sup>	—
c	centi	10 <sup>-2</sup>	See NOTE 2
d	deci	10 <sup>-1</sup>	See NOTE 2
h	hecto	10 <sup>+2</sup>	See NOTE 2
k	kilo	10 <sup>+3</sup>	—
M	mega	10 <sup>+6</sup>	—
G	giga	10 <sup>+9</sup>	—
T	tera	10 <sup>+12</sup>	—
P	peta	10 <sup>+15</sup>	—
E	exa	10 <sup>+18</sup>	—
Z	zetta	10 <sup>+21</sup>	—
Y	yotta	10 <sup>+24</sup>	—

NOTE 1—The preferred symbol for the prefix micro (i.e.,  $\mu$ ) is normally replaced by the symbol u where the character set does not allow Greek letters. By custom, this convention is often used in test requirements.

NOTE 2—By custom, the prefixes for units used in test requirements representing powers of less than 3 (or -3) are not used in test requirements (except for decibel, dB, which is used exclusively).

### B.4.5 Unit prefixes for binary multiples

This standard provides for the use of binary prefixes, which may be used with the symbols for amount of information (b and B). The binary prefixes are listed in Table B.6.

**Table B.6—Binary prefixes**

Prefix	Name	Value	Example
Ki	kibi	$2^{10}$	1 KiB = 1.024 kB
Mi	mebi	$2^{20}$	1 MiB = 1.048576 MB
Gi	gibi	$2^{30}$	1 GiB = 1.073741824 GB
Ti	tebi	$2^{40}$	1 TiB = 1.099511627776 TB
Pi	pebi	$2^{50}$	1 PiB = 1.125899906842624 PB
Ei	exbi	$2^{60}$	1 EiB = 1.152921504606846976 EB
NOTE—These prefixes are defined in IEEE Std 1541™-2002 [B16].			

### B.4.6 Runtime properties

The runtime properties of the Quantity and Physical classes are described as follows:

Quantity Class <Quantity>

- a) **value** <string> contains the full textual description (e.g., “trms 3 V errlmt 100 mV range 1 V to 10 V”).
- b) **magnitude** <real> is the value of the physical type, e.g., 3.0.
- c) **unit** <string> is the read-only unit symbol of the value, e.g., V, Hz, A.
- d) **withUnit (qualifiedunit <string>)** <Quantity> returns an accessor reference for interrogating and setting the value of the quantity with the specified qualifier and unit, e.g., “dBm” or “trms mV”

Physical Class <Physical> derived from <Quantity>

- e) **qualifier** <enum> provides different ways of observing the value and contains one of the following:
  - 1) **trms** (true root mean square)
  - 2) **pk\_pk** (peak-peak)
  - 3) **pk** (peak)
  - 4) **pk\_pos** (positive peak)
  - 5) **pk\_neg** (negative peak)
  - 6) **av** (average)
  - 7) **inst** (instantaneous)
  - 8) **inst\_max** (instantaneous maximum value)
  - 9) **inst\_min** (instantaneous minimum value).

- f) **errlmt** (<enum:UL,LL>) <Quantity>
  - 1) magnitude <real> is the value of the UL or LL error limit, e.g., 0.10.
  - 2) units <enum> is the unit symbol of the UL or LL error limit, e.g., pc.
- g) **range** (<enum:MAX,MIN>) <Quantity>
  - 1) magnitude <real> is the value of the maximum or minimum range, e.g., 10.
  - 2) units <string> is the unit symbol of the maximum or minimum range, e.g., V.
- h) **resolution** <Quantity>
  - 1) magnitude <real> is the value of the resolution, e.g., 25.
  - 2) units <string> is the unit symbol of the resolution, e.g., uV.
- i) **confidence** <Quantity>
  - 1) magnitude <real> is the value of the level of confidence associated with the uncertainty (errlmt), e.g., 95.
  - 2) units <string> is the unit symbol of the level of confidence, e.g., pc or %.
- j) **load\_** <Quantity>
  - 1) magnitude <real> is the value of the load to be used for translation to and from power, e.g., 50.
  - 2) units <string> is the unit symbol of the load, e.g., Ohm.
- k) **name** <String> is the read-only name of the Physical object. The name can be used to extract a specific property from a Signal Function.
- l) **type** <String> is the read-only type of the Physical Type of the quantity.

The **value** property of the **Quantity** and **Physical** class and is internally parsed to complete the **magnitude**, **unit**, **qualifier**, **errlmt**, **resolution**, **confidence**, **load**, and **range** properties. Explicitly changing any **Physical** class properties shall also change the complete **value** property.

As the default unit of a Physical property or attribute is determined by the type of the SignalFunction to which it belongs, **withUnit** is provided to obtain a Physical reference with alternative units. For example, the statement “mySin.amplitude.withUnits("dBm").magnitude = 15.849” provides the magnitude in dBm, whereas the base unit for power is watt (W). The unit of the base property or attribute is not changed.

The **Quantity** and **Physical** class support the default **this** property indexor for getting and setting their numeric value e.g. **amplitude["trms mV"]** returns the numeric value of the property in millivolts trms, and **frequency["kHz"] = 10** sets the frequency value to 10 kHz. The string value of the **this** property indexor has the same format as the **withUnits** qualified unit parameter and represents the equivalent function of **withUnits(qualifiedunit).magnitude**.

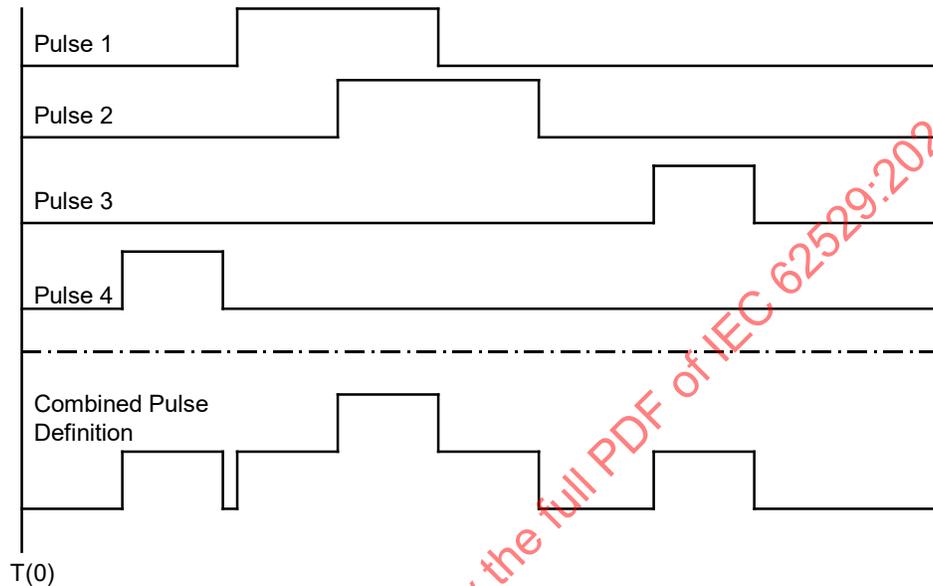
The **Physical** class represents a collection class and supports the following standard enumeration methods and properties **Add**, **Count**, **Item**, **Remove** and **\_NewEnum**. The object returned as an item of the collection supports also the **Physical** interface, but has no collection, and returns a **Count** of zero.

NOTE—The value for the **unit** property never contains the metric prefix, e.g., if the string value “300 mV” is assigned to attribute **value**, then **magnitude** returns SI magnitude 0.3 and **unit** returns V. The default value for the **qualifier** is determined by the associated signal attribute. For example a **Physical** value “pk\_pk 600 mV” is equivalent to **qualifier magnitude unit** “pk\_pk 0.6 V” and equivalent to a **Quantity magnitude** of 0.3 and **unit** V.

## B.5 PulseDefns class

The **PulseDefns** base class (see Table B.1) is used to define BSC signal properties that consist of a set of pulses. The **PulseDefns** is a collection of pulse definitions can be enumerated through.

All pulses defined with the **PulseDefns** class start from the same time frame (i.e., T(0)). All BSCs that use **PulseDefns** superimpose each pulse on top on each other to obtain a complete **PulseDefns**. See Figure B.3.



**Figure B.5—Pulses using PulseDefns**

The complete value of the **PulseDefns** can be defined by using a list of individual **PulseDefn** (as defined by the string format in B.5.1)

### *Runtime Properties*

- **\_NewEnum**<sup>13</sup> returns an object that supports IEnumVARIANT. This property is not visible to users, but allows native languages to iterate through the **PulseDefns**.
- **Item** (index VARIANT) (default indexor) is of type **PulseDefn**. An individual pulse definition is retrieved using the item property, using a numeric index (0-indexed), or using the name of pulse.
- **Count** is of type Long; number of **PulseDefns** in the collection.
- **value** is of type String [i.e., a string list of all the pulses, e.g., “(1ms, 0.5ms, 1) (30us, 2ms, 1)”].

### *Runtime Methods*

- **Add** (name string) is of type **PulseDefn** and adds a **PulseDefn** identified by name. If name is not unique to PulseDefns, the **Add** method retrieves the existing PulseDefn.
- **Remove** (index VARIANT) removes the index **PulseDefn** from the collection. If index does not exist, the **Remove** method returns.

<sup>13</sup> The name “\_NewEnum” is predefined as part of the language interfaces used by IDL[B5]

### B.5.1 PulseDefn class

The **PulseDefn** class defines an individual pulse.

The string format of the complete value of the PulseDefn class is defined as follows:

```
' (<numeric expression> <unit> ', ' <numeric expression> <unit> ', '  
<numeric expression> [pc|%] ' )'
```

- All the occurrences of <unit> shall belong to the same quantity (see Table B.4). In most cases, these shall be units of time.
- The <unit> is made up of the <Unit Symbols> and optionally one of any associated <Metric Prefixes> or <Binary Prefixes>. The unit shall not be omitted unless the quantity is dimensionless.

#### *Runtime Properties*

- **value** is of type String (i.e., a string containing the pulse, e.g., “(30 us, 2 ms, 1)”).
- **start** <Physical> is the point where the pulse occurs. (default 0 s)
- **pulseWidth** <Physical> is the duration of the pulse. (default 0 s)
- **levelFactor** <double> is the multiplication factor that the pulse applies to an input signal. (default 1.0)
- **name** <String> is the name of the pulse. The name can be used to extract a specific PulseDefn from a collection of pulses (PulseDefns).

### B.6 SignalFunction class

All BSCs originate from classes derived from the **SignalFunction** base class (see Table B.1). The **SignalFunction** class is described as a pure virtual class as it can only be used to derive classes rather than to create test objects.

#### *Properties*

- **Out** is of type **Signal**.
- **In** [(at=0)] is of type reference to **Signal**.
- **Sync** is of type reference to **Signal**.
- **Gate** is of type reference to **Signal**.
- **Conn**[(at=0)] is of type reference to **Signal**.
- **pinsIn** is of type pinString.
- **pinsOut** is of type pinString.
- **pinsSync** is of type pinString.
- **pinsGate** is of type pinString.

The **In** and **Conn** properties may both contain multiple signals.

The extensible markup language (XML) defines SignalFunction property **channels** as a list of input channel identifiers (numbers or names), e.g., channels="1 3 2 4". Negative channel numbers can be used to

exclude input channels, e.g., channels="-1 -2". The parameter values 0 or "" implies all signal channels. The special value '-' or '-0' indicates all signal channels in reverse order; and can be mixed with channel exclusions. Channel identifiers are read left to right.

Examples using channels a,b,c,d

- channels = "1 3 2 4" → 'a c b d'
- channels = "-0" → 'd c b a' (all channels reversed)
- channels = "-0 -1" → 'd c b' (all channels reversed 'd c b a' after excluding 'a')
- channels = "-1 1" → 'b c d a' (all channels 'a b c d' excluding 'a' plus 'a')
- channels = "-2 -3" → a d (all channels 'a b c d' after excluding 'b' and excluding 'c')
- channels = "1 -2" → 'a a c d' (channel 'a' plus channels 'a b c d' after excluding 'b')
- channels = "1 -2 -3" → 'a a d' ('a' plus all channels 'a b c d' after excluding 'b' and excluding 'c')
- channels = "1 1 1 1" → a a a a (channel 'a' plus channel 'a' plus channel 'a' plus channel 'a')

The **Out Signal** is an interface to a **Signal** object that the **SignalFunction** creates as part of its function. The behavior between the output **Signal** and the **SignalFunction** is private and depends entirely on the behavior of the **SignalFunction**.

The **pinsIn**, **pinsOut**, **pinsSync**, and **pinsGate** properties provide the means to connect at the level of the signal without reference to a separate connection class. Although these properties are inherited by all STD classes and subclasses, their use with connection classes should be avoided (see B.3.5.7).

Using the **pinsIn**, **pinsOut**, **pinsSync**, and **pinsGate** properties to connect to a signal is restricted to the two-wire (hi, lo) style of connection, e.g., they cannot be used where true, comp, phase, or neutral pin types are required. Refer to B.6.7 for a definition of pinString.

Subclauses B.6.1 through B.6.7 describe all of the BSCs available to the standard covering **Sources**, **Conditioners**, **Sensors**, **EventFunctions**, **Control**, **Digital**, and **Connections**. The description generally takes the form of describing the signal in terms of generating a signal. However, a signal description in the form of a signal model can equally be used as a means of measuring a signal characteristic attribute or signal simulation. The signal descriptions defined by this standard describe the signal, but do not define how the signals are to be used.

Where the following descriptions use the **SignalFunction** template, their type is shown as **Physical**. The BSC can describe any signal based on any physical type listed in Table B.4. Each BSC defines an interface plus the names of the objects that support those interfaces, e.g., **Constant**(Voltage,Time) supports the **Constant** type interface (see Annex D).

NOTE—All BSC behavior to **Gate** and **Sync** events is as described in B.3 unless explicitly changed in the following text. In general, this behavior is consistent across all BSCs and avoids special meanings being invented for these terms.

See C.2.3.3 for description of runtime properties.

### B.6.1 Source ::SignalFunction

- a) *Definition*—**Sources** are the only BSCs from where signals can originate.
- b) *Attributes*—Not applicable

- c) *Description*—A **Source** produces a signal based on its attribute values. The signal is continuously provided, unless gated off, and can be restarted by use of the **Sync** event. Once started, a **Source** generates the signal until explicitly turned off through the **Out Signal** interface.

### B.6.1.1 NonPeriodic ::Source

- a) *Definition*—**NonPeriodic** signals have no implicit period. They identify signals that do not repeat themselves.
- b) *Attributes*—Not applicable
- c) *Description*—**NonPeriodic Sources** are continuous signals, where their final value may be constant, but they do not have a period. A **NonPeriodic Source** represents a transient or single transition, which is repeated on the arrival of each **Sync** event.

#### B.6.1.1.1 Constant<type: Physical> ::NonPeriodic

- a) *Definition*—A **Constant** signal retains its given level. See Figure B.6.
- b) *Attributes*  
**amplitude** <Physical>—the level of the signal (default = 0)
- c) *Description*

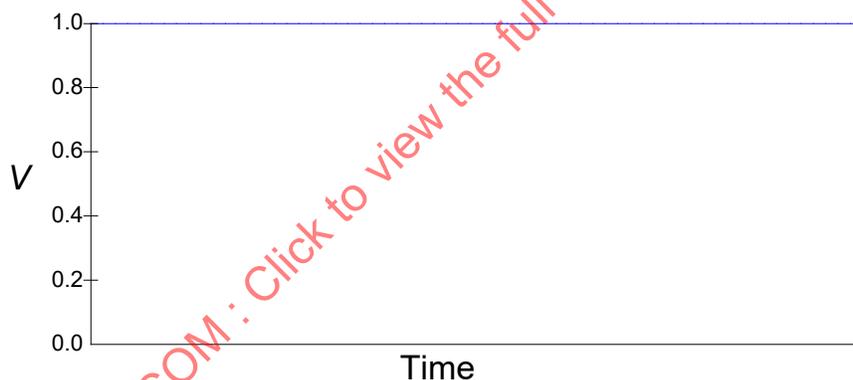


Figure B.6—Constant (amplitude=1 V)

#### B.6.1.1.2 Step<type: Physical> ::NonPeriodic

- a) *Definition*—A **Step** signal makes a transition from zero to a given level. See Figure B.7.
- b) *Attributes*  
**amplitude** <Physical>—final value of Step signal (default = 0)  
**startTime** <Time>—definition of when the step transition starts (default = 0.5 s)
- c) *Description*—A **Step** signal has two properties: the start time of the transition and the final amplitude level. The step transition is regarded as instantaneous. Before start time, the value is 0; after start time, the value is the amplitude.

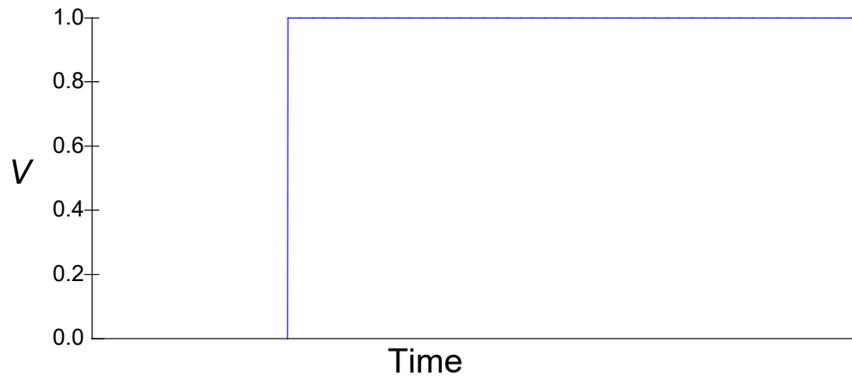


Figure B.7—Step (amplitude=1 V, startTime=0.5 s)

**B.6.1.1.3 SingleTrapezoid<type: Physical> ::NonPeriodic**

- a) *Definition*—A **SingleTrapezoid** is a **NonPeriodic** signal defined by the geometric trapezoid shape. See Figure B.8.
- b) *Attributes*
  - amplitude** <Physical>—value of pulse amplitude (default = 0)
  - startTime** <Time>—time at which trapezoid starts relative to its initialization (default = 0 s)
  - riseTime** <Time>—time taken to reach amplitude (default = 0.25 s)
  - pulseWidth** <Time>—time that trapezoid is stable at amplitude (default = 0.5 s)
  - fallTime** <Time>—time taken to fall back to quiescent state (default = 0.25 s)
- c) *Description*—A **SingleTrapezoid** may have zero values for any of its properties. The trapezoid is regarded as its geometric shape.

Its properties are defined by its amplitude and the times that bound each signal segment of start time, rise time, pulse width, and fall time.

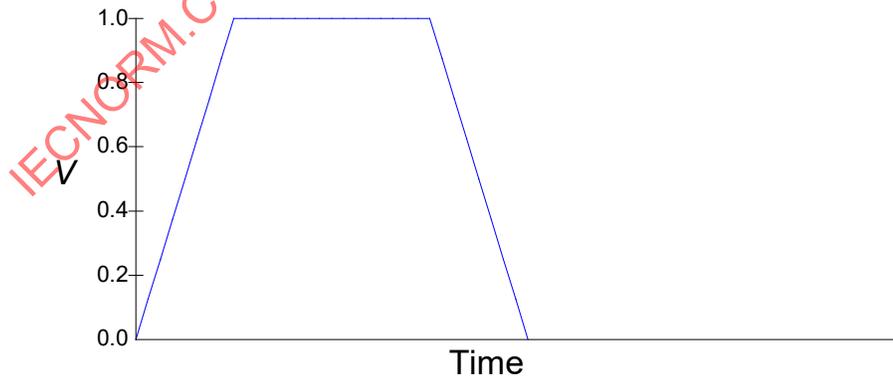


Figure B.8—SingleTrapezoid (amplitude=1 V)

#### B.6.1.1.4 Noise ::NonPeriodic

a) *Definition*—Noise may be considered as unwanted disturbances superimposed upon a useful signal, which tend to obscure the signal’s information content. Noise may be genuinely random (as in white noise) or may be pseudorandom. Noise occurs across a range of frequencies and can be characterized by amplitude; it may take the form of a Sensor or Source signal. Pseudorandom noise is only of interest as a Source signal. In addition to amplitude, it also allows a frequency and an optional seed to be defined. See Figure B.9.

b) *Attributes*

**amplitude** <Physical>—the peak noise amplitude (default = 0)

**seed** <int>—used for pseudorandom noise (default = 0)

**frequency** <Frequency>—upper bound frequency bandwidth for transient disturbances (default = 50 Hz)

c) *Description*—**Noise** is the term most frequently applied to the limiting case where the number of transient disturbances per unit time is large.

Noise has amplitude, frequency, and seed as parameters, of the type of the dependent variable, and has a recurring pattern as determined by the generating algorithms and the seed. These parameters define the mean frequency and amplitude of the transient disturbances.

The pseudorandom effect applies only to multiple sequences of the same implementation, and different implementations shall give different pseudorandom values. A seed value of 0 implies true random noise. Therefore, it could be generated from a thermal noise generator and is not necessarily repeatable.

The frequency attribute provides the bandwidth of the frequency spectrum of the noise. The use of zero (0 Hz) implies noise is independent of frequencies, i.e., white noise.

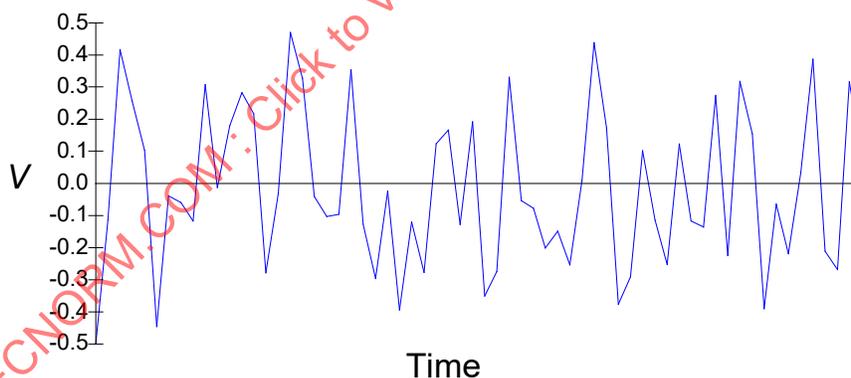


Figure B.9—Noise (seed=1, amplitude=0.5 V, frequency=50 Hz)

#### B.6.1.1.5 SingleRamp<type: Physical> ::NonPeriodic

a) *Definition*—A **SingleRamp** represents a linear transition from 0 to the defined amplitude level during a defined time period. See Figure B.10.

b) *Attributes*

**amplitude** <Physical>—final value of ramp signal (default = 0)

**riseTime** <Time>—time for signal to reach final value (default = 1 s)

**startTime** <Time>—defines when the step transition starts (default = 0 s)

- c) *Description*—A **SingleRamp** takes the form of a linear signal with the transition time defining the event window of that signal. The slope of the linear signal is defined by the difference between the amplitudes divided by the transition time. In a high-to-low transition, the gradient is negative; and in a low-to-high transition, the gradient is positive.

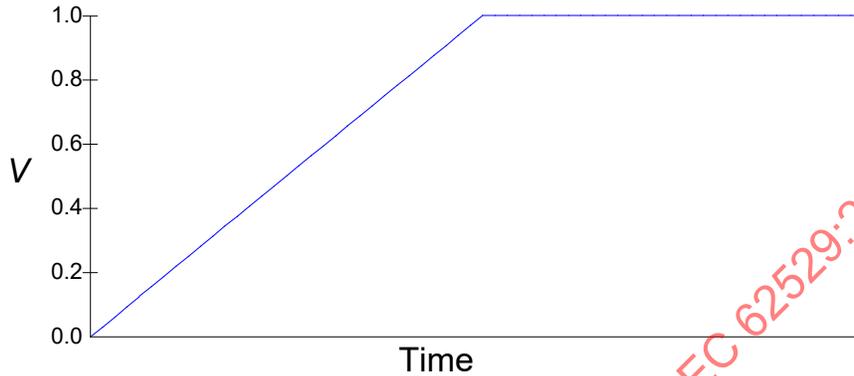


Figure B.10—SingleRamp (amplitude=1 V, riseTime=1 s)

### B.6.1.2 Periodic ::Source

- a) *Definition*—Periodic signals are signals in which the amplitude value (a dependent variable) changes as a periodic function of time (an independent variable). These signals have an implicit period and frequency.
- b) *Attributes*—Not applicable
- c) *Description*—The behavior of any **Periodic** signal defined with a period is that of a single signal that is being synchronized with a clock event equivalent to the period. Therefore, each signal restarts from its initial start time at the start of each period. **Periodic** signals are equivalent to synchronized **NonPeriodic** signals.

#### B.6.1.2.1 Sinusoid<type: Physical> ::Periodic

- a) *Definition*—A **Sinusoid** is a signal where the amplitude of the dependent variable is given by the formula in Equation (B.1):

$$e = A \sin(\omega t + \varphi) \quad (\text{B.1})$$

where

$A$  is the amplitude  
 $\omega$  is  $2\pi \times$  frequency  
 $\varphi$  is the initial phase angle

- b) *Attributes*

**amplitude** <Physical>—amplitude (default = 0)  
**frequency** <Frequency>—frequency (default = 1 Hz)  
**phase** <PlaneAngle>—initial phase angle (default = 0 rad)

- c) *Description*—**Sinusoid** has amplitude, frequency, and phase as parameters. The amplitude has the type of the dependent variable, the frequency is of type Frequency, and the initial phase angle is a PlaneAngle. See Figure B.11.

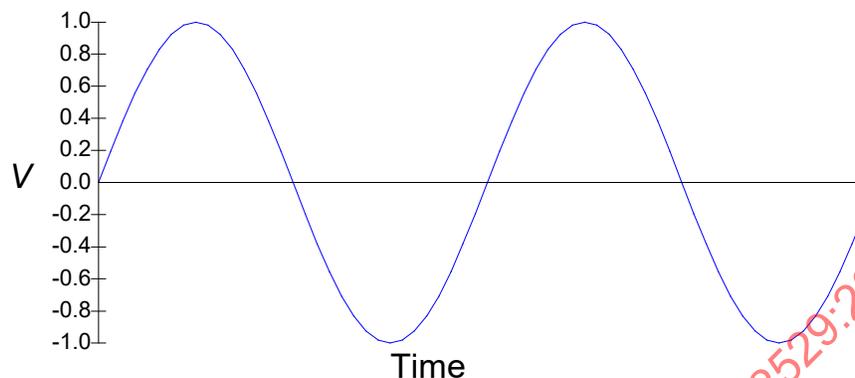


Figure B.11—Sinusoid (amplitude=1 V, frequency=30 Hz)

#### B.6.1.2.2 Trapezoid<type: Physical> ::Periodic

- a) *Definition*—A **Trapezoid** is a **Periodic** signal that sequentially repeats the **SingleTrapezoid**. The period is defined by the duration of the **SingleTrapezoid**. All event times are referenced to local time, which is reset at the start of each pulse. See Figure B.12.
- b) *Attributes*
- |                             |   |                    |
|-----------------------------|---|--------------------|
| <b>amplitude</b> <Physical> | —value of pulse amplitude                   | (default = 0)      |
| <b>period</b> <Time>        | —time in which the signal repeats itself    | (default = 1 s)    |
| <b>riseTime</b> <Time>      | —time taken to reach amplitude              | (default = 0.25 s) |
| <b>pulseWidth</b> <Time>    | —time that Trapezoid is stable at amplitude | (default = 0.5 s)  |
| <b>fallTime</b> <Time>      | —time taken to fall back to quiescent state | (default = 0.25 s) |
- c) *Description*—A **Trapezoid** signal represents the trapezoidal geometric shape. The continuous signal starts on the rising edge, and the trapezoid is repeated every period even if the trapezoid has not been completed.

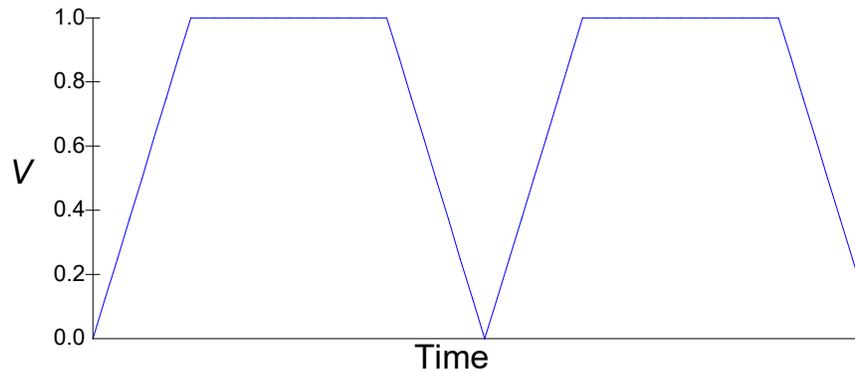


Figure B.12—Trapezoid (amplitude=1 V, pulseWidth=0.5 s)

#### B.6.1.2.3 Ramp<type: Physical> ::Periodic

- a) *Definition*—A **Ramp** signal is a **Periodic** signal whose instantaneous value follows a linear transition from zero to the defined amplitude during the **riseTime** and back to zero during the remainder of the period. In the case that the period is less than the **riseTime**, the linear transition from zero shall stop when the period is reached. See Figure B.13.
- b) *Attributes*
- |                             |                                      |                 |
|-----------------------------|--------------------------------------|-----------------|
| <b>amplitude</b> <Physical> | —final level of the signal           | (default = 0)   |
| <b>period</b> <Time>        | —time in which signal repeats itself | (default = 1 s) |
| <b>riseTime</b> <Time>      | —rise time of Ramp signal            | (default = 1 s) |
- c) *Description*

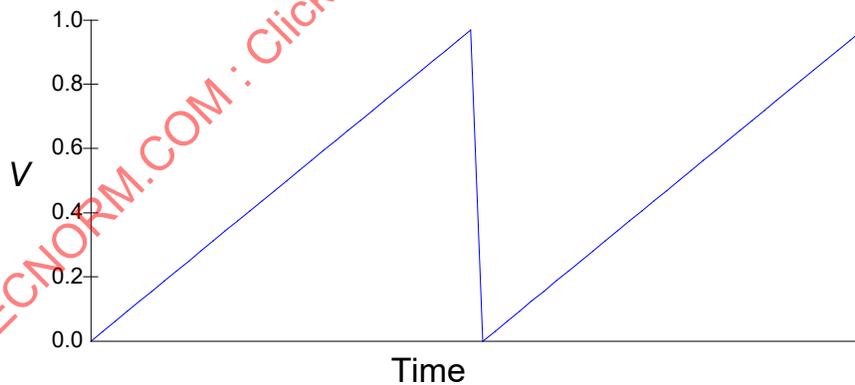


Figure B.13—Ramp (amplitude=1 V)

#### B.6.1.2.4 Triangle<type: Physical> ::Periodic

- a) *Definition*—A **Triangle** signal is a **Periodic** signal whose instantaneous value varies linearly and equally about 0. Duty cycle is a ratio between the time for which it increases to its positive value and the time for which it decreases to its negative value. Its parameters are defined by its amplitude, period, and duty cycle. See Figure B.14.

b) *Attributes*

**amplitude** <Physical>—maximum amplitude level of the signal (default = 0)

**period** <Time>—time in which signal repeats itself (default = 1 s)

**dutyCycle** <Ratio>—ratio between time taken to increase from its minimum to its maximum value and the time for one period (default = 50%)

c) *Description*—**Triangle** has **amplitude** and **period** as parameters. The **amplitude** has the type of the dependent variable; the **period** is of type Time.

NOTE—The value of the attribute **dutyCycle** is a ratio, which can include values outside of the range of 0% to 100% (i.e., 0 to 1). The use of values outside of the range of 0% to 100% may have unintended effects upon the signal.

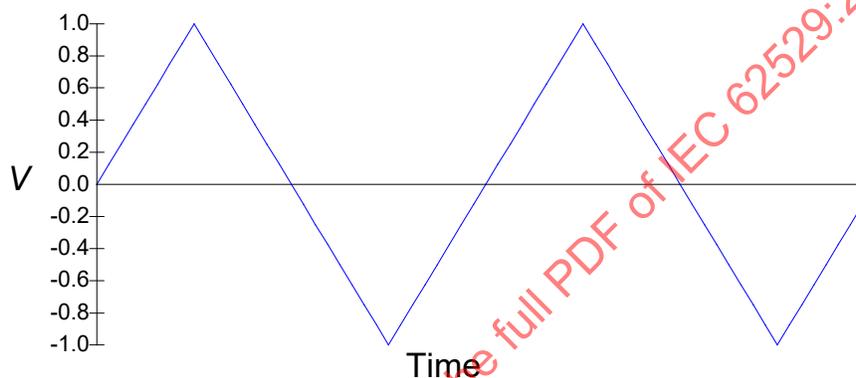


Figure B.14—Triangle (amplitude=1 V)

**B.6.1.2.5 SquareWave**<type: Physical>:Periodic

a) *Definition*—A **SquareWave** is a **Periodic** signal whose amplitude (a dependent variable) alternately assumes one of two fixed values of amplitude. The amplitudes are equal to about 0, which is the reference base line. Duty cycle is a ratio between the time for which it remains at its positive value and the time for which it remains at its negative value. Its parameters are defined by its amplitude, period, and duty cycle. See Figure B.15.

b) *Attributes*

**amplitude** <Physical>—amplitude of signal (default = 0)

**period** <Time>—period of signal (default = 1 s)

**dutyCycle** <Ratio>— ratio between time at its positive value and the time for one period (default = 50%)

c) *Description*—**SquareWave** has **amplitude** and **period** as parameters. The amplitude has the type of the dependent variable; the period is of type Time.

NOTE—The value of the attribute **dutyCycle** is a ratio, which can include values outside of the range of 0% to 100% (i.e., 0 to 1). The use of values outside of the range of 0% to 100% may have unintended effects upon the signal.

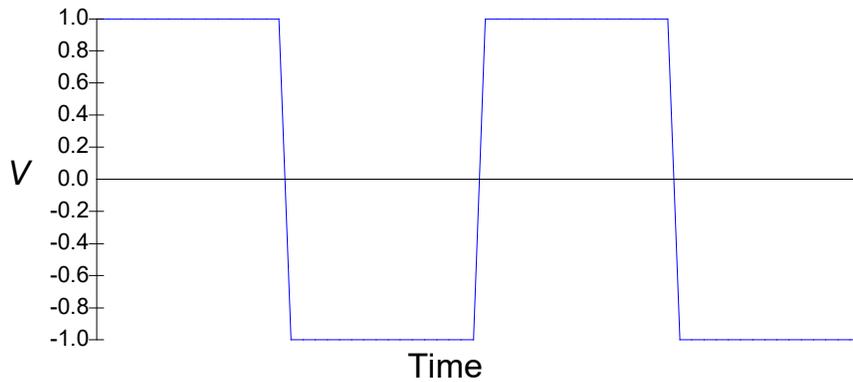


Figure B.15—SquareWave (amplitude=1 V)

#### B.6.1.2.6 WaveformRamp<type: Physical> ::Periodic

- a) *Definition*—A **WaveformRamp** is defined by a sampling interval and a list of values. The **WaveformRamp** cycles through those values sequentially and infinitely, starting from 0. The width of each window is the same, and each window consists of a **Ramp** signal. See Figure B.16.

- b) *Attributes*

**amplitude** <Physical>—amplitude of the output signal where the level factor (in points) is 1  
(default = 1)

**period** <Time>—the time between each sequence (default = 1 s)

**samplingInterval** <Time>—the time between successive (points) outputs (default = 0 s)

**points** <list\_double>—level factor of each waveform sample (default = empty)

If the attribute **samplingInterval** is 0, the complete waveform described by the points is repeated per **period**. Otherwise, the **period** is calculated as (**sampleInterval** × number of points). Assigning a nonzero **period** value sets **samplingInterval** to 0.

- c) *Description*—**WaveformRamp** takes the form of a sequence of linear signals with the sampling interval defining the event window. The slope of the linear signal is defined by the difference between the previous point and the current point divided by the sampling interval. In a high-to-low transition, the slope is negative; and in a low-to-high transition, the slope is positive. The offset is defined by the previous point. **WaveformRamp** cycles through the points sequentially and continuously.

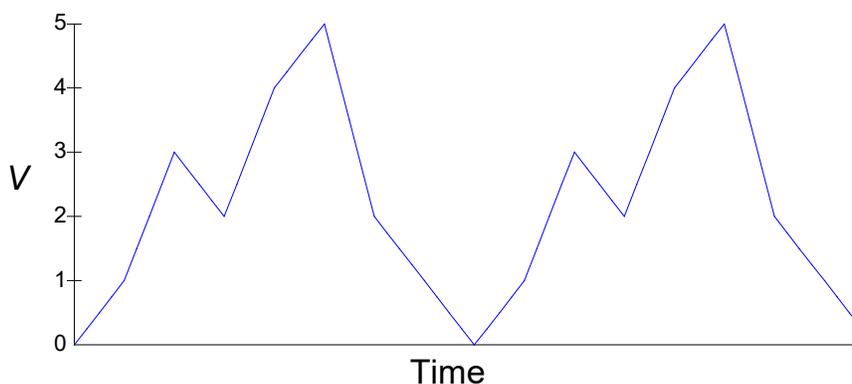


Figure B.16—WaveformRamp (points=[0, 1, 3, 2, 4, 5, 2, 1], period=1 s)

### B.6.1.2.7 WaveformStep<type: Physical> ::Periodic

a) *Definition*—A **WaveformStep** is defined by a sampling interval and a list of values. The **WaveformStep** cycles through those values sequentially and infinitely, starting from 0. The width of each window is the same, and each window consists of a line segment (i.e., a step signal). See Figure B.17.

b) *Attributes*

**amplitude** <Physical>—amplitude of the output signal where the level factor (in points) is 1 (default = 1)

**period** <Time>—the time between each sequence (default = 1 s)

**samplingInterval** <Time>—the time between successive (points) outputs (default = 0 s)

**points** <list\_double>—level factor of each waveform sample (default = empty)

If the attribute **samplingInterval** is 0, the complete waveform described by the points is repeated per **period**. Otherwise, the **period** is calculated as (**sampleInterval** × number of points). Assigning a nonzero **period** value sets **samplingInterval** to 0.

c) *Description*—**WaveformStep** takes the form of a sequence of constant signals. The level of the constant signal is defined by the points, and a transition in level occurs at each increment of the sampling interval. **WaveformStep** cycles through the points sequentially and continuously.

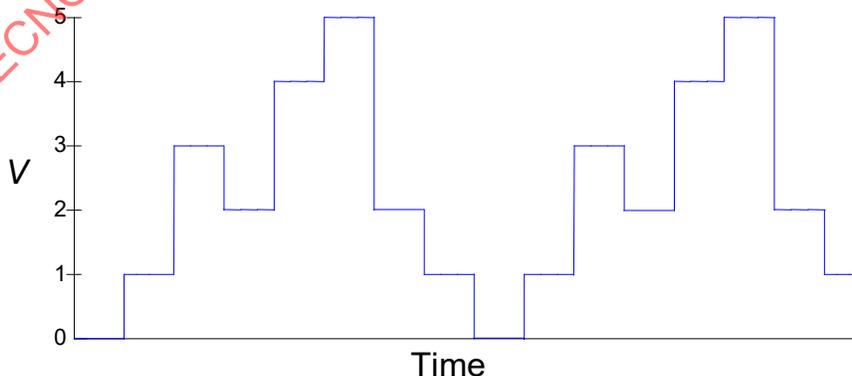


Figure B.17—WaveformStep (points=[0, 1, 3, 2, 4, 5, 2, 1], period=1 s)

**B.6.2 Conditioner ::SignalFunction**

- a) *Definition*—**Conditioners** take one or more signal inputs and transform them to other signals or, as do Product or Sum, take multiple input signals and operate on these to produce a single signal output.
- b) *Attributes*—Not applicable
- c) *Description*—**Conditioners** act on signals, e.g., **Sources**, but not on events (e.g., **EventFunctions**). **Conditioners** can be restarted using the **Sync** property and/or when the input signals become active. Restarting a BSC is where its behavior reverts back to when it first started, i.e., time=0

**B.6.2.1 Filter ::Conditioner**

- a) *Definition*—A Filter is a **Conditioner** that passes a defined set of frequencies from an input signal to produce an output signal.
- b) *Attributes*—Not applicable
- c) *Description*—Filters have attributes that allow a basic filter shape to be defined. If no values are provided for gain, rolloff, passBandRipple, and stopbandRipple, the Filters default to pure filters with instantaneous frequency cutoff across their bandwidths.

**B.6.2.1.1 BandPass ::Filter**

- a) *Definition*—A **BandPass** filter passes frequencies within the pass band from an input signal and filters out all frequencies outside of the band. The **BandPass** filter is a symmetrical filter in which the **rolloff** values for the highpass and lowpass transition slopes are equal and opposite. The **BandPass** filter illustrated in Figure B.18 represents a perfect bandpass filter.
- b) *Attributes*

**centerFrequency** <Frequency>—center frequency of the filter’s band (default = 0 Hz)

**frequencyBand** <Frequency>—bandwidth of filter; zero implies narrowest band (default = 0 Hz)

**gain** <Ratio>—ratio defining the scaling factor for the signal in the pass band (default = 0 dB)

**rolloff** <Ratio>—the rate at which the amplitude of the output signal shall alter over frequency (default = 0)

**passBandRipple** <Ratio>—the maximum allowable variation in the amplitude of the passband signal (default = 0 dB)

**stopBandRipple** <Ratio>—the maximum allowable variation in the amplitude of the stopband signal (default = 0 dB)

The values for **passBandRipple** and **stopBandRipple** are given as a proportion of the value of the **In** signal after any change imparted by the **gain** attribute.

The value for **rolloff** may be given with units of dB/decade or dB/octave in line with customary practice. A value of 0 indicates a pure filter, as defined below, in which case the values for **passBandRipple** and **stopBandRipple** have no meaning.

- c) *Description*—In the case of the pure filter, the output ( $e_{out}$ ) is given as defined in Equation (B.2) and Equation (B.3).

$$e_{out} = e_{in} \quad \text{for } f \geq (f_c - f_{bw}/2) \text{ and } f \leq (f_c + f_{bw}/2) \quad (\text{B.2})$$

$$e_{out} = 0 \quad \text{for } f < (f_c - f_{bw}/2) \text{ and } f > (f_c + f_{bw}/2) \quad (\text{B.3})$$

where

- $e_{in}$  is the input
- $f$  is the frequency of the input signal
- $f_c$  is the center frequency
- $f_{bw}$  is the absolute value of the frequency band

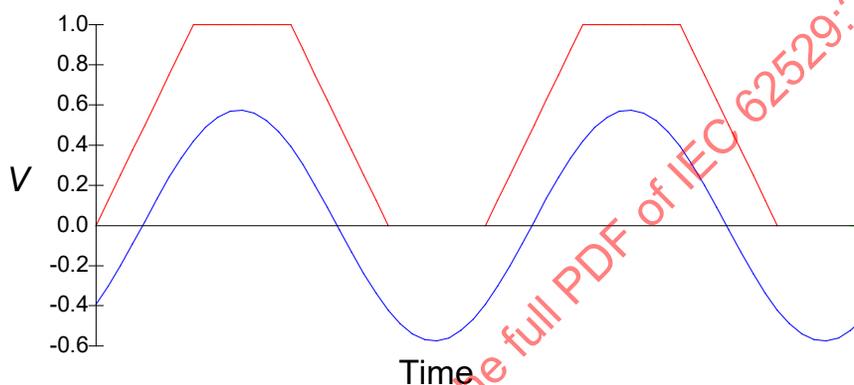


Figure B.18—BandPass (response of Filter to TrapezoidVoltage)

#### B.6.2.1.2 LowPass ::Filter

- a) *Definition*—The **LowPass** filter suppresses all frequencies above the cutoff frequency. Frequencies below the cutoff frequency are passed to the output signal. The **LowPass** filter illustrated in Figure B.19 represents a perfect step filter.

- b) *Attributes*

**cutoff** <Frequency>—cutoff frequency of the filter; zero implies dc only passed (default = 0 Hz)

**gain** <Ratio>—ratio defining the scaling factor for the signal in the pass band (default = 0 dB)

**rollOff** <Ratio>—the rate at which the amplitude of the output signal shall alter over frequency (default = 0)

**passBandRipple** <Ratio>—the maximum allowable variation in the amplitude of the passband signal (default = 0 dB)

**stopBandRipple** <Ratio>—the maximum allowable variation in the amplitude of the stopband signal (default = 0 dB)

The values for **passBandRipple** and **stopBandRipple** are given as a proportion of the value of the **In** signal after any change imparted by the gain attribute.

The value for **rollOff** may be given with units of dB/decade or dB/octave in line with customary practice. Only a single value for magnitude is required. A value of 0 indicates a pure filter, as defined below, in which case the values for **passBandRipple** and **stopBandRipple** have no meaning.

- c) *Description*—In the case of the pure filter, the output ( $e_{out}$ ) is given as defined in Equation (B.4) and Equation (B.5).

$$e_{out} = e_{in} \quad \text{for } f \leq f_c \quad (\text{B.4})$$

$$e_{out} = 0 \quad \text{for } f > f_c \quad (\text{B.5})$$

where

$e_{in}$  is the input  
 $f$  is the frequency of the input signal  
 $f_c$  is the absolute value of the cutoff frequency

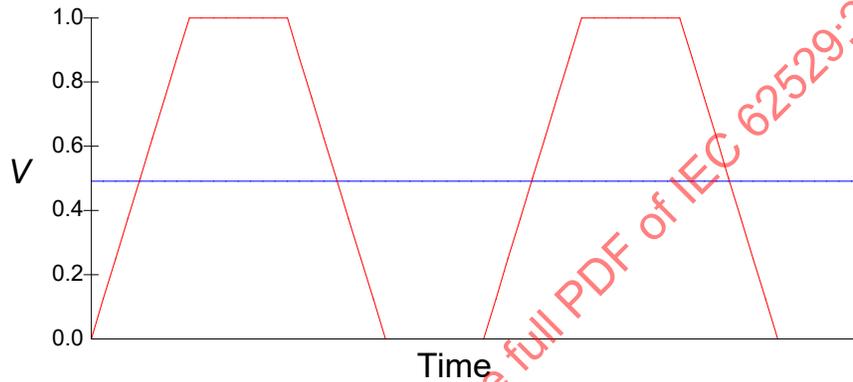


Figure B.19—LowPass (response of Filter to TrapezoidVoltage)

### B.6.2.1.3 HighPass ::Filter

- a) *Definition*—The **HighPass** filter suppresses all frequencies below the cutoff frequency. All frequencies above and including the cutoff frequency are passed (with equal gain) to the output signal. The **HighPass** filter illustrated in Figure B.20 represents a perfect step filter. See Figure B.20.

- b) *Attributes*

**cutoff** <Frequency>—start frequency of the filter; zero implies ac coupled (default = 0 Hz)

**gain** <Ratio>—ratio defining the scaling factor for the signal in the pass band (default = 0 dB)

**rollOff** <Ratio>—the rate at which the amplitude of the output signal shall alter over frequency (default = 0)

**passBandRipple** <Ratio>—the maximum allowable variation in the amplitude of the passband signal (default = 0 dB)

**stopBandRipple** <Ratio>—the maximum allowable variation in the amplitude of the stopband signal (default = 0 dB)

The values for **passBandRipple** and **stopBandRipple** are given as a proportion of the value of the **In** signal after any change imparted by the gain attribute.

The value for **rollOff** may be given with units of dB/decade or dB/octave in line with customary practice. Only a single value for magnitude is required. A value of 0 indicates a pure filter, as defined below, in which case the values for **passBandRipple** and **stopBandRipple** have no meaning.

- c) *Description*—In the case of the pure filter, the output ( $e_{out}$ ) is given as defined in Equation (B.6) and Equation (B.7).

$$e_{out} = e_{in} \quad \text{for } f > f_c \quad (\text{B.6})$$

$$e_{out} = 0 \quad \text{for } f \leq f_c \quad (\text{B.7})$$

where

$e_{in}$  is the input  
 $f$  is the frequency of the input signal  
 $f_c$  is the absolute value of the cutoff frequency

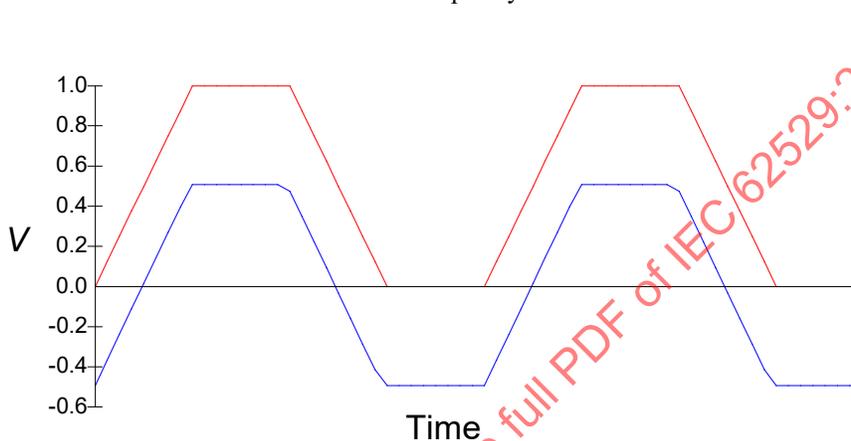


Figure B.20—HighPass (response of Filter to TrapezoidVoltage)

#### B.6.2.1.4 Notch ::Filter

- a) *Definition*—A **Notch** filter blocks frequencies within the pass band from an input signal and passes all frequencies outside of the band. The **Notch** is a symmetrical filter in which the **rollOff** values for the highpass and lowpass transition slopes are equal and opposite. The **Notch** filter illustrated in Figure B.21 represents a perfect notch (bandstop) filter.
- b) *Attributes*

**centerFrequency** <Frequency>—center frequency of the Filter’s notch (default = 0 Hz)

**frequencyBand** <Frequency>—stop band of Filter; zero implies minimum band (default = 0 Hz)

**gain** <Ratio>—ratio defining the scaling factor for the signal in the pass band (default = 0 dB)

**rollOff** <Ratio>—the rate at which the amplitude of the output signal shall alter over frequency (default = 0)

**passBandRipple** <Ratio>—the maximum allowable variation in the amplitude of the passband signal (default = 0 dB)

**stopBandRipple** <Ratio>—the maximum allowable variation in the amplitude of the stopband signal (default = 0 dB)

The values for **passBandRipple** and **stopBandRipple** are given as a proportion of the value of the **In** signal after any change imparted by the gain attribute.

The value for **rollOff** may be given with units of dB/decade or dB/octave in line with customary practice. Only a single value for magnitude is required. A value of 0 indicates a pure filter, as defined below, in which case the values for **passBandRipple** and **stopBandRipple** have no meaning.

- c) *Description*—In the case of the pure filter, the output ( $e_{out}$ ) is given as defined in Equation (B.8) and Equation (B.9).

$$e_{out} = e_{in} \quad \text{for } f \leq (f_c - f_{bw}/2) \text{ and } f \geq (f_c + f_{bw}/2) \quad (\text{B.8})$$

$$e_{out} = 0 \quad \text{for } f > (f_c - f_{bw}/2) \text{ and } f < (f_c + f_{bw}/2) \quad (\text{B.9})$$

where

- $e_{in}$  is the input
- $f$  is the frequency of the input signal
- $f_c$  is the center frequency
- $f_{bw}$  is the absolute value of the frequency band

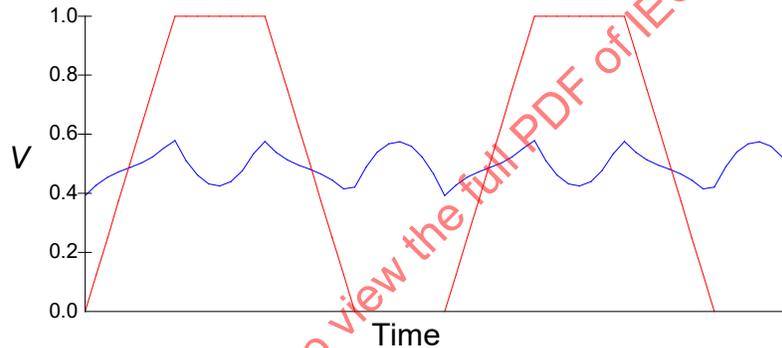


Figure B.21—Notch (response of Filter to TrapezoidVoltage)

### B.6.2.2 Combiner ::Conditioner

- a) *Definition*—**Combiners** take multiple input signals and combine them into a single output signal.
- b) *Attributes*—Not applicable
- c) *Description*

#### B.6.2.2.1 Sum ::Combiner

- a) *Definition*—**Sum** makes signals from other signals by summing them together.
- b) *Attributes*—Not applicable
- c) *Description*—Figure B.22 shows the sum of two sinusoidal signals, one with an amplitude of 1 V and a frequency of 30 Hz and the other with an amplitude of 1 V and a frequency of 960 Hz.

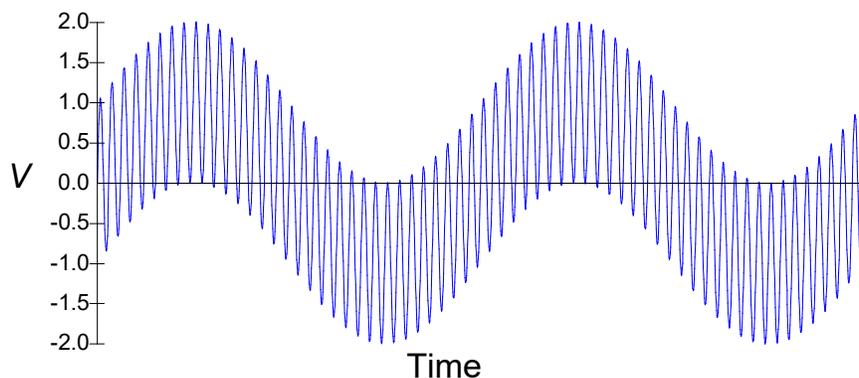


Figure B.22—Sum

#### B.6.2.2.2 Product ::Combiner

- a) *Definition*—**Product** makes signals from other signals by multiplying them together.
- b) *Attributes*—Not applicable
- c) *Description*—Figure B.23 shows the product of two sinusoidal signals, one with an amplitude of 1 V and a frequency of 30 Hz and the other with an amplitude of 1 V and a frequency of 960 Hz).

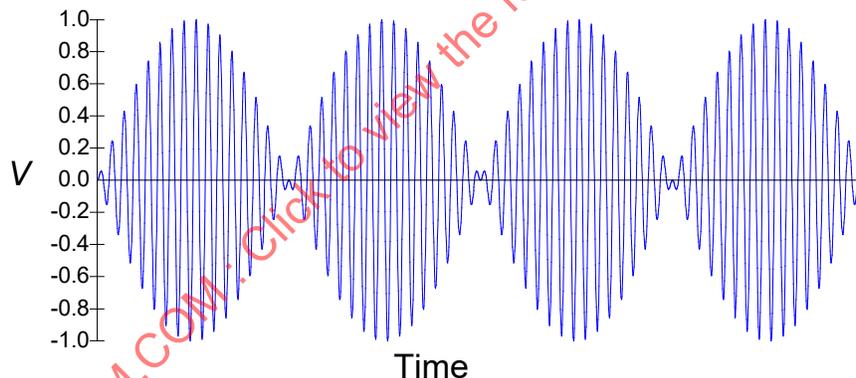


Figure B.23—Product

#### B.6.2.2.3 Diff ::Combiner

- a) *Definition*—**Diff** makes a signal from other signals by subtracting the second and subsequent signals from the first signal.
- b) *Attributes*—Not applicable
- c) *Description*—Figure B.24 shows the difference between two sinusoidal signals, one with an amplitude of 1 V and a frequency of 30 Hz and the other with an amplitude of 1 V and a frequency of 960 Hz.

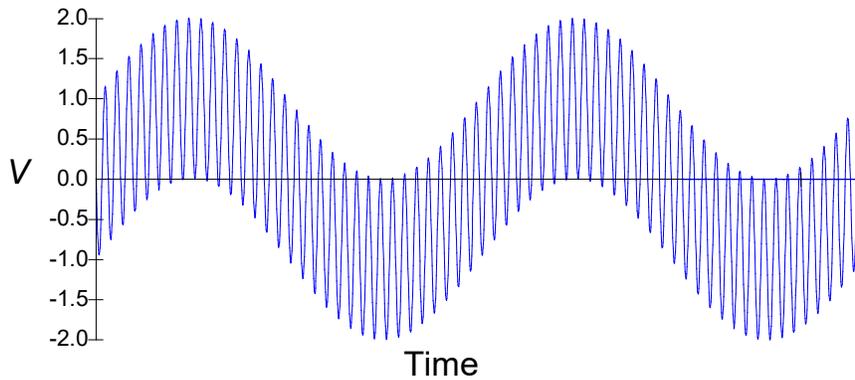


Figure B.24—Diff

### B.6.2.3 Modulator ::Conditioner

- Definition*—**Modulator** provides facilities to create a modulated signal where the modulation is proportional to the input signal.
- Attributes*—Not applicable
- Description*—Where there is no **In** signal (or the **In** signal is Gated Off (tri-state)), the **Out** represents the carrier signal without modulation.

#### B.6.2.3.1 FM<type: Physical> ::Modulator

- Definition*—**FM** is a modulator where the instantaneous frequency of the sinusoidal carrier varies with the amplitude of the modulating input signal.
- Attributes*

**amplitude** <Physical>—peak amplitude of sinusoidal carrier wave (default = 1 V)

**carrierFrequency** <Frequency>—frequency of sinusoidal carrier wave (default = 1 kHz)

**frequencyDeviation** <Frequency>—frequency deviation (default = 100 Hz)
- Description*—The instantaneous frequency of a signal is defined as rate of change of  $\varphi$  ( $d\varphi/dt$ ). For FM, the general solution is given as defined in Equation (B.10):

$$e = E_c \sin(d\varphi/dt) \quad (\text{B.10})$$

where

$$d\varphi/dt = f_c + \text{frequencyDeviation} \times m(t).$$

the general solution for  $d\varphi/dt = f_c + \text{frequencyDeviation} \times m(t)$  is given as

$$\varphi_{ct} + \text{frequencyDeviation} \left( \int_{0-2\pi} m(t) dt \right)$$

and

$E_c$  is the carrier amplitude (unmodulated)  
 $\varphi$  is the phase angle  
 $m(t)$  is the modulating signal  
 $d\varphi/dt$  is the instantaneous frequency  
 $f_c$  is carrier frequency  
 $\int_{0-2\pi}$  is the integral

As an example, the output for a FM-modulated cosine waveform is given by Equation (B.11):

$$e = E_c \sin(\omega_c t + m_f \sin \omega_m t) \quad (\text{B.11})$$

where

- $\omega_c$  is  $2\pi f_c$
- $\omega_m$  is  $2\pi \times$  modulating frequency
- $m_f$  deviation ratio ( $\equiv$  modulation index)

In order that the output signal has the defined deviation ratio, this BSC requires that the amplitude of the modulating input signal has a value of 1 (unity).

Figure B.25 shows frequency modulation where the carrier has a frequency of 960 Hz with an amplitude of 1 V and the modulating signal has a frequency of 30 Hz.

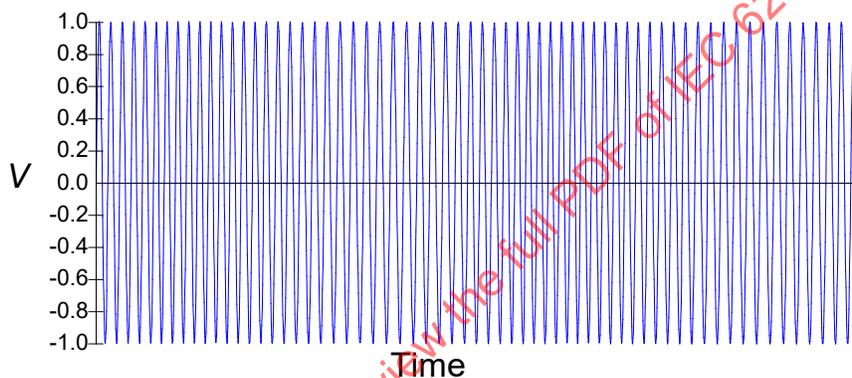


Figure B.25—FM

### B.6.2.3.2 AM :: Modulator

- a) *Definition*—**AM** is a modulator where the amplitude of the carrier varies with the amplitude of the modulating input signal.
- b) *Attributes*
  - modIndex** <Ratio>—modulation index (depth of modulation) (default = 0.3)
  - Carrier** <SignalFunction>—sinusoidal signal to be modulated
- c) *Description*—The formula for AM signal is given in Equation (B.12):

$$e = E_c(1 + \text{modIndex } m(t)) \sin \omega_c t \quad (\text{B.12})$$

where

- $E_c$  is the carrier amplitude (unmodulated)
- $m(t)$  is the modulating signal
- $\omega_c$  is  $2\pi \times$  carrier frequency

As an example, the output for an AM-modulated sinusoid signal is given by Equation (B.13):

$$e = E_c(1+m_a\sin\omega_m t)\sin\omega_c t \quad (\text{B.13})$$

where

- $E_c$  is the carrier amplitude (unmodulated)
- $m_a$  is the depth of modulation ( $\equiv$  modulation index)
- $\omega_m$  is  $2\pi \times$  modulating frequency
- $\omega_c$  is  $2\pi \times$  carrier frequency

In order that the output signal has the defined modulation index, the BSC requires that the amplitude of the modulating input signal has a value of 1 (unity).

Figure B.26 shows amplitude modulation where the carrier has a frequency of 960 Hz with an amplitude of 1 V and the modulating signal has a frequency of 30 Hz.

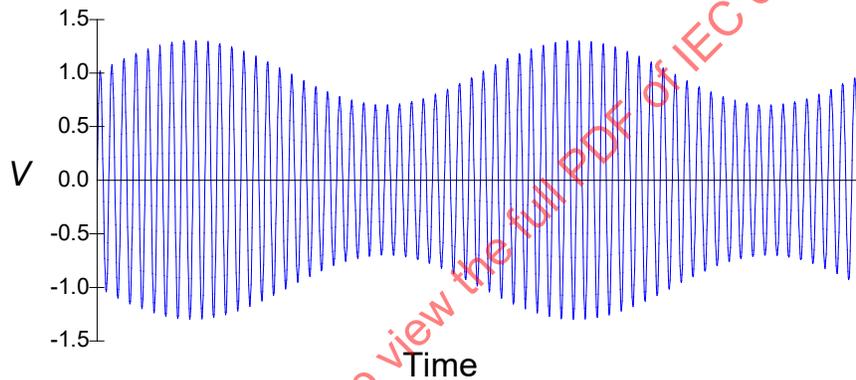


Figure B.26—AM

### B.6.2.3.3 PM<type: Physical> :: Modulator

- a) *Definition*—**PM** is a modulator where the phase of the sinusoidal carrier varies with the amplitude of the modulating input signal.
- b) *Attributes*
  - amplitude** <Physical>—amplitude of sinusoidal carrier wave (default = 1 V)
  - carrierFrequency** <Frequency>—frequency of sinusoidal carrier wave (default = 1 kHz)
  - phaseDeviation** <PlaneAngle>—phase deviation (default =  $\pi/4$  rad)
- c) *Description*—The formula for a PM signal is given in Equation (B.14):

$$e = E_c\sin(\omega_c t + \text{phaseDeviation } m(t)) \quad (\text{B.14})$$

where

- $E_c$  is the carrier amplitude (unmodulated)
- $\omega_m$  is  $2\pi \times$  modulating frequency
- $m(t)$  is the modulating signal

As an example, the output PM-modulated cosine signal is given by Equation (B.15):

$$e = E_c \sin (\omega_c t + \varphi_d \cos \omega_m t) \tag{B.15}$$

where

- $E_c$  is the carrier amplitude (unmodulated)
- $\omega_c$  is  $2\pi \times$  carrier frequency
- $\varphi_d$  is phase deviation ( $\equiv$  modulation index)
- $\omega_m$  is  $2\pi \times$  modulating frequency

In order that the output signal has the correct phase deviation, the BSC requires that the amplitude of the modulating input signal has a value of 1 (unity).

Figure B.27 shows phase modulation where the carrier has a frequency of 960 Hz with an amplitude of 1 V and the modulating signal has a frequency of 30 Hz.

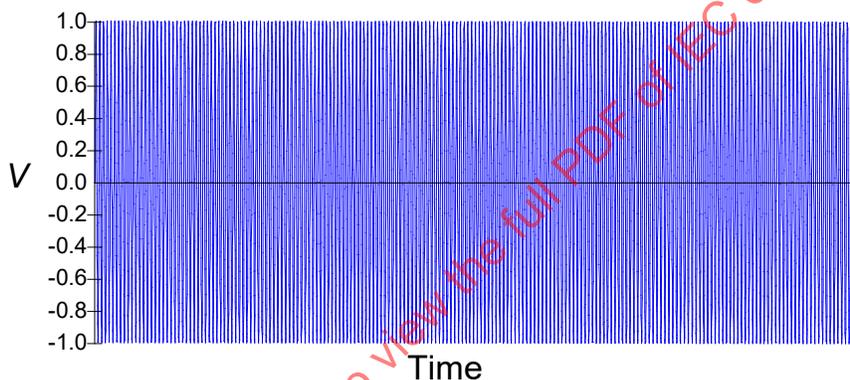


Figure B.27—PM

#### B.6.2.4 Transformation ::Conditioner

- a) *Definition*—**Transformation** takes a signal and transforms it (e.g., converting it from the time domain to the frequency domain).
- b) *Attributes*—Not applicable
- c) *Description*

#### B.6.2.4.1 SignalDelay ::Transformation

- a) *Definition*—With **SignalDelay**, the **In** signal is delayed to become the **Out** signal, where the delay is defined by an initial fixed delay and where the delay may change over time.
- b) *Attributes*

**acceleration** <Frequency>—the rate at which the rate shall alter over time (default = 0 s<sup>-1</sup>)

**delay** <Time>—the fixed delay that signal shall be delayed (default = 0 s)

**rate** <Ratio>—the rate at which the delay shall alter over time (default = 0)

c) *Description*—**SignalDelay** can be applied to both signals and events. The **SignalDelay** can be used for two distinct operations on the input signal:

- 1) Delay signals (**delay**)
- 2) Change the time base (**rate, acceleration**)

Both these operations can be combined into a single **SignalDelay**.

The delay at time  $t$  ( $t_d$ ) between the input and output is calculated from the initialization time ( $t_0$ ) as defined in Equation (B.16):

$$t_d = \text{SignalDelay} = \text{Delay} + (\text{Rate} \times t) + (\text{Acceleration} \times t^2/2) \quad (\text{B.16})$$

When the signal delay time ( $t_d$ ) is greater than the current time ( $t$ ), signal delay presents a null output signal.

A signal delay time that is negative refers to events that shall happen in the future. This value is a valid signal specification and represents a change in the time base of the signal. For example, a rate of 1 has the effect of doubling the frequency (i.e., halving the period) of any input signal.

*Example:*

An example of **SignalDelay** is radar, the delay for a signal to travel to a moving target, where all properties are defined with respect to distance (meters) and the speed of light (meters/second).

The delay for a signal to travel to a moving target is defined as follows:

- Delay is the fixed delay (due to distance from target to the observer),  $\text{m}/(\text{m}/\text{s})=\text{s}$ .
- Rate is the velocity at which the target is moving away from the observer,  $(\text{m}/\text{s})/(\text{m}/\text{s})=\text{dimensionless}$ .
- Acceleration is the rate at which the velocity of the target (from the observer) is changing,  $(\text{m}/\text{s}^2)/(\text{m}/\text{s})=\text{s}^{-1}$ .

Using the **SignalDelay** for radar defines both the radar pulse delay for the target plus any Doppler effect, due to the target movement, through changes to the signal time base.

Figure B.28 shows the effect of a **SignalDelay** on a waveform ramp. In this example, the delay is 0.1 s, the acceleration is  $-0.1 \text{ s}^{-1}$ , and the rate is  $-0.1$ .

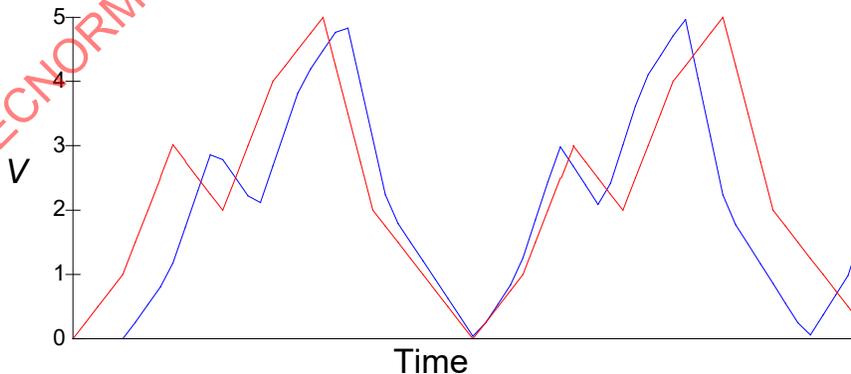


Figure B.28—SignalDelay

#### B.6.2.4.2 Exponential Transformation

- a) *Definition*—**Exponential** is a transformation that multiplies the input signal with a coefficient that decays exponentially over time. See Figure B.29.
- b) *Attributes*  
**dampingFactor** <double>—value of damping factor (default = 1.0)
- c) *Description*—Any signal may be damped over a given time, according to a floating-point damping factor. An **Exponential** is determined by the damping factor, using the expression  $e^{-t/\tau}$ , where  $\tau$  is equal to the damping factor.

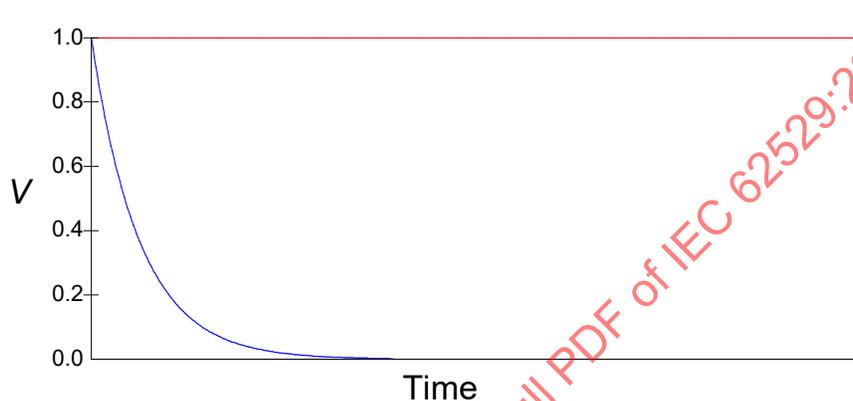


Figure B.29—Exponential (constant amplitude = 1 V)

#### B.6.2.4.3 E Transformation

- a) *Definition*—**E** is an exponential operation on a signal. See Figure B.30.
- b) *Attributes*—Not applicable
- c) *Description*—The output of the signal may be expressed by Equation (B.17):

$$y = e^x \tag{B.17}$$

where

$y$  is the signal output  
 $x$  is the signal input

NOTE—In Equation (B.17), the symbol  $e$  refers to the mathematical constant, the base of the natural logarithm. The equation could also have been expressed as  $y=\exp(x)$ .

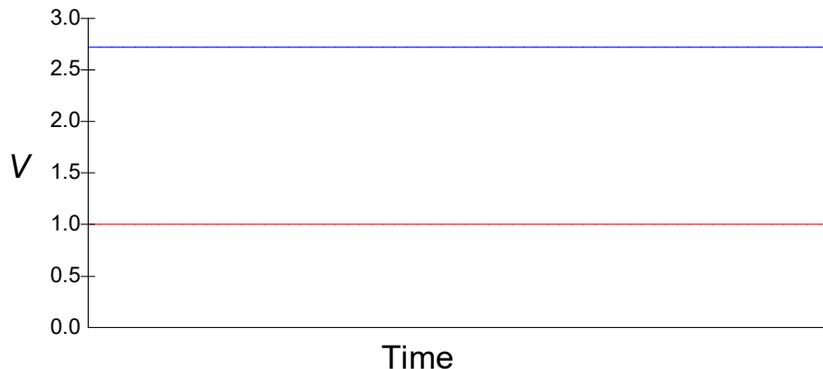


Figure B.30—E (constant amplitude = 1 V)

**B.6.2.4.4 Ln ::Transformation**

- a) *Definition*—**Ln** is a natural logarithmic (inverse exponential) operation on a signal. See Figure B.31.
- b) *Attributes*—Not applicable
- c) *Description*—The output of the signal may be expressed by Equation (B.18):

$$y = \ln(x) \tag{B.18}$$

where

- $y$  is the signal output
- $x$  is the signal input

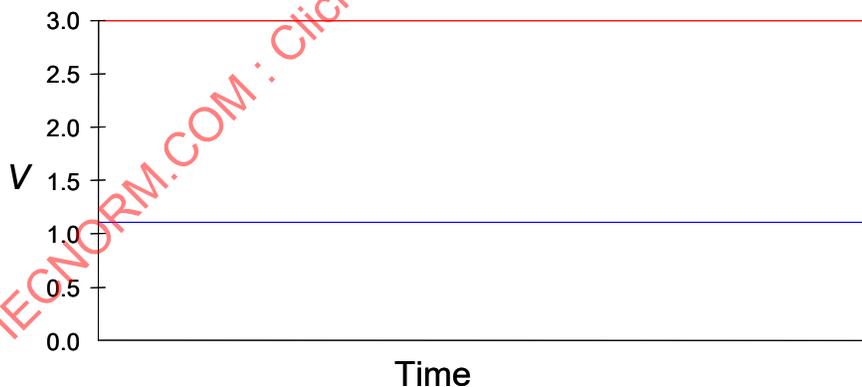


Figure B.31—Ln (constant amplitude = 3 V)

**B.6.2.4.5 Negate ::Transformation**

- a) *Definition*—**Negate** modifies a signal so that its amplitude is the negative of the **In** signal amplitude. See Figure B.32.
- b) *Attributes*—Not applicable

- c) *Description*—The output of the signal may be expressed by Equation (B.19):

$$y = -x \tag{B.19}$$

where

$y$  is the signal output  
 $x$  is the signal input

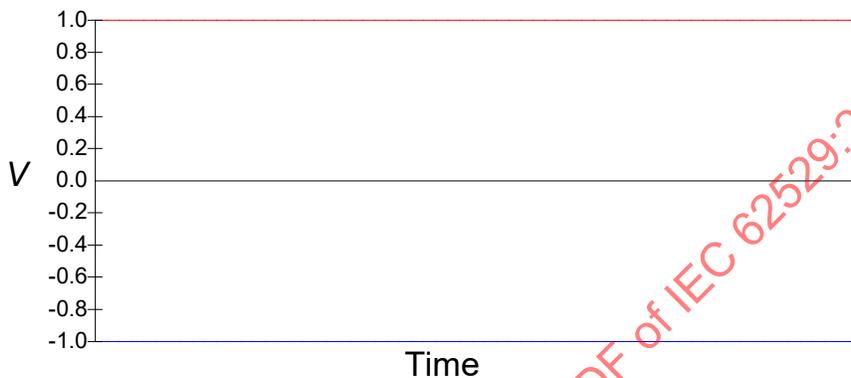


Figure B.32—Negate (constant amplitude = 1 V)

#### B.6.2.4.6 Inverse ::Transformation

- a) *Definition*—**Inverse** is the mathematical reciprocal of a signal. See Figure B.33.  
b) *Attributes*—Not applicable  
c) *Description*—The output of the signal may be expressed by Equation (B.20):

$$y = 1/x \tag{B.20}$$

where

$y$  is the signal output  
 $x$  is the signal input

NOTE—The value of  $y$  is indeterminate when the value of  $x$  is 0.

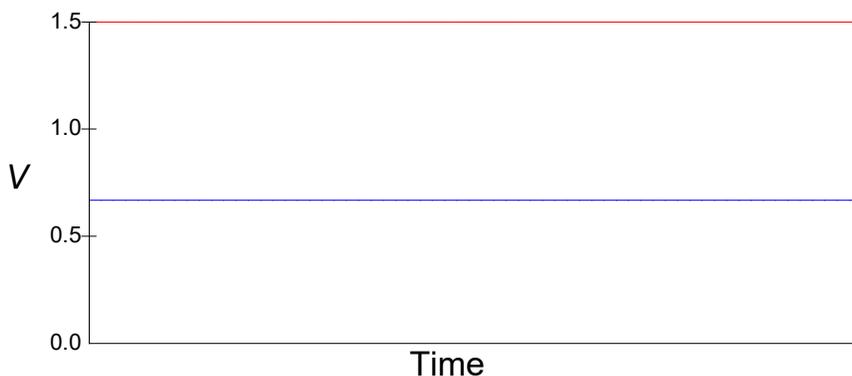


Figure B.33—Inverse (constant amplitude = 1 V)

#### B.6.2.4.7 PulseTrain ::Transformation

- a) *Definition*—**PulseTrain** creates a train of pulses of the **In** signal by multiplying the input signal with the amplitude of the pulses.
- b) *Attributes*
  - pulses** <PulseDefns>—a list defining the shape of the pulses to be created
  - repetition** <int>—the number of times the list of pulses is output; zero indicates that the sequence is repeated indefinitely (default = 0)
- c) *Description*—Figure B.34 shows the creation of a **PulseTrain** where the **In** signal is a sinusoid of amplitude 1 V with a frequency of 30 Hz and the pulses are defined by the PulseDefns [(0.2, 0.5), (0.4, 0.3, 0.5)]. The default **repetition** value of 0 is assumed.

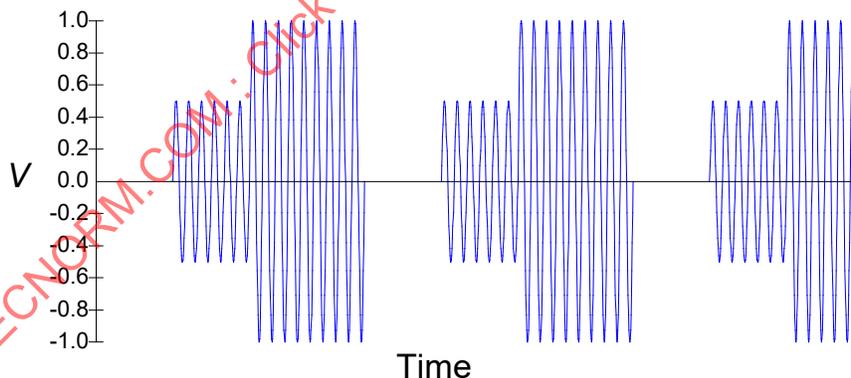


Figure B.34—PulseTrain

#### B.6.2.4.8 Attenuator ::Transformation

- a) *Definition*—**Attenuator** scales the amplitude (a dependent variable) of the **In** signal and allows both the increase and decrease of the signal. See Figure B.35.
- b) *Attributes*
  - gain** <Ratio>—ratio defining the scaling factor for the signal (default = 1)

- c) *Description*—The output of the signal may be expressed by Equation (B.21):

$$y = mx \tag{B.21}$$

where

- $y$  is the signal output
- $x$  is the signal input
- $m$  is the gain

NOTE—If the value of **gain** is negative, a dc signal shall also be negated, and an ac signal shall acquire a 180° phase shift.

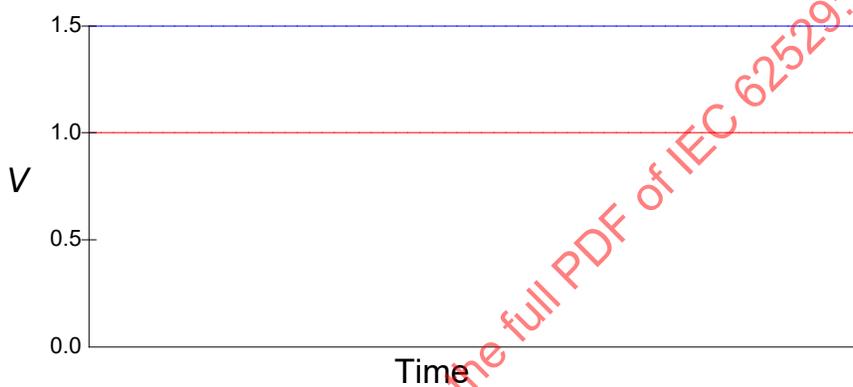


Figure B.35—Attenuator (gain = 1.5, constant amplitude = 1 V)

#### B.6.2.4.9 Load :: Transformation

- a) *Definition*—**Load** provides an impedance to load a signal. See Figure B.36.
- b) *Attributes*
- resistance** <Resistance>—the impedance (in ohms) of the resistive part of the load  
(default = 0 Ω)
  - reactance** <Reactance>—the impedance (in ohms) of the reactive part of the load  
(default = 0 Ω)
- c) *Description*—**Load** provides an impedance, defined in terms of resistance and reactance, which can load a signal.

The **Load** does not modify a signal but is used to indicate an impedance required to set out the correct operation of a signal at its point of connection. It is not to be used as a circuit element, in which case a **Constant** of type Impedance may be used.

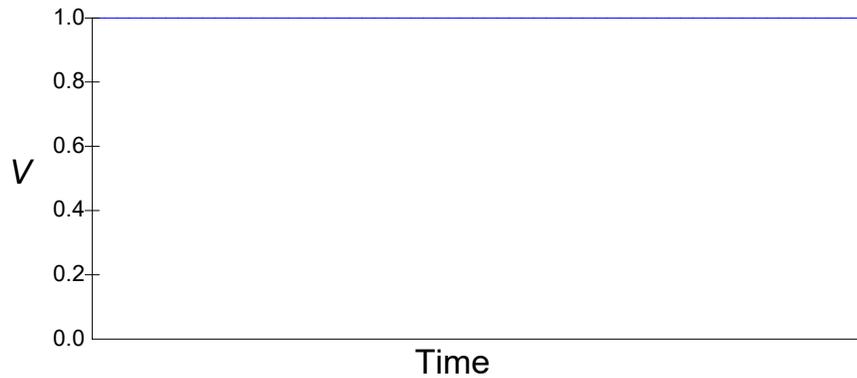


Figure B.36—Load (resistance=50  $\Omega$ , constant amplitude=1 V)

#### B.6.2.4.10 Limit<type: Physical> ::Transformation

- a) *Definition*—**Limit** restricts the values of the signal to  $\pm$  the limit value.
- b) *Attributes*

**limit** <Physical>—the absolute value of the maximum or minimum signal (default = 1)

- c) *Description*—**Limit** has a generic type. Therefore, using a **Limit**(Voltage) on a voltage signal limits the signal voltage. Using a **Limit**(Current) on a voltage signal restricts the voltage to limit the current using the equation  $V=IR$ . Using **Limit**(Power) restricts the voltage to limit the power using the expression  $I.V$ .

Figure B.37 shows the effect of a limit of 0.90 V on a sinusoid of amplitude 1 V and frequency 30 Hz.

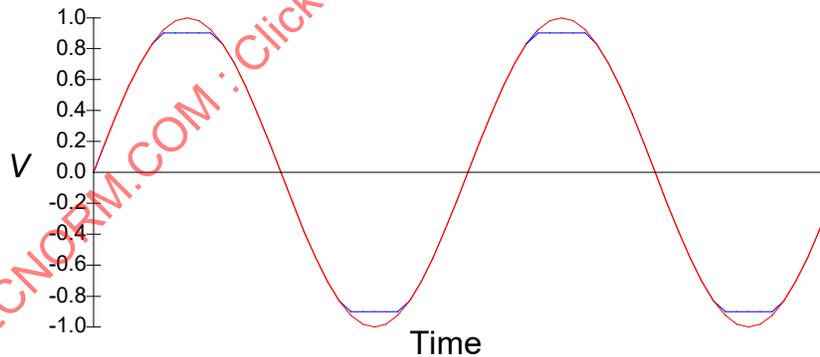


Figure B.37—Limit

#### B.6.2.4.11 FFT ::Transformation

- a) *Definition*—**FFT** (i.e., Fourier transform) characterizes time domain signals in the frequency domain at discrete time intervals. It is more restricted than the other BSC signal combination mechanisms. It uses a number of samples (which is rounded up to the nearest power of 2), the time over which the signal shall be sampled, and the signal to be converted into a waveform representing the frequency domain.

b) *Attributes*

**samples** <int>—number of samples to be used (before rounding) (default = 1023)

**interval** <Time>—time to sample signal (default = 1 s)

c) *Description*—The number of **samples** used shall be the next power of 2.

**FFT** provides a Discrete Fourier Transform to convert time to frequency domain signals, useful for measuring frequency characteristics or performing signal analysis. The **FFT** returns the magnitude of the value of the signal within each frequency band, where the frequency band is defined by  $1/\text{interval}$ , and the axis defined from 0 Hz to the Nyquist frequency defined by half of the sampling frequency ( $\text{samples}/(2 \times \text{interval})$ ).

**FFT** loses any signal phase information because it provides only magnitude values and does not provide any complex components.

An inverse FFT is available using the InverseTransform BSC (Ref B.6.5.5) and with the FFT as the **As** reference.

NOTE—The use of the **FFT** as a method of providing the different characteristics of a signal in the frequency domain at sample intervals. Signal transformation is also inherent in the standard for related physical types and reference types (see B.3.3).

Figure B.38 shows the **FFT** of a phase-modulated signal where the carrier has a frequency of 960 Hz with an amplitude of 1 V and the modulating signal has a frequency of 30 Hz.

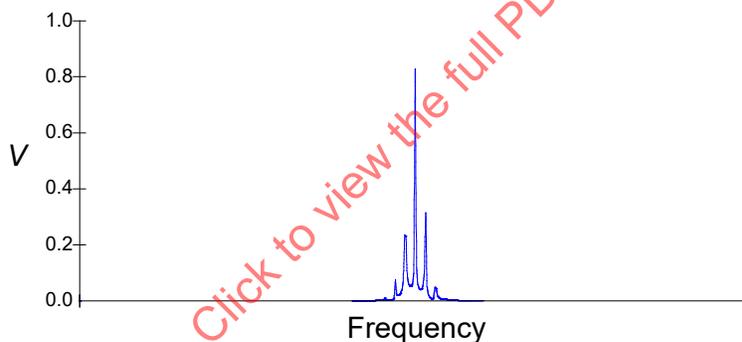


Figure B.38—FFT of PM signal

### B.6.3 EventFunction ::SignalFunction

a) *Definition*—An **EventFunction** creates and manipulates events. Events are signals without value information, where the important information is when they become active and inactive.

b) *Attributes*—Not applicable

c) *Description*—A signal can be used as an event, but an event cannot be used as a signal. In general, **EventFunctions** can be combined to create complex events that are used to synchronize or gate other BSCs.

#### B.6.3.1 EventSource ::EventFunction

a) *Definition*—**EventSources** generate events.

b) *Attributes*—Not applicable

c) *Description*

### B.6.3.1.1 Clock ::EventSource

- a) *Definition*—**Clock** generates an event at regular intervals. Each event is active for the first half of the clock period. See Figure B.39.
- b) *Attributes*
  - clockRate** <Frequency>—frequency of the clock (default = 1 Hz)
- c) *Description*

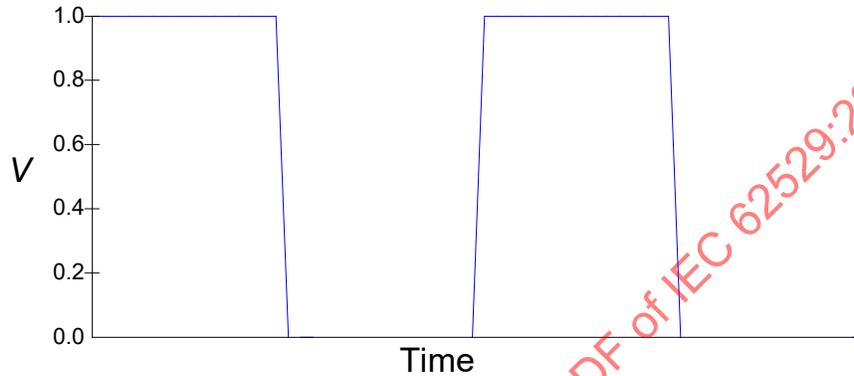


Figure B.39—Clock (clockRate = 20 Hz)

### B.6.3.1.2 TimedEvent ::EventSource

- a) *Definition*—**TimedEvent** generates an **Out** event at regular intervals. Each event is active for a specific duration. If the duration is longer than the event interval, the **Out** event is signaled active at each interval, but never becomes paused. See Figure B.40.
- b) *Attributes*
  - delay** <Time>—the delay time before the first event shall start (default = 0 s)
  - duration** <Time>—the duration for which each event is active (default = 1 s)
  - period** <Time>—the time interval between the start of each successive event (default = 1 s)
  - repetition** <int>—the number of events to be output; zero indicates that events are generated indefinitely (default = 0)
- c) *Description*

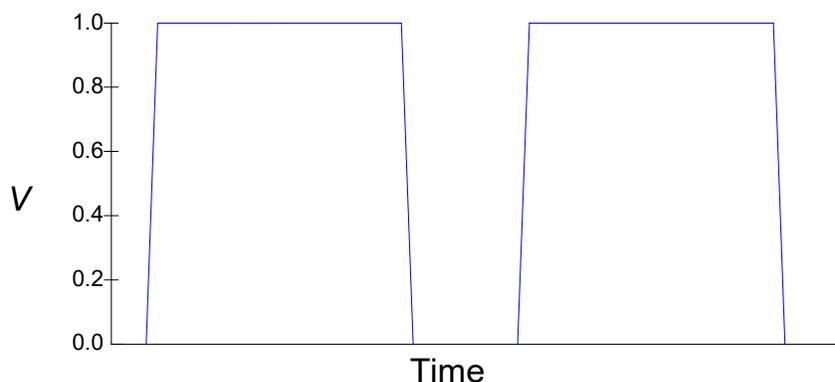


Figure B.40—TimedEvent (delay = 0.1 s, duration = 0.7 s)

### B.6.3.1.3 PulsedEvent ::EventSource

- a) *Definition*—**PulsedEvent** generates an **Out** event in the form of a sequence of timing pulses primarily intended for use in generating **Source** signals. The sequence consists of a train of  $N$  pulses (where  $N$  may be any integer greater than 0). Where  $N$  is greater than 1, the pulses may be of unequal duration and spacing. The pulse train may be either output once or repeated infinitely and continuously for a periodic timing sequence. See Figure B.41.
- b) *Attributes*
  - pulses** <PulseDefns>—a list defining the shape of the pulses to be created
  - repetition** <int>—the number of times the list of pulses is output; zero indicates that the sequence is repeated indefinitely (default = 0)
- c) *Description*—Changes in state (e.g., pulse start and stop) are specified from  $t = 0$ . Cycling is facilitated by resetting the time ( $t$ ) to 0.

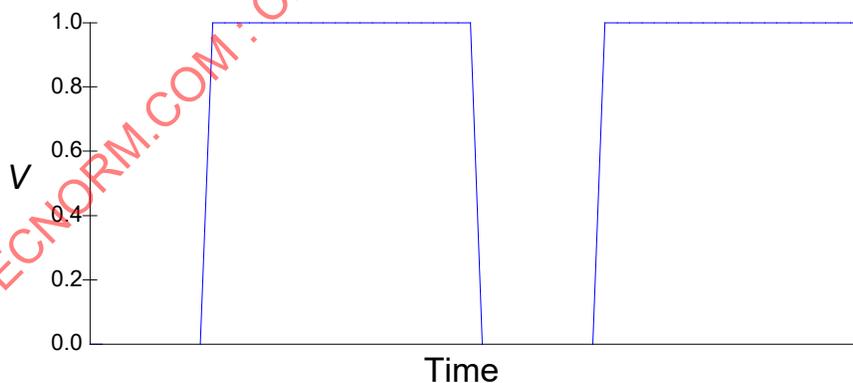


Figure B.41—PulsedEvent (pulses = (0.3, 0.7, 1) )

### B.6.3.2 EventConditioner ::EventFunction

- a) *Definition*—**EventConditioner** takes a signal or event and outputs the event when the event conditions occur. **EventConditioner** allows events to be created and modified, based on the action of the events and signals.

- b) *Attributes*—Not applicable
- c) *Description*

### B.6.3.2.1 EventedEvent ::EventConditioner

- a) *Definition*—**EventedEvent** conditioner allows events to be combined to produce complex event streams.
- b) *Attributes*—Not applicable
- c) *Description*—**EventedEvent** uses multiple inputs to successively enable and disable its own output. The first input (**In(at=1)**) is regarded as the enable event; subsequent inputs are regarded as disable inputs.

The output is enabled (i.e., active) when the input goes active.

If a second input is assigned, the output is disabled (i.e., inactive) when the second input goes active. The output is then enabled (i.e., active) when the first input goes active again, and so forth.

If multiple inputs are assigned, the behavior is determined by cascading the inputs through multiple **EventedEvent** pairs.

Figure B.42 shows an event stream as created by the combination of two clocks, one with a clock rate of 20 Hz and the other with a clock rate of 15 Hz.

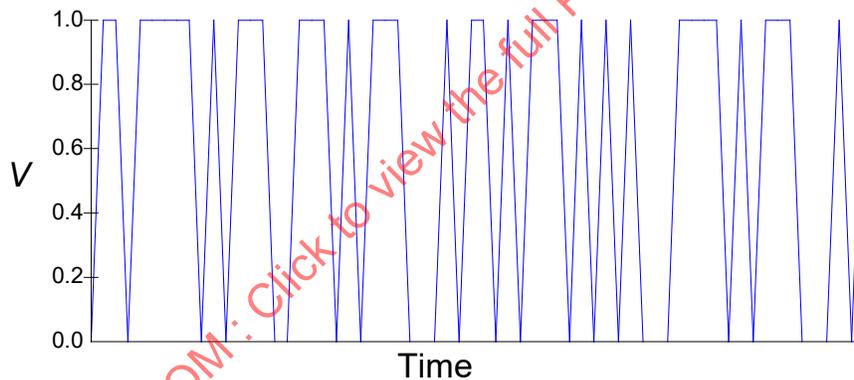


Figure B.42—EventedEvent

### B.6.3.2.2 EventCount ::EventConditioner

- a) *Definition*—**EventCount** filters out input events to only allow every **count** input event. See Figure B.43.
- b) *Attributes*
  - count** <int>—identifies the number of events that shall occur before a event is generated (default = 0)
- c) *Description*—The **EventCount** counts events and produces an event when **count** events are received. The **EventCount** acts as an event divider in which the divider is dependent on the value of the **count** property.

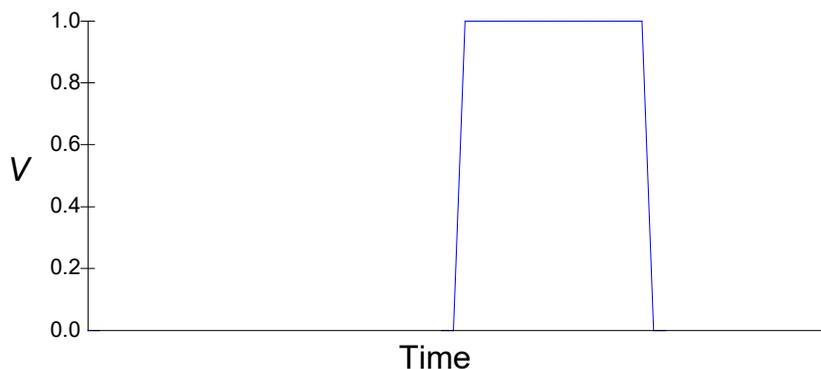


Figure B.43—EventCount (count = 1)

### B.6.3.2.3 ProbabilityEvent ::EventConditioner

- a) *Definition*—**ProbabilityEvent** conditioner generates an event stream based upon the event stream present at the **In** event. It shall replicate the same timing information, but shall randomly suppress **In** pulses. Conceptually, the **ProbabilityEvent** comprises a random number generator and a comparator. At each event, the comparator compares the random number with the value of the attribute **probability** to determine whether to generate an event in the **Out** event stream. A seed is included so that the user can reliably reproduce test results. See Figure B.44.
- b) *Attributes*
  - seed** <int>—for pseudorandom probabilities (default = 0)
  - probability** <Ratio>—value for comparison with random number (default = 50%)
- c) *Description*—**ProbabilityEvent** filters out a proportion of input events. The number it lets through is determined by the probability event. The bigger the ratio, the more events pass through; the lower the ratio, the more events are filtered out. **ProbabilityEvent** filters out complete active sections regardless of their length.

NOTE—The value of **probability** is a ratio, which can include values outside of the range of 0% to 100% (i.e., 0 to 1). The use of values outside of the range of 0% to 100% may have unintended effects upon the signal.

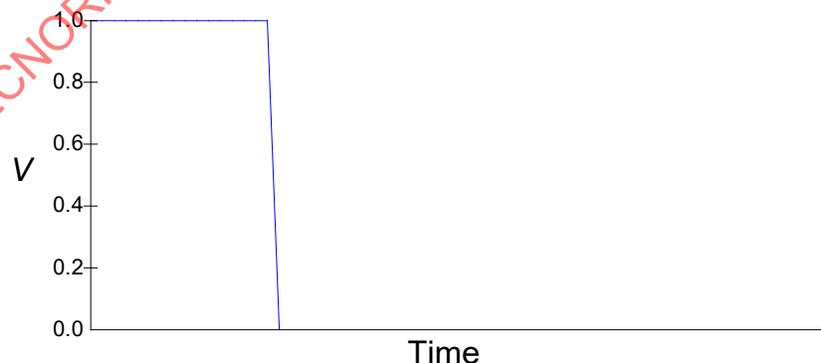
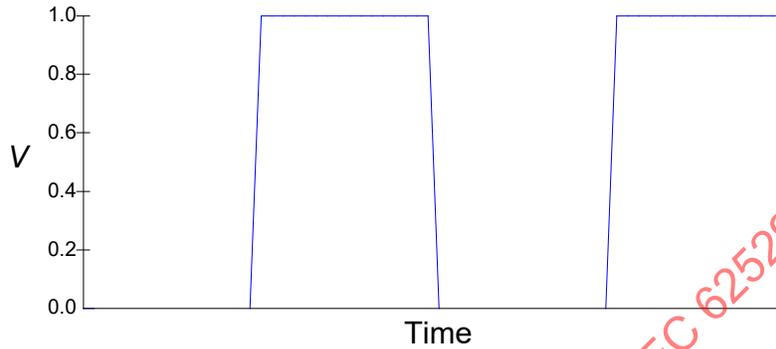


Figure B.44—ProbabilityEvent

**B.6.3.2.4 NotEvent ::EventConditioner**

- a) *Definition*—The **NotEvent** conditioner is active when the **In** signal is not active. See Figure B.45.
- b) *Attributes*—Not applicable
- c) *Description*



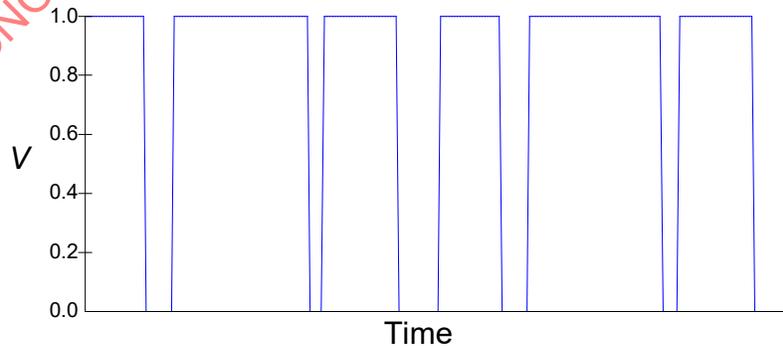
**Figure B.45—NotEvent**

**B.6.3.2.5 Logical ::EventConditioner**

- a) *Definition*—**Logical** event conditioners take multiple input event streams and combine them into a single event stream.
- b) *Attributes*—Not applicable
- c) *Description*

**B.6.3.2.5.1 OrEvent ::Logical**

- a) *Definition*—**OrEvent** is active when any **In** events are active.
- b) *Attributes*—Not applicable
- c) *Description*—Figure B.46 shows an event stream as created by the combination of two clocks, one with a clock rate of 15 Hz and the other with a clock rate of 20 Hz.



**Figure B.46—OrEvent**

### B.6.3.2.5.2 XOrEvent ::Logical

- a) *Definition*—**XOrEvent** is active when an odd number of **In** events is active.
- b) *Attributes*—Not applicable
- c) *Description*—Figure B.47 shows an event stream as created by the combination of two clocks, one with a clock rate of 15 Hz and the other with a clock rate of 20 Hz.

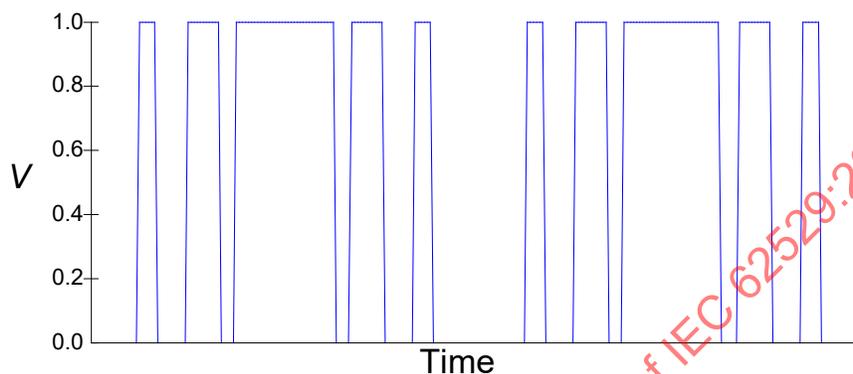


Figure B.47—XOrEvent

### B.6.3.2.5.3 AndEvent ::Logical

- a) *Definition*—**AndEvent** is active when all **In** events are active.
- b) *Attributes*—Not applicable
- c) *Description*—Figure B.48 shows an event stream as created by the combination of two clocks, one with a clock rate of 15 Hz and the other with a clock rate of 20 Hz.

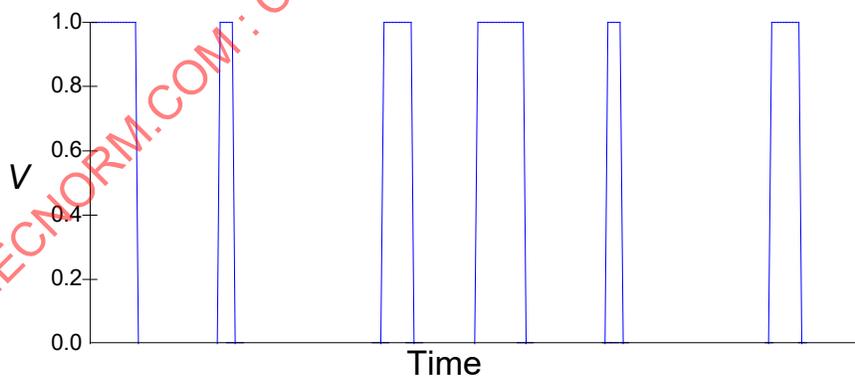


Figure B.48—AndEvent

### B.6.4 Sensor ::SignalFunction

- a) *Definition*—**Sensors** allow signal characteristics to be measured, monitored, and compared. A **Sensor** takes an input signal and generates measurement values. Any **Sensor** can be applied to any signal; however, the resultant value only has meaning when applied to the correct type of signal.

b) *Attributes*

**measuredVariable** <enumMeasuredVariable>—whether the measurement made is of the dependent or independent variable.

**measurement** <any attribute type>—read-only, most recent value(s) measured.

**measurements** <any attribute type>—read-only, multi dimensional array of measurements made.

**samples** <int>—number of discrete consecutive measurements to be made; zero indicates no measurement is to be taken and indicates the **Sensor** is acting as a monitor only; negative value indicates continuous discrete consecutive measurements where the last set of **samples measurements**, are available. (default = 1)

**count** < int >—read-only number of measurements currently made.

**gateTime** <Physical>—continuous range of independent variable (Time) over which measurement is made.

**nominal** <Physical>—primary value against which any condition is checked; can be either an absolute value (e.g., 5 V) or a ratio value (e.g., 50%) representing the percentage value between the low-peak and high-peak values as defined in IEEE Std 181™.

**condition** <enumCondition>—test made between measurement and nominal value. (default = NONE)

**GO** <int>—read-only variable indicating the number of measurement that passed; if no measurement is taken, **GO** is zero.

**NOGO** <int>—read-only variable indicating the number of measurements that failed; if no measurement is taken, **NOGO** is zero.

**HI** <int>—read-only variable indicating the number of measurement with high status; if no measurement is taken, **HI** is zero.

**LO** <int>—read-only variable indicating the number of measurement with low status; if no measurement is taken, **LO** is zero.

**UL** <Physical>—upper limit value against which condition is checked.

**LL** <Physical>—lower limit value against which condition is checked.

**As** <SignalFunction>—reference to a Signal representing a signal model of the expected input signal.

- c) *Description*—**Sensors** are used to measure physical characteristics of signals, which can then be read back through measurement(s) values. **Sensors** are primarily used to take measurements such as root mean square (rms) or average. The output abstract signal value of the **Sensor** is primarily the last measurement(s) or the current monitored value. When the inputs contain multiple channels, each **measurement** represents an array, where each array item is a measured value for each channel and where **measurements** is the collection of these measured arrays.

The **Sensor** generates an output value that is held in the attributes **measurement** and **measurements**. By combining various **Sensors** into a signal model, the compound physical characteristics of a signal can be defined, e.g., signal-to-noise ratio or average rms value.

A **Sensor** can take multiple measurements. The number of measurements required is defined by the attribute **samples**. A value of zero indicates that the measurement values are never required and the **Sensor** is a monitor only. A negative value of **samples** indicates that measurements are made continuously where the absolute value of **samples** represents the maximum number of previous measurements that can be retrieved. The number of measurements currently taken is held in the read-only attribute **count**. In measurement mode, the **Out** abstract signal of the **Sensor** is active after all of the measurement(s) have been taken and has the value of the last measurement made. In

Monitor mode, the **Out** signal of a **Sensor** is active while the monitor condition is being met and has the current monitored value, e.g., rms or average value.

The condition and nominal value can also be used to define the window over which the measurement is taken. For example, take 10 peak measurements (samples) when the input signal value is **GT** (condition) trms 100mV (nominal), as defined by IEEE Std 181.

The attribute **gateTime** defines the explicit measurement window over which the signal is monitored or the measurement is taken. If the value of **gateTime** is zero (0.0), the measurement window is implementation dependent. If the value of **gateTime** is negative (<0.0), the measurement window is the width of the event on the **Gate**.

If no **Gate** is provided (i.e. unassigned), the measurement value is evaluated whenever the input signal becomes active. When a **Gate** is provided (i.e. assigned), the measurement value is evaluated whenever the Gate event becomes active, provided the input signal is active. Measurements continue to be taken until the number in attribute **samples** have been taken, where each measurement is taken after the gate time elapses, and if a **Gate** is assigned, whenever the next Gate event arrives.

Each **Sync** event restarts the measurement operation from the beginning so that the **Sync** event clears the count, (**count** = 0) and resets the measurement.

A **Sensor** is operational only while it is taking measurements. When the number of measurements in the attribute **samples** has been made, the **Sensor** calls **Change** (see Annex C) on the input signal. A monitor where the attribute **samples** is zero never implicitly calls **Change** on the input.

The attribute **measuredVariable** may contain one of the enumerated values **DEPENDENT** or **INDEPENDENT**.

The attribute **condition** may contain one of the enumerated values **NONE**, **GT**, **GE**, **LE**, **LT**, **EQ**, or **NE**.

The **GO**, **NOGO**, **HI**, and **LO** variables are set or updated by one of the following (in order of precedence):

- 1) As a result of a comparison between a measured value (or series of measured values) and the specified limits (UL and LL).
- 2) As a result of an equivalence comparison between a measured value (or series of measured values) and the nominal value.
- 3) As a result of an equivalence comparison between a measured value (or series of measured values) and the value of the appropriate property in the **As** signal.

NOTE—A sensor shall output an Active Event at any time that the value of the monitored signal satisfies the condition with respect to the nominal value.

#### B.6.4.1 Counter ::Sensor

- a) *Definition*—For all **Sensors**, every time a measurement is taken, the attribute **count** is incremented. Counter is a **Sensor** that counts when a measurement would be taken, but does not take any specific measurement.
- b) *Attributes*—Not applicable
- c) *Description*

NOTE—When measuring the independent variable, the measurement value is the same as for the dependent variable.

### B.6.4.2 Interval ::Sensor

- a) *Definition*—**Interval** measures the interval between the **In/Sync** event going active and the **Gate** event going active.
- b) *Attributes*—Not applicable
- c) *Description*—The measurement is taken only when the **In** event is active and **Gate** event goes active. The counter is set to 0 every time the **In** or **Sync** signal becomes active.

If no **Gate** is present (i.e., unassigned), the interval is measured between consecutive **In** events going active (e.g., period). If the **Gate** is present (i.e., assigned), the interval is recorded when the **Gate** event becomes active. The measurement is reset when the **In** event goes active.

The **Sync** event clears the count (**count** = 0) and resets the measurement.

NOTE—The independent measurement value corresponds to the width of the active signal being monitored. Therefore, if no **Gate** is assigned, the value is the width of the Active state of the **In** signal.

### B.6.4.3 Instantaneous<type: Physical> ::Sensor

- a) *Definition*—**Instantaneous** measures the amplitude (i.e., value) of the signal in the dimension “type” at specified instances in time.
- b) *Attributes*—Not applicable
- c) *Description*—The instantaneous type value of the signal with respect to the independent variable (e.g., time) is returned. The signal is not sampled over a gate time or **Gate**. The **Gate** is used only to indicate when the Instantaneous measurement should be made.

NOTE—When measuring the independent variable, the measurement value is the value of the independent variable at the instant that the measurement was taken.

Figure B.49 shows the instantaneous value of the sum of two signals: a sinusoid with an amplitude of 1 V and a frequency of 1 Hz and a constant with an amplitude of 1.5 V.

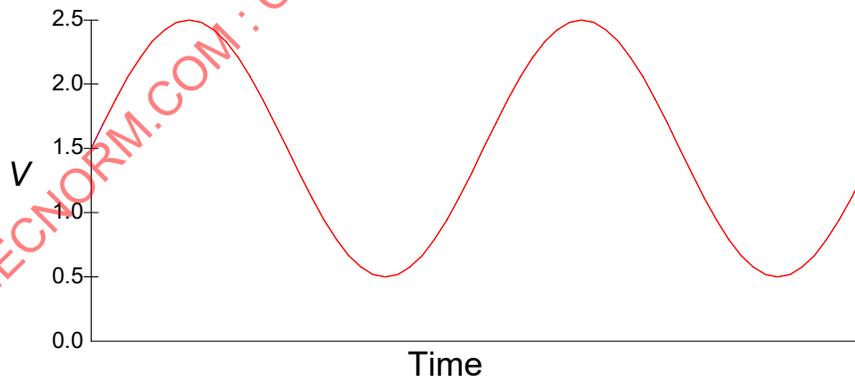


Figure B.49—Instantaneous

### B.6.4.4 RMS<type: Physical> ::Sensor

- a) *Definition*—**RMS** measures the root-mean-square (rms) value of a signal.
- b) *Attributes*—Not applicable

- c) *Description*—The default rms gate time should be a whole number of periods of the input signal to achieve maximum accuracy.

NOTE—When measuring the independent variable, the measurement value is the RMS value of the independent variable while the measurement was taken.

Figure B.50 shows the rms value of the sum of two signals: a sinusoid with an amplitude 1 V and a frequency of 1 Hz and a constant with an amplitude of 1.5 V.

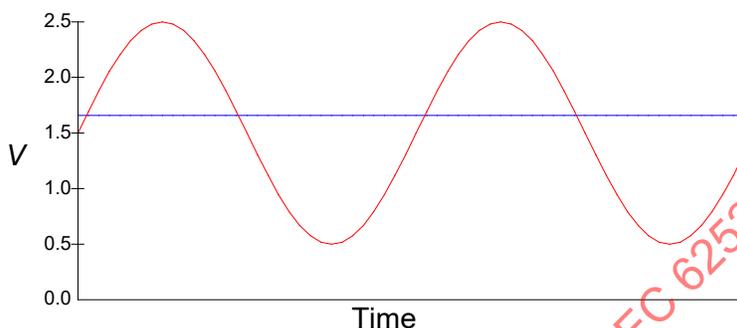


Figure B.50—RMS

#### B.6.4.5 Average<type: Physical> ::Sensor

- a) *Definition*—**Average** is the arithmetic mean of all the signal values during the gate time.
- b) *Attributes*—Not applicable
- c) *Description*—The default average gate time should be a whole number of periods of the input signal to achieve maximum accuracy.

NOTE—When measuring the independent variable, the measurement value is the average of the independent variable while the measurement was taken, e.g., average time when measurement was made, or center frequency of the bandwidth.

Figure B.51 shows the average value of the sum of two signals: a sinusoid with an amplitude of 1 V and a frequency of 1 Hz and a constant with an amplitude of 1.5 V.

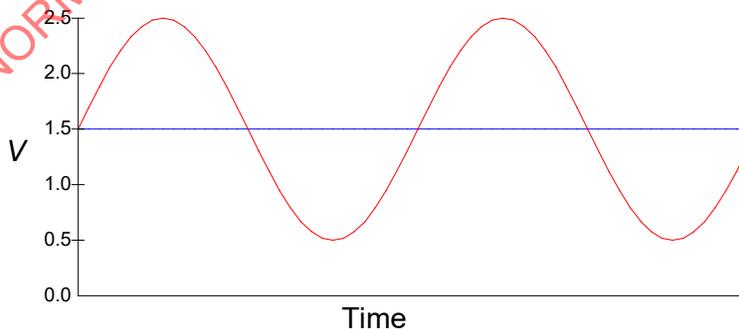


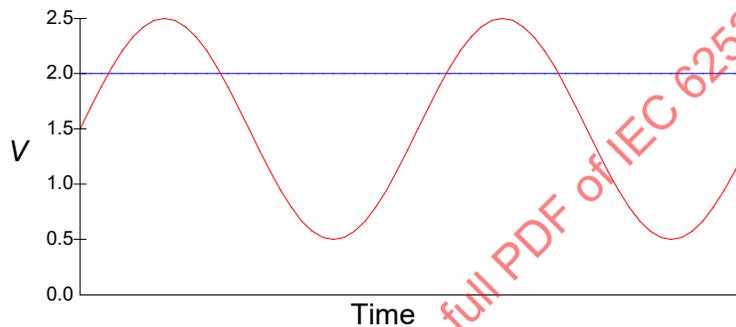
Figure B.51—Average

**B.6.4.6 PeakToPeak<type: Physical> ::Sensor**

- a) *Definition*—**PeakToPeak** is the difference between the highest value and the lowest value during the gate time.
- b) *Attributes*—Not applicable
- c) *Description*

NOTE—When measuring the independent variable, the measurement value is the difference in the values of the independent variable when the (last) maximum and the (first) minimum values of the dependent variable occurred.

Figure B.52 shows the peak-to-peak value of the sum of two signals: a sinusoid with an amplitude of 1 V and a frequency of 1 Hz and a constant with an amplitude of 1.5 V.



**Figure B.52—PeakToPeak**

**B.6.4.7 Peak<type: Physical> ::Sensor**

- a) *Definition*—**Peak** is the measured value that is furthest away from the mean value.
- b) *Attributes*—Not applicable
- c) *Description*—**Peak** returns either the **PeakNeg** or **PeakPos**, whichever has the largest absolute value.

NOTE—When measuring the independent variable, the measurement value is the difference in the values of the independent variable when the (last) peak and the average (center) values of the dependent variable occurred, e.g., how far the peak is from the center.

Figure B.54 shows the peak value of the sum of two signals: a sinusoid with an amplitude of 1 V and a frequency of 1 Hz and a constant with an amplitude of 1.5 V.

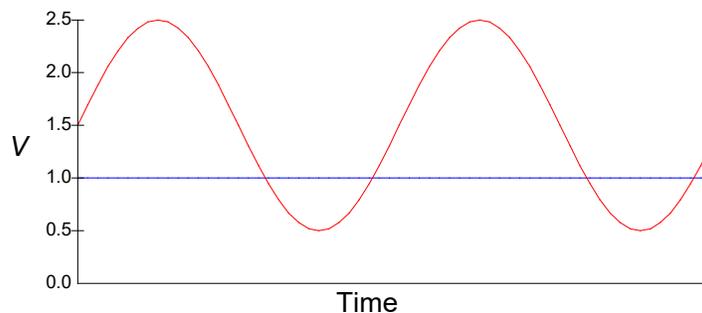


Figure B.53—Peak

#### B.6.4.8 PeakPos<type: Physical> ::Sensor

- Definition*—**PeakPos** is the value obtained by subtracting the mean value from the maximum measurement of the signal during the gate time.
- Attributes*—Not applicable
- Description*

NOTE—When measuring the independent variable, the measurement value is the difference in the values of the independent variable when the (last) peak and the average (center) values of the dependent variable occurred, e.g., how far the peak is from the center.

Figure B.54 shows the positive peak value of the sum of two signals: a sinusoid with an amplitude of 1 V and a frequency of 1 Hz and a constant with an amplitude of 1.5 V.

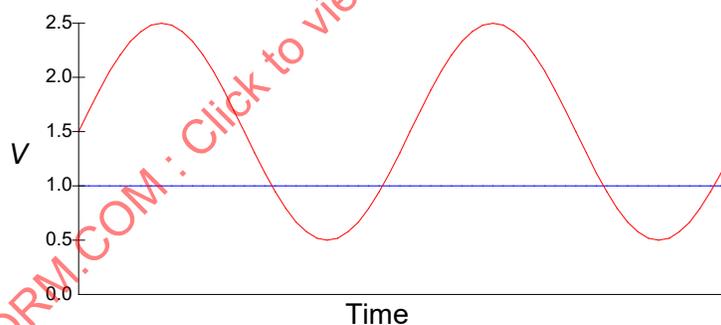


Figure B.54—PeakPos

#### B.6.4.9 PeakNeg<type: Physical> ::Sensor

- Definition*—**PeakNeg** measurement is the value obtained by subtracting the mean value from the minimum measurement of the signal during the gate time.
- Attributes*—Not applicable
- Description*

NOTE—When measuring the independent variable, the measurement value is the difference in the values of the independent variable when the average (center) and the (last) minimum values of the dependent variable occurred, e.g., how far the peak negative point is from the center.

Figure B.55 shows the peak negative value of the sum of two signals: a sinusoid with an amplitude of 1 V and a frequency of 1 Hz and a constant with an amplitude of 1.5 V.

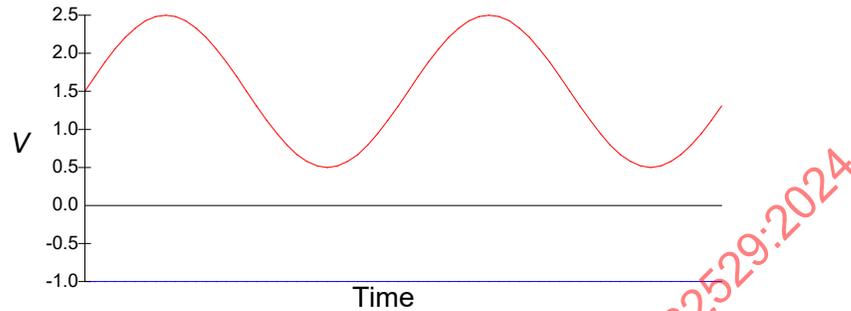


Figure B.55—PeakNeg

#### B.6.4.10 MaxInstantaneous<type: Physical> ::Sensor

- a) *Definition*—**MaxInstantaneous** measurement is the maximum measurement of the signal during the gate time.
- b) *Attributes*—Not applicable
- c) *Description*

NOTE—When measuring the independent variable, the measurement value is the value of the independent variable at the instant that the maximum peak occurred.

Figure B.56 shows the maximum instantaneous value of the sum of two signals: a sinusoid with an amplitude of 1 V and a frequency of 1 Hz and a constant with an amplitude of 1.5 V.

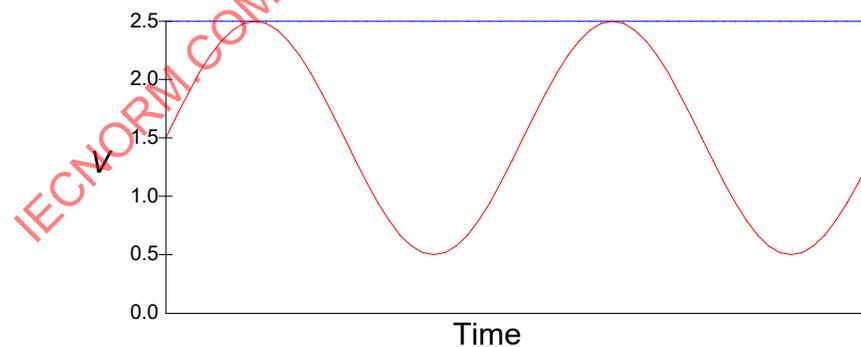


Figure B.56—MaxInstantaneous

#### B.6.4.11 MinInstantaneous<type: Physical> ::Sensor

- a) *Definition*—**MinInstantaneous** measurement is the minimum measurement of the signal during the gate time.

- b) *Attributes*—Not applicable
- c) *Description*

NOTE—When measuring the independent variable, the measurement value is the value of the independent variable at the instant that the minimum peak occurred.

Figure B.57 shows the minimum instantaneous value of the sum of two signals: a sinusoid with an amplitude of 1 V and a frequency of 1 Hz and a constant with an amplitude of 1.5 V.

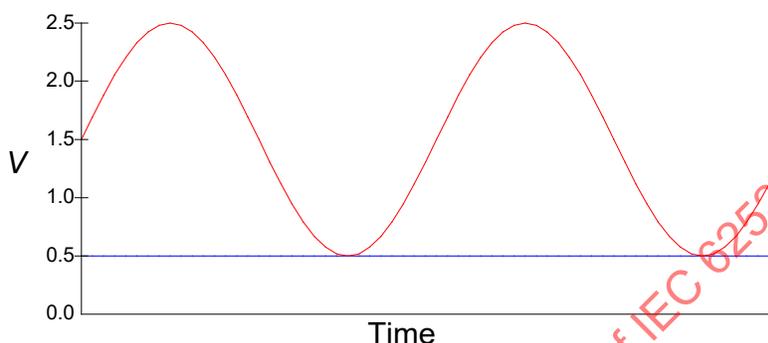


Figure B.57—MinInstantaneous

#### B.6.4.12 Measure ::Sensor

- a) *Definition*—The **Measure** BSC measures any control attribute, as opposed to a capability attribute, of a BSC or TSF, or input signal value of a BSC or TSF.
- b) *Attributes*  
**attribute** <string>—name of the signal attribute that is to be measured, e.g., **car\_ampl** or **In**.
- c) *Description*—Provides the ability to measure any attribute of any TSF or BSC. The Signal and properties referenced by **As** and **attribute** are used to indicate the measurement required. The method of measurement is not defined by this standard, however the standard does provide a method of validation that the correct measurement process has been met.

The **Measure** BSC conceptually compares the actual input signal against all possible allowed reference signals and selects a resultant reference signal that provides the best match. The returned values are the corresponding values of the identified resultant reference signal. The best match is defined as the minimum rms value of the difference between the actual input signal and the reference signal defined by **As**.

The **attribute** property can be represented by one or more of four distinct cases:

- 1) Not present or empty
- 2) Control attribute name, e.g., **amplitude**
- 3) Input Signal name, e.g., **In**
- 4) Non-Control attribute such as value or Capability, e.g., **measurement**

If no **attribute** property is defined, the value returned is the minimum rms difference between the reference source signal and the input signal. This is equivalent to the signal's standard deviation error from the reference signal, and represents the RMS error value.

When the value of the **attribute** property is a control attribute name the returned value represents the value of the control attribute that provides the best match to the input signal.

When the value of the **attribute** property represents an input signal name, the Out abstract Signal represents the input signal that, when applied to the Reference signal, best matches the input signal being observed; this provides an inverse transform function as defined by *InverseTransform*. Any measurements made are made on that waveform.

For example, when the **As** signal represents a conditioner, this has the equivalent effect of applying an inverse conditioner to the input signal. For example, the following two models are equivalent, but only because of the behavior of Negate, i.e.,  $\text{InverseTransform}(\text{Negate}) = \text{Negate}$ :

- `<Measure As="Negate" In="..." />`
- `<Measure In="InverseNegate" /><Negate name="InverseNegate" In="..." />`

When the **attribute** properties represent a NonControl attribute name such as a measured value or Capability, e.g., **measurement**, that value or Capability shall be returned based on the actual input signal.

The **attribute** property, if provided, shall be the name of a property of the reference signal defined by **As**.

If the **As** property is not specified, the **attribute** property is ignored, and the **Measure** BSC measures the value indicated by the nominal value as illustrated by the following examples:

```
<Measure nominal="5 V" /> implies <Instantaneous nominal="5 V" />
<Measure nominal="trms 10 V" /> implies <RMS nominal="trms 10 V" />
<Measure nominal="av 5 V" /> implies <Average nominal="av 5 V" />
<Measure nominal="pk_pk 5 V" /> implies <PeakToPeak nominal="pk_pk 5 V" />
<Measure nominal="pk 5 V" /> implies <Peak nominal="pk 5 V" />
<Measure nominal="pk_pos 5 V" /> implies <PeakPos nominal="pk_pos 5 V" />
<Measure nominal="pk_neg 5 V" /> implies <PeakNeg nominal="pk_neg 5 V" />
<Measure nominal="inst_max 5 V" /> implies <InstantaneousMax nominal="inst_max 5 V" />
<Measure nominal="inst_min 5 V" /> implies <InstantaneousMin nominal="inst_min 5 V" />
```

These examples are considered to have an implied **As** referenced to a corresponding Intrinsic measurement type, e.g., `<Measure nominal="trms 10 V" As="rmsSig"/>` (where "rmsSig" is the name of an RMS sensor). Where **As** is included in an expression, the signal referenced by the **As** defines the specific measurement type, e.g., the expression `<Measure nominal="pk_pk 10 V" As="rmsSig" />` defines a measurement to calculate the peak-to-peak value of the rmsSig measurement. This facility is intended for use with multiple measurements and is best described by using an example. Consider the following:

```
<RMS name="rmsSig" samples="5">
<Measure nominal="pk_pk 10 V" As="rmsSig" samples="3">
```

This multiple measurement provides a set of three results (samples="3") in which the peak-to-peak value of a set of five rms measurements (`<RMS name="rmsSig" samples="5">`) is calculated.

The **As** property is used in a different way in an intrinsic measurement (such as RMS) from in a generic measurement (such as **Measure**). In intrinsic measurements, it is used to identify the type of signal being measured, whereas in generic measurements, the reference signal is used as part of the measurement of the attribute or signal with the least rms error deviation. The following examples show the effect of including an **As** property:

```
<Measure nominal="trms 10 mV" .../> measures the trms of the signal.
<Measure nominal="trms 10 mV" As="SquareWaveSig" .../> measures the trms of the deviation
of the input signal from a signal named SquareWaveSig.
<RMS As="SquareWaveSig" .../> measures the trms of an input signal that is expected to be the
same as a signal named SquareWaveSig.
```

NOTE—SquareWaveSig is defining a square wave signal and takes the form `<SquareWave name="SquareWaveSig" ampl="trms 10 mV" .../>`.

The following examples show the use of **As** in several measurements with different qualifiers. In these examples, the **As** refers to the type of signal expected. In practice, the name of a signal would be used that described a signal of that type:

- <Measure As="Sinusoid" attribute="amplitude" /> calculates the amplitude of a sinusoidal wave with the least rms error from the input waveform.
- <Measure As="Sinusoid" /> provides the “error aberration” between the input waveform and the defined **As** signal.
- <Measure As="SQUARE\_WAVE" nominal="trms"/> returns the value of the rms error.
- <Measure As="SQUARE\_WAVE" nominal="inst"/> returns the instantaneous error value.
- <Measure As="SQUARE\_WAVE" nominal="inst" samples="1000"/> returns an array of instantaneous error values. If the error is subtracted from the input, the result is the reference waveform.
- <Measure As="SQUARE\_WAVE" nominal="av"/> returns the average error value.
- <Measure As="SQUARE\_WAVE" nominal="pk"/> returns the peak error value.
- <Measure As="SQUARE\_WAVE" nominal="pk\_pk"/> returns the peak-to-peak error value.

Where the reference signal (defined by **As**) utilizes ranges to constrain the allowed value of its attributes, the generic Measure shall consider only results that are within the ranges of possible signals associated with the constrained attributes, and only these signals shall be compared in determining the least rms value in arriving at a measurement value.

#### B.6.4.13 Decode ::Sensor

- a) *Definition*—**Decode** converts a stream of bits represented by events back into data information.

Active, Digital H → 1

Inactive, Digital L → 0

The TriSate/Digital Z is not measured

- b) *Attributes*

**datatype** <string>—The base type of the encoded data and the associated measurement’s resultant variant’s datatype. (default = 0)

**encoding** <string>—Identifies the character set to be used to convert the encoded bit stream into a string message. This attribute allows alternative character set mappings and code pages to be applied. (default = CP-1252)

**stepsize** <Physical>—Shall specify the size of the encoding step (default = 1.0)

**offset** <Physical>—Shall specify the offset to be applied to the decoded value after decoding. This represents the value of encoded zero(0) (default = 0)

- c) *Description*—**Decodes** the digital stream and converts it via a stream of bits into a required (datatype). Each measurement is collected in the measurements property and the last value read is available through the measurement attribute. The UL, LL, or nominal value can be used where the physical value shall be initially converted to measurement type prior to any comparison.

The use of *stepsize* and *offset* apply the following expression such that each data value (x) is decoded from the encoded value e(x).

$$x = e(x) * \textit{stepsize} + \textit{offset}$$

**B.6.5 Control ::SignalFunction**

- a) *Definition*—**Control** is a class of signal that modifies the signal model depending on the *Select* value. The **Control** inputs (**In**) represent the various signal alternatives available, which are selected in turn as the selector changes; and, if no **Gate** is provided, an output corresponding to the selected input is produced. If a **Gate** is provided, the output is produced only when the Gate event becomes Active.

NOTE—The multi-input behavior differs from the default described in B.3.2.

- b) *Attributes*—Not applicable  
c) *Description*

**B.6.5.1 SelectIf ::Control**

- a) *Definition*—**SelectIf** converts a single channel event stream into a physical signal.  
b) *Attributes*  
**Selector** <Signal>—the digital event stream.  
c) *Description*—**SelectIf** cycles through its **In** signals starting from **In(1)** on each subsequent change of the selector event state between Low and High. If the selector enters the X state (tri-state), the output is gated off. If the selector enters the Z state (no signal), the previous output continues. Once started, the initial state of selector is considered Inactive, and the initial signal selected for output is **In(1)**. Prior to starting, the output is considered to be in the ZRep (No Signal) state.

As an example, when there are two Inputs, the selector event state Inactive corresponds to **In(1)**, and Active corresponds to **In(2)**.

**B.6.5.2 SelectCase ::Control**

- a) *Definition*—**SelectCase** converts a multichannel digital stream into a physical signal.  
b) *Attributes*  
**Selector** <Signal>—the digital event stream.  
**mask** <int>—selector channel mask; identifies which channels of the selector shall be used to select the input. The value can be expressed as a decimal number (e.g., 13) or a hexadecimal number (e.g., 0xC) to identify the binary pattern, e.g., “1101” selects channels 1, 3 & 4. A value of 0 implies all channels. (default = –1)  
c) *Description*—**SelectCase** selects the Input that corresponds to the masked **Selector** value (Active/Inactive state). The mask is converted into a bit mask and applied (as an And function) over the **Selector** digital stream, where channel 1 is LSB. The resulting pattern is converted to a one based index to select the corresponding input signal.

When the resultant **Selector** value exceeds the number of inputs, a Z state (tri-state) output signal is provided. When the selector channel is Active, it is considered '1' and Inactive is considered '0'.

**B.6.5.3 Encode ::Control**

- a) *Definition*—**Encode** provides a basic digital signal as a stream of bits derived from the data information, where the bit value 0 is represented by the event state InActive or digital L and bit value 1 is represented by the event state Active or digital H. The tri-state/Digital Z/gated Off are all identical and cannot generally be deduced from the data information.

b) *Attributes*

**data** <any>—The information to be streamed, or the URI identifying location of information. The type of the <any> defines the default datatype. When using XML descriptions, the attribute datatype shall be used. (default = empty)

**width** <int>—The number of output channels. Zero indicates that the number of outputs is the minimum required to represents a complete data symbol. (default = 0)

**repetition** <int>—the number of times the data is output. Zero indicates that the sequence is repeated indefinitely. (default = 1 )

**datatype** <string>—Used to define the base type of the encoded data. (default = "")

**encoding** <string>— Identifies the character set used to translate a string messages to character codes for transmission. This attribute allows alternative character set mappings and code pages to be applied. (default = CP-1252)

**stepsize** <Physical>—Shall specify the size of the encoding step (default = 1.0)

**offset** < Physical >—Shall specify the offset to be removed before encoding. This represents the value which encodes to zero(0) (default = 0)

- c) *Description*—**Encode** allows any data representation to be packaged and streamed as a sequence of parallel bits. It turns messages such as “Hello World” into a bit stream represented by events where the event state represents the individual bit state. Since the width attribute does not necessarily need to match the data information type, the data are converted into a stream of bits and each bit assigned to consecutive channels so that channel 1 is the LSB and the channel number corresponding to the **channelWidth** is the MSB. In this way, the following are equivalent:

```
data="11010100" datatype="xs:boolean"
data="HHLHLHLL" datatype="digitalString"
data="C4" datatype="hex"
data="212" datatype="byte"
data="Å" datatype="char" encoding="Windows Western"
```

Table B.7 shows the different bit streams that the same block of data provides depending on the values selected for the **width** and **repetition** attributes.

**Table B.7—Bit streams for the same data with different attributes**

Attributes	Channel	Bit stream
data = "11010100" width = "4" repetition = "2"	1 (LSB)	0101 →
	2	1010 →
	3	0101 →
	4 (MSB)	0101 →
data = "11010100" width = "8" repetition = "2"	1 (LSB)	00 →
	2	00 →
	3	11 →
	4	00 →
	5	11 →
	6	00 →
	7	11 →
	8 (MSB)	11 →
data = "11010100" width = "2"	1 (LSB)	0111 →
	2 (MSB)	0001 →
data = "11010100" width = 3 repetition = "3"	1 (LSB)	00101110 →
	2	01110001 →
	3 (MSB)	10001011 →
data = "11010100" width = "3"	1 (LSB)	Z10 →
	2	001 →
	3 (MSB)	011 →

Each pattern is delivered when **In** goes Active (1,H), or when the **In** is in the Active state whenever the **Change** notification event is received or the **Gate** signal goes Active. This provides for both pull and push data operation.

When the **data** attribute is empty, the data symbol output value represents the instantaneous value at the input signal in SI units converted to the datatype.

A **datatype** value of “Physical(<unit\_symbol>)” can be used to convert a Physical signal stream into discrete values represented by the <unit\_symbol>. Any <unit\_symbol> allowed by Physical can be used appropriate to the input signals type.

If the input becomes tri-state (Z) or Gated Off (X), the output is correspondingly tri-stated(Z) or Gated off(X).

The use of **stepsize** and **offset** apply the following expression to each data value (x) such that the final encoded value e(x) transmitted represents the closest possible value to

$$e(x) = (x - \text{offset}) / \text{stepsize}$$

Digital signals are slightly different in that their values do not represent physical values; rather, they take on event states that can be converted into data type values and thus converted to Physical values

#### B.6.5.4 Transform::Control

- a) *Definition*—**Transform** allows the signal model to use a “variable” signal definition as a template parameter. **Transform** acts like a template where the behavior is replaced with the signal referenced to the **As** signal.

b) *Attributes*

**mapping** <string> - the mapping attribute provides a user defined mapping represented by an set of attribute name-value pairs “required attribute name” “instance name, value or expression” (default=’’)”)

**As** <SignalFunction>—reference to a signal model to be used within the template model

c) *Description*

Transform provides a placeholder for a signal model to provide a variable signal function when creating a compound signal model. If used in a TSF, any signal defined as part of the “As”, provides the interface and “shape” of the signals that can be used with the TSF.

In a signal model the *As* signal function replaces the transform with the same inputs and output.

The *mapping* property is used to identify properties that the *As* signal function shall support and their instance values, this is defined in terms of other property attributes or expressions. Alternative required properties (and values) can be defined by using the ‘|’ character e.g. *frequency|period prf* which would indicate the *As* signal function shall have a *frequency* or *period* attribute and it shall have the value associated with the local *prf* property. Attribute name-value pairs are whitespace separated, single ‘ ’ or double “ ” quotes may be used as delimiters of values which are empty or may contain spaces.

The use of the *mapping* attribute allows for TSFs to be used which have different interface and attribute names, and maps the required attribute name to the instance attribute value.

#### B.6.5.5 InverseTransform::Control

a) *Definition*—**InverseTransform** allows the signal model to use a “variable” signal definition as a template parameter. InverseTransform acts like a template where the behavior represents the pseudo inverse transform of the *As* signal.

b) *Attributes*

**mapping** <string> - the mapping attribute provides a user defined mapping represented by an set of attribute name-value pairs “required attribute name” “instance name, value or expression” (default=’’)”)

**As** <SignalFunction>—reference to a signal model component whose inverse is to be used within the template model

c) *Description*

**InverseTransform** provides a placeholder for a signal model to provide a variable signal when creating a compound signal model. If used in a TSF, any signal defined as part of the “As”, provides the interface and “shape” of the signals that can be used with the TSF. See Transform description above.

The behavior is described as a pseudo inverse transform of the *As* signal

Inverse Transforms are defined in the same way as the generic measurement using a measurement the input attribute, such that the output of the Inverse Transform would be the input signal without the effect of the *As* signal. When the *As* signal represents some form of transformation the following is considered to hold true, and represent the identity or passthrough function.

i) Signal → Transform (As=’x’) → input → Inverse Transform (As=’x’) → Signal

When the *As* signal represents some source function (with no input) the following is considered to hold true and represents a signal with no value

ii) Transform (As=’x’) → input → Inverse Transform (As=’x’) → No Signal value

When the Inverse Transform is applied to signal which has a component of the Transform, the resultant signal is the input signal without the transform component

iii) Signal + Transform ( $As='x'$ )  $\rightarrow$  input  $\rightarrow$  Inverse Transform ( $As='x'$ )  $\rightarrow$  Signal

In these examples the key point to observe is that the *As* signal model shall not fully define the actual operation, it shall be described with possible ranges of values to be considered. Given an input signal the properties of both the *As* signal model and the resultant output Signal are calculated such that any other set of values would yield a larger rms error when compacted to the actual input. The RMS error can be calculated by applying the *As* signal model to the InverseFunction output **Signal**, then compare (subtract) the input signal and calculate the RMS value which is considered as the error. The best result is defined to be solution which provides the minimum RMS error. Note: There may not be a unique solution.

As an example consider our *As* signal model to be a Sinusoid, however the input is a SquareWave. This represents InverseFunction described by iii) above. Considering the SquareWave as sum of the odd harmonics 1,3,5,7 etc the solution shall be a signal without the 1<sup>st</sup> harmonic and only containing the other harmonics 3, 5, 7 etc, By calculating an RMS error between the input and various Sinusoids the least rms error is obtained when the Sinusoid represents the 1<sup>st</sup> harmonic of the input signal.

#### B.6.5.6 Channels ::Control

- d) *Definition*—**Channels** combines multiple signals into a single multiple channel signal, and allows signal functions to operate across input channels rather than signals.
- e) *Attribute*

**channelNames** <pinString>—list of channel names associated with the channels.

**As** <SignalFunction>—reference to a Signal representing a signal model component to be inserted into the model

NOTE 1—The channel name follows the same syntax as UUT pin names, but is chosen by the user for convenience and is not necessarily related to real UUT pin names, e.g., names may be chosen to indicate function such as “Reset” or “Enable.”

NOTE 2—If a channel name is selected that is the same as a UUT pin name, the UUT pin name takes precedence, and the channel then refers to that pin name.

- f) *Description*—**Channels** allows SignalFunctions to work across channels. For each input Signal, It separates each channel into a single channel signal and then combines them into a linear set of channels, optionally identified by the attribute *channels*, to form a multiple channel output. All signal phase information is maintained.

When *As* is provided, each input’s intermediate single channel signal is used as separate inputs into the *As* SignalFunction the outputs of each input signal are then combined. Multiple input signals create multiple channel outputs.

As an example a multi channel input and where *As* refers to a Sum, the resultant output shall be a single channel output whose value is the sum across each channel. By contrast If there were two multi channel inputs the output would be a two channel output with channel one being sum across input-1’s channels and second channel being the sum across input-2’s channels

Gate and Sync are passed to the *As* signal, where no *As* exists Gating off turns all output channels to the Z state. Prior to the first **Sync**, there is no value on any channel (Null). Subsequent **SynCs** have no effect.

Figure B.58 Represents an example where the Channels template is using "Sum" SignalFunction. It has two multi channel inputs, Channel 1 of Sig A is summed with Channel 2 of Sig A, and Channel 1 of Sig B is summed with Channel 2 of Sig B. e.g.

```
<Channels As="Sum">
  <In name='A'>
    <Sinusoid amplitude="1V" frequency="1kHz"/>
    <Constant amplitude="1500mV" />
  </In>
  <In name='B'>
    <Constant amplitude="750mV" />
    <Constant amplitude="-250mV" />
  </In>
</Channels>
```

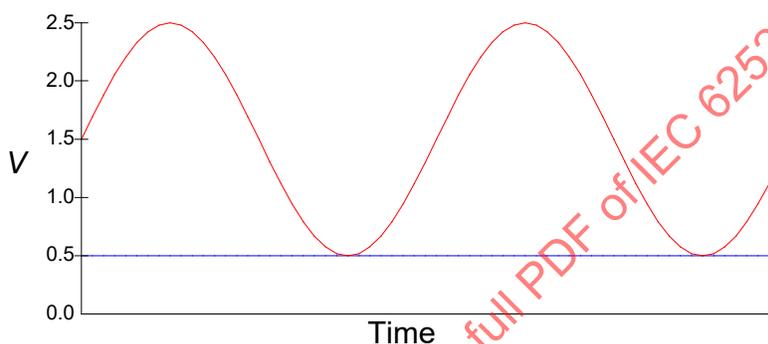


Figure B.58—Channel Summation example

### B.6.6 Digital ::SignalFunction

- Definition*—**Digital** is a class of signal that represents one of two values (which are sometimes called by names such as true and false, low and high, 1 and 0, etc.) and which can be gated off into a tri-state. A digital signal is a collection of one or more event signals, which can be converted into a range of physical signals.
- Attributes*—Not applicable
- Description*—**Digital** may have a number of representations. This standard defines two: digital signal and control signal. A digital signal is an abstract representation of the values that are encountered in engineering design; these values are defined more precisely in B.6.5.1 and B.6.5.2. A control signal is an analog signal whose value varies between low and high thresholds. Control signals are useful for translating to various logic families.

Digital signals are unique in that their values do not take on physical values; rather, they take on enumeration values that represent physical values.

#### B.6.6.1 SerialDigital<type: Physical> ::Digital

- Definition*—**SerialDigital** provides a (digital) control signal. These signals may take the form of a low signal, a high signal, or no signal (i.e., high impedance). They are defined by the characters L, H, Z, and X (“do not care”) in a character string.

b) *Attributes*

**data** <digitalString>—string containing characters H, L, Z, which identify digital state, or X as a “do not care” mask.

**period** <Time>—internal digital clock rate.

**logic\_H\_value** <Physical>—analog logic high (or logic 1).

**logic\_L\_value** <Physical>—analog logic low (or logic 0).

**pulseClass** <enumPulseClass>—pulse class type. (default = NRZ)

c) *Description*—The characters represent the digital signals as follows:

H logic high (or logic 1)

1 logic high (or logic 1)

L logic low (or logic 0)

0 logic low (or logic 0)

Z high impedance (absence of logic signal)

X unknown or indeterminate logic level

, delimiter between blocks

; delimiter between blocks

The digitalString comprises a list of digital characters separated with delimiters. Each comma ',' or semicolon ';' is treated as a delimiter between blocks.

The digitalString may also include whitespace characters (namely, space, new-line, carriage-return, line-feed, and tab). These whitespace characters are available for formatting purposes to make the data more readable. They are ignored when the digitalString is processed.

Where an external clock is provided at **In**, this becomes the clock used as the digital clock rate. The internal clock defined by the period attribute is not used.

The attribute **pulseClass** may take one of the following pulse class types:

NRZ nonreturn to zero

RZ return to zero

R1 return to one

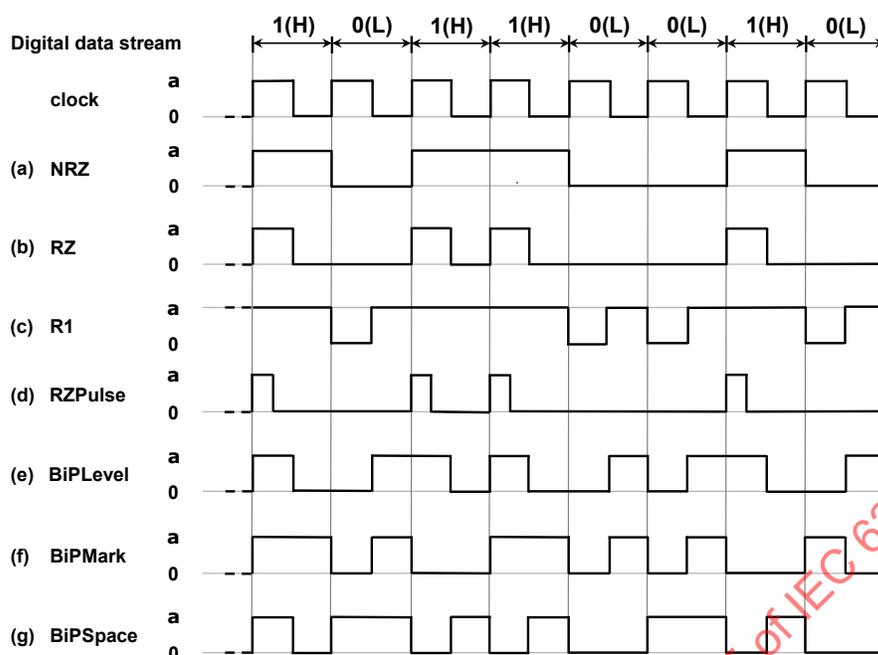
RZPulse Pulse return to zero

BiPLevel Bi-phase Level

BiPMark Bi-phase Mark pulse 0

BiPSpace Bi-phase Space pulse 1

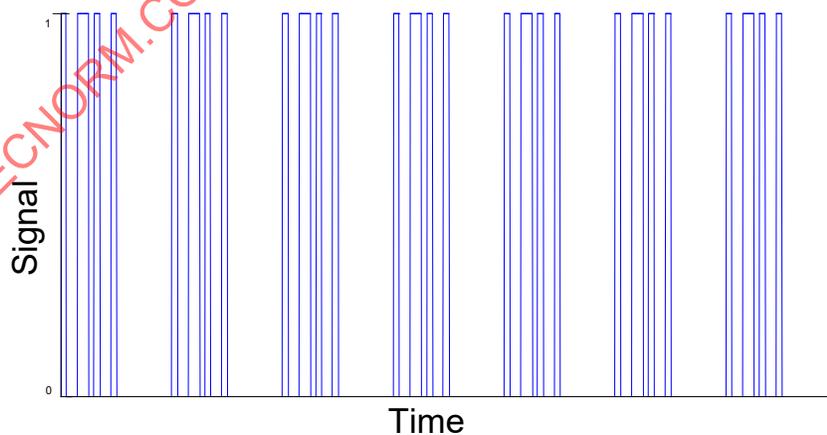
Figure B.59 illustrates the various pulse class types that may be used with the **pulseClass** attribute. At the top of the diagram is the digital data stream that is conveyed by the signal in each pulse class, i.e., the pattern "HLHLLHL" or "10110010".



**Figure B.59—Pulse class types used with the attribute pulseClass**

The values on the left of the diagram indicate the amplitude that the physical signal takes while transmitting the data. In the simplest case, NRZ or nonreturn to zero, the nonzero amplitude represents a logic one or High, and the zero amplitude represents a logic 0 or Low. Other pulse classes involve amplitude transitions or have both positive.

Figure B.60 shows a serial digital sequence where data = "HLLHHLHZZHL" is sent at a digital clock rate of 20 Hz. The sequence is synchronized and repeated periodically.



**Figure B.60—SerialDigital**

**B.6.6.2 ParallelDigital<type: Physical> ::Digital**

- a) *Definition*—**ParallelDigital** provides parallel streams of [digital] control signals. These signals may take the form of a low signal, a high signal, or no signal (i.e., high impedance). They are represented by the characters L, H, Z, and X in an array of character strings. The output is multichanneled, and the number of channels is defined by the width of digital statements.

- b) *Attributes*

**data** <digitalString>—each string String containing characters H, L, Z, which identify digital state, or X as a “do not care” mask.

**period** <Time>—internal digital clock rate.

**logic\_H\_value** <Physical>—analog logic high (or logic 1).

**logic\_L\_value** <Physical>—analog logic low (or logic 0).

**pulseClass** <enumPulseClass>—pulse class type. (default = NRZ)

- c) *Description*—The characters represent the digital signals as follows:

H	logic high (or logic 1)
1	logic high (or logic 1)
L	logic low (or logic 0)
0	logic low (or logic 0)
Z	high impedance (absence of logic signal)
X	unknown or indeterminate logic level
,	delimiter between blocks
;	delimiter between blocks

The digitalString comprises a list of digital characters separated with delimiters. Each comma ',' or semicolon ';' is treated as a delimiter between blocks. Each block represents a separate parallel step.

The digitalString may also include whitespace characters (namely, space, new-line, carriage-return, line-feed, and tab). These whitespace characters are available for formatting purposes to make the data more readable. They are ignored when the digitalString is processed.

Where an external clock is provided at In, this becomes the clock used as the digital clock rate. The period attribute is not used.

The attribute **pulseClass** may take one of the following pulse class types:

NRZ	nonreturn to zero
RZ	return to zero
R1	return to one
RZPulse	Pulse return to zero
BiPLevel	Bi-phase Level
BiPMark	Bi-phase Mark pulse 0
BiPSpace	Bi-phase Space pulse 1

See B.6.6.1 for a description of the pulse class types.

Figure B.61 shows a parallel digital sequence where data = "[HLHLZH, LHLHHL, LLHHLZ]" is sent with a period of 30 μs.

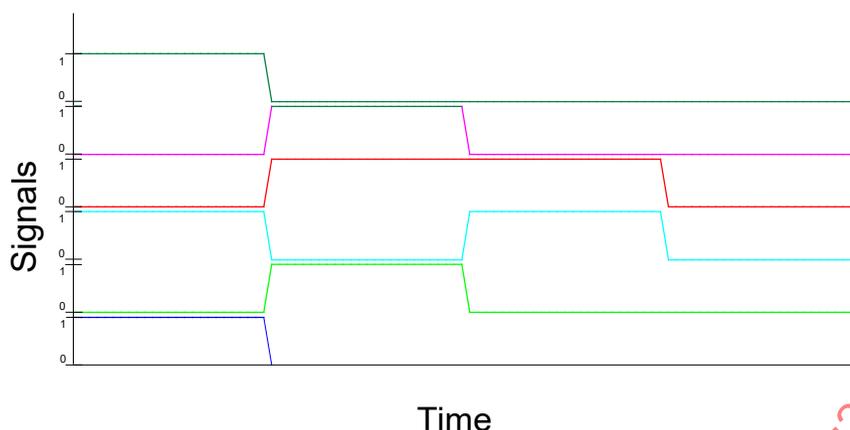


Figure B.61—ParallelDigital

### B.6.7 Connection ::SignalFunction

- a) *Definition*—**Connection** is the base class that collects signals into multiple channels.
- b) *Attributes*  
**channelWidth** <int>—maximum number of channels to be connected. (default = 0, unbounded)
- c) *Description*—**Connections** collect signal channels and allow them to be mapped onto real pins such as UUT pins. **Connection** subclasses are used to attach real pins names.

Unless otherwise stated, pin attributes may contain multiple pin names. This technique is analogous to using multiple **Connections**, each with a single pin name.

In the following connection definitions, the attribute value <pinString> is defined as follows:

A pinString may comprise one or more pin names or an expression that provides one or more pin names.

A pin name shall be a contiguous string of characters that may include alphanumeric, hyphen, and underscore characters. Pin names may not include a comma, semicolon, or whitespace character (namely, space, new-line, carriage-return, line-feed, and tab).

Multiple pin names are delimited by one or more whitespace, comma, or semicolon characters.

NOTE—For the **DigitalBus pins** attribute, a comma or semicolon is used as a channel delimiter (see B.6.7.14).

#### B.6.7.1 TwoWire ::Connection

- a) *Definition*—**TwoWire** is a two-wire connection in which the **hi** terminal represents the hot, or live, side of a circuit and the **lo** terminal represents the cold, or return, side of the circuit.
- b) *Attributes*  
**channelWidth** = 1  
**hi** <pinString>  
**lo** <pinString>
- c) *Description*

### B.6.7.2 TwoWireComp ::Connection

- a) *Definition*—**TwoWireComp** is a two-wire connection in which the true terminal represents the true signal for a differential digital signal and the comp terminal represents the complement signal for the differential digital signal.
- b) *Attributes*
  - channelWidth** = 1
  - true** <pinString>
  - comp** <pinString>
- c) *Description*

### B.6.7.3 ThreeWireComp ::Connection

- a) *Definition*—**ThreeWireComp** is a three-wire connection in which the true terminal represents the true signal for a differential digital signal, the comp terminal represents the complement signal for the differential digital signal, and the lo terminal represents a ground, or screen, connection.
- b) *Attributes*
  - channelWidth** = 1
  - true** <pinString>
  - comp** <pinString>
  - lo**<pinString>
- c) *Description*

### B.6.7.4 SinglePhase ::Connection

- a) *Definition*—**SinglePhase** is a two-wire connection in which terminal **a** represents the live connection to one phase of a one-phase (or more) circuit and the terminal **n** represents the neutral connection to the circuit.
- b) *Attributes*
  - channelWidth** = 1
  - a** <pinString>
  - n** <pinString>
- c) *Description*

### B.6.7.5 TwoPhase ::Connection

- a) *Definition*—**TwoPhase** is a three-wire connection in which terminal **a** represents the live connection to one phase of a two-phase (or more) circuit, terminal **b** represents the live connection to the second phase of a two-phase (or more) circuit, and terminal **n** represents the neutral connection to the circuit.
- b) *Attributes*
  - channelWidth** = 2
  - a** <pinString>

**b** <pinString>

**n** <pinString>

c) *Description*

#### B.6.7.6 ThreePhaseDelta ::Connection

a) *Definition*—**ThreePhaseDelta** is a three-wire connection in which terminal **a** represents the live connection to one phase of a three-phase circuit, terminal **b** represents the live connection to the second phase of a three-phase circuit, and terminal **c** represents the live connection to the third phase of a three-phase circuit. There is no neutral connection to the circuit.

b) *Attributes*

**channelWidth** = 3

**a** <pinString>

**b** <pinString>

**c** <pinString>

c) *Description*

#### B.6.7.7 ThreePhaseWye ::Connection

a) *Definition*—**ThreePhaseWye** is a four-wire connection in which terminal **a** represents the live connection to one phase of a three-phase circuit, terminal **b** represents the live connection to the second phase of a three-phase circuit, terminal **c** represents the live connection to the third phase of a three-phase circuit, and terminal **n** represents the neutral connection to the circuit.

b) *Attributes*

**channelWidth** = 3

**a** <pinString>

**b** <pinString>

**c** <pinString>

**n** <pinString>

c) *Description*

#### B.6.7.8 ThreePhaseSynchro ::Connection

a) *Definition*—**ThreePhaseSynchro** is a three-wire connection for use with the three-stator outputs of a Synchro.

b) *Attributes*

**channelWidth** = 3

**x** <pinString>

**y** <pinString>

**z** <pinString>

c) *Description*—Terminals **x**, **y**, and **z** represent the stator terminals S1, S2, and S3. The stator is connected in a delta format, and the output voltages are developed between **x** and **y**, **y** and **z**, and **x** and **z**.

#### B.6.7.9 FourWireResolver ::Connection

- a) *Definition*—**FourWireResolver** is a four-wire connection for use with the four-stator terminals of a Resolver.
- b) *Attributes*  
**channelWidth** = 2  
**s1** <pinString>  
**s2** <pinString>  
**s3** <pinString>  
**s4** <pinString>
- c) *Description*—Terminals **s1** and **s3** are used for the sine output, and terminals **s2** and **s4** are used for the cosine output.

#### B.6.7.10 SynchroResolver ::Connection

- a) *Definition*—**SynchroResolver** consists of up to four connections for use with the rotor terminals of a Synchro or Resolver.
- b) *Attributes*  
**channelWidth** = 1 or 2 (default = 2)  
**r1** <pinString>  
**r2** <pinString>  
**r3** <pinString>  
**r4** <pinString>
- c) *Description*—In many applications, only two terminals (i.e., **r1** and **r2**) are required for the R1 and R2 excitation connections of a Synchro or a Resolver unit.

#### B.6.7.11 Series ::Connection

- a) *Definition*—A **Series** connection **via** is used when only one connection is required at the test subject.
- b) *Attributes*  
**channelWidth** = 1  
**via** <pinString>
- c) *Description*—This connection is used for series signals (such as the application or measurement of current) where only one terminal is connected to the test subject.

#### B.6.7.12 FourWire ::Connection

- a) *Definition*—**FourWire** is a four-wire connection in which the **hi** terminal represents the hot, or live, side of a circuit; the **lo** terminal represents the cold, or return, side of the circuit; **hiRef** represents a terminal for a reference associated with the **hi** terminal; and the **loRef** represents a terminal for a reference associated with the **lo** terminal.

- b) *Attributes*
- channelWidth** = 1
  - hi** <pinString>
  - lo** <pinString>
  - hiRef** <pinString>
  - loRef** <pinString>
- c) *Description*—This connector is intended for use where the UUT requires four pins for what is effectively a two-wire type of connection, e.g., the UUT has power connections with sense terminals that need to be identified separately from the force terminals.

#### B.6.7.13 NonElectrical ::Connection

- a) *Definition*—**NonElectrical** is a connection for use with nonelectrical signals (such as the connection of fluids and gasses).
- b) *Attributes*
- channelWidth** = 1
  - to** <pinString>
  - from** <pinString>
- c) *Description*—The terminals **to** and **from** are both used where a fluid flows to and from the test subject. Either terminal may be used on its own if the fluid passes only one way (to or from the test subject).

#### B.6.7.14 DigitalBus ::Connection

- a) *Definition*—**DigitalBus** is a connection comprising one or more terminals. One terminal is used for each simultaneous (i.e., parallel) digital data channel.
- b) *Attributes*
- pins** <pinString>—List of pin names associated with the digital channels.
- c) *Description*—The number of parallel connections is specified by the <**channelWidth**> of the signal. Each pin name is associated with its corresponding channel. Ground or signal return connections may be added after the active channel pins. The last ground pin shall be used to return any remaining channels without a specified signal return pin. If no return pin is specified, a common return is assumed.

Each channel is delimited by a single comma or semicolon character, e.g., for a two channel system (**channelWidth** = 2), the pinString “PL1-1, PL1-2 SK1-2, GND” indicates that channel 1 uses connection pin PL1-1, channel 2 uses connection pins PL1-1 and SK1-2, and the common return pin is GND.

## Annex C

(normative)

### Dynamic signal descriptions

#### C.1 Introduction

This annex describes the dynamic interactions of basic signal components (BSCs) in signal models and test signal framework (TSF) components and what happens when they are programmed in any native carrier language through their control interface.

Signal descriptions can use both static and dynamic signal definitions. This standard defines all actions available through the control interface, using the API defined in the interface definition language (IDL), and provides actions that are deterministic. In order to provide a consistency to dynamic signal descriptions, the following concepts are introduced:

- a) A signal model defines a signal.
- b) Signal models have a single state. The state attributed to the signal model shall be the state of its output **Signal**.
- c) Signal models synchronized by different events represent signals that exist in different time frames.

To allow future implementers the maximum scope, the interactions are described only with reference to **Signal** state changes (i.e., events) and method calls (i.e., actions). Any carrier program attempting to use **Signal** state changes to synchronize shall be non-deterministic because the actual software notification of the **Signal** state change can only happen after the event.

In order that multiple BSCs can describe a single signal without ambiguity, this standard defines the interactions between the **SignalFunction** and **Signal** objects. A signal model consisting of more than one component describes one signal, e.g., a damped sinusoid signal has sinusoidal and “exponential decay” components, but is only one signal. Because the damped sinusoid example defines one signal, the notion of having one of its components running without the other cannot happen. The dilemma is that, in a dynamic signal model, the test program via the API can start either component, that is, start the sinusoidal component or the exponential decay component. Rather than attempt to forbid certain control actions, the dynamic behavior is defined so that all cases are semantically described. This standard specifies that starting any component within a signal model shall start the whole signal model and, therefore, activate the whole signal.

Starting different components may lead to transient differences even though the stable signals shall be the same.

This standard describes dynamic behavior by describing what happens to individual components when events and actions happen. This approach provides a very low level view of the components’ interaction, but does not provide an overall description; such big-picture descriptions have to be inferred on a case-by-case basis. This approach allows the standard to be used to describe more complex and varied scenarios without considering each action sequence in detail. On the down side, there may be many scenarios that are undesirable or meaningless to a test system, e.g., having a signal that goes nowhere and is not connected to any test locations. In these cases, this standard does not make sure such signals have a useful purpose, but does provide behavior that is deterministic.

Unlike static signals, dynamic signal descriptions can have their signal model definitions changed via the API, e.g., changing an attribute value or being connected to another signal model. These changes are buffered up and become the signal's next settings; the signal holds the next set of signal characteristic until the signal is requested to change. At that point, all the changes for the pending settings are applied together, and the signal has its new characteristics. Any subsequent changes become the next settings, and so on.

## C.2 Basic classes

### C.2.1 ResourceManager

**ResourceManagers** are resource managers that are used to create either single signal objects or signal models, using the **Require** method, within a native carrier program.

The **ResourceManager** is the only directly creatable class object specified by this standard. The minimum number of **ResourceManagers** that a valid system can have is one.

The use of different **ResourceManagers** within a native carrier program allows concurrent support of different test environments, e.g., an intermix of **ResourceManagers** that represent simulation environment and automatic test equipment (ATE) subsystem environments all within the same test program.

#### C.2.1.1 Runtime method

**Require** (SignalDescriptor [,UniqueId] )—The **Require** method provides the signal components or signal component models.

- a) SignalDescriptor is a string with one of the following:
  - 1) The name of a signal class, optionally followed by a comma separated attribute value pair
  - 2) An extensible markup language (XML) static signal model description as prescribed in Annex I
  - 3) An XML element derived from the SignalFunctionType defined in Annex I
  - 4) A URL that references a static signal model description as prescribed in Annex I
- b) UniqueId is an optional VARIANT value providing a unique signal identifier that may be used internally by the underlying implementation.

#### C.2.1.2 Comments

The **SignalDescriptor** is text string that is either a name corresponding to the BSC type or an XML description conforming to Annex I for XML signal schema description. Examples of valid signal description for a constant voltage are as follows:

- "Constant"
- "Constant(Voltage)"
- "Constant(Voltage, Time)"
- "<Signal Out='dc'> <Constant name='dc' amplitude='2V+5%'></Signal>"
- "<Constant name='dc' amplitude='2V+5%'>"
- "Constant amplitude 2V+5%"

— "file://lookhere.xml"

*Example 1*—signal class name

```
Set myAM = Std.Require("urn:IEEE-1641:2022:STDTSFLib\AM_SIGNAL")
myAM.car_ampl.value = "100kHz"
myAM.car_freq.value = "1V"
myAM.mod_freq.value = "660Hz"
myAM.mod_ampl.value = "0.5V"
```

*Example 2*—XML static signal model description

```
Set myAM = Std.Require(
  "<Signal Out='amSig'>" &
  "<tsf716:AM_SIGNAL name='amSig'" &
  "car_ampl='1V' car_freq='100kHz' mod_freq='660Hz' mod_ampl='0.5V' />"
  & "</Signal>")
```

The **UniqueId** is reserved for implementations, e.g., used by a test procedure language (TPL) processor to help its runtime system determine which resource is best to supply the signal. It allows helper information to be held in a common way.

## C.2.2 Signal

The **Signal** class provides a control interface for all signals and events described by BSCs. This control interface is used to describe signal models and to control input signals. Because TSFs are built up using BSCs, the same rules apply equally to TSFs.

### C.2.2.1 Runtime properties

**state**—The state property reflects the state of the signal or event being described by the associated signal model. The state property of the **Signal** interface is a read-only, bindable, and “edit request” property and cannot be changed directly through the **Signal**’s control interface. The values that the state property can take are as follows:

*Stopped*—The *Stopped* state indicates that the signal is in a generalized reset condition, i.e., no signal activity is present. Thus a *Stopped Signal* can represent either no signal at all or a signal from an allocated resource that has not been activated or triggered. All **Signals** initiate to the *Stopped* state.

*Paused*—The *Paused* signal is waiting to be triggered into the *Running* state by an external event. A *Paused* signal does not yet exist, but all the necessary resources have been acquired and prepared and are awaiting the final on or go event.

*Running*—The *Running* signal is active and exists as a signal or gated event stream. A *Running Signal* is measurable and available for use.

**Channels**—Provides a collection of individual channels, where each channel supports the **Signal** interface.

**SignalFunction**—Returns a reference to the **SignalFunction** object (TSF or BSC) representing the signal model which is defining the signal.

### C.2.2.2 Runtime methods

**Stop([timeout=0])**—The **Stop** method resets, disconnects, or turns off any *Paused* or *Running* **Signal** and thereby frees any associated signal resources. Following a successful **Stop**, the state of the **Signal** shall become *Stopped*.

**Run([timeout=0])**—The **Run** method sets up, starts, connects, or turns on a *Stopped* signal. Following a successful **Run**, the state of the **Signal** is *Paused* and subsequently becomes *Running*. The **Run** method on an *Running* or *Paused* **Signal** shall reinitialize the **Signal** to its value at time  $t = 0$ .

**Change([timeout=0])**—The **Change** method initiates the **Signal** to its next setting. If no further settings are pending, **Change()** indicates that the current **Signal** is finished and no longer needed. This knowledge allows the source BSCs to change the signal to the next available setup. If no further signal conditions are available, **Change()** resets the signal to the *Stopped* state.

The **timeout** value indicates the minimum time in milliseconds that the method call shall wait for the **Signal** to enter the expected **Signal** state. If the signal enters the expected state, the method returns the IDL HRESULT success code `S_OK` (0x00000000L). If the signal does not enter the expected state, the method returns the IDL HRESULT success code `S_FALSE` (0x00000001L).

A method called with a **timeout** value of zero is asynchronous.

NOTE—A timeout shall not be included in a signal definition. Waiting for a UUT response is part of test definition and waiting for an instrument response is a system function.

### C.2.2.3 Comments

The **Signal** class is provided to define interactions between BSCs and defines the control API for test programs. The way that BSCs use and create **Signals** is local to the implementation and defined by the common interfaces. These interactions are characterized in two ways:

- a) Notification of changes through **Signal** state changes
- b) Requests for **Signals** to alter their state through **Signal** method calls

It is important not to confuse the terms *Running* and active. A **Signal** or an **Event** may be active only while it is *Running*. This is best illustrated with examples. Consider a **Clock** BSC with its output connected to the input of a **NotEvent** BSC: While the BSCs are *Running*, the output of the **Clock** is alternately active and inactive, while the output of the **NotEvent** is alternately inactive and active, i.e., one is active while the other is inactive (see Figure C.1).

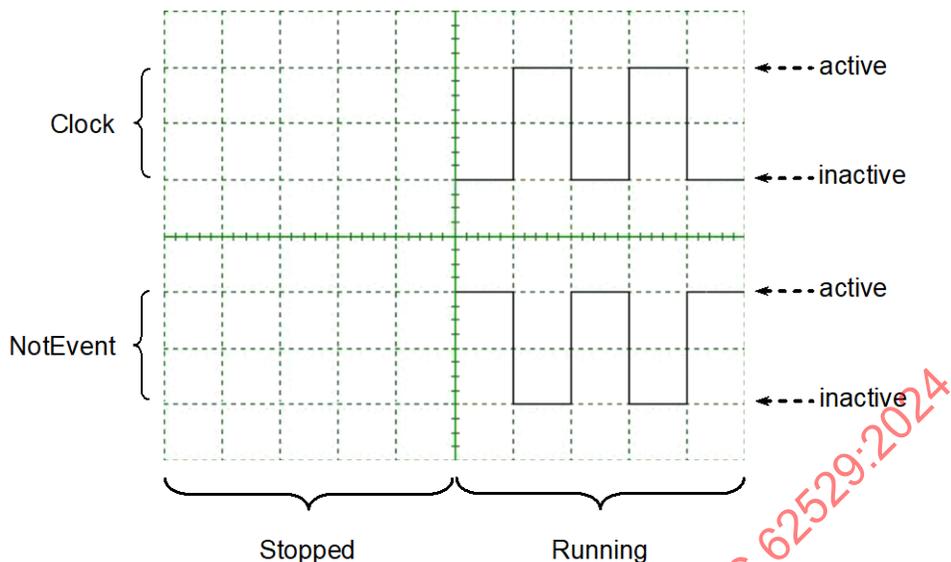


Figure C.1—Illustration of Events changing between active and inactive

Consider a simple **Signal** producing a sinusoidal output, which is being gated on and off via its Gate input. The **Signal** shall be *Running* and active continuously, but at the output there shall be a **Signal** or no **Signal** according to the state of the Gate input (see Figure C.2).

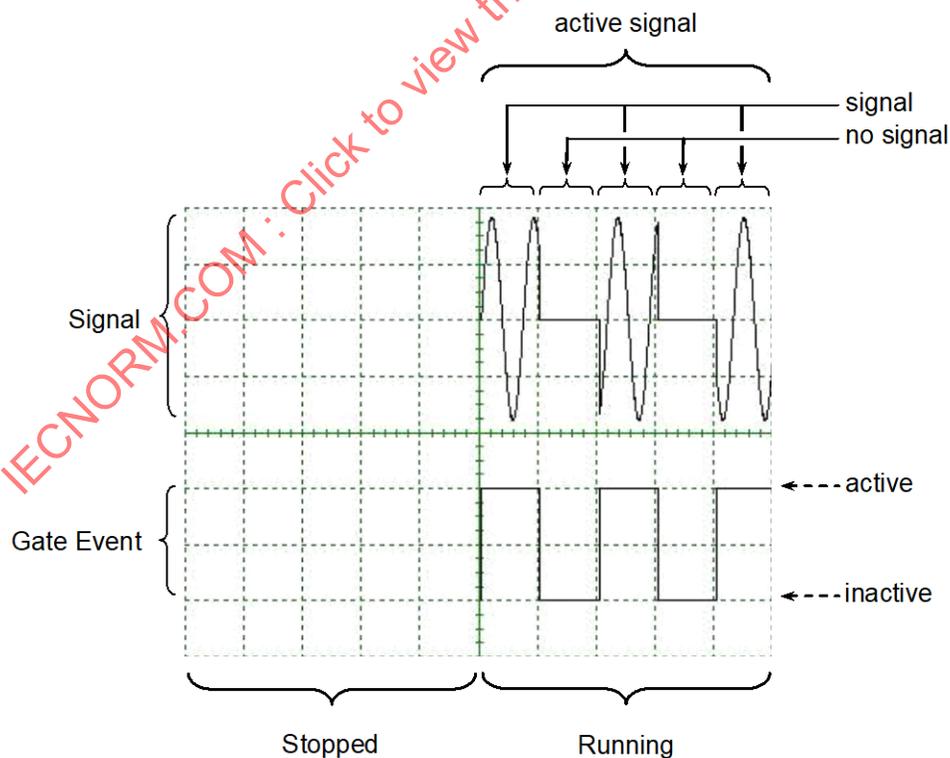


Figure C.2—Effect of a Gate Event on a Signal

In general, an analog **Signal** is regarded as active when it enters the *Running* state, while an Event is active when it is *Running* and its operational state is active.

The **Sync** reference considers the event occurrence when the **Sync Signal** becomes active.

The **Gate** reference considers the event gated active while the **Gate Signal** is in the active state.

The *state* property of a **Signal** may reflect the actual state of the physical event such as *Stopped*, *Paused*, or *Running*. Provided that both the source and all sinks of the signal can be implemented in hardware, then the *state* property of the **Signal** does not have to reflect the physical *Paused* and *Running* states.

The **Signal** interface also provides an enumeration interface so that a native code program can enumerate through all of a **Signal**'s assigned BSCs.

*Example:*

```
For Each sf in SinusoidWave.Out
    'sf is an Basic Signal Component where the Out is assigned
Next
```

The **Signal Channels** property provides an enumeration interface so that a native code program can enumerate through all of a **Signal**'s individual channels. The channels property is empty if the signal is not multichannel.

*Example:*

```
For Each s in SinusoidWave.Out.Channels
    's is a Signal representing a single channel
    's.Channels.Count=0
Next
```

Individual channels signals can be named provided they have been identified by use of the **Connector** or **Channels** BSC, e.g., threePhase.Out.Channels.Item("a"). or threePhase.Out.Channels["a"].

### C.2.2.4 State diagram

Methods and states are interdependent. Calling a method indicates an intention for a state to change. See Figure C.3.

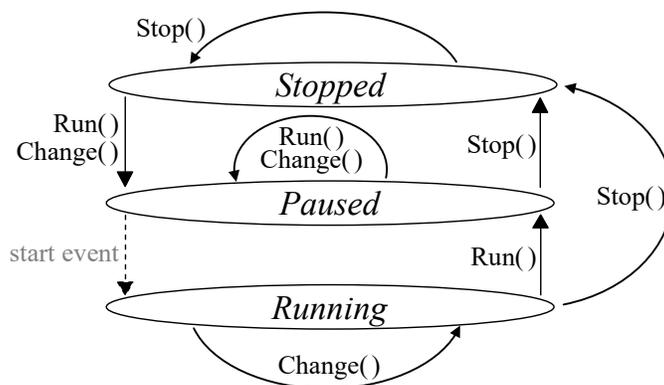


Figure C.3—Signal state changes

For example, when calling the **Run** method, the state of a **Signal** may never become *Running* if the **Sync** event never becomes active. The reason is that **Run()** tells the BSC associated with a **Signal** that a *Running* signal state is required. This knowledge in turn causes the BSC to call **Run()** on all its inputs and then on all its input events and returns with the **Signal** in the *Paused* or *Running* state. When the method returns with the **Signal** in the *Paused* state, the BSC is waiting for some internal event to become active. When this expected event becomes active, the signal becomes *Running*. **Run()** does not cause a **Signal** state to become *Running*, but it indicates that a *Running* signal is required.

All methods are asynchronous. A native test program, therefore, needs to monitor the **Signal** state or use a timeout value in the method call to determine when the signal has become *Running*.

### C.2.3 SignalFunctions

The SignalFunction operation is controlled both through its **Out Signal** interface methods and the state of any **In**, **Gate**, and **Sync Signals** and provides a two-way control mechanism.

The general behavior of a BSC is that the **Out Signal** state reflects the **In Signal** reference states.

- When the input **Signal** state is *Stopped*, the output **Signal** state is *Stopped*.
- When the input **Signal** state is *Paused*, the output **Signal** state is *Paused*.
- When the input **Signal** state is *Running*, the output **Signal** state is either *Running* or, if a **Sync** reference is assigned, *Paused* awaiting a **Sync** event to become active.

All BSCs have two input event references called **Sync** and **Gate**. These events affect the behavior of a BSC as follows:

- **Sync** unassigned—starts and then resumes when the **In Signal** enters the *Running* state.
- **Sync** assigned—restarted when the **Sync** event becomes active while an **In Signal** is *Running*.
- **Gate** unassigned—is operational while the **In Signal** is in the *Running* state.
- **Gate** assigned—is operational while the **Gate** and **In** event is in the *Running* state so that the signal's characteristics are available only while the **Gate** event is gated active.

When a signal re-starts, it repeats its operation from time zero, when it resumes it continues from where it was paused.

Any of the method calls, **Stop()**, **Change()**, or **Run()**, made on the **Out Signal** interface of the BSC shall affect the BSC behavior. The BSC shall in turn make similar calls on its input and event **Signal** references as follows:

**Stop**—The **Out Signal** proceeds to the *Stopped* state, and the BSC calls the **Stop** method of all of its assigned **In**, **Gate**, and **Sync Signals**.

**Run**—The **Out Signal** proceeds to the *Paused* state, and the BSC calls **Run()** on any unassigned **Signals**, then on all its assigned **In Signals** and then calls **Run()** on any assigned **Gate** and **Sync Signals**. This sequence allows any signal model to start by calling the **Run** method to act upon any on the signal interfaces.

**Change**—This method causes the **Signal**'s associated BSC to go on to its next internal function settings and calls **Change()** on all its input **Signals**.

A BSC may have pending settings that represent different signal characteristics. These characteristics may have been changes to signal attributes or built-in Control changes. Calling the Change or Run method shall cause the BSC to change its signal characteristics.

### C.2.3.1 Signal state diagram

As well as the methods affecting the state of a signal, the **In** and **Sync Signal** of a **SignalFunction** may also change the state of the BSC's **Out Signal**. In Figure C.4, the => symbol should be read as “enters the state,” e.g., **In => Paused** is read as “**In** enters the state *Paused*.”

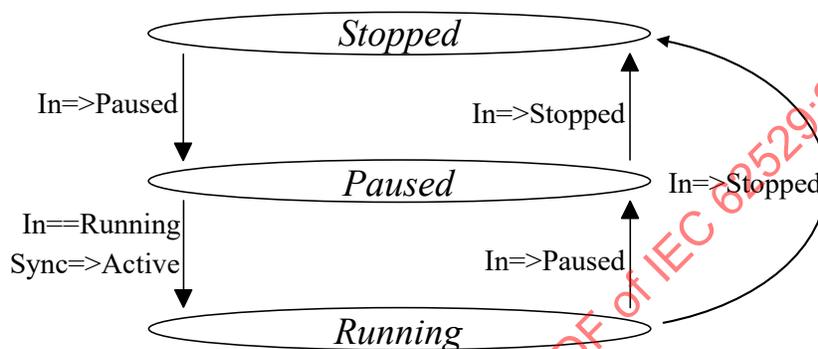


Figure C.4—In (Event) state changes

If a **Sync** event is unassigned and the Signal becomes *Paused*, then the **Signal** represents a continuous signal, and the **Signal** immediately becomes *Running*. If a **Sync** event is assigned, it represents a synchronized signal, and the **Signal** becomes *Paused* and is then driven *Running* when the **Sync** event becomes active.

When a Signal proceeds to a state, it enters each state in turn, e.g., *Running* -> *Paused* -> *Stopped*.

### C.2.3.2 Comments

The behavior of all BSCs is governed by the following rules:

- When any BSC is not directly referenced, i.e., nobody is using the BSC, it shall be immediately destroyed.
- When a BSC is destroyed and prior to its destruction, any **Out Signals** of the BSCs have the **Stop** method called, and the BSC waits until they have all become *Stopped*. **In Signals** of the BSCs are unassigned, and the **Change** method is called if all BSCs have finished with that **Signal**.
- When all assigned BSCs have finished with an **In Signal**, the **Change** method of the **In Signal** is called.
- When the output **Signal** is *Running* and a BSCs properties are altered, these properties represent its next signal settings that shall take effect when the **Out Signal Change** or **Run** method is next called.
- If the **Sync** reference is unassigned, the **Out Signal** shall enter *Running* from *Paused* immediately, when its Inputs become *Running*.

- If the **Sync** reference is assigned, then the **Out Signal** shall enter *Running* from *Paused* when the **Sync** event becomes active. If the **Sync** event becomes active again while the **Out Signal** is *Running*, then the **Out Signal** is synchronized to this **Sync** event, i.e., it starts again from time zero (T0). Once *Running*, the **Out Signal** state is not affected by the **Sync** state.
- Once the **Out Signal** is *Running*, the BSCs are operational only while the **Gate** reference event is active (gated on). In other words, once triggered, the **Out Signal** enters the *Running* state, but the signal is present only while the **Gate** event is gated on.
- When the **Out Signal** enters the *Paused* state from the *Stopped* state, it acquires any necessary resources and prepares the signal ready for output.
- When the **Out Signal** enters *Running* from *Paused*, the real signal is activated, and the output **Signal** state becomes *Running*.
- When the **Out Signal** enters *Paused* from *Running*, the real signal is deactivated, and the output **Signal** state becomes *Paused*.
- When the **Out Signal** enters *Stopped* from *Paused*, the resources are unprepared and released.
- If an attribute does not have a value available and has no default value, the **SignalFunction** waits in the paused state until a value is available.

The **Conn** property allows a user to specify connectivity of BSCs without any implied activation, which is implicit with the **In** property. **Conn** is used for a dynamic model, where the user wants to show connectivity of signals without any implied activation. All BSCs connected solely through the **Conn** property exist in separate time frame and have no implicit synchronization between them.

The **SignalFunction** also provides an enumeration property (*SignalFunctions*) so that a native code program can enumerate through all of the contained **SignalFunctions** of the signal model. BSCs and TSFs contain no accessible **SignalFunctions**. This feature returns accessible **SignalFunctions** only within anonymous TSFs or user-defined signal models, where the order of the **SignalFunctions** represents their position in their *SignalDescription*.

*Example:*

```
For Each sf in mySig.SignalFunctions
    sf.name 'sf is a SignalFunction within the mySig model
Next
```

An anonymous TSF also supports the interfaces of each of its sub items held within its *SignalFunctions*. The interface represents the first sub items within *SignalFunctions* supporting the interface.

Note: the order of the sub items within *SignalFunctions* is defined by the *SignalDescription* provided to the *Require* method. This order is important when there are duplicates as it is used to identify both the priority attribute names and interface, and the later duplicates become implicitly hidden.

### C.2.3.3 Runtime properties

- **SignalFunctions** collection of accessible components
- **name** is of type String
- **Out** is of type **Signal**.
- **In [(at=0)]** is of type reference to **Signal**.
- **Sync** is of type reference to **Signal**.

- **Gate** is of type reference to **Signal**.
- **Conn[(at=0)]** is of type reference to **Signal**.
- **attributes [(name="")]**– returns the named attribute or collection of attributes
- **pinsIn** is of type pinString.
- **pinsOut** is of type pinString.
- **pinsSync** is of type pinString.
- **pinsGate** is of type pinString.

The parameter “at” can be used to identify which signal input is being referenced, e.g., **In(1)**.

The Signal Function enumerating sub item **SignalFunctions** support the **Count** and **Item** properties, for example, in an anonymous TSF.

The Signal Function **attributes** property returns the named property or using default value ("" or null) returns the collection of named properties, where the collection represents all the attributes of the BSC, TSF or anonymous TSF and where the collection supports **Count** and **Item** properties.

*Example:*

```
For Each att in sig.attributes
  att.name   att is a named property within the sig model
Next
```

### C.3 Dynamic signal goals and use cases

The use of BSC or TSF components within the system is identical. This standard does not differentiate between the behavior of BSC and TSF components.

Using a TSF component within a user signal model is identical to using the internal signal model definition of the TSF within the same model. This equivalence means that packaging a signal model in a TSF does not change the behavior of the signal model.

Calling **Run()** on any **Signal** component of a signal model shall activate every **Signal** within the model so that the whole signal model including any **Gate/Sync** events becomes *Running*.

Calling **Change()** on any **Signal** component of a signal model does not cause **Sync/Gate** events to be initiated.

The **Run**, **Change**, and **Stop** methods may return before the **Signal** state has changed. The time at which the signal returns may be synchronized with the signal state change by using the timeout.

There is no implied phase relationship across a **Connection** object. If a connection object connects two signal models, then the signal models exist in two separate time frames.

An event synchronizing a TSF component has the effect of synchronizing all internal TSF signal model components.

An event gating a TSF component has the effect of gating the TSF output signal model components.

When two sub signal models exist in different time frames, activating the first signal model does not call **Run()** on the event of the second signal model.

The **Run** method makes sure that **Run()** is called on all inputs and **Gate** and **Sync** events.

The **Change** method calls **Change()** on all inputs, but does not affect any **Gate** or **Sync** events.

IECNORM.COM : Click to view the full PDF of IEC 62529:2024

## Annex D

(normative)

### Interface definition language (IDL) basic components

#### D.1 Introduction

The referenced IDL provides the common interface description for all the basic components described within this standard. The use of this IDL allows test programs written in native carrier languages to use a common interface and successfully use basic signal components (BSCs) regardless of which carrier environment is used, provided it supports IDL. The IDL can be compiled into a type library to support implementations of BSCs. All implementations should use the same IDL to provide compatibility between native carrier language test programs and different BSC implementations.

The standard requirement is that the interface used by the native carrier language shall be via use of the IDL or associated TypeLibrary. The native carrier language syntax and IDL import process shall govern the names of the classes and attributes used within the carrier language. As an example importing the IDL into Visual Studio shall convert attributes with parameters into equivalent method calls prefixed with "get\_" and "set\_" such that the Physical attribute "range(...)" becomes "get\_range(...)".

The IDL defines the types, interfaces, classes, methods, properties, and attributes used to support BSCs described in this standard.

NOTE—Annex D is a normative annex in that it provides the normative descriptions for the BSCs in IDL. Inclusion of this annex as normative does not mean that the BSCs may not be described in other interface languages.

#### D.2 IDL BSC library

The IDL BSC library is maintained at the IEEE Standards download site (see IEEE 1641™) <https://standards.ieee.org/downloads.html>.

## Annex E

(informative)

### Test signal framework (TSF) for C/ATLAS

#### E.1 Introduction

This annex provides an example TSF representing many of the signals defined in IEEE Std 716-1995 [B13] for the Common/Abbreviated Test Language for All Systems (C/ATLAS). It is provided so that a user may create test requirements (using this standard) equivalent to the requirements written using the C/ATLAS standard.

Not every signal (noun) and attribute (noun modifier) described in the C/ATLAS standard is covered by an equivalent in STD. If a user requires a signal or attribute not described in this annex, that signal may be created using the basic signal components (BSCs).

A diagram is provided with each signal to illustrate graphically the relationship between the BSCs and interface attributes that make up the signal. In order to reduce the amount of information included in each diagram, inputs to BSCs that are at zero or the default values are omitted.

#### E.2 TSF library definition in extensible markup language (XML)

Where examples are given, their static signal description is provided in XML. The information provided in Annex I, together with the detailed description of each TSF model in this annex, may be used to create the example TSF library for C/ATLAS that conforms to the XML Schema document defined in Annex I.

A complete XML instance document conforming to the requirements of this standard may be obtained from the IEEE Standards download site (see IEEE 1641™) <https://standards.ieee.org/downloads.html>

#### E.3 Interface definition language (IDL) for the TSF for C/ATLAS

##### E.3.1 Introduction

The IDL referenced in E.3.2 provides the common interface description for all the TSF models for the C/ATLAS example described within this annex. The use of this IDL allows test programs written in native carrier languages to use a common interface and successfully use TSF components regardless of which carrier environment is used, provided it supports IDL. The IDL can be compiled into a type library to support implementations of TSF for C/ATLAS components. All implementations that use these TSFs should use the same IDL to provide compatibility between native carrier language test programs and different BSC implementations.

The IDL defines the types, interfaces, classes, methods, properties, and attributes used to support BSCs described in this standard.

Where additional TSFs have been created for other test domains, they shall require a new IDL library so that they can be used by multiple programming environments. In such cases, the style and layout shown in the IDL referenced in E.3.2 can be used.

The IDL for the C/ATLAS TSF library is derived from the corresponding XML library.

### E.3.2 IDL for the TSF for C/ATLAS library

The IDL for the example ATLAS TSF library is maintained at the IEEE Standards download site (see IEEE 1641™) <https://standards.ieee.org/downloads.html>.

## E.4 AC\_SIGNAL<type: Current|| Power|| Voltage>

### E.4.1 Definition

A sinusoidal time-varying electrical signal. See Figure E.1.

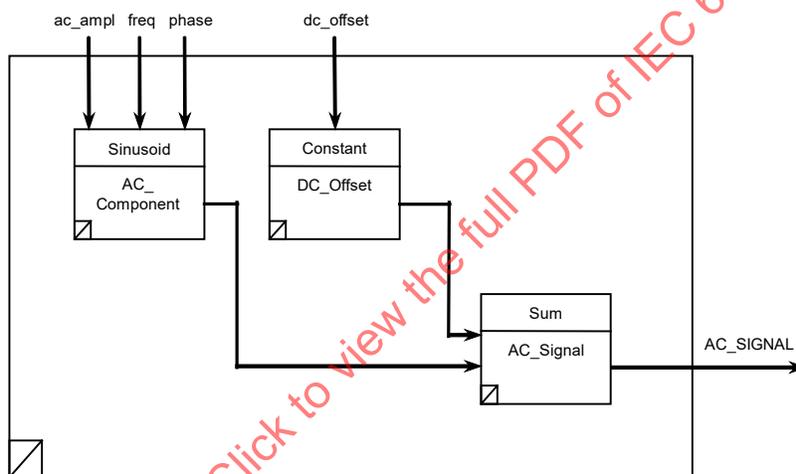


Figure E.1—TSF AC\_SIGNAL

### E.4.2 Interface properties

See Table E.1 for details of the TSF AC\_SIGNAL interface.

Table E.1—TSF AC\_SIGNAL interface

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 – 2π rad

### E.4.3 Notes

There are no special notes for this TSF.

### E.4.4 Model description

See Table E.2 for details of the TSF AC\_SIGNAL model.

**Table E.2— TSF AC\_SIGNAL model**

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	—	—

### E.4.5 Rules

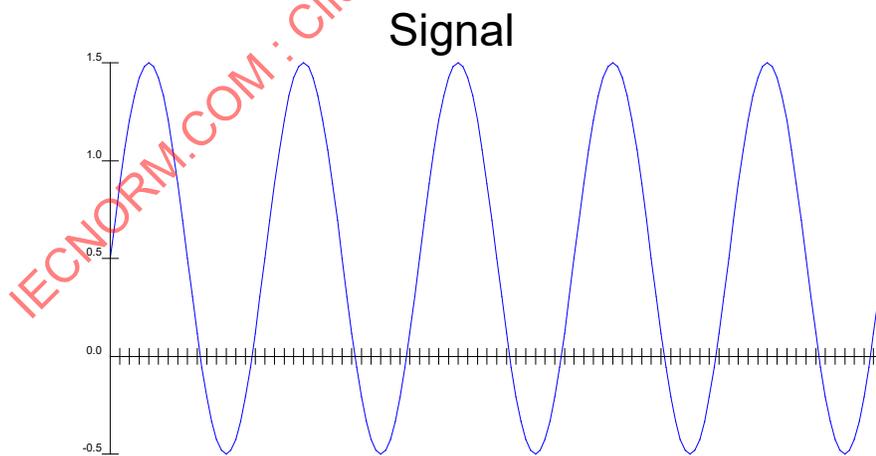
For this signal, the allowable types are Voltage, Current, and Power. All types need to be consistent. Thus for example, if ac signal amplitude is specified in volts, then the dc offset shall also be specified in volts.

### E.4.6 Example

See Figure E.2 for an example of AC\_SIGNAL.

*XML Static Signal Description:*

```
<AC_SIGNAL ac_ampl="1 V" dc_offset="0.5 V" freq="1000 Hz" />
```



**Figure E.2—AC\_SIGNAL example**

## E.5 AM\_SIGNAL

### E.5.1 Definition

A continuous sinusoidal wave (carrier) whose amplitude is varied as a function of the instantaneous value of a second wave (modulating). See Figure E.3.

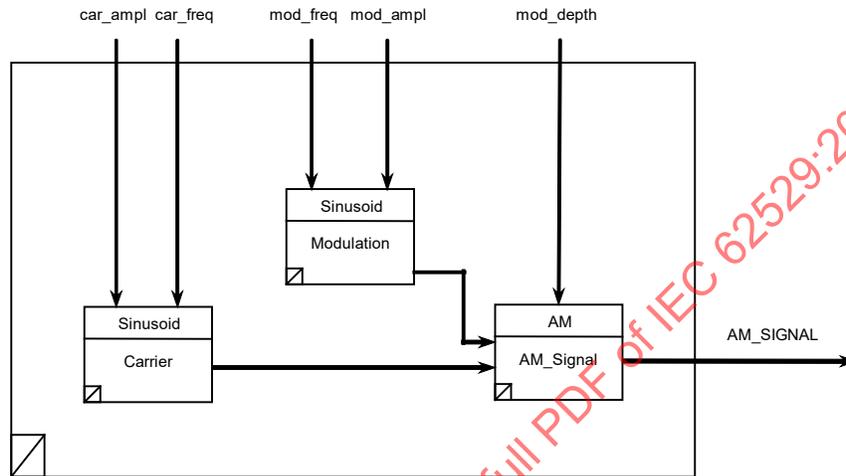


Figure E.3—TSF AM\_SIGNAL

### E.5.2 Interface properties

See Table E.3 for details of the TSF AM\_SIGNAL interface.

Table E.3—TSF AM\_SIGNAL interface

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Voltage	—	—
Carrier frequency	car_freq	Frequency	—	—
Modulation frequency	mod_freq	Frequency	—	—
Depth of modulation	mod_depth	Ratio	—	0 – 1
Modulation amplitude	mod_ampl	Voltage	1 V	—

### E.5.3 Notes

There are no special notes for this TSF.

### E.5.4 Model description

See Table E.4 for details of the TSF AM\_SIGNAL model.

**Table E.4—TSF AM\_SIGNAL model**

Name	Type	Terminal	Inputs	Output	Formula
AM_Signal	AM	Signal [Out]	—	AM_SIGNAL	—
		modIndex	mod_depth	—	—
		Carrier [In]	Carrier	—	—
		Signal [In]	Modulation	—	—
Carrier	Sinusoid	Signal [Out]	—	AM_Signal	—
		amplitude	car_ampl	—	—
		frequency	car_freq	—	—
		phase	—	—	0 rad
Modulation	Sinusoid	Signal [Out]	—	AM_Signal	—
		amplitude	mod_ampl	—	—
		frequency	mod_freq	—	—
		phase	—	—	0 rad

**E.5.5 Rules**

The output is given by the following equation:

$$e = E_c(1+m_aE_m\sin(\omega_m t))\sin(\omega_c t) \tag{E.1}$$

where

- $E_c$  is the carrier amplitude (unmodulated)
- $E_m$  is the modulation amplitude
- $m_a$  is the depth of modulation ( $\equiv$  modulation index)
- $\omega_m$  is  $2\pi \times$  modulating frequency
- $\omega_c$  is  $2\pi \times$  carrier frequency

**E.5.6 Example**

See Figure E.4 for an example of AM\_SIGNAL.

*XML Static Signal Description:*

```
<AM_SIGNAL car_ampl="1 V" car_freq="40 kHz" mod_freq="1 kHz" />
```

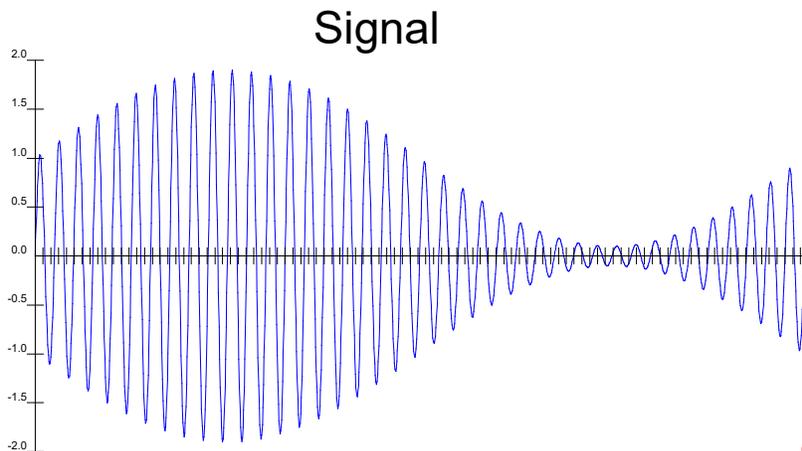


Figure E.4—AM\_SIGNAL example

## E.6 DC\_SIGNAL<type: Voltage|| Current|| Power>

### E.6.1 Definition

An unvarying electrical signal with an optional ac component. Figure E.5.

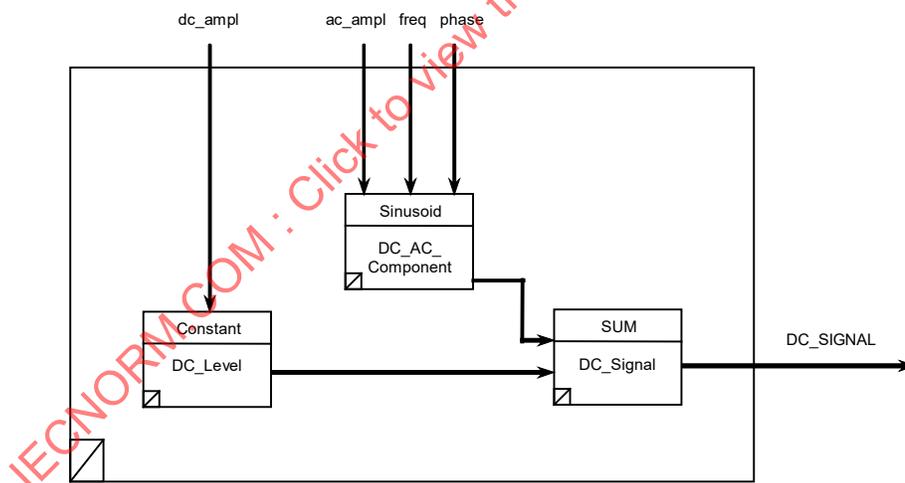


Figure E.5—TSF DC\_SIGNAL

### E.6.2 Interface properties

See Table E.5 for details of the TSF DC\_SIGNAL interface.

**Table E.5—TSF DC\_SIGNAL interface**

Description	Name	Type	Default	Range
DC level	dc_ampl	Physical	—	—
AC Component amplitude	ac_ampl	Physical	0	—
AC Component frequency	freq	Frequency	0 Hz	—
AC Component phase angle	phase	PlaneAngle	0 rad	0 – 2π rad

### E.6.3 Notes

There are no special notes for this TSF.

### E.6.4 Model description

See Table E.6 for details of the TSF DC\_SIGNAL model.

**Table E.6—TSF DC\_SIGNAL model**

Name	Type	Terminal	Inputs	Output	Formula
DC_Signal	Sum	Signal [Out]	—	DC_SIGNAL	—
		Signal [In]	DC_Level	—	—
		Signal [In]	AC_Component	—	—
DC_Level	Constant	Signal [Out]	—	DC_Signal	—
		amplitude	dc_ampl	—	—
DC_AC_Component	Sinusoid	Signal [Out]	—	DC_Signal	—
		amplitude	ac_ampl	—	—
		frequency	freq	—	—
		phase	phase	—	—

### E.6.5 Rules

For this signal, the allowable types are Voltage, Current, and Power. All types need to be consistent. Thus for example, if dc level is specified in volts, then the ac component amplitude shall also be specified in volts.

### E.6.6 Example

See Figure E.6 for an example of DC\_SIGNAL.

*XML Static Signal Description:*

```
<DC_SIGNAL name="DC_SIGNAL7" ac_ampl="0.03" dc_ampl="1" freq="50" />
```

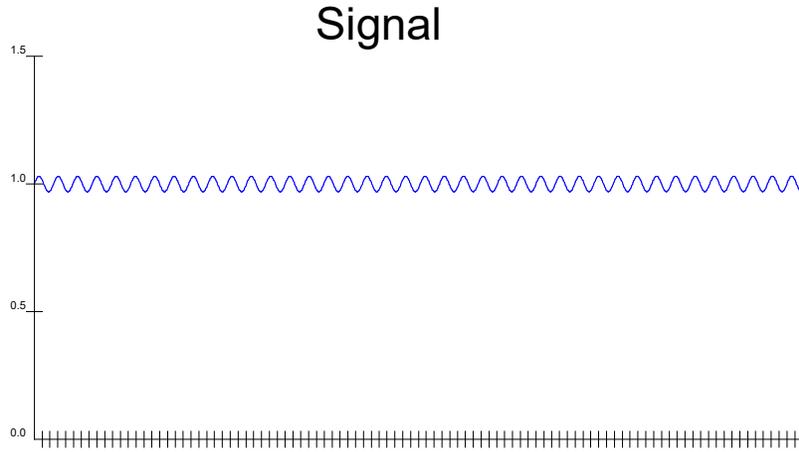


Figure E.6—DC\_SIGNAL example

## E.7 DIGITAL\_PARALLEL

### E.7.1 Definition

A parallel digital source that creates a digital logic signal in which the physical values for logic 1, logic 0, and high impedance data values are determined by the logic threshold values specified. See Figure E.7.

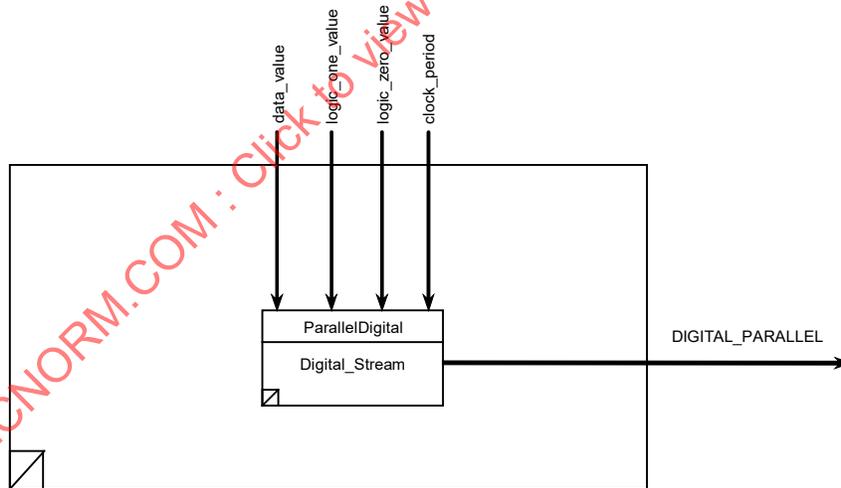


Figure E.7—TSF DIGITAL\_PARALLEL

### E.7.2 Interface properties

See Table E.7 for details of the TSF DIGITAL\_PARALLEL interface.

**Table E.7—TSF DIGITAL\_PARALLEL interface**

Description	Name	Type	Default	Range
Data value	data_value	digitalString	—	H L Z X
Clock period	clock_period	Time	—	—
Logic One level	logic_one_value	Voltage	—	—
Logic Zero level	logic_zero_value	Voltage	—	—

**E.7.3 Notes**

The width of the signal (and hence the minimum associated connection width) is implied by the number of logic elements in each array element.

The default condition for clock period (clock\_period = 0) denotes infinite time for static digital data.

**E.7.4 Model description**

See Table E.8 for details of the TSF DIGITAL\_PARALLEL model.

**Table E.8—TSF DIGITAL\_PARALLEL model**

Name	Type	Terminal	Inputs	Output	Formula
Digital_Stream	ParallelDigital	Signal [Out]		DIGITAL_PARALLEL	—
		data	data_value	—	—
		period	clock_period	—	—
		logic_H_value	logic_one_value	—	—
		logic_L_value	logic_zero_value	—	—

**E.7.5 Rules**

A high impedance is generated when the digital signal value character is Z, i.e., no digital signal is present.

A logic 1 (output voltage is equal to logic\_one\_value) is generated when the digital signal value character is H.

A logic 0 (output voltage is equal to logic\_zero\_value) is generated when the digital signal value character is L.

An unknown value cannot be generated by the digital source model. When the digital signal value character is X, the model may generate a logic 1 or a logic 0.

The output values are held at the defined levels for the duration of the clock\_period.

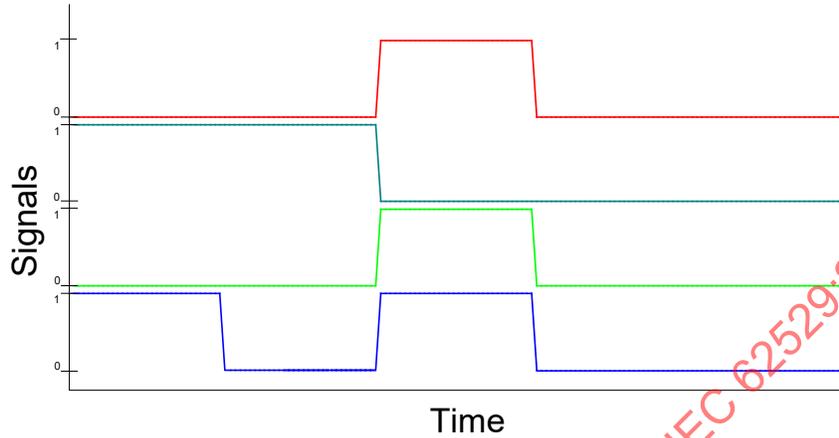
For this signal, the data values are transmitted via the parallel connections. Data received via these connections shall be available when the signal is used in a measurement.

**E.7.6 Example**

See Figure E.8 for an example of DIGITAL\_PARALLEL.

*XML Static Signal Description:*

```
<DIGITAL_PARALLEL data_value=' "HLHL", "LLHL", "HHLH"' clock_period="1 us" />
```

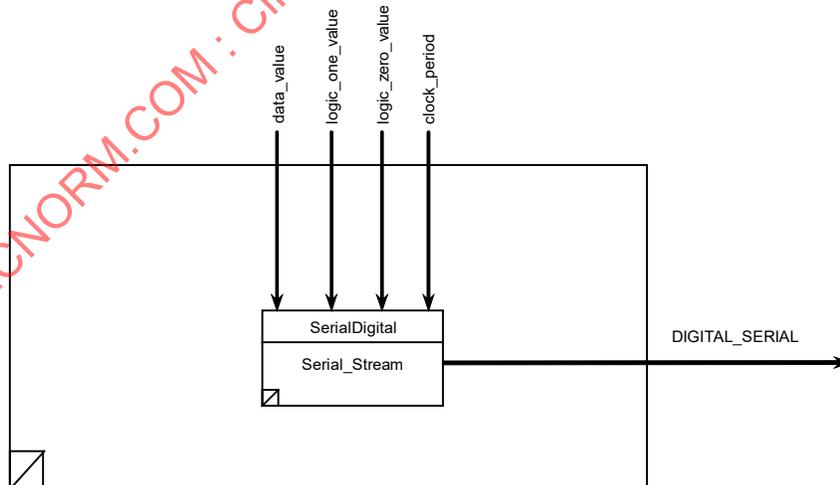


**Figure E.8—DIGITAL\_PARALLEL example**

## E.8 DIGITAL\_SERIAL

### E.8.1 Definition

A serial digital source that creates a digital logic signal in which the physical values for logic 1, logic 0, and high impedance data values are determined by the logic threshold values specified. See Figure E.9.



**Figure E.9—TSF DIGITAL\_SERIAL**

## E.8.2 Interface properties

See Table E.9 for details of the TSF DIGITAL\_SERIAL interface.

**Table E.9—TSF DIGITAL\_SERIAL interface**

Description	Name	Type	Default	Range
Data value	data_value	digitalString	—	H L Z X
Clock period	clock_period	Time	—	—
Logic One level	logic_one_value	Voltage	—	—
Logic Zero level	logic_zero_value	Voltage	—	—

## E.8.3 Notes

The default condition for clock period ( $\text{clock\_period} = 0$ ) denotes infinite time for static digital data.

The serial TSF deals only with serial data where the data value is conveyed as the value of the signal rather than any transition of the signal.

## E.8.4 Model description

See Table E.10 for details of the TSF DIGITAL\_SERIAL model.

**Table E.10—TSF DIGITAL\_SERIAL model**

Name	Type	Terminal	Inputs	Output	Formula
Serial_Stream	SerialDigital	Signal [Out]		DIGITAL_SERIAL	—
		data	data_value	—	—
		period	clock_period	—	—
		logic_H_value	logic_one_value	—	—
		logic_L_value	logic_zero_value	—	—

## E.8.5 Rules

A high impedance is generated when the digital signal value character is Z, i.e., no digital signal present.

A logic 1 (output voltage is equal to  $\text{logic\_one\_value}$ ) is generated when the digital signal value character is H.

A logic 0 (output voltage is equal to  $\text{logic\_zero\_value}$ ) is generated when the digital signal value character is L.

An unknown value cannot be generated by the digital source model. When the digital signal value character is X, the model may generate a logic 1 or a logic 0.

The output values are held at the defined levels for the duration of the  $\text{clock\_period}$ .

For this signal, the data value supplied is transmitted via the serial connections. Data received via the serial connections shall be available when the signal is used in a measurement.

### E.8.6 Example

See Figure E.10 for an example of DIGITAL\_SERIAL.

*XML Static Signal Description:*

```
<DIGITAL_SERIAL data_value="LZHLHHLHHHLLZL" clock_period="1 us"
    logic_one_value="3.6 V" logic_zero_value="-2.6 V" />
```

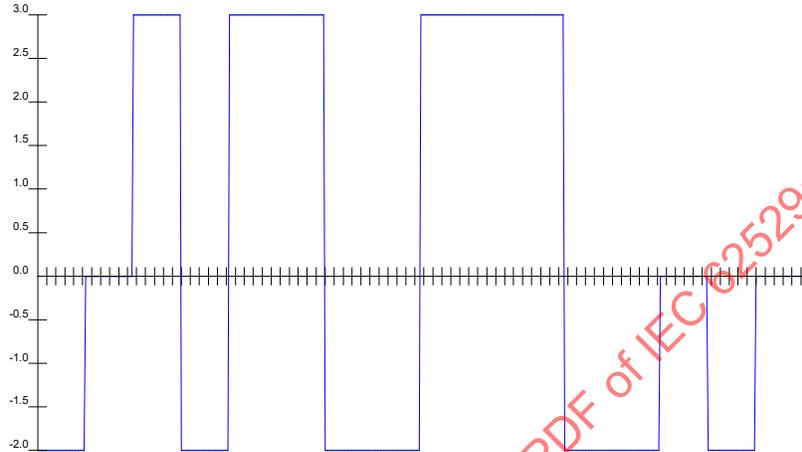


Figure E.10—DIGITAL\_SERIAL example

## E.9 DIGITAL\_TEST

### E.9.1 Definition

The digital test TSF uses both stimulus and response data together with the appropriate clock information to perform a bidirectional digital test. See Figure E.11.

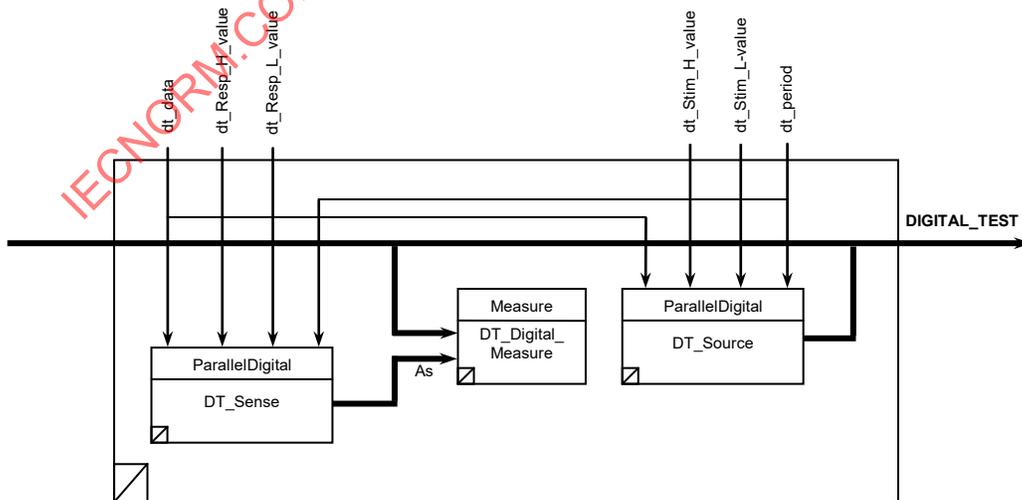


Figure E.11—TSF DIGITAL\_TEST

## E.9.2 Interface properties

See Table E.11 for details of the TSF DIGITAL\_TEST interface.

**Table E.11—TSF DIGITAL\_TEST interface**

Description	Name	Type	Default	Range
Clock Period	dt_Period	Time	—	—
Logic One level	dt_Stim_H_value	Physical	—	—
Logic Zero level	dt_Stim_L_value	Physical	—	—
Logic One level	dt_Resp_H_value	Physical	—	—
Logic Zero level	dt_Resp_L_value	Physical	—	—
Logic Data	dt_Data	digitalString	—	H L Z Xh l z x

## E.9.3 Notes

When using the DIGITAL\_TEST TSF, the stimulus and response logic levels should be provided, together with the digital clock period to define the characteristics of the signal waveform.

The default conditions for this TSF have no significance other than to provide an example of its use.

Stimulus data is defined using the following syntax:

- H for High or logic 1
- L for Low or logic 0.
- Z for Tri-state or high impedance.
- X for unspecified, usually implemented using the Z state.

Response data are defined using the following syntax:

- h for High or logic 1
- l for Low or logic 0.
- Z for Tri-state or high impedance.
- X for “do not care,” i.e., the value is not measured.

The DIGITAL\_TEST TSF deals only with data where the data value is conveyed as the value of the signal rather than any transition of the signal.

## E.9.4 Model description

See Table E.12 for details of the TSF DIGITAL\_TEST model.

**Table E.12—TSF DIGITAL\_TEST model**

Name	Type	Terminal	Inputs	Output	Formula
DT_Source	ParallelDigital	Signal [Out]	—	DIGITAL_TEST	—
		data	dt_Data	—	—
		period	dt_Period	—	—
		logic_H_value	dt_Stim_H_value	—	—
		logic_L_value	dt_Stim_L_value	—	—
DT_Sense	ParallelDigital	Signal [Out]	—	DT_Digital_Measure	—
		data	—	—	—
		period	dt_Period	—	—
		logic_H_value	dt_Resp_H_value	—	—
		logic_L_value	dt_Resp_L_value	—	—
DT_Digital_Measure	Measure	[Out]	—	—	—
		measuredVariable	—	—	—
		measurement	—	—	—
		measurements	—	—	—
		sample	—	—	—
		count	—	—	—
		gateTime	—	—	—
		nominal	—	—	—
		condition	—	—	—
		GO	—	—	—
		NOGO	—	—	—
		HI	—	—	—
		LO	—	—	—
		UL	—	—	—
		LL	—	—	—
Signal [As]	DT_Sense	—	—		
Signal [In]	DIGITAL_TEST	—	—		

**E.9.5 Rules**

A high impedance is generated when the digital signal value character is Z, i.e., no digital signal present.

A logic 1 (output voltage is equal to logic\_one\_value) is generated when the digital signal value character is H.

A logic 0 (output voltage is equal to logic\_zero\_value) is generated when the digital signal value character is L.

An unknown value cannot be generated by the digital source model. When the digital signal value character is X, the model may generate a high impedance, a logic 1 or a logic 0.

The output values are held at the defined levels for the duration of the clock\_period.

For this signal, the data value supplied is transmitted via the bidirectional connections. Data received via the bidirectional connections shall be available when the signal is used in a measurement.

### E.9.6 Example

See Figure E.12 for an example of DIGITAL\_TEST.

*XML Static Signal Description:*

```
<DIGITAL_TEST dt_data="HLLLLHlh, LLZHhLxx, HLzzHZZH" dt_period="1 us"
dt_Stim_H_value="5.0 V" dt_Stim_L_value="0.0 V" dt_Resp_H_value="3.5 V"
dt_Resp_L_value="0.5 V"/>
```

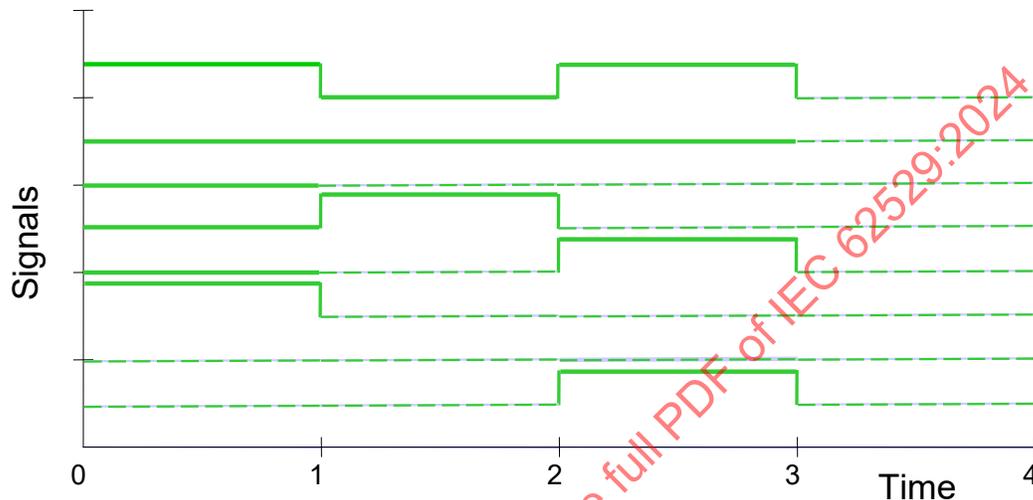


Figure E.12—DIGITAL\_TEST example

## E.10 DME\_INTERROGATION

### E.10.1 Definition

A radio aid-to-air navigation that provides distance information by measuring the time of transmission from an interrogator to a transponder and return. See Figure E.13.

The distance measuring equipment (DME) system is composed of a transponder in the ground base unit and an interrogator in the airborne unit. The interrogator on the aircraft emits a pulse signal that, once received by the DME transponder on the ground, starts a response sequence that sends a return pulse signal on a different (paired) channel to the aircraft. The aircraft equipment receives the response from the ground station, computes the elapsed time between interrogation and response, subtracts 50  $\mu$ s (to cover ground station processing time), and divides the result by 2. This result is then displayed on the DME indicator.

The DME operates on the ultra high frequency (UHF) band in the range of 962 MHz to 1213 MHz with a step of 1 MHz. The frequencies used by the interrogator are between 1025 MHz and 1150 MHz, and the transponder on the ground replies using two set frequencies: the first from 962 MHz to 1024 MHz and the second from 1151 MHz to 1213 MHz. The number of available frequencies is 252; therefore, there are 126 available channels. Each channel has 2 frequencies: one for interrogation and the other for the response from the ground station. On each pair of frequencies, the difference between the interrogator frequency and the response frequency is 63 MHz. For the channels between 1 and 63, the interrogator frequency is 63 MHz higher than the response frequency; and for channels from 63 to 126, the response frequency is 63 MHz higher than the interrogator frequency.

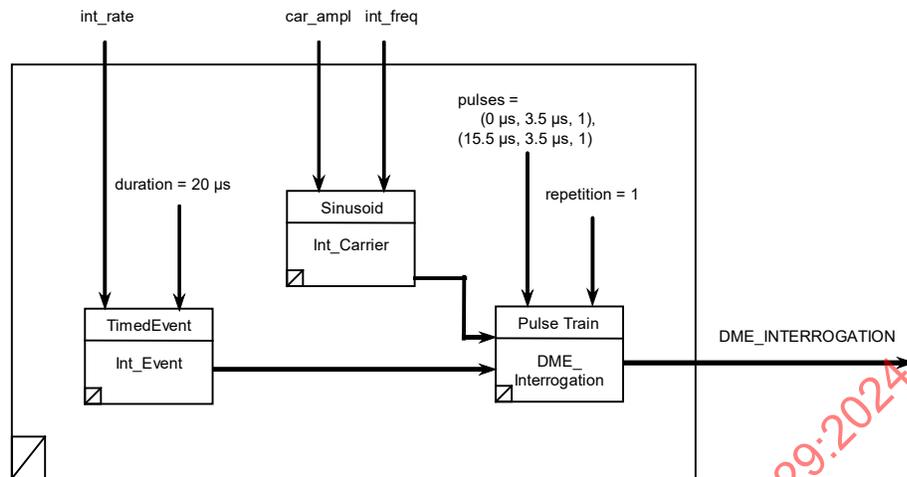


Figure E.13—TSF DME\_INTERROGATION

### E.10.2 Interface properties

See Table E.13 for details of the TSF DME\_INTERROGATION interface.

Table E.13—TSF DME\_INTERROGATION interface

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Voltage	—	—
Interrogator frequency	int_freq	Frequency	1025 MHz	1025 MHz – 1150 MHz
Interrogation rate	int_rate	Frequency	27 Hz	27 Hz   150 Hz

### E.10.3 Notes

This model has limited functionality. It does not provide for the variation of some of the parameters (such as the pulse timing and level). The model may be modified by the user to include such parameters in the interface properties.

### E.10.4 Model description

See Table E.14 for details of the TSF DME\_INTERROGATION model.

**Table E.14—TSF DME\_INTERROGATION model**

Name	Type	Terminal	Inputs	Output	Formula
DME_Interrogation	PulseTrain	Signal [Out]	—	DME_INTERROGATION	—
		pulses	—	—	(0 μs , 3.5 μs, 1), (15.5 μs, 3.5 μs, 1)
		repetition	—	—	1
		Signal [In]	Int_Carrier	—	—
		Sync[In]	Int_Event	—	—
Int_Carrier	Sinusoid	Signal [Out]	—	DME_Interrogation	—
		amplitude	car_ampl	—	—
		frequency	int_freq	—	—
		phase	—	—	0 rad
Int_Event	TimedEvent	Event [Out]	—	DME_Interrogation	—
		delay	—	—	0 s
		duration	—	—	20 μs
		period	—	—	(1/int_rate)
		repetition	—	—	0

**E.10.5 Rules**

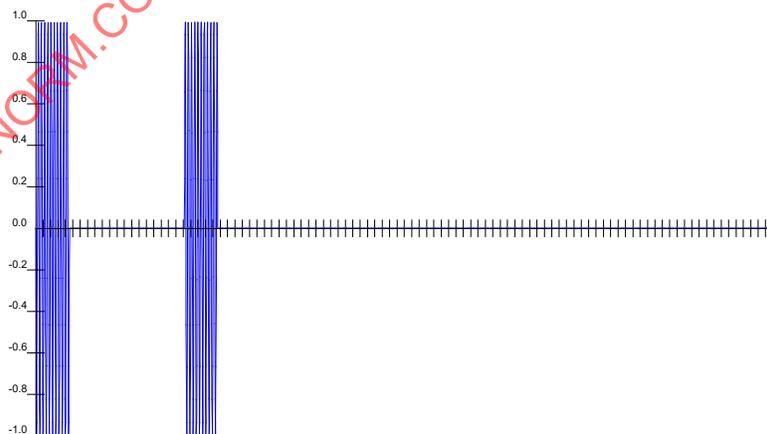
There are no special rules for this TSF.

**E.10.6 Example**

See Figure E.14 for an example of DME\_INTERROGATION.

*XML Static Signal Description:*

```
<DME_INTERROGATION name="DME_INTERROGATION6" int_freq="1050 MHz"
int_rate="150 Hz" />
```



**Figure E.14—DME\_INTERROGATION example**

## E.11 DME\_RESPONSE

### E.11.1 Definition

A radio aid-to-air navigation that provides distance information by measuring the time of transmission from an interrogator to a transponder and return. See Figure E.15.

The DME system is composed of a transponder in the ground base unit and an interrogator in the airborne unit. The interrogator on the aircraft emits a pulse signal that, once received by the DME transponder on the ground, starts a response sequence that sends a return pulse signal on a different (paired) channel to the aircraft. The aircraft equipment receives the answer from the ground station, computes the elapsed time between interrogation and response, subtracts 50  $\mu\text{s}$  (to cover ground station processing time), and divides the result by 2. This result is then displayed on the DME indicator.

The DME operates on the UHF band in the range of 962 MHz to 1213 MHz with a step of 1 MHz. The frequencies used by the interrogator are between 1025 MHz and 1150 MHz, and the transponder on the ground replies using two set frequencies: the first from 962 MHz to 1024 MHz and the second from 1151 MHz to 1213 MHz. The number of available frequencies is 252; therefore, there are 126 available channels. Each channel has 2 frequencies: one for interrogation and the other for the response from the ground station. On each pair of frequencies, the difference between the interrogator frequency and the response frequency is 63 MHz. For the channels between 1 and 63, the interrogator frequency is 63 MHz higher than the response frequency; and for channels from 63 to 126, the response frequency is 63 MHz higher than the interrogator frequency.

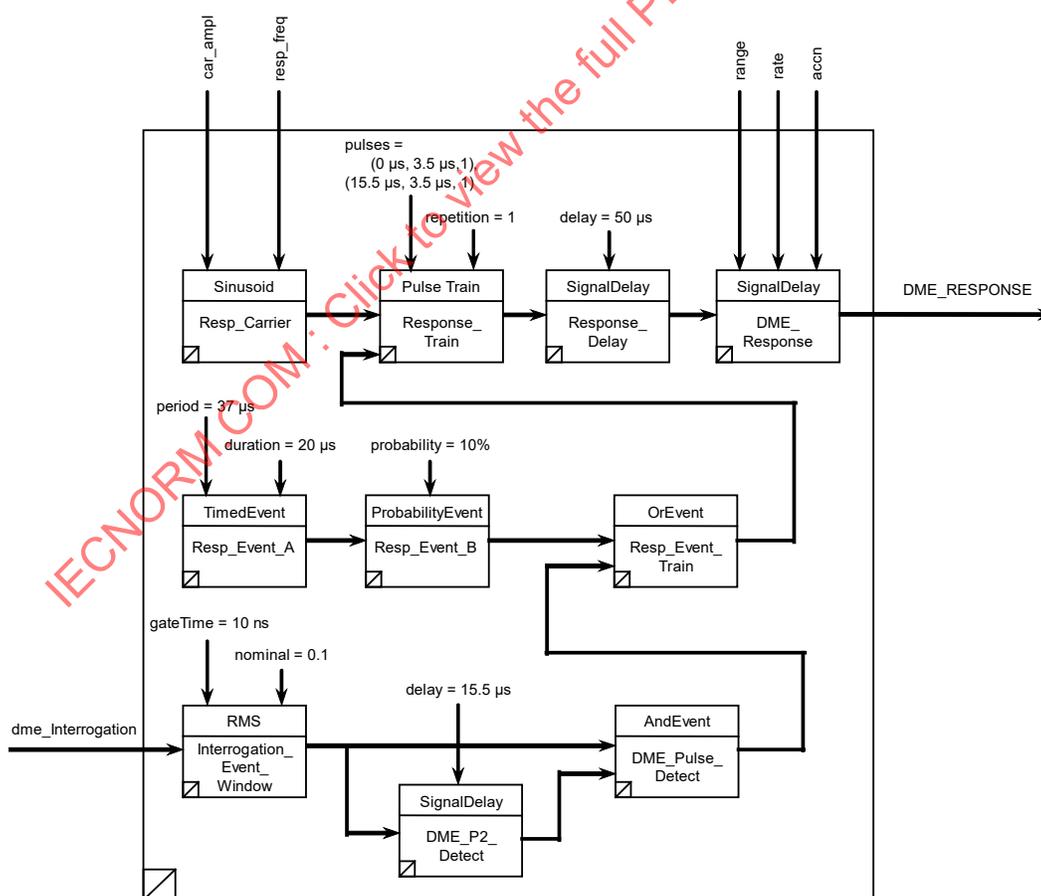


Figure E.15—TSF DME\_RESPONSE

### E.11.2 Interface properties

See Table E.15 for details of the TSF DME\_RESPONSE interface.

**Table E.15—TSF DME\_RESPONSE interface**

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Voltage	—	—
Transponder frequency	resp_freq	Frequency	962 MHz	962 MHz – 1213 MHz
Slant range	range	Distance	0 m	—
Range rate	rate	Speed	0 m/s	—
Rate of change of Range Rate	accn	Acceleration	0 m/s <sup>2</sup>	—
DME Interrogation signal	dme_Interrogation	SignalFunction	—	—

### E.11.3 Notes

Slant range of DME is dependent on aircraft height, transponder location and its associated environment, and geographical topography. Maximum range in ARINC 568 [B1] is quoted as up to 300 nmi (550 km) up to an altitude of 75 000 ft (23 000 m). The delay range quoted shall allow for a transponder transmission range of approximately 400 nmi (740 km) and its lower value is 0 nmi (0 km), the default 50 μs usually allowed from receipt of an interrogator signal to the transponder response within the transponder itself. These values cannot be exceeded.

This model has limited functionality. It does not provide for the variation of some of the parameters (such as the pulse timing and level). The model may be modified by the user to include such parameters in the interface properties.

### E.11.4 Model description

See Table E.16 for details of the DME\_RESPONSE model.

**Table E.16—TSF DME\_RESPONSE model**

Name	Type	Terminal	Inputs	Output	Formula
DME_Response	SignalDelay	Signal [Out]	—	DME_RESPONSE	—
		acceleration	—	—	(accn*2/3.0e8)
		delay	—	—	(range*2/3.0e8)
		rate	—	—	(rate*2/3.0e8)
		Signal [In]	Response_Delay	—	—
Response_Delay	SignalDelay	Signal [Out]	—	DME_Response	—
		—	—	—	—
		acceleration	—	—	0 Hz
		delay	—	—	50 μs
		rate	—	—	0%
Response_Train	PulseTrain	Signal [In]	Response_Train	—	—
		Signal [Out]	—	Response_Delay	—
		pulses	—	—	(0 μs , 3.5 μs, 1), (15.5 μs, 3.5 μs, 1)
		repetition	—	—	1
		Signal [In]	Resp_Carrier	—	—
		Sync[In]	Resp_Event_Train	—	—

**Table E.16—TSF DME\_RESPONSE model (continued)**

Name	Type	Terminal	Inputs	Output	Formula
Resp_Carrier	Sinusoid	Signal [Out]	—	Response_Train	—
		amplitude	car_ampl	—	—
		frequency	resp_freq	—	—
		phase	—	—	0 rad
Resp_Event_Train	OrEvent	Event [Out]	—	Response_Train	—
		Signal [In]	Resp_Event_B	—	—
		Signal [In]	DME_Pulse_Detect	—	—
Resp_Event_B	ProbabilityEvent	Event [Out]	—	Resp_Event_Train	—
		seed	—	—	0
		probability	—	—	10%
		Signal [In]	Resp_Event_A	—	—
DME_Pulse_Detect	AndEvent	Event [Out]	—	Resp_Event_Train	—
		Signal [In]	DME_P2_Detect	—	—
		Signal [In]	Interrogation_Event_Window	—	—
Resp_Event_A	TimedEvent	Event [Out]	—	Resp_Event_B	—
		delay	—	—	0 s
		duration	—	—	20 μs
		period	—	—	37 μs
		repetition	—	—	0
DME_P2_Detect	SignalDelay	Signal [Out]	—	DME_Pulse_Detect	—
		acceleration	—	—	0 Hz
		delay	—	—	15.5 μs
		rate	—	—	0%
		Signal [In]	Interrogation_Event_Window	—	—
Interrogation_Event_Window	RMS	[Out]	—	DME_Pulse_Detect , DME_P2_Detect	—
		measuredVariable	—	—	—
		measurement	—	—	—
		measurements	—	—	—
		sample	—	—	—
		count	—	—	—
		gateTime	—	—	1.0e-8
		nominal	—	—	0.1
		condition	—	—	GE
		GO	—	—	—
		NOGO	—	—	—
		HI	—	—	—
		LO	—	—	—
		UL	—	—	—
		LL	—	—	—
Signal [As]	—	—	—		
Signal [In]	dme_Interrogation	—	—		

### E.11.5 Rules

There are no special rules for this TSF.

### E.11.6 Example

See Figure E.16 for an example of DME\_RESPONSE.

*XML Static Signal Description:*

```
<DME_RESPONSE name="DME_RESPONSE5" range="2 nmi" rate="600 kt"  
In="DME_INTERROGATION6"/>  
<DME_INTERROGATION name="DME_INTERROGATION6"  
int_freq="1050 MHz" int_rate="150 Hz" />
```

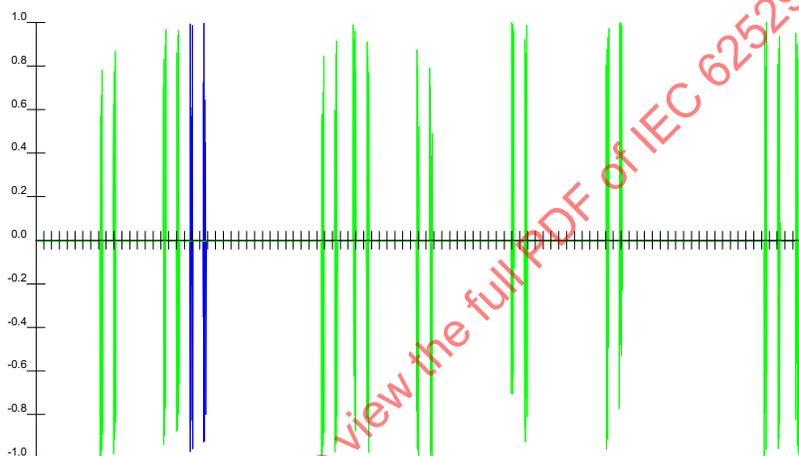


Figure E.16—DME\_RESPONSE example

## E.12 FM\_SIGNAL<type: Voltage|| Power|| Current>

### E.12.1 Definition

A continuous sinusoidal (carrier) wave generated when the frequency of one wave is varied in accordance with the amplitude of another (modulating) wave (modulating). See Figure E.17.

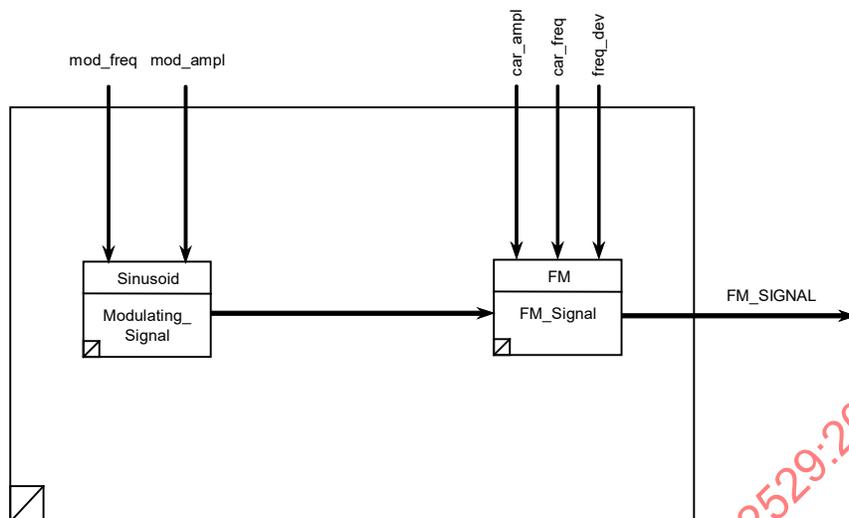


Figure E.17—TSF FM\_SIGNAL

### E.12.2 Interface properties

See Table E.17 for details of the TSF FM\_SIGNAL interface.

Table E.17—TSF FM\_SIGNAL interface

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Physical	—	—
Carrier frequency	car_freq	Frequency	—	—
Frequency deviation	freq_dev	Frequency	—	—
Modulation frequency	mod_freq	Frequency	—	—
Modulation amplitude	mod_ampl	Physical	1	—

### E.12.3 Notes

There are no special notes for this TSF.

### E.12.4 Model description

See Table E.18 for details of the TSF FM\_SIGNAL model.

**Table E.18—TSF FM\_SIGNAL model**

Name	Type	Terminal	Inputs	Output	Formula
FM_Signal	FM	Signal [Out]	—	FM_SIGNAL	—
		amplitude	car_ampl	—	—
		carrierFrequency	car_freq	—	—
		frequencyDeviation	freq_dev	—	—
		Signal [In]	Modulating_Signal	—	—
Modulating_Signal	Sinusoid	Signal [Out]	—	FM_Signal	—
		amplitude	mod_ampl	—	—
		frequency	mod_freq	—	—
		phase	—	—	0 rad

**E.12.5 Rules**

The output is given by Equation (E.2) and Equation (E.3).

$$e = E_c \sin(\omega_c t + m_f \sin(\omega_m t)) \tag{E.2}$$

$$m_f = k_f (E_m / \omega_m) \tag{E.3}$$

where

- $E_c$  is the carrier amplitude (unmodulated)
- $E_m$  is the modulation amplitude
- $\omega_c$  is  $2\pi \times$  carrier frequency
- $m_f$  deviation ratio ( $\equiv$  modulation index)
- $\omega_m$  is  $2\pi \times$  modulating frequency
- $k_f$  is the frequency deviation

**E.12.6 Example**

See Figure E.18 for an example of FM\_SIGNAL.

*XML Static Signal Description:*

```
<FM_SIGNAL name="FM_SIGNAL9" car_freq="100kHz"
freq_dev="10kHz"mod_freq="1200Hz" />
```

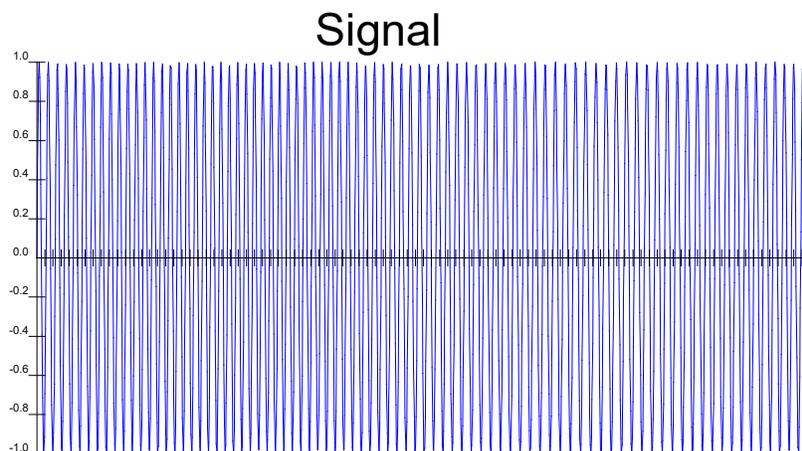


Figure E.18—FM\_SIGNAL example

### E.13 ILS\_GLIDE\_SLOPE<type: Voltage|| Power>

#### E.13.1 Definition

The vertical guidance portion of an instrument landing system (ILS).

At present, 40 glide slope channels exist with 150 kHz channel separation in the frequency range from 328.6 MHz to 335.4 MHz. The carrier is amplitude-modulated at 90 Hz and 150 Hz in a spatial pattern, with the 90 Hz modulation predominant when the airplane is above the glide path, and the 150 Hz modulation predominant if the airplane is below the glide path. The glide slope signal is achieved by transmitting two beams with equal offset about the correct glide slope angle. The upper beam is modulated to a depth of 40% with a 90 Hz tone, and the lower beam is modulated to a depth of 40% with a 150 Hz tone. The carrier of both beams is phase-locked so that any receiver shall treat them as a single-carrier signal with two modulating tones. If the aircraft is positioned off the glide slope, the ILS receiver shall detect one signal as stronger than the other. As a result, the demodulated amplitude (or apparent depth of modulation) of one tone shall be greater than the tone of the other. If the receiver is exactly on the glide slope, it shall receive a radio frequency (RF) carrier where the 90 Hz and 150 Hz modulation depths appear exactly the same. The greater the deviation from the glide slope, the greater shall be the difference in amplitude of the tones. Figure E.19.

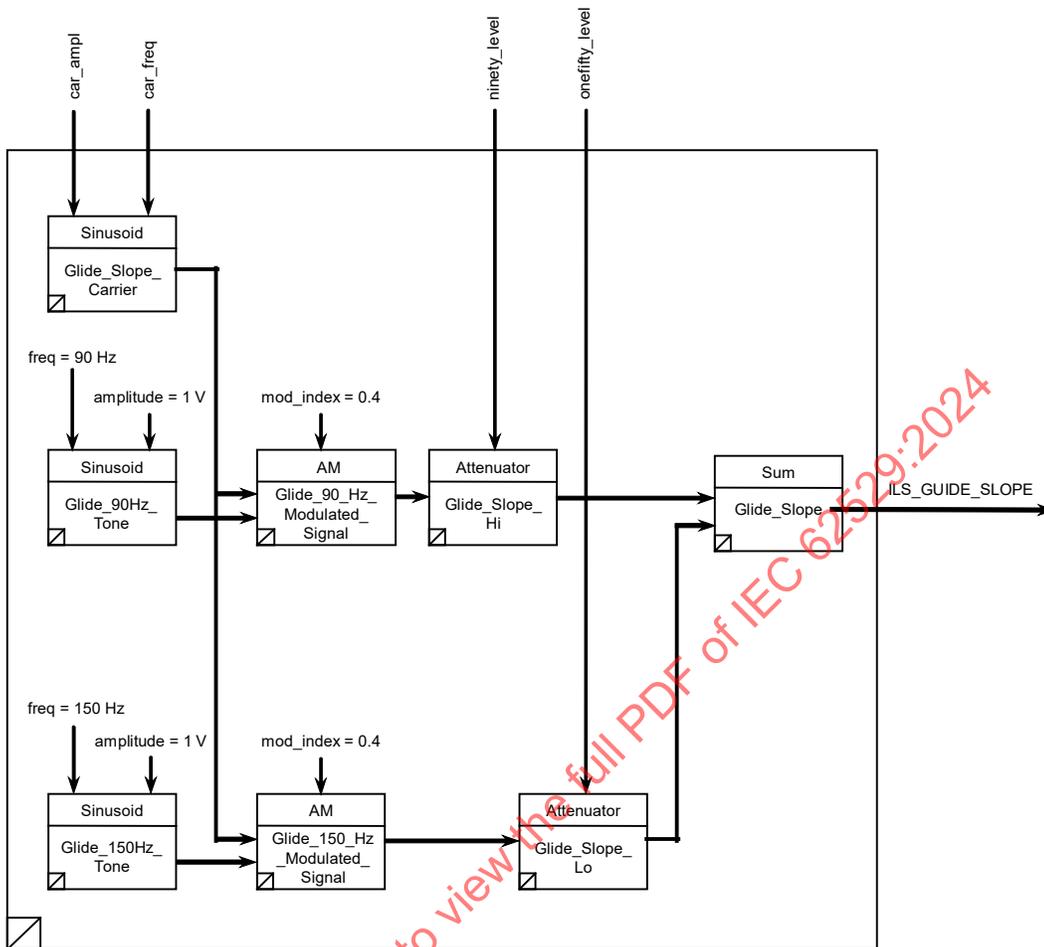


Figure E.19—TSF ILS\_GLIDE\_SLOPE

### E.13.2 Interface properties

See Table E.19 for details of the TSF ILS\_GLIDE\_SLOPE interface.

Table E.19—TSF ILS\_GLIDE\_SLOPE interface

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Physical	2 mV	—
Frequency	car_freq	Frequency	328.6 MHz	328.6 MHz – 335.4 MHz
150 Hz attenuation depth	oneifty_level	Ratio	1	0 – 1
90 Hz attenuation depth	ninety_level	Ratio	1	0 – 1

### E.13.3 Notes

This model has limited functionality. It does not provide for the variation of some of the parameters (such as the tone frequencies). The model may be modified by the user to include such parameters in the interface properties.

### E.13.4 Model description

See Table E.20 for details of the TSF ILS\_GLIDE\_SLOPE model.

**Table E.20—TSF ILS\_GLIDE\_SLOPE model**

Name	Type	Terminal	Inputs	Output	Formula
Glide_Slope	Sum	Signal [Out]	—	ILS_GLIDE_SLOPE	—
		Signal [In]	Glide_Slope_Lo	—	—
		Signal [In]	Glide_Slope_Hi	—	—
Glide_Slope_Hi	Attenuator	Signal [Out]	—	Glide_Slope	—
		gain	ninety level	—	—
		Signal [In]	Glide_90_Hz_Modulated_Signal	—	—
Glide_Slope_Lo	Attenuator	Signal [Out]	—	Glide_Slope	—
		gain	onefifty level	—	—
		Signal [In]	Glide_150_Hz_Modulated_Signal	—	—
Glide_90_Hz_Modulated_Signal	AM	Signal [Out]	—	Glide_Slope_Hi	—
		modIndex	—	—	0.4
		Carrier [In]	Glide_Slope_Carrier	—	—
		Signal [In]	Glide_90Hz_Tone	—	—
Glide_150_Hz_Modulated_Signal	AM	Signal [Out]	—	Glide_Slope_Lo	—
		modIndex	—	—	0.4
		Carrier [In]	Glide_Slope_Carrier	—	—
		Signal [In]	Glide_150Hz_Tone	—	—
Glide_Slope_Carrier	Sinusoid	Signal [Out]	—	Glide_150_Hz_Modulated_Signal, Glide_90_Hz_Modulated_Signal	—
		amplitude	car_ampl	—	—
		frequency	car_freq	—	—
		phase	—	—	—
Glide_90Hz_Tone	Sinusoid	Signal [Out]	—	Glide_90_Hz_Modulated_Signal	—
		amplitude	—	—	1 (see NOTE)
		frequency	—	—	90 Hz
		phase	—	—	0 rad
Glide_150Hz_Tone	Sinusoid	Signal [Out]	—	Glide_150_Hz_Modulated_Signal	—
		amplitude	—	—	1 (see NOTE)
		frequency	—	—	150 Hz
		phase	—	—	0 rad

NOTE—The BSC requires a unity value for the amplitude of the modulating signal.

### E.13.5 Rules

For this signal, the allowable types for carrier amplitudes are Voltage and Power.

### E.13.6 Example

See Figure E.20 for an example of ILS\_GLIDE\_SLOPE.

*XML Static Signal Description:*

```
<ILS_GLIDE_SLOPE name="ILS_GLIDE_SLOPE7" onefifty_level="1.1"  
ninety_level="0.9" />
```

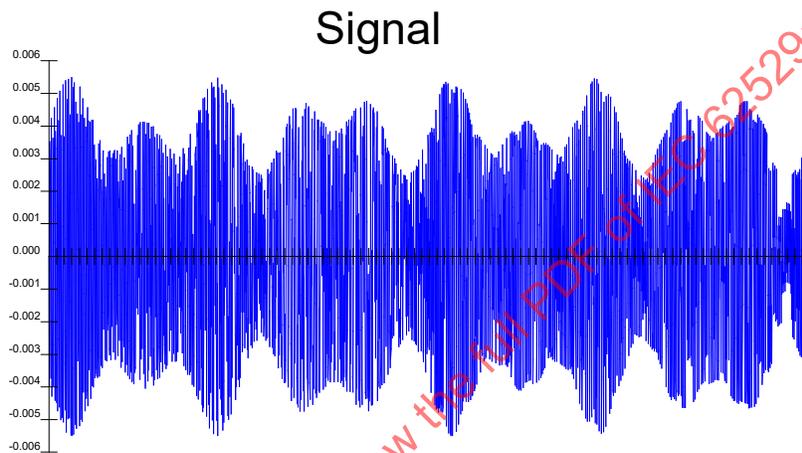


Figure E.20—ILS\_GLIDE\_SLOPE example

## E.14 ILS\_LOCALIZER<type: Power|| Voltage>

### E.14.1 Definition

The localizer is the lateral guidance portion of the ILS, giving azimuth guidance with reference to the runway center line. It operates using the same principles as the glide slope, but with 40 channels in the very high frequency (VHF) band of 108.0 MHz to 112.0 MHz. Each localizer channel is paired with a glide slope channel. The carrier is modulated with 90 Hz and 150 Hz tones in a spatial pattern that makes the 90 Hz tone predominant when the aircraft is to the left of the course and the 150 Hz tone predominant when the aircraft is to the right of the course. The localizer carrier contains a Morse code signal identifying the runway and approach direction and also may carry a ground-to-air communication channel. See Figure E.21.

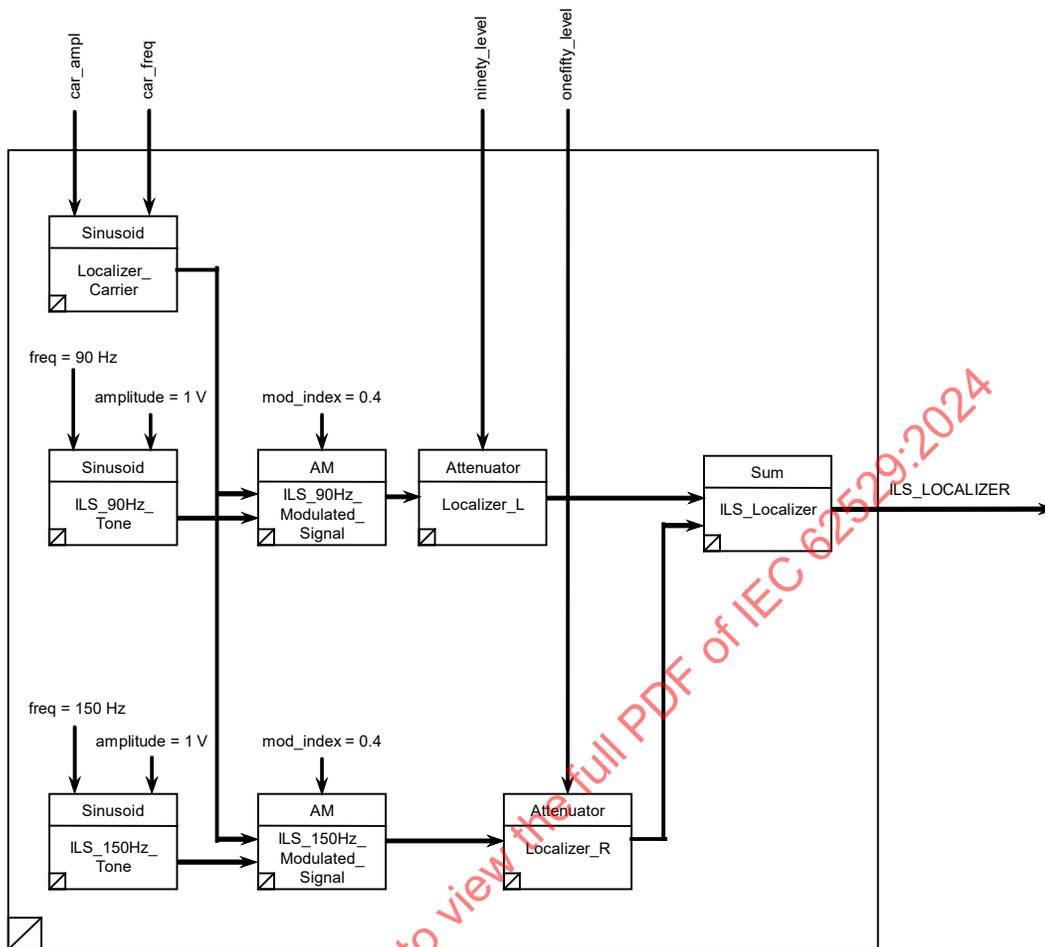


Figure E.21—TSF ILS\_LOCALIZER

### E.14.2 Interface properties

See Table E.21 for details of the TSF ILS\_LOCALIZER interface.

Table E.21—TSF ILS\_LOCALIZER interface

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Physical	2 mW	—
Carrier frequency	car_freq	Frequency	108.1 MHz	108.1 MHz –111.9 MHz
150 Hz attenuation depth	oneifty_level	Ratio	1	0 – 1
90 Hz attenuation depth	ninety_level	Ratio	1	0 – 1

### E.14.3 Notes

This model represents a limited implementation of the signal. It represents only the two-tone directional signal and does not allow for inclusion of coded information. It does not provide for the variation of some of the parameters (such as the tone frequencies). The model may be modified by the user to include such parameters in the interface properties.

**E.14.4 Model description**

See Table E.22 for details of the TSF ILS\_LOCALIZER model.

**Table E.22—TSF ILS\_LOCALIZER model**

Name	Type	Terminal	Inputs	Output	Formula
ILS_Localizer	Sum	Signal [Out]	—	ILS_LOCALIZER	—
		Signal [In]	Localizer_R	—	—
		Signal [In]	Localizer_L	—	—
Localizer_R	Attenuator	Signal [Out]	—	ILS_Localizer	—
		gain	onefifty level	—	—
		Signal [In]	ILS_150Hz_Modulated_Signal	—	—
Localizer_L	Attenuator	Signal [Out]	—	ILS_Localizer	—
		gain	ninety level	—	—
		Signal [In]	ILS_90Hz_Modulated_Signal	—	—
ILS_150Hz_Modulated_Signal	AM	Signal [Out]	—	Localizer_R	—
		modIndex	—	—	0.2
		Carrier [In]	Localizer_Carrier	—	—
		Signal [In]	ILS_150Hz_Tone	—	—
ILS_90Hz_Modulated_Signal	AM	Signal [Out]	—	Localizer_L	—
		modIndex	—	—	0.2
		Carrier [In]	Localizer_Carrier	—	—
		Signal [In]	ILS_90Hz_Tone	—	—
ILS_150Hz_Tone	Sinusoid	Signal [Out]	—	ILS_150Hz_Modulated_Signal	—
		amplitude	—	—	1 (see note)
		frequency	—	—	150 Hz
		phase	—	—	0 rad
ILS_90Hz_Tone	Sinusoid	Signal [Out]	—	ILS_90Hz_Modulated_Signal	—
		amplitude	—	—	1 (see note)
		frequency	—	—	90 Hz
		phase	—	—	0 rad
Localizer_Carrier	Sinusoid	Signal [Out]	—	ILS_90Hz_Modulated_Signal,	—
		amplitude	car ampl	—	—
		frequency	car freq	—	—
		phase	—	—	0 rad

NOTE—The BSC requires a unity value for the amplitude of the modulating signal.

### E.14.5 Rules

For this signal, the allowable types for carrier amplitudes are Voltage and Power.

### E.14.6 Example

See Figure E.22 for an example of ILS\_LOCALIZER.

*XML Static Signal Description:*

```
<ILS_LOCALIZER name="ILS_LOCALIZER6" ninety_level="0.9"  
  onefifty_level="1.1" />
```

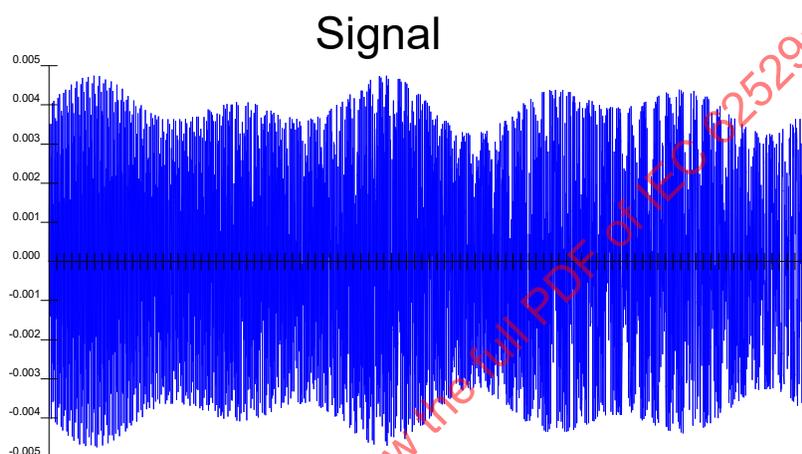


Figure E.22—ILS\_LOCALIZER example

## E.15 ILS\_MARKER

### E.15.1 Definition

Two or three marker beacons operate at 75 MHz to give a range with reference to the touchdown point. The outer marker is modulated with a 400 Hz tone to a depth of 95%. It is located 3½ nmi to 6 nmi (6 km to 11 km) from the end of the runway where the glide slope intersects the procedure turn altitude ± 15 m (50 ft) vertically. It radiates a fan-shaped pattern vertically and normal to the localizer and activates a marker receiver when the aircraft passes through.

The middle marker is a second fan-shaped marker similar to the outer marker. It is located approximately 0.5 nmi to 0.8 nmi (1 km to 1.5 km) from the ILS approach end of the runway and modulated at 1300 Hz. The inner marker, when used for category II approaches, intercepts the glide path at about the 100 ft (30 m) height to mark the overshoot decision point (if the runway is still not visible). The marker is recognized by its 3000 Hz modulation. Category II approaches allow operation down to 100 ft (30 m) and 1300 ft (400 m) visibility. See Figure E.23.

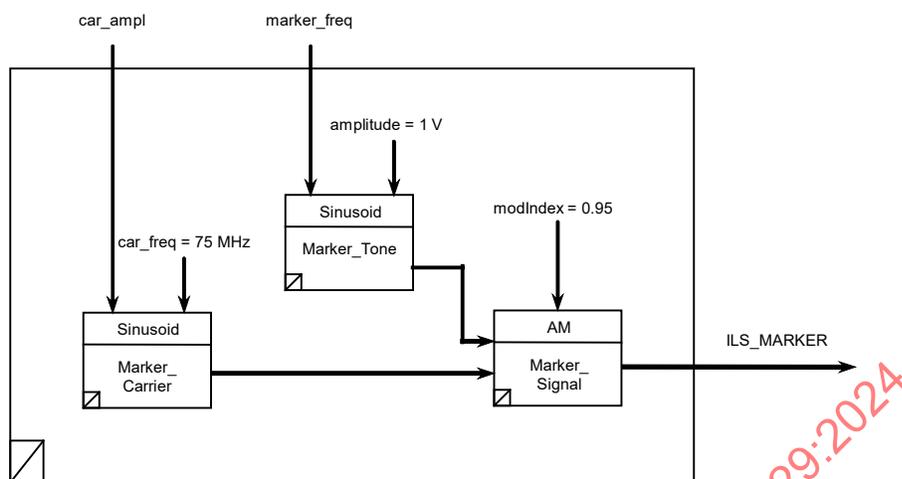


Figure E.23—TSF ILS\_MARKER

**E.15.2 Interface properties**

See Table E.23 for details of the TSF ILS\_MARKER interface.

Table E.23—TSF ILS\_MARKER interface

Description	Name	Type	Default	Range
Marker frequency	marker_freq	Frequency	400 Hz	400 Hz   1.3 kHz   3 kHz
Carrier amplitude	car_ampl	Power	2 mW	—

**E.15.3 Notes**

This model represents a limited implementation of the signal. It does not provide for the variation of some of the parameters (such as the carrier frequency). The model may be modified by the user to include such parameters in the interface properties.

**E.15.4 Model description**

See Table E.24 for details of the TSF ILS\_MARKER model.

**Table E.24—TSF ILS\_MARKER model**

Name	Type	Terminal	Inputs	Output	Formula
Marker_Signal	AM	Signal [Out]	—	ILS_MARKER	—
		modIndex	—	—	0.95
		Carrier [In]	Marker_Carrier	—	—
		Signal [In]	Marker_Tone	—	—
Marker_Carrier	Sinusoid	Signal [Out]	—	Marker_Signal	—
		amplitude	car_ampl	—	—
		frequency	—	—	75 MHz
		phase	—	—	0 rad
Marker_Tone	Sinusoid	Signal [Out]	—	Marker_Signal	—
		amplitude	—	—	1 (see NOTE)
		frequency	marker_freq	—	—
		phase	—	—	0 rad

NOTE—The BSC requires a unity value for the amplitude of the modulating signal.

**E.15.5 Rules**

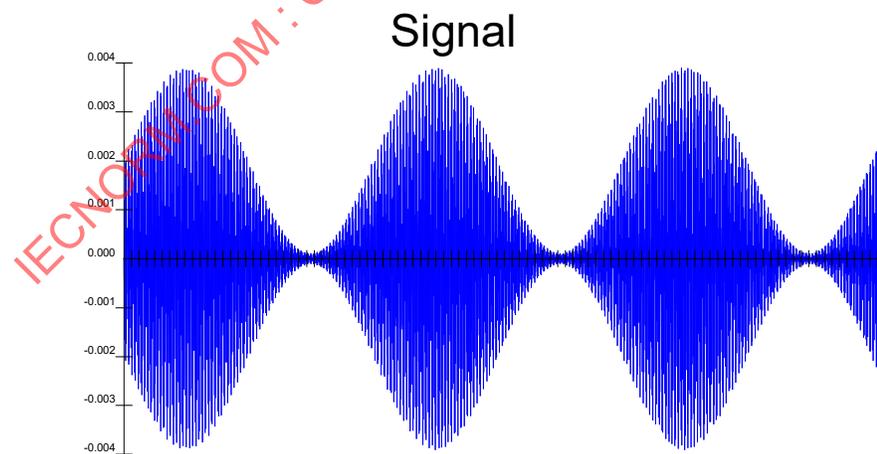
For this signal, the carrier amplitudes can be expressed only in terms of power.

**E.15.6 Example**

See Figure E.24 for an example of ILS\_MARKER.

*XML Static Signal Description:*

```
<ILS_MARKER name="ILS_MARKER5" />
```



**Figure E.24—ILS\_MARKER example**

## E.16 PM\_SIGNAL

### E.16.1 Definition

A continuous sinusoidal wave (carrier) whose phase is varied in accordance with the amplitude of another wave. See Figure E.25.

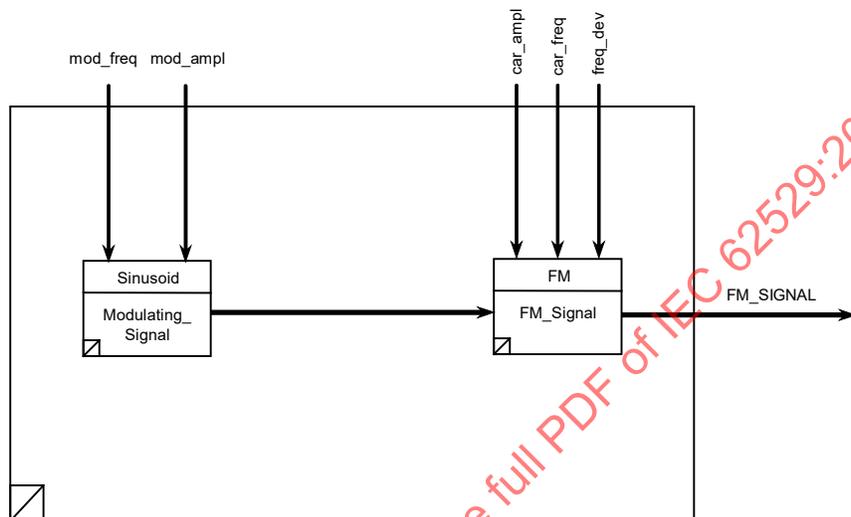


Figure E.25—TSF PM\_SIGNAL

### E.16.2 Interface properties

See Table E.25 for details of the TSF PM\_SIGNAL interface.

Table E.25—TSF PM\_SIGNAL interface

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Voltage	—	—
Carrier frequency	car_freq	Frequency	—	—
Phase deviation	phase_dev	PlaneAngle	—	—
Modulation frequency	mod_freq	Frequency	—	—
Modulation amplitude	mod_ampl	Voltage	1 V	—

### E.16.3 Notes

There are no special notes for this TSF.

### E.16.4 Model description

See Table E.26 for details of the TSF PM\_SIGNAL model.

**Table E.26—TSF PM\_SIGNAL model**

Name	Type	Terminal	Inputs	Output	Formula
PM_Signal	PM	Signal [Out]	—	PM_SIGNAL	—
		amplitude	car_ampl	—	—
		carrierFrequency	car_freq	—	—
		phaseDeviation	phase_dev	—	—
		Signal [In]	PModulating_Signal	—	—
PModulating_Signal	Sinusoid	Signal [Out]	—	PM_Signal	—
		amplitude	mod_ampl	—	—
		frequency	mod_freq	—	—
		phase	—	—	0 rad

### E.16.5 Rules

The output is given by Equation (E.4) and Equation (E.5).

$$e = E_c \sin(\omega_c t + k_p E_m \sin(\omega_m t)) \quad (E.4)$$

$$m_f = k_f (E_m / \omega_m) \quad (E.5)$$

where

- $E_c$  is the carrier amplitude (unmodulated)
- $E_m$  is the modulation amplitude
- $\omega_c$  is  $2\pi \times$  carrier frequency
- $\omega_m$  is  $2\pi \times$  modulating frequency
- $k_p$  is the phase deviation

### E.16.6 Example

See Figure E.26 for an example of PM\_SIGNAL.

*XML Static Signal Description:*

```
<PM_SIGNAL name="PM_SIGNAL9" phase_dev="(pi*8)" car_ampl="1 V"
car_freq="100 kHz" mod_freq="1240 Hz" />
```

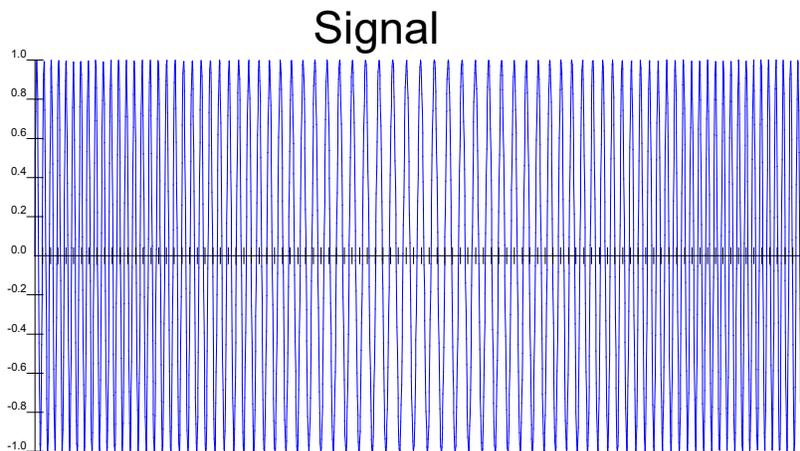


Figure E.26—PM\_SIGNAL example

**E.17 PULSED\_AC\_SIGNAL<type: Current|| Power|| Voltage>**

**E.17.1 Definition**

A signal characterized by short duration periods of (sinusoidal) ac electrical potential. See Figure E.27.

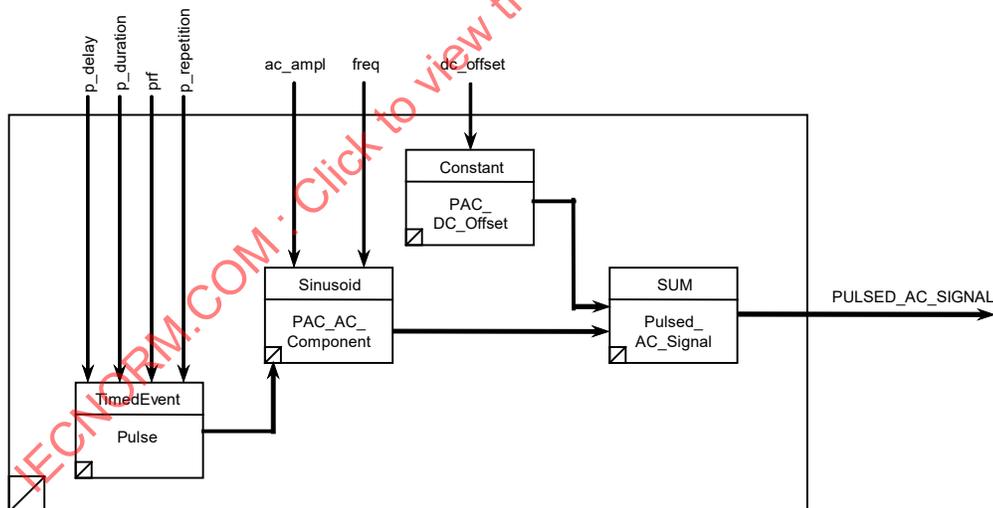


Figure E.27—TSF PULSED\_AC\_SIGNAL

**E.17.2 Interface properties**

See Table E.27 for details of the TSF PULSED\_AC\_SIGNAL interface.

**Table E.27—TSF PULSED\_AC\_SIGNAL interface**

Description	Name	Type	Default	Range
AC Signal amplitude	ac_ampl	Physical	—	—
AC Signal frequency	Freq	Frequency	—	—
DC Offset	dc_offset	Physical	0	—
Initial delay	p_delay	Time	0 s	—
Pulse width	p_duration	Time	—	—
Pulse repetition frequency	Prf	Frequency	—	—
Number of pulses	p_repetition	int	0	—

**E.17.3 Notes**

Default condition (where p\_repetition = 0) is for continuously repeating pulses.

This model represents a pulsed ac signal with a permanent dc offset. An alternative model may be created where only the pulses have a dc offset.

**E.17.4 Model description**

See Table E.28 for details of the TSF PULSED\_AC\_SIGNAL model.

**Table E.28—TSF PULSED\_AC\_SIGNAL model**

Name	Type	Terminal	Inputs	Output	Formula
Pulsed_AC_Signal	Sum	Signal [Out]	—	PULSED_AC_SIGNAL	—
		Signal [In]	PAC_DC_Offset	—	—
		Signal [In]	PAC_AC_Component	—	—
PAC_AC_Component	Sinusoid	Signal [Out]	—	Pulsed_AC_Signal	—
		amplitude	ac_ampl	—	—
		frequency	freq	—	—
		phase	—	—	0 rad
		Gate [In]	Pulse	—	—
PAC_DC_Offset	Constant	Signal [Out]	—	Pulsed_AC_Signal	—
		amplitude	dc_offset	—	—
Pulse	TimedEvent	Event [Out]	—	PAC_AC-Component	—
		delay	p_delay	—	—
		duration	p_duration	—	—
		period	—	—	1/prf
		repetition	p_repetition	—	—

**E.17.5 Rules**

For this signal, the allowable types are Voltage, Current, and Power. All types need to be consistent. Thus, for example, if the ac signal amplitude is specified in volts, then the dc offset shall also be specified in volts.

### E.17.6 Example

See Figure E.28 for an example of PULSED\_AC\_SIGNAL.

XML Static Signal Description:

```
<PULSED_AC_SIGNAL name="PULSED_AC_SIGNAL11" dc_offset="0.5 V"  
p_delay="7 ms" p_duration="3 ms" p_period="5 ms" p_repetition="10" />
```

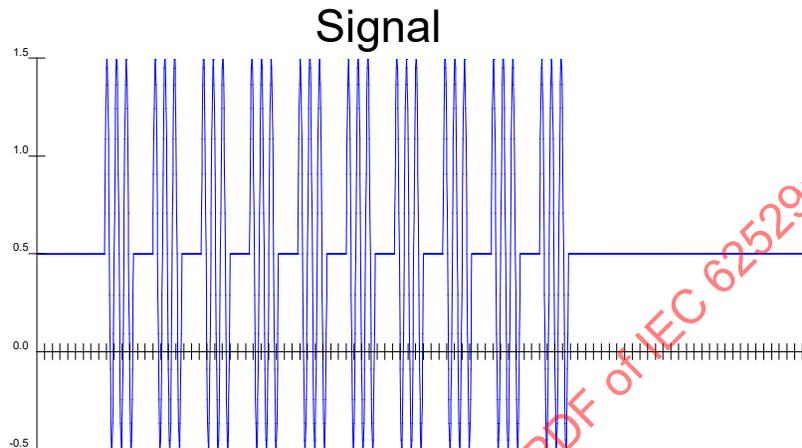


Figure E.28—PULSED\_AC\_SIGNAL example

### E.18 PULSED\_AC\_TRAIN<type: Voltage|| Current|| Power>

#### E.18.1 Definition

A signal characterized by a train of pulses of sinusoidal electrical ac activity with different durations and amplitudes. See Figure E.29.

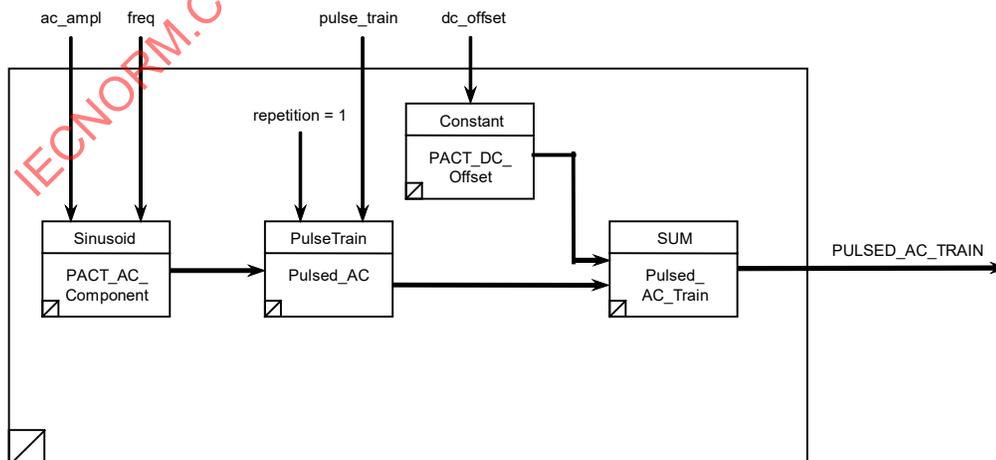


Figure E.29—TSF PULSED\_AC\_TRAIN

### E.18.2 Interface properties

See Table E.29 for details of the TSF PULSED\_AC\_TRAIN interface.

**Table E.29—TSF PULSED\_AC\_TRAIN interface**

Description	Name	Type	Default	Range
AC amplitude	ac_ampl	Physical	—	—
AC frequency	Freq	Frequency	—	—
DC Offset	dc_offset	Physical	0	—
Pulse train	pulse_train	PulseDefns	—	—

### E.18.3 Notes

This model represents a pulsed ac train with a permanent dc offset. An alternative model may be created where only the pulses have a dc offset.

### E.18.4 Model description

See Table E.30 for details of the TSF PULSED\_AC\_TRAIN model.

**Table E.30—TSF PULSED\_AC\_TRAIN model**

Name	Type	Terminal	Inputs	Output	Formula
Pulsed_AC_Train	Sum	Signal [Out]	—	PULSED_AC_TRAIN	—
		Signal [In]	Pulsed_AC	—	—
		Signal [In]	PACT_DC_Offset	—	—
Pulsed_AC	PulseTrain	Signal [Out]	—	Pulsed_AC_Train	—
		pulses	pulse_train	—	—
		repetition	—	—	1
		Signal [In]	PACT_AC_Component	—	—
PACT_DC_Offset	Constant	Signal [Out]	—	Pulsed_AC_Train	—
		amplitude	dc_offset	—	—
PACT_AC_Component	Sinusoid	Signal [Out]	—	Pulsed_AC	—
		amplitude	ac_ampl	—	—
		frequency	freq	—	—
		phase	—	—	0 rad

### E.18.5 Rules

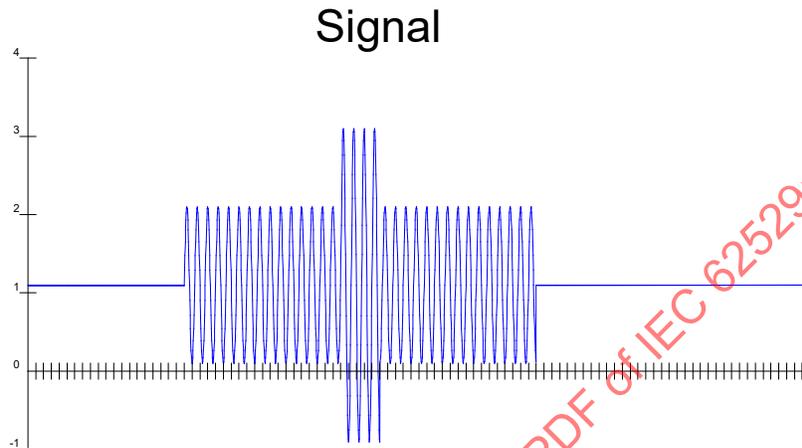
For this signal, the allowable types are Voltage, Current, and Power. All types need to be consistent. Thus, for example, if ac amplitude is specified in volts, then the dc offset shall also be specified in volts.

**E.18.6 Example**

See Figure E.30 for an example of PULSED\_AC\_TRAIN.

*XML Static Signal Description:*

```
<PULSED_AC_TRAIN name="PULSED_AC_TRAIN9" dc_offset="1.1 V" freq="150 Hz" pulse_train="(0.1,0.125,1), (0.2,0.125,1)" />
```

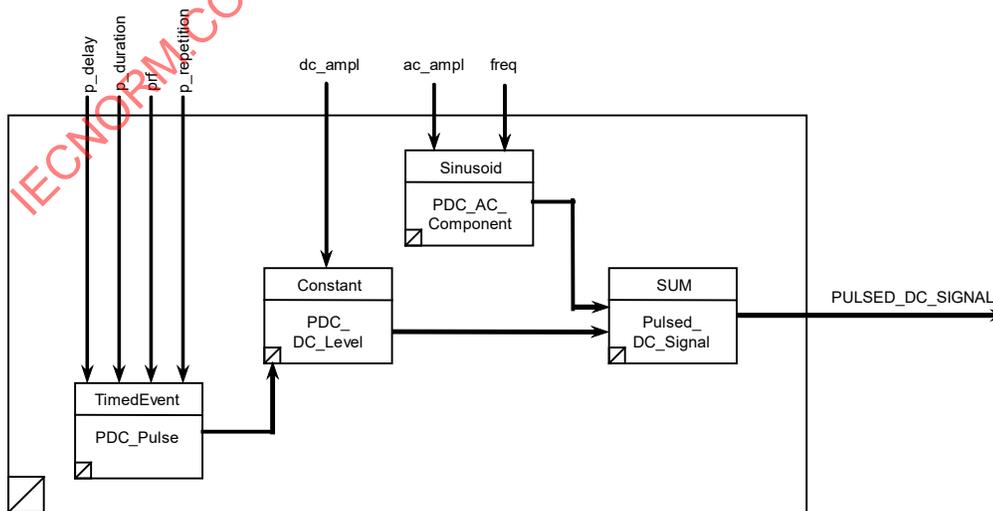


**Figure E.30—PULSED\_AC\_TRAIN example**

**E.19 PULSED\_DC\_SIGNAL<type: Voltage|| Current|| Power>**

**E.19.1 Definition**

A signal characterized by a train of pulses of electrical dc activity with different durations and amplitudes with an optional ac component. See Figure E.31.



**Figure E.31—TSF PULSED\_DC\_SIGNAL**

### E.19.2 Interface properties

See Table E.31 for details of the TSF PULSED\_DC\_SIGNAL interface.

**Table E.31—TSF PULSED\_DC\_SIGNAL interface**

Description	Name	Type	Default	Range
DC level	dc_ampl	Physical	—	—
AC Component amplitude	ac_ampl	Physical	0	—
AC Component frequency	Freq	Frequency	0 Hz	—
Delay before first pulse	p_delay	Time	0 s	—
Pulse width	p_duration	Time	—	—
Pulse repetition frequency	Prf	Frequency	—	—
Number of pulses	p_repetition	int	0	—

### E.19.3 Notes

Default condition (where p\_repetition = 0) is for continuously repeating pulses.

This model represents a pulsed dc signal with a permanent ac component (ripple). An alternative model may be created where only the pulses have an ac component.

### E.19.4 Model description

See Table E.32 for details of the TSF PULSED\_DC\_SIGNAL model.

**Table E.32—TSF PULSED\_DC\_SIGNAL model**

Name	Type	Terminal	Inputs	Output	Formula
Pulsed_DC_Signal	Sum	Signal [Out]	—	PULSED_DC_SIGNAL	—
		Signal [In]	PDC_DC_Level	—	—
		Signal [In]	PDC_AC_Component	—	—
PDC_DC_Level	Constant	Signal [Out]	—	Pulsed_DC_Signal	—
		amplitude	dc_ampl	—	—
		Gate[In]	PDC_Pulse	—	—
PDC_AC_Component	Sinusoid	Signal [Out]	—	Pulsed_DC_Signal	—
		amplitude	ac_ampl	—	—
		frequency	freq	—	—
		phase	—	—	0 rad
PDC_Pulse	TimedEvent	Event [Out]	—	PDC_DC_Level	—
		delay	p_delay	—	—
		duration	p_duration	—	—
		period	—	—	1/prf
		repetition	p_repetition	—	—

### E.19.5 Rules

For this signal, the allowable types are Voltage, Current, and Power. All types need to be consistent. Thus, for example, if a dc level is specified in volts, then the ac component amplitude shall also be specified in volts.

### E.19.6 Example

See Figure E.32 for an example of PULSED\_DC\_SIGNAL.

*XML Static Signal Description:*

```
<PULSED_DC_SIGNAL name="PULSED_DC_SIGNAL11" ac_ampl="0.2" dc_ampl="1"
freq="1 kHz" p_delay="0.02" p_duration="6 ms" p_period="10 ms"
p_repetition="5" />
```

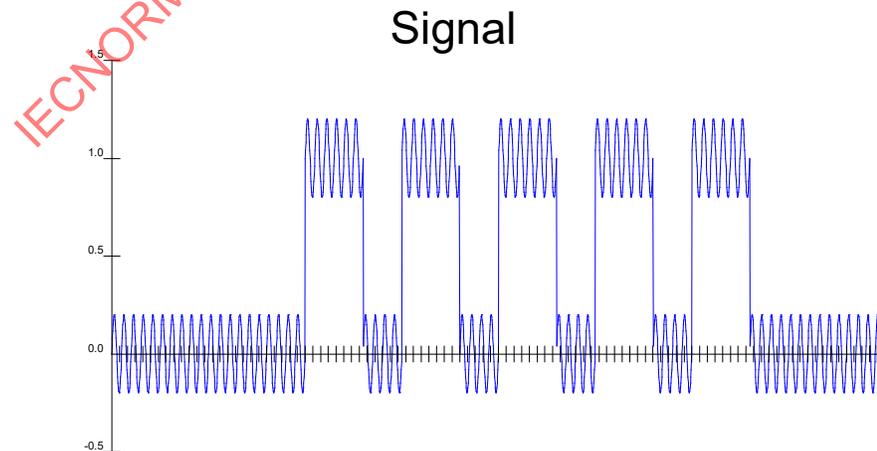


Figure E.32—PULSED\_DC\_SIGNAL example

## E.20 PULSED\_DC\_TRAIN<type: Voltage|| Current|| Power>

### E.20.1 Definition

A signal characterized by a train of different, short-duration periods of dc electrical activity. See Figure E.33.

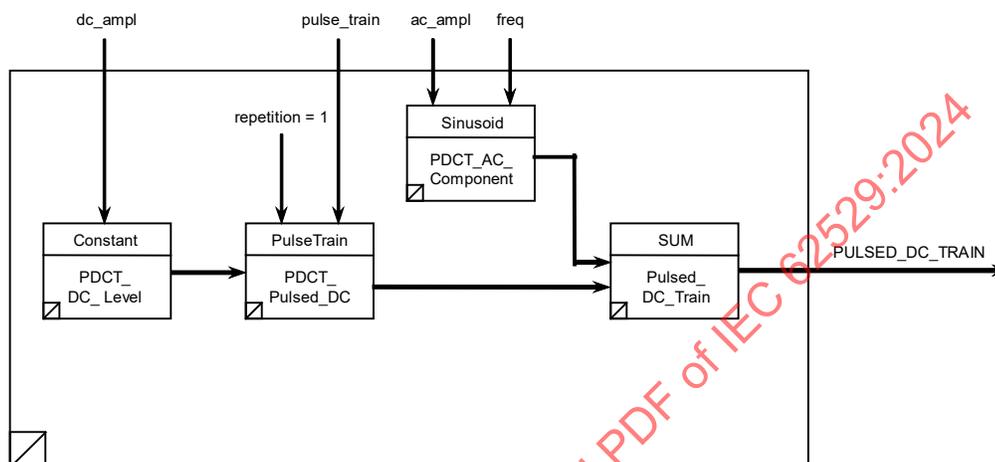


Figure E.33—TSF PULSED\_DC\_TRAIN

### E.20.2 Interface properties

See Table E.33 for details of the TSF PULSED\_DC\_TRAIN interface.

Table E.33—TSF PULSED\_DC\_TRAIN interface

Description	Name	Type	Default	Range
DC level	dc_ampl	Physical	—	—
Pulse train	pulse_train	PulseDefns	—	—
AC Component amplitude	ac_ampl	Physical	0	—
AC Component frequency	Freq	Frequency	0 Hz	—

### E.20.3 Notes

For this signal, the allowable types are Voltage, Current, and Power. All types need to be consistent. Thus, for example, if dc level is specified in volts, then the ac component amplitude shall also be specified in volts.

This model represents a pulsed dc train with a permanent ac component (ripple). An alternative model may be created where only the pulses have an ac component.

### E.20.4 Model description

See Table E.34 for details of the TSF PULSED\_DC\_TRAIN model.

**Table E.34—TSF PULSED\_DC\_TRAIN model**

Name	Type	Terminal	Inputs	Output	Formula
Pulsed_DC_Train	Sum	Signal [Out]	—	PULSED_DC_TRAIN	—
		Signal [In]	PDCT_Pulsed_DC	—	—
		Signal [In]	PDCT_AC_Component	—	—
PDCT_Pulsed_DC	PulseTrain	Signal [Out]	—	Pulsed_DC_Train	—
		pulses	pulse_train	—	—
		repetition	—	—	1
		Signal [In]	PDCT_DCLevel	—	—
PDCT_AC_Component	Sinusoid	Signal [Out]	—	Pulsed_DC_Train	—
		amplitude	ac_ampl	—	—
		frequency	freq	—	—
		phase	—	—	0 rad
PDCT_DC_Level	Constant	Signal [Out]	—	PDCT_Pulsed_DC	—
		amplitude	dc_ampl	—	—

**E.20.5 Rules**

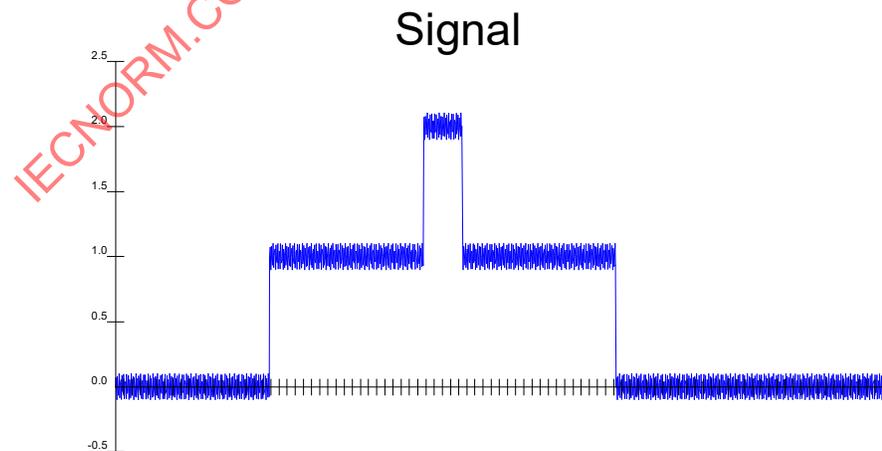
There are no special rules for this TSF.

**E.20.6 Example**

See Figure E.33 for an example of PULSED\_DC\_TRAIN.

*XML Static Signal Description:*

```
<PULSED_DC_TRAIN name="PULSED_DC_TRAIN6" ac_ampl="100 mV" freq="1 kHz" pulse_train="(0.1,0.125,1), (0.2,0.125,1)" />
```



**Figure E.34—PULSED\_DC\_TRAIN example**

## E.21 RADAR\_RX\_SIGNAL

### E.21.1 Definition

An appropriate delayed signal response to an input radar signal. See Figure E.35.

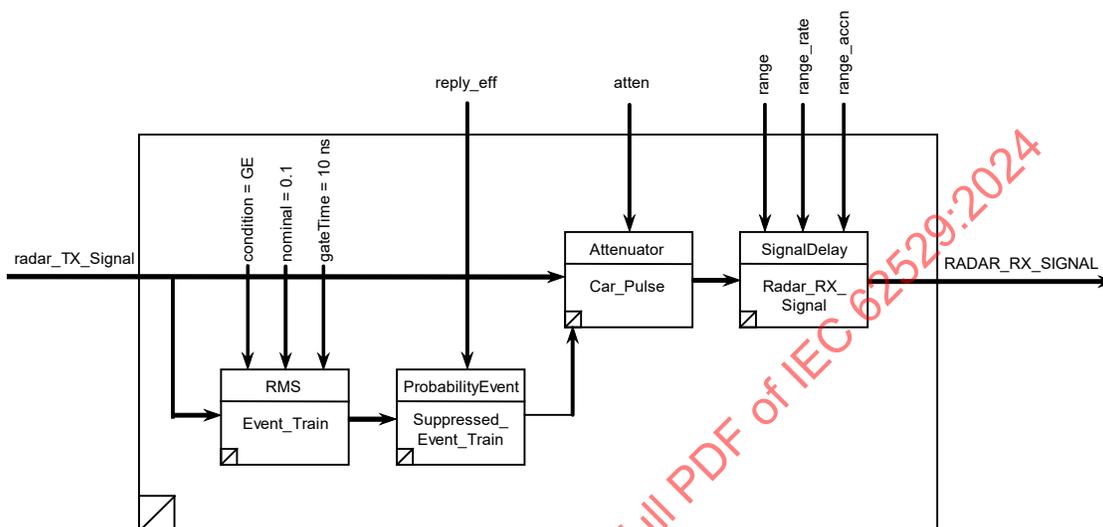


Figure E.35—TSF RADAR\_RX\_SIGNAL

### E.21.2 Interface properties

See Table E.35 for details of the TSF RADAR\_RX\_SIGNAL interface.

Table E.35—TSF RADAR\_RX\_SIGNAL interface

Description	Name	Type	Default	Range
Attenuation	atten	Ratio	1	—
Range of simulated target	range	Distance	—	—
Rate of change of rate change	range_accn	Acceleration	0	—
Rate of change of target range	range_rate	Speed	0	—
Proportion of Tx pulses returned	reply_eff	Ratio	100%	0 – 100%
Transmitted Radar Signal	radar_TX_Signal	SignalFunction	—	—

### E.21.3 Notes

This annex describes a transmitted signal as a reference. Thus, the TSF library provides a description for both the transmitted (i.e., Radar\_TX\_Signal) and received (i.e., Radar\_RX\_Signal) signals.

The Radar\_RX\_Signal takes an input radar signal and delays the signal response. In addition, the signal does not respond to all transmitted radar pulses (a feature that gives rise to a reply efficiency).

To achieve reply efficiency, the Radar\_RX\_Signal needs to detect the incoming radar pulses and suppress some individual pulses. To detect a radar pulse, an RMS monitor is used with a selected gate time. This

monitoring provides an event while the continuous rms value is greater than a nominal threshold value. The RMS monitor is used solely to detect a signal.

The default values for range\_rate and range\_accn (i.e., range\_rate = 0 and range\_accn = 0) represent a stationary target.

### E.21.4 Model description

See Table E.36 for details of the TSF RADAR\_RX\_SIGNAL model.

**Table E.36—TSF RADAR\_RX\_SIGNAL model**

Name	Type	Terminal	Inputs	Output	Formula
Radar_RX_Signal	SignalDelay	Signal [Out]	—	RADAR_RX_SIGNAL	—
		acceleration	—	—	(range_accn *2/3.0e8)
		delay	—	—	(range *2/3.0e8)
		rate	—	—	(range_rate *2/3.0e8)
		Signal [In]	Car Pulse	—	—
Car_Pulse	Attenuator	Signal [Out]	—	Radar RX Signal	—
		gain	atten	—	—
		Signal [In]	radar TX Signal	—	—
		Gate[In]	Suppressed_Event Train	—	—

**Table E.36—TSF RADAR\_RX\_SIGNAL model (continued)**

Name	Type	Terminal	Inputs	Output	Formula
Suppressed_Event_Train	ProbabilityEvent	Event [Out]	—	Car Pulse	—
		seed	—	—	0
		probability	reply eff	—	—
		Signal [In]	Event_Train	—	—
Event_Train	RMS	[Out]	—	Suppressed_Event_Train	—
		measuredVariable	—	—	—
		measurement	—	—	—
		measurements	—	—	—
		sample	—	—	—
		count	—	—	—
		gateTime	—	—	1.0e-8
		nominal	—	—	0.1
		condition	—	—	GE
		GO	—	—	—
		NOGO	—	—	—
		HI	—	—	—
		LO	—	—	—
		UL	—	—	—
LL	—	—	—		
Signal [As]	—	—	—		
Signal [In]	radar TX Signal	—	—		

### E.21.5 Rules

For this signal, the allowable types are Voltage, Current, and Power. However, for this signal, the type is determined by the RADAR\_TX\_SIGNAL to which it is referenced.

### E.21.6 Example

See Figure E.36 for an example of RADAR\_RX\_SIGNAL.

*XML Static Signal Description:*

```
<RADAR_RX_SIGNAL name="RADAR_RX_SIGNAL2" atten="0.6" range="2 nmi"  
range_rate="650 kt" In="RADAR_TX_SIGNAL10"/>  
<RADAR_TX_SIGNAL name="RADAR_TX_SIGNAL10" ampl="1" delay="0"  
duration="10 us" freq="100 MHz" period="120 us" />
```

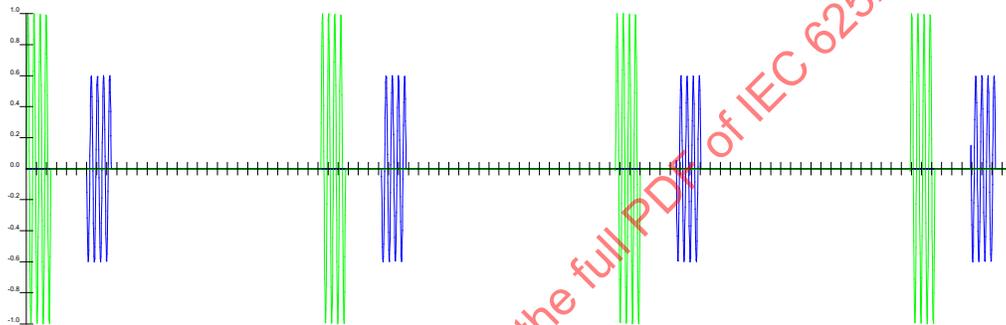


Figure E.36—RADAR\_RX\_SIGNAL example

## E.22 RADAR\_TX\_SIGNAL<type: Current|| Voltage|| Power>

### E.22.1 Definition

A pulsed ac signal used as a reference for received radar signals (i.e., Radar\_RX\_Signal). See Figure E.37.

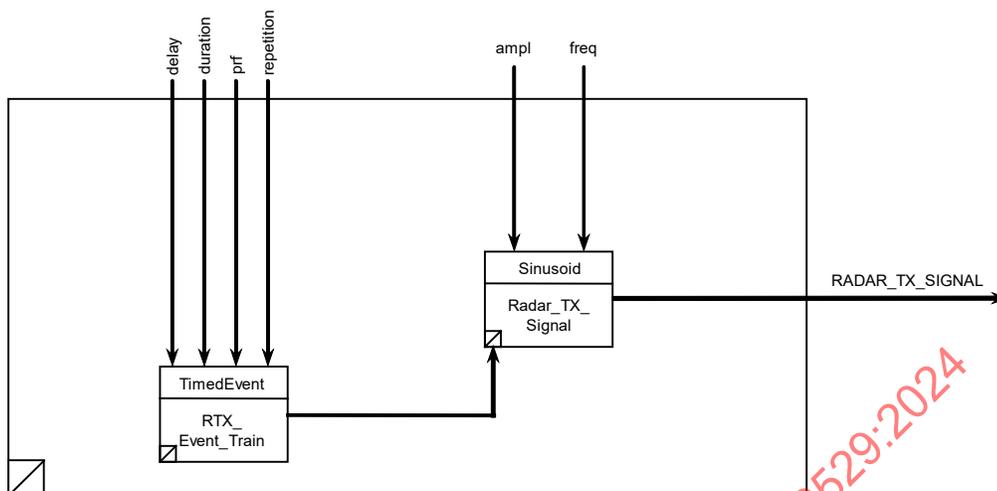


Figure E.37—TSF RADAR\_TX\_SIGNAL

**E.22.2 Interface properties**

See Table E.37 for details of the TSF RADAR\_TX\_SIGNAL interface.

Table E.37—TSF RADAR\_TX\_SIGNAL interface

Description	Name	Type	Default	Range
Tx signal amplitude	Ampl	Physical	—	—
Tx signal frequency	Freq	Frequency	—	—
Initial delay	Delay	Time	0 s	—
Pulse duration	duration	Time	—	—
Pulse repetition frequency	Prf	Frequency	—	—
Number of pulses	repetition	int	0	—

**E.22.3 Notes**

Default condition (where repetition = 0) is for continuously repeating pulses.

**E.22.4 Model description**

See Table E.38 for details of the TSF RADAR\_TX\_SIGNAL model.

**Table E.38—TSF RADAR\_TX\_SIGNAL model**

Name	Type	Terminal	Inputs	Output	Formula
RADAR_TX_Signal	Sinusoid	Signal [Out]	—	RADAR_TX_SIGNAL	—
		amplitude	ampl	—	—
		frequency	freq	—	—
		phase	—	—	0 rad
		Gate[In]	RTX_Event_Train	—	—
RTX_Event_Train	TimedEvent	Event [Out]	—	RADAR_TX_Signal	—
		delay	delay	—	—
		duration	duration	—	—
		period	—	—	1/prf
		repetition	repetition	—	—

**E.22.5 Rules**

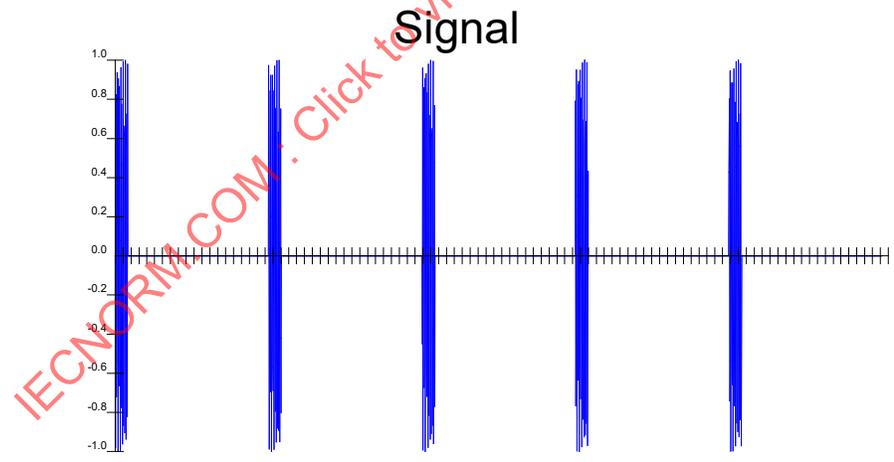
For this signal, the allowable types are Voltage, Current, and Power.

**E.22.6 Example**

See Figure E.38 for an example of RADAR\_TX\_SIGNAL.

*XML Static Signal Description:*

```
<RADAR_TX_SIGNAL name="RADAR_TX_SIGNAL10" ampl="1" delay="0"
duration="10 us" freq="100 MHz" prf="120 us" />
```



**Figure E.38—RADAR\_TX\_SIGNAL example**

**E.23 RAMP\_SIGNAL<type: Voltage|| Current|| Power>**

**E.23.1 Definition**

A periodic wave whose instantaneous value varies alternately and linearly between two specified values (i.e., initial and alternate). See Figure E.39.

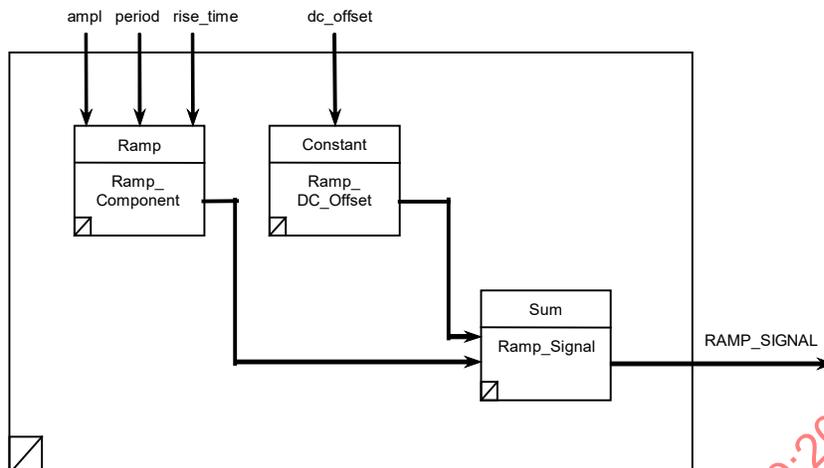


Figure E.39—TSF RAMP\_SIGNAL

**E.23.2 Interface properties**

See Table E.39 for details of the TSF RAMP\_SIGNAL interface.

Table E.39—TSF RAMP\_SIGNAL interface

Description	Name	Type	Default	Range
Ramp signal amplitude	Ampl	Physical	—	—
DC Offset	dc_offset	Physical	0	—
Ramp signal period	Period	Time	—	—
Ramp signal time to rise	rise_time	Time	—	—

**E.23.3 Notes**

There are no special notes for this TSF.

**E.23.4 Model description**

See Table E.40 for details of the TSF RAMP\_SIGNAL model.

Table E.40—TSF RAMP\_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
Ramp_Signal	Sum	Signal [Out]	—	RAMP_SIGNAL	—
		Signal [In]	Ramp_Component	—	—
		Signal [In]	Ramp_DC_Offset	—	—
Ramp_Component	Ramp	Signal [Out]	—	Ramp_Signal	—
		amplitude	ampl	—	—
		period	period	—	—
		riseTime	rise_time	—	—
Ramp-DC_Offset	Constant	Signal [Out]	—	Ramp_Signal	—
		amplitude	dc_offset	—	—

### E.23.5 Rules

For this signal, the allowable types are Voltage, Current, and Power. All types need to be consistent. Thus, for example, if the ramp signal amplitude is specified in volts, then the dc offset shall also be specified in volts.

### E.23.6 Example

See Figure E.40 for an example of RAMP\_SIGNAL.

*XML Static Signal Description:*

```
<RAMP_SIGNAL name="RAMP_SIGNAL7" dc_offset="0.5 V" period="1 kHz" rise_time="1 ms" />
```

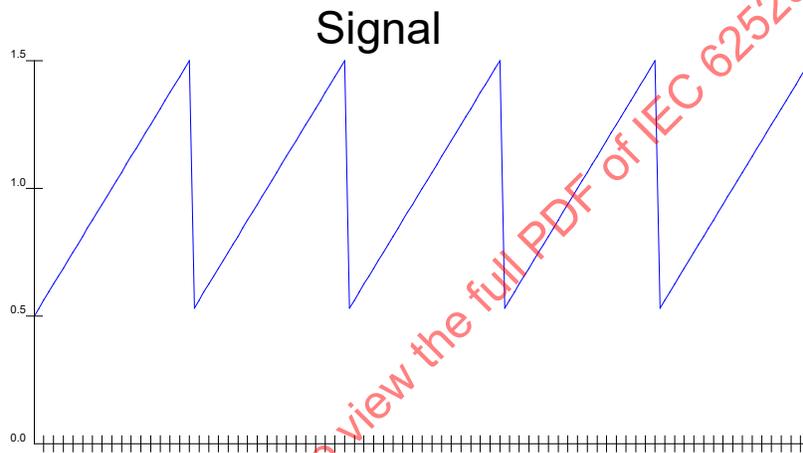


Figure E.40—RAMP\_SIGNAL example

## E.24 RANDOM\_NOISE

### E.24.1 Definition

Transient disturbances occurring unpredictably, except in a statistical sense. See Figure E.41.

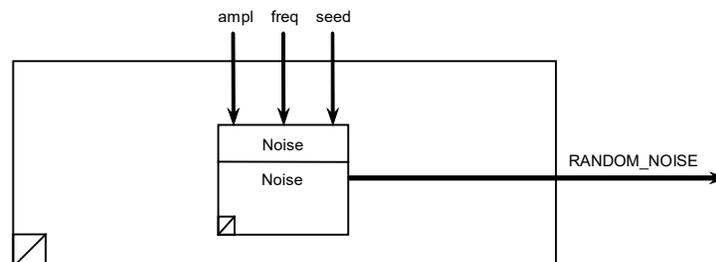


Figure E.41—TSF RANDOM\_NOISE

### E.24.2 Interface properties

See Table E.41 for details of the TSF RANDOM\_NOISE interface.

**Table E.41—TSF RANDOM\_NOISE interface**

Description	Name	Type	Default	Range
Noise signal amplitude	ampl	Physical	—	—
Pseudo random noise frequency	freq	Frequency	0	—
Pseudo random noise seed	seed	int	0	—

### E.24.3 Notes

The default for random noise is white noise (characterized by a flat frequency spectrum in the frequency range of interest). White noise needs only noise signal amplitude to be defined.

For repeatable pseudorandom noise, both the frequency upper bound and seed need to be specified. Specifying the frequency upper bound provides noise in the frequency band bounded by the freq value. If no seed is specified, this signal may not be repeatable.

### E.24.4 Model description

See Table E.42 for details of the TSF RANDOM\_NOISE model.

**Table E.42—TSF RANDOM\_NOISE model**

Name	Type	Terminal	Inputs	Output	Formula
Noise	Noise	Signal [Out]	—	RANDOM_NOISE	—
		amplitude	ampl	—	—
		seed	seed	—	—
		frequency	freq	—	—

### E.24.5 Rules

For this signal, the allowable types are Voltage and Power.

### E.24.6 Example

See Figure E.42 for an example of RANDOM\_NOISE.

*XML Static Signal Description:*

```
<RANDOM_NOISE ampl="100 mV" freq="500 Hz" seed="0" />
```

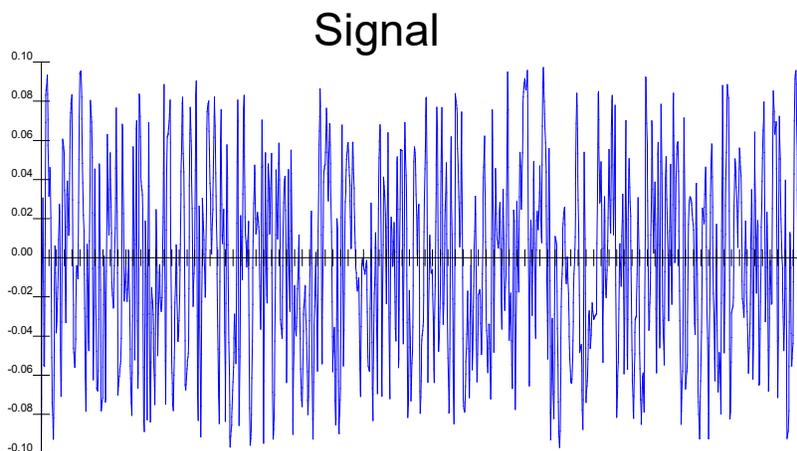


Figure E.42—RANDOM\_NOISE example

## E.25 RESOLVER

### E.25.1 Definition

Two ac sine wave voltages whose relationships of amplitude represent the rotation of a shaft position of an electromechanical transducer. See Figure E.43.

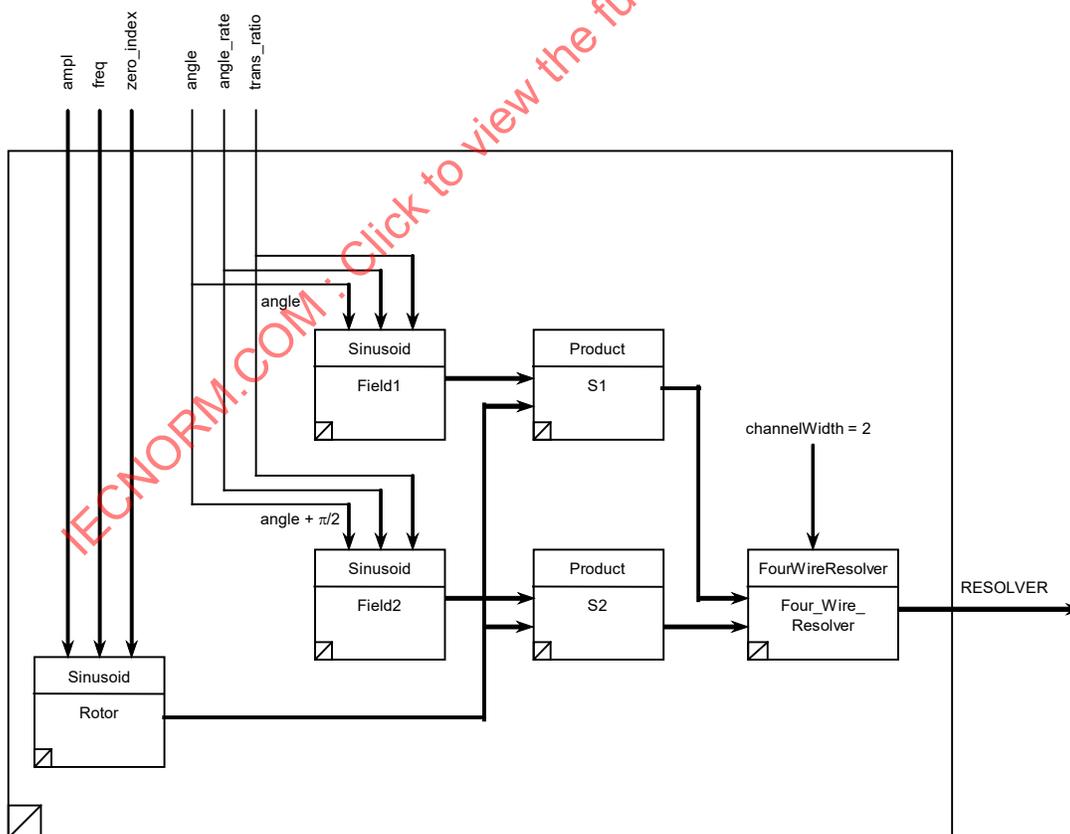


Figure E.43—TSF RESOLVER

### E.25.2 Interface properties

See Table E.43 for details of the TSF RESOLVER interface.

**Table E.43—TSF RESOLVER interface**

Description	Name	Type	Default	Range
Shaft angle	Angle	PlaneAngle	0	—
Reference amplitude	Ampl	Voltage	26 V	26 V-119 V
Reference frequency	Freq	Frequency	400 Hz	30 Hz – 54 kHz
Zero index	zero_index	PlaneAngle	0 rad	0 – $2\pi$ rad
Shaft angle rate	angle_rate	Frequency	0 Hz	—
Transformer Ratio	trans_ratio	Ratio	1	—

### E.25.3 Notes

This model does not consider the effects of angular velocity of the rotor and the quadrature voltages generated in the secondaries.

### E.25.4 Model description

See Table E.44 for details of the TSF RESOLVER model.

**Table E.44—TSF RESOLVER model**

Name	Type	Terminal	Inputs	Output	Formula
Four_Wire_Resolver	FourWireResolver	Signal [Out]	—	RESOLVER	—
		channelWidth	—	—	2
		Signal [In]	S1	—	—
		Signal [In]	S2	—	—
S1	Product	Signal [Out]	—	Four_Wire_Resolver	—
		Signal [In]	Rotor	—	—
		Signal [In]	Field1	—	—
S2	Product	Signal [Out]	—	Four_Wire_Resolver	—
		Signal [In]	Rotor	—	—
		Signal [In]	Field2	—	—
Rotor	Sinusoid	Signal [Out]	—	S1 S2	—
		amplitude	ampl	—	—
		frequency	freq	—	—
		phase	zero_index	—	—
Field1	Sinusoid	Signal [Out]	—	S1	—
		amplitude	—	—	trans_ratio
		frequency	—	—	(angle_rate)
		phase	angle	—	—
Field2	Sinusoid	Signal [Out]	—	S1	—
		amplitude	—	—	trans_ratio
		frequency	—	—	(angle_rate)
		phase	—	—	angle+ $\pi/2$

### E.25.5 Rules

The outputs of the resolver secondaries are given by Equation (E.6) (for sine) and either Equation (E.7) or Equation (E.8) (for cosine).

Sine output

$$e_{s1} = KE_r \sin\theta \sin(2\pi f_r t + \varphi) \quad (\text{E.6})$$

Cosine output

$$e_{s2} = KE_r \cos\theta \sin(2\pi f_r t + \varphi) \quad (\text{E.7})$$

or

$$e_{s2} = KE_r \sin(\theta + \pi/2) \sin(2\pi f_r t + \varphi) \quad (\text{E.8})$$

where

- $K$  is the transformer ratio (trans\_ratio), assuming  $K$  to be the same for both secondaries
- $E_r$  is the reference amplitude in the primary (ampl)
- $\theta$  is angular displacement of the rotor (angle)
- $f_r$  is the reference frequency of the signal in the primary (freq)
- $\varphi$  is the zero index position of the rotor (zero\_index)

Thus, the operation of the resolver may be modeled as the product of two signals for each output:

Sine output

$$e_{s1} = (E_r \sin(2\pi f_r t + \varphi)) \times (K \sin\theta) \quad (\text{E.9})$$

Cosine output

$$e_{s2} = (E_r \sin(2\pi f_r t + \varphi)) \times (K \sin(\theta + \pi/2)) \quad (\text{E.10})$$

### E.25.6 Example

See Figure E.44 for an example of RESOLVER.

*XML Static Signal Description:*

```
<RESOLVER name="RESOLVER9" angle_rate="5 Hz" freq="100 Hz" />
```

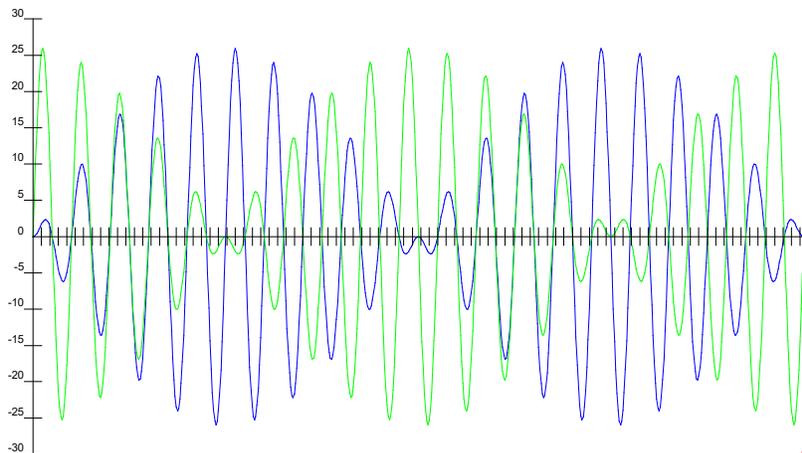


Figure E.44—RESOLVER example

## E.26 RS\_232

### E.26.1 Definition

A serial databus signal that transmits and receives strings of characters and operates according to TIA-232 [B20].

### E.26.2 Interface properties

See Table E.45 for details of the TSF RS\_232 interface.

Table E.45—TSF RS\_232 interface

Description	Name	Type	Default	Range
Data Word	data_word	string		
Baud Rate	baud_rate	Int	9600	75  110  134  150  300  600  1200  1800  2400  4800  7200  9600  14400  19200  38400  57600  115200
Data Bits	data_bits	Int	8	4  5  6  7  8
Parity	parity	enumeration	None	Even  Odd  None  Mark  Space
Stop Bits	stop_bits	enumeration	1	1  1.5  2
Flow Control	flow_control	enumeration	None	None  Hardware  Xon-Xoff
NOTE—The range of values for baud_rate and data_bits is for information only. Any value may be used with this model.				

### E.26.3 Notes

When using the TSF RS\_232 model, only the active data connection (and its ground) are considered, i.e., the connections hi and lo may be used. Some or all of the other control connections required by the TIA-232 specification may need to be physically connected, but are not considered by this TSF.

### E.26.4 Model description

See Table E.46 for details of the TSF RS\_232 model.

**Table E.46—TSF RS\_232 model**

Name	Type	Terminal	Inputs	Output	Formula
TIA_EIA_232	TIA/EIA-232	Signal [Out]	—	RS_232	—
		baud_rate	baud_rate	—	—
		data_bits	data_bits	—	—
		parity	parity	—	—
		stop_bits	stop_bits	—	—
		flow_control	flow_control	—	—
		data_word	data_word	—	—

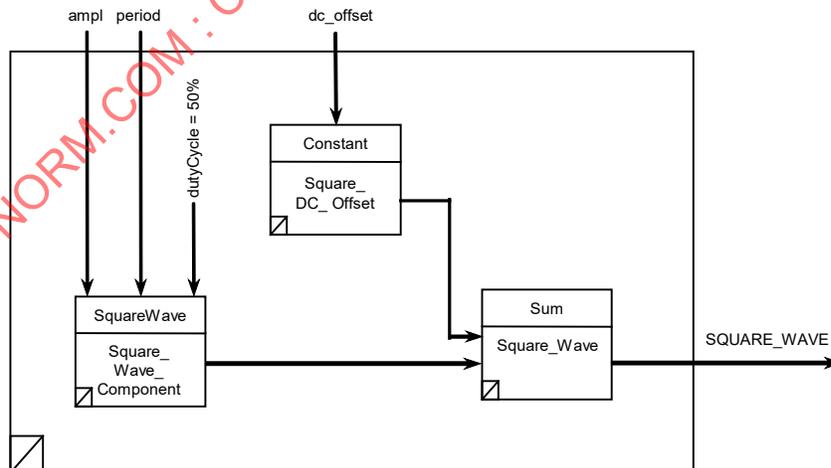
### E.26.5 Rules

For this signal, the data word supplied is transmitted via the serial bus connections according to the rules specified in TIA-232 [B20]. Data received via the serial bus connections shall be available when the signal is used in a measurement.

## E.27 SQUARE\_WAVE<type: Current|| Voltage|| Power>

### E.27.1 Definition

A periodic wave that alternately assumes one of two fixed values of amplitude for equal lengths of time. See Figure E.45.



**Figure E.45—TSF SQUARE\_WAVE**

### E.27.2 Interface properties

See Table E.47 for details of the TSF SQUARE\_WAVE interface.

**Table E.47—TSF SQUARE\_WAVE interface**

Description	Name	Type	Default	Range
Square wave amplitude	Ampl	Physical	—	—
Square wave period	Period	Time	—	—
DC offset	dc_offset	Physical	0	—

### E.27.3 Notes

There are no special notes for this TSF.

### E.27.4 Model description

See Table E.48 for details of the TSF SQUARE\_WAVE model.

**Table E.48—TSF SQUARE\_WAVE model**

Name	Type	Terminal	Inputs	Output	Formula
Square_Wave	Sum	Signal [Out]	—	SQUARE_WAVE	—
		Signal [In]	Square_Wave_Component	—	—
		Signal [In]	Square_DC_Offset	—	—
Square_Wave_Component	SquareWave	Signal [Out]	—	Square_Wave	—
		amplitude	ampl	—	—
		period	period	—	—
		dutyCycle	—	—	50%
Square_DC_Offset	Constant	Signal [Out]	—	Square_Wave	—
		amplitude	dc_offset	—	—

### E.27.5 Rules

For this signal, the allowable types are Voltage, Current, and Power. All types need to be consistent. Thus, for example, if the square wave amplitude is specified in volts, then the dc offset shall also be specified in volts.

### E.27.6 Example

See Figure E.46 for an example of SQUARE\_WAVE.

*XML Static Signal Description:*

```
<SQUARE_WAVE name="SQUARE_WAVE6" ampl="1" dc_offset="500 mV" period="10 us" />
```

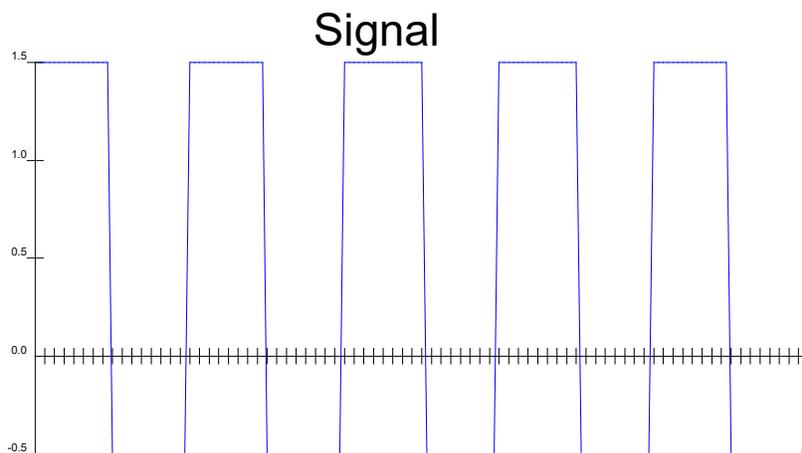


Figure E.46—SQUARE\_WAVE example

## E.28 SSR\_INTERROGATION<type: Voltage|| Current|| Power>

### E.28.1 Definition

Secondary surveillance radar (SSR) provides information to supplement the information obtained from a primary radar. Governing documents for civilian air traffic control (ATC) are ARINC 572 [B2] and ARINC 711-10 [B4] and for the military's identification, friend or foe, (IFF) system, STANAG 4193 [B19]. An aircraft on-board transponder shall sense an interrogation from a ground (or airborne) station on a specific frequency (i.e., 1030 MHz) and shall respond with coded signals on another frequency (i.e., 1090 MHz). See Figure E.47.

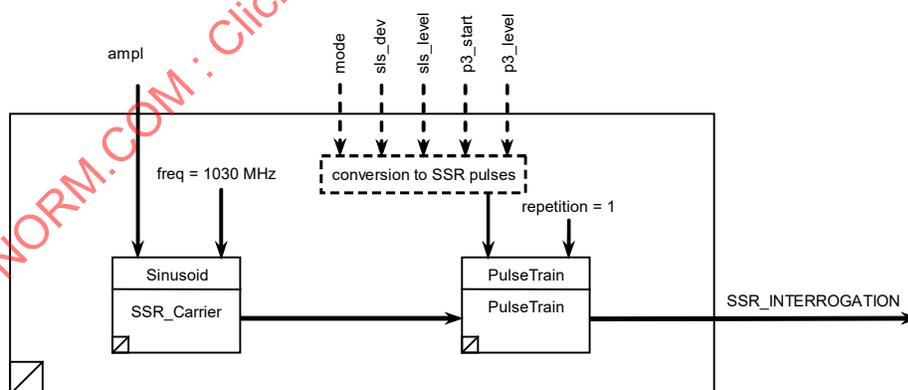


Figure E.47—TSF SSR\_INTERROGATION

### E.28.2 Interface properties

See Table E.49 for details of the TSF SSR\_INTERROGATION interface.

**Table E.49—TSF SSR\_INTERROGATION interface**

Description	Name	Type	Default	Range
P1 amplitude	Ampl	Physical	—	—
Interrogation mode	Mode	enumeration	1	1   2   3   A   B   C   D
P3 start time	p3_start	Time	3us	3 μs   5 μs   8 μs   17 μs   21 μs   25 μs
P3 level	p3_level	Ratio	1	—
SLS deviation	sls_dev	Time	0 μs	—
SLS level	sls_level	Ratio	1	—

**E.28.3 Notes**

The interrogation signal comprises three pulses, called P1, P2, and P3. See Table E.50. The normal spacing between P1 and P2 is 2 μs. Normal spacing between P1 and P3 depends on the choice of mode.

While interrogators shall repeat the interrogation sequence approximately every 2 ms and are capable of interlacing several modes alternately (most commonly 3-A and C, known as Mode 3-C), the model is set up for a single interrogation and thus allows each mode to be interrogated individually to verify the correct response.

The interface allows for the indirect programming of the pulse information. The pulse attributes are not directly entered as an array. The interface is used to select various SSR-specific parameters, which are then converted by the interface into the appropriate pulse definitions.

**Table E.50—SSR\_INTERROGATION pulse descriptions**

Pulse	Start time (μs)	Pulse width (μs)	Level factor
P1	0	0.8	1
P2	2 + SLS Deviation	0.8	SLS Level
P3	Mode 1     3 Mode 2     5 Mode 3     8 Mode A     8 Mode B     17 Mode C     21 Mode D     25	0.8	P3 Level

The output is given by the following equation:

$$\text{SSR\_pulses} \in (0 \mu\text{s}, 0.8 \mu\text{s}, 1), \\ (\{0.000002+\text{sls\_dev}\}, 0.8 \mu\text{s}, \{\text{sls\_level}\}), \\ (\{\text{p3\_start}\}, 0.8 \mu\text{s}, \{\text{p3\_level}\})$$

where

sls_dev	is the SLS deviation from the interface properties
sls_level	is the SLS level from the interface properties
p3_start	is the P3 start time as defined by the interrogation mode (see Table E.51)
p3_level	is the P3 level from the interface properties

**Table E.51—Pulse P3 start times**

Interrogation mode (mode)	P3 start time (p3_start) (μs)
1	3
2	5
3	8
A	8
B	17
C	21
D	25

#### E.28.4 Model description

See Table E.52 for details of the TSF SSR\_INTERROGATION model.

**Table E.52—TSF SSR\_INTERROGATION model**

Name	Type	Terminal	Inputs	Output	Formula
PulseTrain	PulseTrain	Signal [Out]	—	SSR_INTERROGATION	
		pulses	—	—	SSR_pulses See equation
		repetition	—	—	1
		Signal [In]	SSR_Carrier	—	—
SSR_Carrier	Sinusoid	Signal [Out]	—	PulseTrain	—
		amplitude	ampl	—	—
		frequency	—	—	1030 MHz
		phase	—	—	0 rad

#### E.28.5 Rules

For this signal, the allowable types are Voltage, Current, and Power.

### E.28.6 Example

See Figure E.48 for an example of SSR\_INTERROGATION.

*XML Static Signal Description:*

```
<SSR_INTERROGATION name="SSR_INTERROGATION3" mode="3" p3_start="8 us"  
p3_level="2" />
```

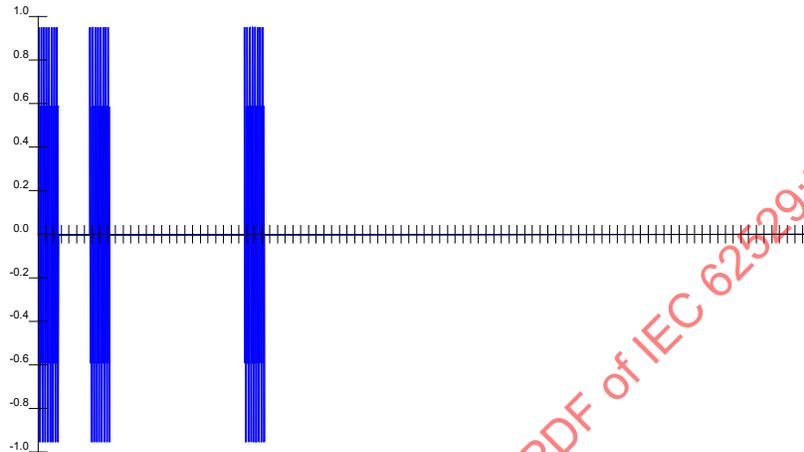


Figure E.48—SSR\_INTERROGATION example

## E.29 SSR\_RESPONSE<type: Voltage|| Current|| Power>

### E.29.1 Definition

The transponder response to a valid SSR interrogation. It consists of an encoded pulse train. Each pulse train consists of a number of data pulses. The number and position of these data pulses (after the start pulse) are determined by the mode selected. There are 16 pulse positions in the pulse train; however, the code or (height) information carried by the response shall determine which pulses are present. See Figure E.49.

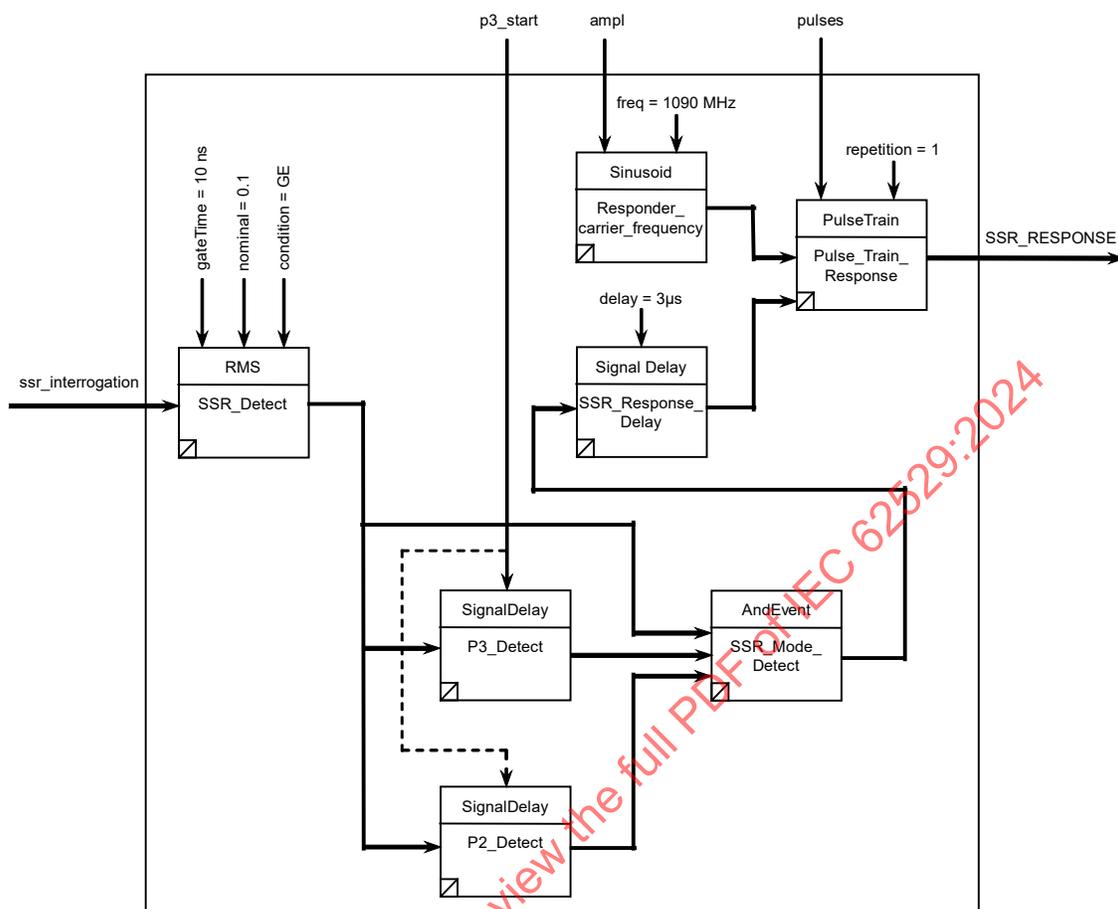


Figure E.49—TSF SSR\_RESPONSE

### E.29.2 Interface properties

See Table E.53 for details of the TSF SSR\_RESPONSE interface.

Table E.53—TSF SSR\_RESPONSE interface

Description	Name	Type	Default	Range
Carrier amplitude	Ampl	Physical	—	—
P3 pulse start time	p3_start	Time	3 us	3 μs   5 μs   8 μs   17 μs   21 μs   25 μs
SSR Response pulse train	Pulses	PulseDefns	[ ]	—
Transmitted Interrogation signal	ssr_interrogation	SignalFunction	—	—

**E.29.3 Notes**

The response is initiated 3 μs after the third pulse of a valid interrogation is received.

The parameters of the array of pulses are defined in Table E.54. Pulse F1 and pulse F2 have to be present. Pulse X is not currently used and should be omitted. Other pulses may be specified as required.

**Table E.54—SSR\_RESPONSE pulse descriptions**

Pulse	Start (μs)	PulseWidth (μs)	LevelFactor
F1	0	0.45	1
C1	1.45	0.45	1
A1	2.9	0.45	1
C2	4.35	0.45	1
A2	5.8	0.45	1
C4	7.25	0.45	1
A4	8.7	0.45	1
X	10.15	0.45	1
B1	11.6	0.45	1
D1	13.05	0.45	1
B2	14.5	0.45	1
D2	15.95	0.45	1
B4	17.4	0.45	1
D4	18.85	0.45	1
F2	20.3	0.45	1
P1	24.65	0.45	1

**E.29.4 Model description**

See Table E.55 for details of the TSF SSR\_RESPONSE model.

**Table E.55—TSF SSR\_RESPONSE model**

Name	Type	Terminal	Inputs	Output	Formula
Pulse_Train_Response	PulseTrain	Signal [Out]	—	SSR_RESPONSE	—
		pulses	pulses	—	—
		repetition	—	—	1
		Signal [In]	Responder_carrier frequency	—	—
		Sync [In]	SSR_Response_Delay	—	—
Responder_carrier_frequency	Sinusoid	Signal [Out]	—	Pulse Train Response	—
		amplitude	ampl	—	—
		frequency	—	—	1090 MHz
		phase	—	—	0 rad

**Table E.55—TSF SSR\_RESPONSE model (continued)**

Name	Type	Terminal	Inputs	Output	Formula
SSR_Response_Delay	SignalDelay	Signal [Out]	—	Pulse Train Response	—
			—	—	—
		acceleration	—	—	0 Hz
		delay	—	—	3 μs
		rate	—	—	0%
SSR_Mode_Detect	AndEvent	Event [Out]	—	SSR Response Delay	—
		Signal [In]	SSR_Detect	—	—
		Signal [In]	P2_Detect	—	—
		Signal [In]	P3_Detect	—	—
P3_Detect	SignalDelay	Signal [Out]	—	SSR_Mode_Detect	—
		acceleration	—	—	0 Hz
		delay	p3_start	—	—
		rate	—	—	0%
		Signal [In]	SSR_Detect	—	—
P2_Detect	SignalDelay	Signal [Out]	—	SSR_Mode_Detect	—
		acceleration	—	—	0 Hz
		delay	—	—	p3_start - 2.0e-6
		rate	—	—	0%
SSR_Detect	RMS	[Out]	—	SSR_Mode_Detect, P2_Detect, P3_Detect	—
		measuredVariable	—	—	—
		measurement	—	—	—
		measurements	—	—	—
		sample	—	—	—
		count	—	—	—
		gateTime	—	—	10.0e-9 s
		nominal	—	—	0.1
		condition	—	—	GE
		GO	—	—	—
		NOGO	—	—	—
		HI	—	—	—
		LO	—	—	—
		UL	—	—	—
LL	—	—	—		
Signal [As]	—	—	—		
Signal [In]	ssr_interrogation	—	—		

**E.29.5 Rules**

For this signal, the allowable types are Voltage, Current, and Power. The type selected needs to agree with the type of the SSR\_INTERROGATION signal that triggers the SSR\_RESPONSE.

**E.29.6 Example**

See Figure E.50 for an example of SSR\_RESPONSE.

*XML Static Signal Description:*

```
<SSR_RESPONSE name="SSR_RESPONSE4" p3_start="8 us"
pulses="(0,0.00000045,1), (0.00000145,0.00000045,1),
```

```
(0.0000029,0.00000045,1),
(0.0000058,0.00000045,1),
(0.0000087,0.00000045,1),
(0.0000116,0.00000045,1),
(0.0000145,0.00000045,1),
(0.0000174,0.00000045,1),
(0.0000203,0.00000045,1),
(0.0000435,0.00000045,1),
(0.0000725,0.00000045,1),
(0.0001015,0.00000045,1),
(0.0001305,0.00000045,1),
(0.0001595,0.00000045,1),
(0.0001885,0.00000045,1),
(0.0002465,0.00000045,1)"
In="SSR_INTERROGATION3"/>
<SSR_INTERROGATION name="SSR_INTERROGATION3" mode="3" p3_start="8 us"
p3_level="2" />
```

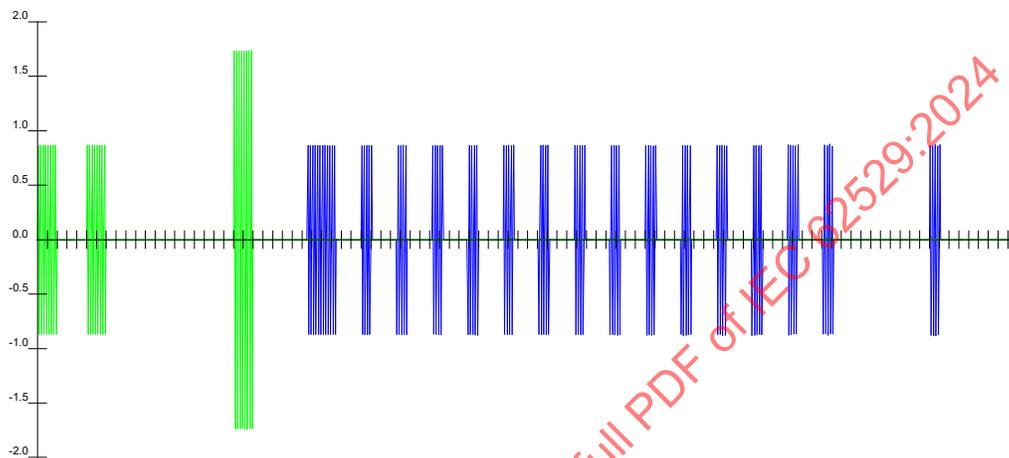


Figure E.50—SSR\_RESPONSE example

### E.30 STEP\_SIGNAL

#### E.30.1 Definition

A change of dc electrical potential from one level to another, either positive or negative. See Figure E.51.

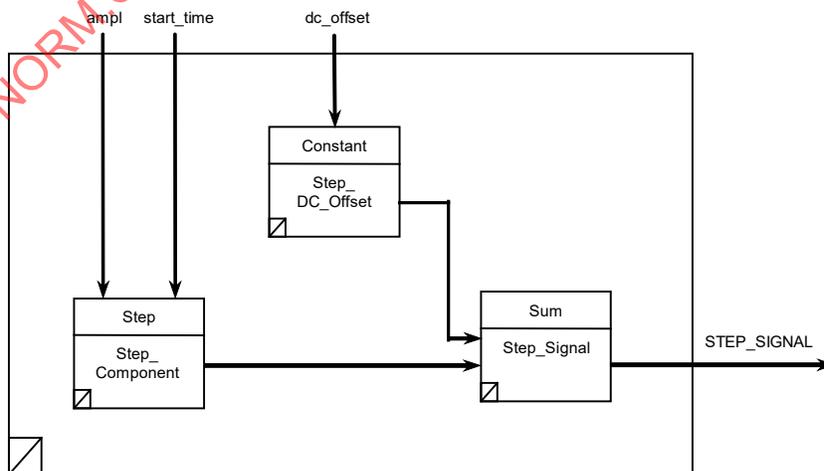


Figure E.51—TSF STEP\_SIGNAL

### E.30.2 Interface properties

See Table E.56 for details of the TSF STEP\_SIGNAL interface.

**Table E.56—TSF STEP\_SIGNAL interface**

Description	Name	Type	Default	Range
Step size	ampl	Voltage	—	—
DC Offset	dc_offset	Voltage	0 V	—
Step time	start_time	Time	—	—

### E.30.3 Notes

There are no special notes for this TSF.

### E.30.4 Model description

See Table E.57 for details of the TSF STEP\_SIGNAL model.

**Table E.57—TSF STEP\_SIGNAL model**

Name	Type	Terminal	Inputs	Output	Formula
Step_Signal	Sum	Signal [Out]		STEP_SIGNAL	—
		Signal [In]	Step	—	—
		Signal [In]	Step_DC_Offset	—	—
Step_Component	Step	Signal [Out]	—	Step_Signal	—
		amplitude	ampl	—	—
		startTime	start_time	—	—
Step_DC_Offset	Constant	Signal [Out]	—	Step_Signal	—
		amplitude	dc_offset	—	—

### E.30.5 Rules

There are no special rules for this TSF.

### E.30.6 Example

See Figure E.52 for an example of STEP\_SIGNAL.

*XML Static Signal Description:*

```
<STEP_SIGNAL name="STEP_SIGNAL4" ampl="1 V" dc_offset="1 V"
start_time="0.5 s" />
```

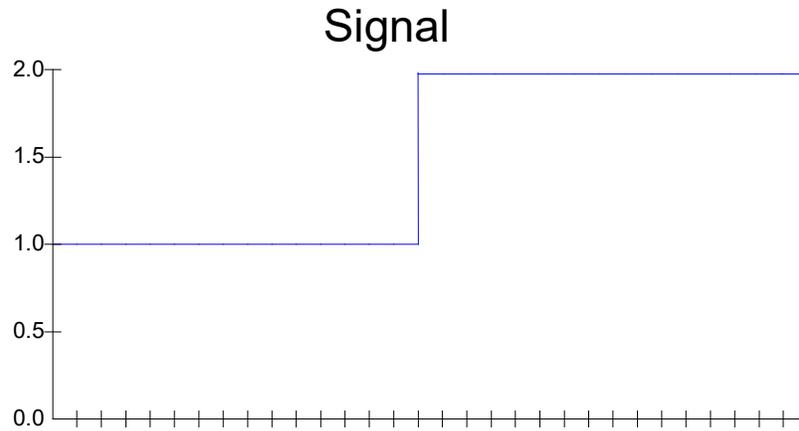


Figure E.52—STEP\_SIGNAL example

### E.31 SUP\_CAR\_SIGNAL

#### E.31.1 Definition

An amplitude-modulated signal in which the carrier is suppressed. See Figure E.53.

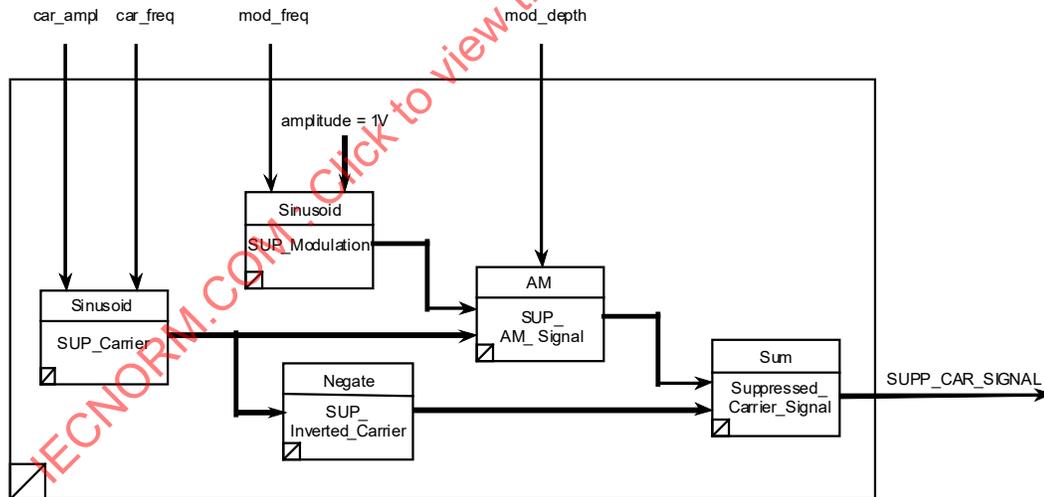


Figure E.53—TSF SUP\_CAR\_SIGNAL

#### E.31.2 Interface properties

See Table E.58 for details of the TSF SUP\_CAR\_SIGNAL interface.

**Table E.58—TSF SUP\_CAR\_SIGNAL interface**

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Voltage	—	—
Carrier frequency	car_freq	Frequency	—	—
Modulation frequency	mod_freq	Frequency	—	—
Depth of modulation	mod_depth	Ratio	—	—

**E.31.3 Notes**

There are no special notes for this TSF.

**E.31.4 Model description**

See Table E.59 for details of the TSF SUP\_CAR\_SIGNAL model.

**Table E.59—TSF SUP\_CAR\_SIGNAL model**

Name	Type	Terminal	Inputs	Output	Formula
Suppressed_Carrier_Signal	Sum	Signal [Out]	—	SUP_CAR_SIGNAL	—
		Signal [In]	SUP_Inverted_Carrier	—	—
		Signal [In]	SUP_AM_Signal	—	—
SUP_Inverted_Carrier	Negate	Signal [Out]	—	Suppressed_Carrier_Signal	—
		Signal [In]	SUP_Carrier	—	—
SUP_AM_Signal	AM	Signal [Out]	—	Suppressed_Carrier_Signal	—
		modIndex	mod_depth	—	—
		Carrier [In]	SUP_Carrier	—	—
		Signal [In]	SUP_Modulation	—	—
SUP_Modulation	Sinusoid	Signal [Out]	—	SUP_AM_Signal	—
		amplitude	—	—	1 V (see NOTE)
		frequency	mod_freq	—	—
		phase	—	—	0 rad
SUP_Carrier	Sinusoid	Signal [Out]	—	SUP_Inverted_Carrier, SUP_AM_Signal	—
		amplitude	car_ampl	—	—
		frequency	car_freq	—	—
		phase	—	—	0 rad

NOTE—The BSC requires a unity value for the amplitude of the modulating signal.

### E.31.5 Rules

The output is defined by Equation (E.11).

$$e = (E_m E_c / 2) \cos(\omega_c + \omega_m)t + (E_m E_c / 2) \cos(\omega_c - \omega_m)t \tag{E.11}$$

where

- $E_m$  is the modulation signal amplitude
- $E_c$  is the carrier amplitude (unmodulated)
- $\omega_m$  is  $2\pi \times$  modulating frequency
- $\omega_c$  is  $2\pi \times$  carrier frequency

### E.31.6 Example

See Figure E.54 for an example of SUP\_CAR\_SIGNAL.

*XML Static Signal Description:*

```
<SUP_CAR_SIGNAL name="SUP_CAR_SIGNAL8" car_ampl="1" car_freq="10 kHz"
mod_freq="1 kHz" mod_index="0.3" />
```

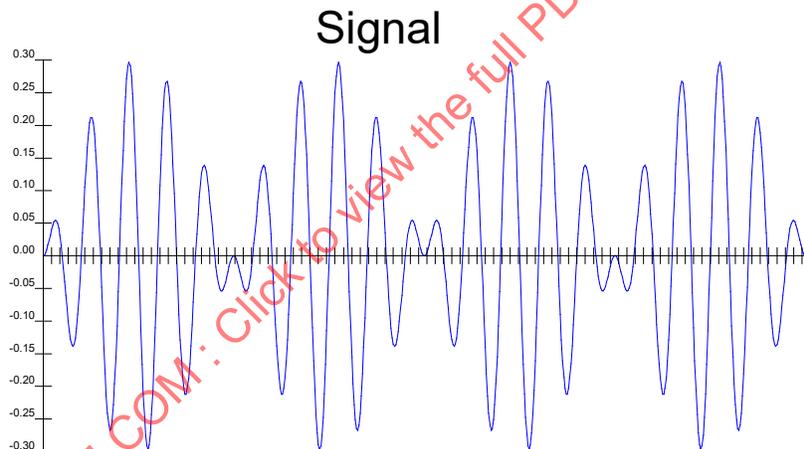


Figure E.54—SUP\_CAR\_SIGNAL example

## E.32 SYNCHRO

### E.32.1 Definition

Three ac sinusoid voltages whose relationships of amplitude represent the rotational shaft position of an electromechanical transducer. See Figure E.55.

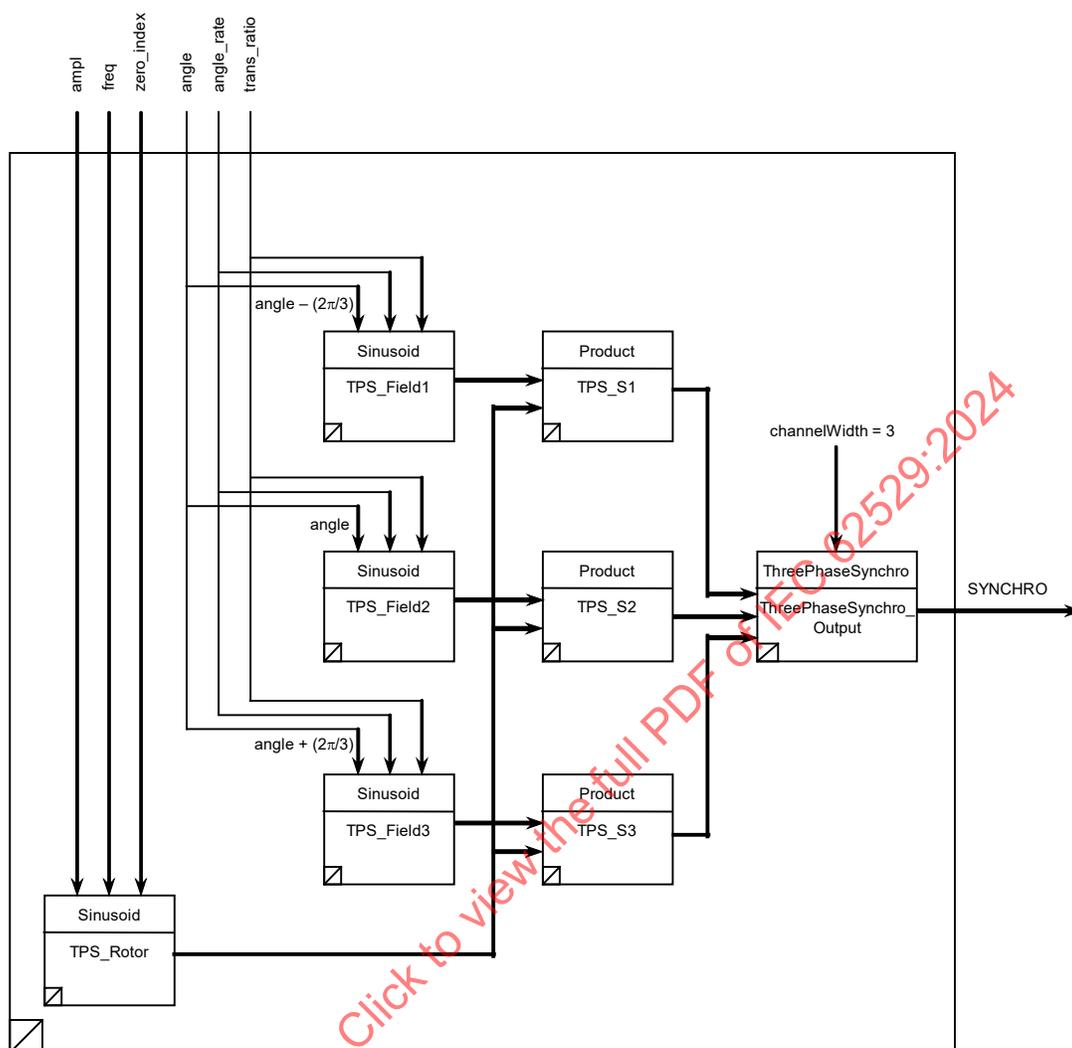


Figure E.55—TSF SYNCHRO

### E.32.2 Interface properties

See Table E.60 for details of the TSF SYNCHRO interface.

Table E.60—TSF SYNCHRO interface

Description	Name	Type	Default	Range
Shaft angle	angle	PlaneAngle	0	—
Reference amplitude	ampl	Voltage	26 V	26 V – 119 V
Reference frequency	freq	Frequency	400 Hz	30 Hz – 54 kHz
Zero index	zero_index	PlaneAngle	0 rad	0 – 2π rad
Shaft angle rate	angle_rate	Frequency	0 Hz	—
Transformer ratio	trans_ratio	Ratio	1	—

### E.32.3 Notes

This model does not consider the effects of angular velocity of the rotor and the quadrature voltages generated in the stator windings.

### E.32.4 Model description

See Table E.61 for details of the TSF SYNCHRO model.

**Table E.61—TSF SYNCHRO model**

Name	Type	Terminal	Inputs	Output	Formula
ThreePhaseSynchro_ Output	ThreePhaseSynchro	Signal [Out]	—	SYNCHRO	—
		channelWidth	—	—	3
		Signal [In]	TPS_S1	—	—
		Signal [In]	TPS_S2	—	—
TPS_S1	Product	Signal [Out]	—	ThreePhaseSynchro_Output	—
		Signal [In]	TPS_Field1	—	—
		Signal [In]	TPS_Rotor	—	—
TPS_S2	Product	Signal [Out]	—	ThreePhaseSynchro_Output	—
		Signal [In]	TPS_Field2	—	—
		Signal [In]	TPS_Rotor	—	—
TPS_S3	Product	Signal [Out]	—	ThreePhaseSynchro_Output	—
		Signal [In]	TPS_Field3	—	—
		Signal [In]	TPS_Rotor	—	—
TPS_Field1	Sinusoid	Signal [Out]	—	S1	—
		amplitude	trans_ratio	—	—
		frequency	angle_rate	—	—
		phase	—	—	angle - (2π/3)
TPS_Field2	Sinusoid	Signal [Out]	—	TPS_S2	—
		amplitude	trans_ratio	—	—
		frequency	angle_rate	—	—
		phase	angle	—	—
TPS_Field3	Sinusoid	Signal [Out]	—	TPS_S3	—
		amplitude	trans_ratio	—	—
		frequency	angle_rate	—	—
		phase	—	—	angle + (2π/3)
TPS_Rotor	Sinusoid	Signal [Out]	—	TPS_S2, TPS_S1, TPS_S3	—
		amplitude	ampl	—	—
		frequency	freq	—	—
		phase	zero_index	—	—

### E.32.5 Rules

The outputs of the synchro stator windings are given by Equation (E.12), Equation (E.13), and Equation (E.14).

S1

$$E_{s1} = KE_r \sin(\theta - 2\pi/3) \sin(2\pi f_r t + \varphi) \quad (\text{E.12})$$

S2

$$E_{s2} = KE_r \sin \theta \sin(2\pi f_r t + \varphi) \quad (\text{E.13})$$

S3

$$E_{s3} = KE_r \sin(\theta + 2\pi/3) \sin(2\pi f_r t + \varphi) \quad (\text{E.14})$$

where

- $K$  is the transformer ratio (trans\_ratio), assuming  $K$  to be the same for all stator windings
- $E_r$  is the reference amplitude in the primary (ampl)
- $\theta$  is angular displacement of the rotor (angle)
- $f_r$  is the reference frequency of the signal in the primary (freq)
- $\varphi$  is the zero index position of the rotor (zero\_index)

Thus, the operation of the synchro may be modeled as the product of two signals for each output:

S1

$$E_{s1} = (E_r \sin(2\pi f_r t + \varphi)) \times (K \sin(\theta - 2\pi/3)) \quad (\text{E.15})$$

S2

$$E_{s2} = (E_r \sin(2\pi f_r t + \varphi)) \times (K \sin(\theta)) \quad (\text{E.16})$$

S3

$$E_{s3} = (E_r \sin(2\pi f_r t + \varphi)) \times (K \sin(\theta + 2\pi/3)) \quad (\text{E.17})$$

### E.32.6 Example

See Figure E.56 for an example of SYNCHRO.

*XML Static Signal Description:*

```
<SYNCHRO name="SYNCHRO5" angle_rate="5" freq="20 Hz" />
```

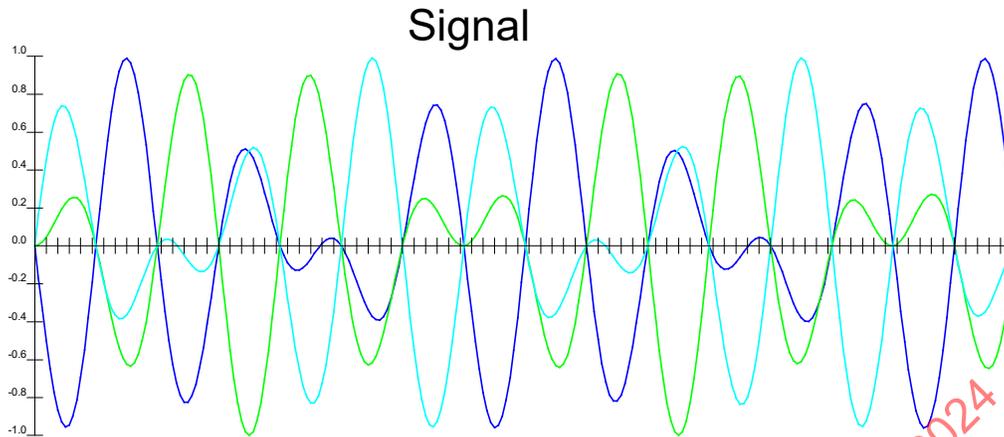


Figure E.56—SYNCHRO example

### E.33 TACAN

#### E.33.1 Definition

Tactical air navigation (TACAN) is a complete UHF polar coordinate navigation system using pulse techniques. The function operates identically as a DME, and the bearing function is derived by rotating the ground transponder antenna to obtain a rotating multilobe pattern for coarse and fine bearing information, as defined in MIL-STD-291B [B17]. See Figure E.57.

The model defines a subset of the TACAN X signal concerned with bearing, rather than the complete signal, as test requirements dealing with TACAN distance can be refined using the DME model.

The transponder emits RF pulses that are amplitude-modulated to provide bearing information. The amplitude modulation is produced by rotating a parasitic reflector array about the antenna radiating element. The array consists of one 15 Hz and nine 135 Hz reflectors. As the pattern from the 15 Hz reflector passes through the magnetic east azimuth, a main reference burst (MRB) is transmitted. As the pattern from the 135 Hz reflectors passes through east, an auxiliary reference burst (ARB) is transmitted, except when the pattern is coincident with the 15 Hz pattern. This sequence produces a total of one MRB and eight ARB bursts per antenna rotation. The airborne receiving equipment determines the aircraft bearing from the ground station by measuring elapsed time, first, from the MRB to the 0° phase of the 15 Hz component and, second, from the ARB to 0° of the 135 Hz component.

The TACAN beacon also generates a two- or three-letter Morse identification signal every 37.5 s.



**Table E.62—TSF TACAN interface**

Description	Name	Type	Default	Range
Transponder frequency	freq	Frequency	962 MHz	962 MHz – 1213 MHz
Modulation index	mod_index	Ratio	0.3	0 – 1
Magnetic bearing	bearing	PlaneAngle	0°	0° – 360°
Carrier amplitude	car_ampl	Voltage	—	—

**E.33.3 Notes**

The transponder generates 2700 pulse pairs per second, but with a jittered pulse repetition frequency (PRF).

The rotating antenna modulates this signal at 15 Hz and 135 Hz using the same principles as the variable phase in a VHF omnidirectional range (VOR) signal.

The MRB and ARB comprise 12 and 6 equally spaced pulse pairs, respectively. Spacing has been assumed to be 30 s in the model. MRB and ARB pulse trains take priority over interrogator and randomly generated pulse pairs; therefore, the model suppresses these pulse pairs at the appropriate time.

This model is a limited implementation to provide the basic TACAN signal. Many properties have been included as fixed parameters and have not been made externally accessible to the user. Some parameters, such as the beacon identification signal (comprising two or three Morse letters) and speed (i.e., variable pulse width and spacing) have not been addressed in model.

**E.33.4 Model description**

See Table E.63 for details of the TSF TACAN model.

**Table E.63—TSF TACAN model**

Name	Type	Terminal	Inputs	Output	Formula
TACAN_Response_Train	PulseTrain	Signal [Out]	—	TACAN	—
		pulses	—	—	(0 μs, 3.5 μs, 1), (15.5 μs, 3.5 μs, 1)
		repetition	—	—	0
		Signal [In]	Tacan_Modulated_Carrier	—	—
		Gate [In]	Tacan_Event_Train	—	—
Tacan_Modulated_Carrier	AM	Signal [Out]	—	Tacan_Response_Train	—
		modIndex	mod_index	—	—
		Carrier [In]	Tacan_Carrier	—	—
		Signal [In]	Tacan_Modulation	—	—
Tacan_Event_Train	OrEvent	Event [Out]	—	Tacan_Response_Train	—
		Signal [In]	Random_Event_B	—	—
		Signal [In]	Reference_Burst	—	—

**Table E.63—TSF TACAN model (continued)**

Name	Type	Terminal	Inputs	Output	Formula
Tacan_Modulation	Sum	Signal [Out]	—	Tacan_Modulated_Carrier	—
		Signal [In]	Tacan_135Hz_Modulation	—	—
		Signal [In]	Tacan_15Hz_Modulation	—	—
Reference_Burst	TimedEvent	Event [Out]	—	Tacan_Event_Train	—
		delay	—	—	10 μs
		duration	—	—	20 μs
		period	—	—	50 μs
		repetition	—	—	0
Random_Event_B	ProbabilityEvent	Gate [In]	RB_Gate	—	—
		Event [Out]	—	Tacan_Event_Train	—
		seed	—	—	0
		probability	—	—	10% (reply efficiency)
		Signal [In]	Random_Event_A	—	—
Tacan_Carrier	Sinusoid	Gate [In]	RB_Mask	—	—
		Signal [Out]	—	Tacan_Modulated_Carrier	—
		amplitude	car_ampl	—	—
		frequency	freq	—	—
Random_Event_A	TimedEvent	phase	—	—	0 rad
		Event [Out]	—	Random_Event_B	—
		delay	—	—	0 s
		duration	—	—	20 μs
		period	—	—	37 μs
Tacan_15Hz_Modulation	Sinusoid	repetition	—	—	0
		Signal [Out]	—	Tacan_Modulation	—
		amplitude	—	—	0.5 V
		frequency	—	—	15 Hz
Tacan_135Hz_Modulation	Sinusoid	phase	bearing	—	—
		Signal [Out]	—	Tacan_Modulation	—
		amplitude	—	—	0.5 V
		frequency	—	—	135 Hz
RB_Mask	NotEvent	phase	bearing	—	—
		Event [Out]	—	Random_Event_B	—
RB_Gate	OrEvent	Signal [In]	RB_Gate	—	—
		Event [Out]	—	RB_Mask, Reference_Burst	—
		Signal [In]	ARB_Gate	—	—
MRB_Gate	TimedEvent	Signal [In]	MRB_Gate	—	—
		Event [Out]	—	RB_Gate	—
		delay	—	—	0 s
		duration	—	—	600 μs
		period	—	—	15 Hz
repetition	—	—	0		

**Table E.63—TSF TACAN model (continued)**

Name	Type	Terminal	Inputs	Output	Formula
ARB_Gate	TimedEvent	Event [Out]	—	RB_Gate	—
		delay	—	—	0 s
		duration	—	—	300 μs
		period	—	—	135 Hz
		repetition	—	—	0

**E.33.5 Rules**

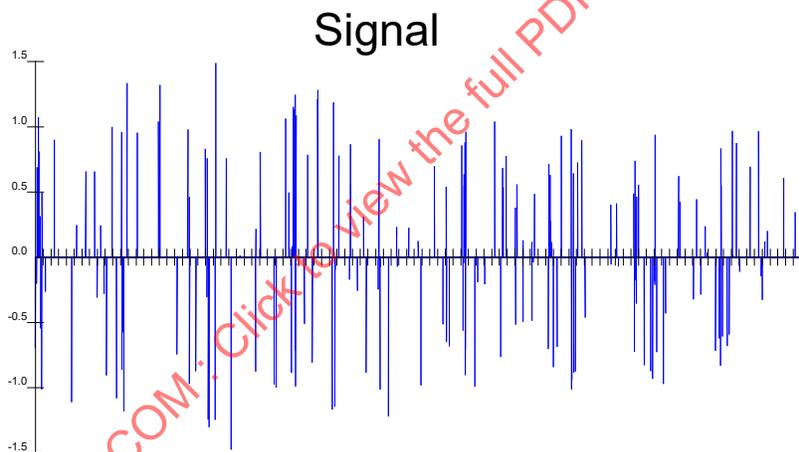
There are no special rules for this TSF.

**E.33.6 Example**

See Figure E.58 for an example of TACAN.

*XML Static Signal Description:*

```
<TACAN name="TACAN2" />
```



**Figure E.58—TACAN example**

**E.34 TRIANGULAR\_WAVE\_SIGNAL<type: Voltage|| Current|| Power>**

**E.34.1 Definition**

A periodic wave whose instantaneous value varies alternately and linearly between two specified values (i.e., initial and alternate). The interval required to transit from the initial value to the alternate value is equal to the interval to transition from the alternate value to the initial value. See Figure E.59.

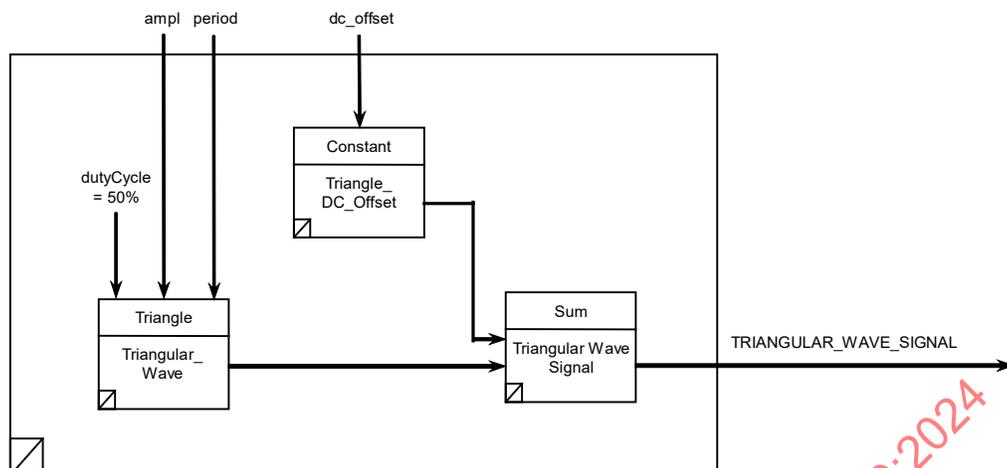


Figure E.59—TSF TRIANGULAR\_WAVE\_SIGNAL

### E.34.2 Interface properties

See Table E.64 for details of the TSF TRIANGULAR\_WAVE\_SIGNAL interface.

Table E.64—TSF TRIANGULAR\_WAVE\_SIGNAL interface

Description	Name	Type	Default	Range
Triangular wave signal amplitude	ampl	Physical	—	—
Triangular wave signal period	period	Time	—	—
DC offset	dc_offset	Physical	0	—

### E.34.3 Notes

There are no special notes for this TSF.

### E.34.4 Model description

See Table E.65 for details of the TSF TRIANGULAR\_WAVE\_SIGNAL model.

Table E.65—TSF TRIANGULAR\_WAVE\_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
Triangular_Wave_Signal	Sum	Signal [Out]	—	TRIANGULAR_WAVE_SIGNAL	—
		Signal [In]	Triangle_DC_Offset	—	—
		Signal [In]	Triangular_Wave	—	—
Triangular_Wave	Triangle	Signal [Out]	—	Triangular_Wave_Signal	—
		amplitude	ampl	—	—
		period	period	—	—
		dutyCycle	—	—	50%
Triangle_DC_Offset	Constant	Signal [Out]	—	Triangular_Wave_Signal	—
		amplitude	dc_offset	—	—

### E.34.5 Rules

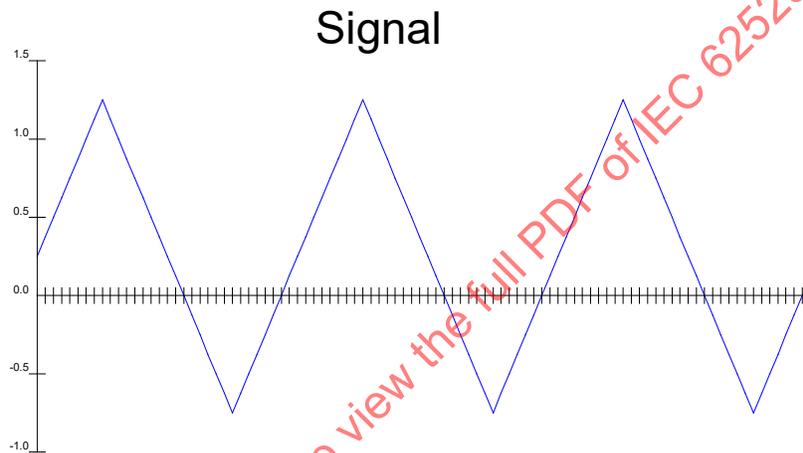
For this signal, the allowable types are Voltage, Current, and Power. All types need to be consistent. Thus, for example, if the triangular wave signal amplitude is specified in volts, then the dc offset shall also be specified in volts.

### E.34.6 Example

See Figure E.60 for an example of TRIANGULAR\_WAVE\_SIGNAL.

*XML Static Signal Description:*

```
<TRIANGULAR_WAVE_SIGNAL name="TRIANGULAR_WAVE_SIGNAL6" ampl="1"  
dc_offset="250 mV" period="0.001 s" />
```



**Figure E.60—TRIANGULAR\_WAVE\_SIGNAL example**

## E.35 VOR

### E.35.1 Definition

VHF omnidirectional range (VOR) is a system combining ground and airborne equipment to provide bearing to or from a ground station, as defined in ARINC 579-2 [B3]. See Figure E.61. The VOR radiates a RF carrier in the band of 108.0 MHz to 117.975 MHz, with which are associated two separate 30 Hz modulations. The phase of one of these modulations is independent of the point of observation (i.e., reference phase). The phase of the other modulation (variable phase) is such that, at a point of observation, it differs from the reference phase by an angle equal to the bearing of the point of observation with respect to the VOR. The two separate modulations consist of the following:

- A subcarrier of 9960 Hz, frequency-modulated at 30 Hz, modulating the carrier to a nominal depth of 30%. This 30 Hz component is fixed independently of the azimuth and is termed the reference phase.
- A 30 Hz component, modulating the carrier to a nominal depth of 30%. This 30 Hz component is caused by a rotating antenna that produces a change in phase with azimuth and is termed the variable phase.

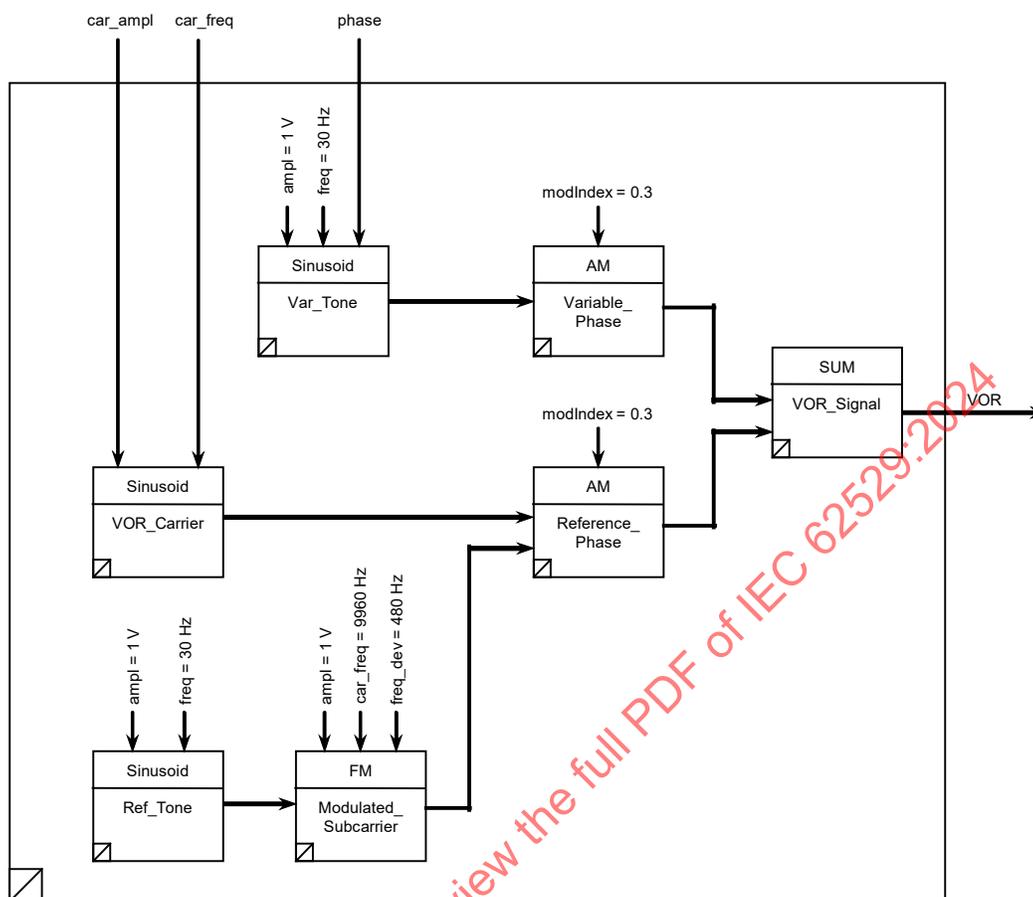


Figure E.61—TSF VOR

### E.35.2 Interface properties

See Table E.66 for details of the TSF VOR interface.

Table E.66—TSF VOR interface

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Voltage	2 mV	—
Carrier frequency	car_freq	Frequency	107.975 MHz	107.975 MHz – 117.975 MHz
Radial bearing	phase	PlaneAngle	90°	0° – 360°

### E.35.3 Notes

This model has limited functionality. It does not provide for the variation of some of the parameters (such as the tone frequencies). The model may be modified by the user to include such parameters in the interface properties.

### E.35.4 Model description

See Table E.67 for details of the TSF VOR model.

**Table E.67—TSF VOR model**

Name	Type	Terminal	Inputs	Output	Formula
VOR_Signal	Sum	Signal [Out]	—	VOR	—
		Signal [In]	Reference_Phase	—	—
		Signal [In]	Variable_Phase	—	—
Reference_Phase	AM	Signal [Out]	—	VOR	—
		modIndex	—	—	0.3
		Carrier [In]	VOR_Carrier	—	—
		Signal [In]	Modulated_Subcarrier	—	—
Variable_Phase	AM	Signal [Out]	—	VOR	—
		modIndex	—	—	0.3
		Carrier [In]	VOR_Carrier	—	—
		Signal [In]	Var_Tone	—	—
VOR_Carrier	Sinusoid	Signal [Out]	—	Reference_Phase, Variable_Phase	—
		amplitude	car-ampl	—	car_ampl/2
		frequency	car_freq	—	—
		phase	—	—	0°
Modulated_Subcarrier	FM	Signal [Out]	—	Reference_Phase	—
		amplitude	—	—	1 V (see NOTE)
		carrierFrequency	—	—	9960 Hz
		frequencyDeviation	—	—	480 Hz
		Signal [In]	Ref_Tone	—	—
Var_Tone	Sinusoid	Signal [Out]	—	Variable_Phase	—
		amplitude	—	—	1 V (see NOTE)
		frequency	—	—	30 Hz
		phase	phase	—	—
Ref_Tone	Sinusoid	Signal [Out]	—	Modulated_Subcarrier	—
		amplitude	—	—	1 V (see NOTE)
		frequency	—	—	30 Hz
		phase	—	—	0

NOTE—The BSC requires a unity value for the amplitude of the modulating signal.

### E.35.5 Rules

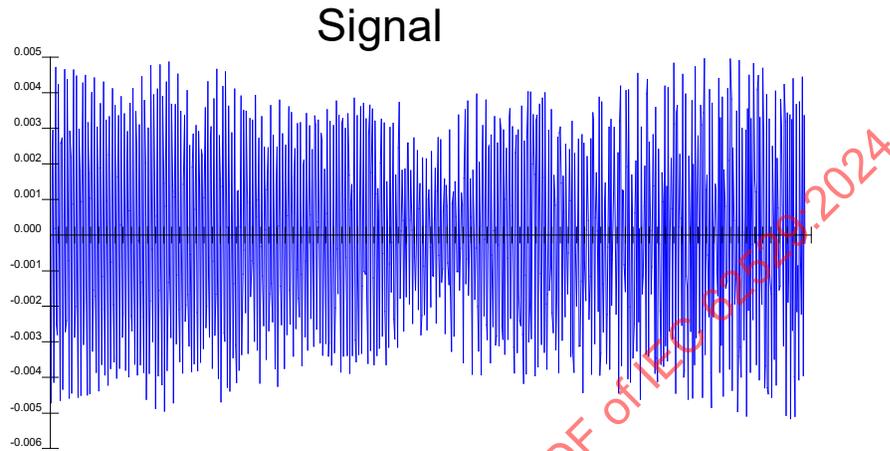
There are no special rules for this TSF.

### E.35.6 Example

See Figure E.62 for an example of VOR.

*XML Static Signal Description:*

```
<VOR name="VOR7" />
```



**Figure E.62—VOR example**

## Annex F

(informative)

### Test signal framework (TSF) library for digital pulse classes

#### F.1 Introduction

This annex provides a TSF library representing several digital pulse class signals. It illustrates the use of the digital stream basic signal components (BSCs) to create typical digital signals using different pulse classes. An example of a TSF to reference digital files using the Digital Test Interchange Format (DTIF) standard (i.e., IEEE Std 1445™-1998 [B15]) is also provided.

#### F.2 TSF library definition in extensible markup language (XML)

Where examples are given, their static signal description is provided in XML. The information provided in Annex I, together with the detailed description of each TSF model in this annex, may be used to create the example TSF library for the digital pulse classes that conforms to the XML Schema document defined in Annex I.

A complete XML instance document conforming to the requirements of this standard may be obtained from the IEEE Standards download site (see IEEE 1641™) <https://standards.ieee.org/downloads.html>

#### F.3 Graphical models of TSFs

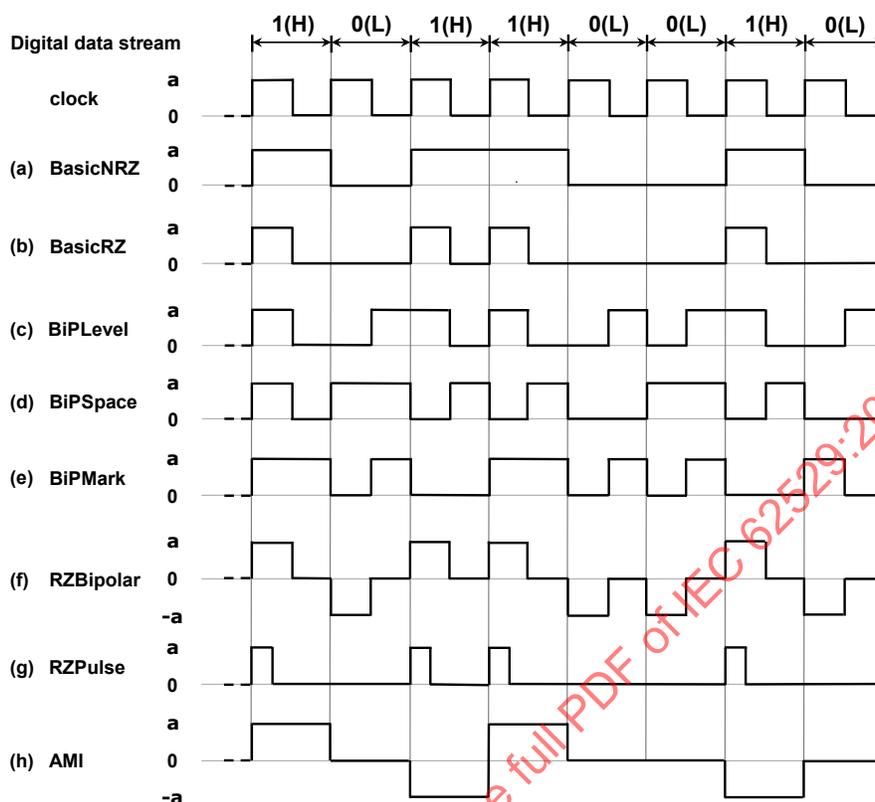
A diagram is provided with each signal to illustrate graphically the relationship between the BSCs and interface attributes that make up the signal. In order to reduce the amount of information included in each diagram, inputs to BSCs that are at zero or the default values are omitted.

#### F.4 Pulse class family of TSFs

The Pulse Class TSFs are designed to be used with an event stream. The Encode BSC is used, which provides a basic digital signal as a stream of bits derived from the data information. A Pulse Class TSF may then be used to turn the digital stream into a real physical signal. Similarly, Decode is used to allow incoming digital signals to be measured and compared to the expected values.

##### F.4.1 Pulse classes

Figure F.1 illustrates the various pulse classes that are covered by TSFs in this annex. At the top of the diagram is the digital data stream that is conveyed by the signal in each pulse class, i.e., the pattern "HLHLLHL" or "10110010".



**Figure F.1—Digital pulse classes**

The values on the left of the diagram indicate the amplitude that the physical signal takes while transmitting the data. In the simplest case, Basic NRZ or basic nonreturn to zero, the nonzero amplitude represents a logic one or High, and the zero amplitude represents a logic 0 or Low. Other pulse classes involve amplitude transitions or have both positive and negative amplitudes within the signal. This is explained further in each specific TSF example.

Many of the TSFs in this annex produce the same pulse class waveform as the pulse class attribute types defined for use with the digital BSCs (see Annex B). The following table shows the equivalence between the BSC pulse class attributes types and the digital TSFs in this annex.

BSC pulse class attribute types	Equivalent digital TSF	Description
NRZ	BasicNRZ	Nonreturn to zero
RZ	BasicRZ	Return to zero
R1	—	Return to one
RZPulse	RZPulse	Pulse return to zero
BiPLevel	BiPLevel	Bi-phase level
BiPMark	BiPMark	Bi-phase mark (pulse 0)
BiPSpace	BiPSpace	BiPhase space (pulse 1)

This annex include two TSF models, RZBipolar (return to zero bipolar) and AMI (alternate mark inversion), which are not available as BSC pulse class types.

## F.4.2 BasicNRZ

### F.4.2.1 Definition

A pulse class in which the digital data is carried by two physical signal levels (most often two voltage levels), in which one level represents a logic one or High and the other represents a logic zero or Low.

See Figure F.1 for a typical waveform and Figure F.2 for a graphical model of the TSF.

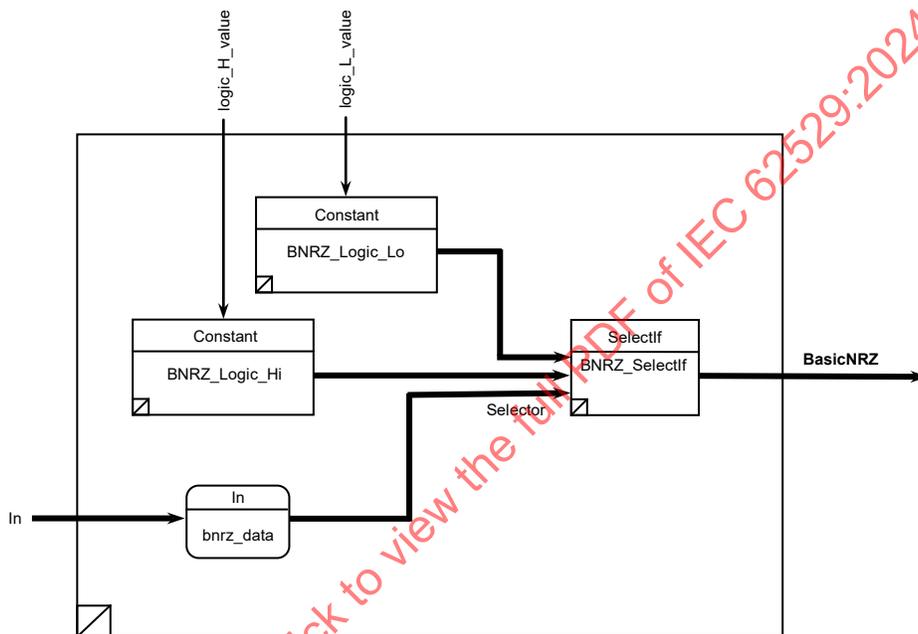


Figure F.2—TSF BasicNRZ

### F.4.2.2 Interface properties

See Table F.1 for details of the TSF BasicNRZ interface.

Table F.1—TSF BasicNRZ interface

Description	Name	Type	Default	Range
Logic High Value	logic_H_value	Physical	—	—
Logic Low Value	logic_L_value	Physical	—	—
Digital Stream Input	bnrz_data	SignalFunction	—	—

### F.4.2.3 Notes

There are no special notes for this TSF.

#### F.4.2.4 Model description

See Table F.2 for details of the TSF BasicNRZ model.

**Table F.2—TSF BasicNRZ model**

Name	Type	Terminal	Inputs	Output	Formula
BNRZ_SelectIf	SelectIf	Signal [Out]	—	BasicNRZ	—
		Selector	bnrz_data	—	—
		Signal [In]	BNRZ_Logic_Lo	—	—
		Signal [In]	BNRZ_Logic_Hi	—	—
BNRZ_Logic_Hi	Constant	Signal [Out]	—	BNRZ_SelectIf	—
		amplitude	logic_H_value	—	—
BNRZ_Logic_Lo	Constant	Signal [Out]	—	BNRZ_SelectIf	—
		amplitude	logic_L_value	—	—

#### F.4.2.5 Rules

There are no special rules for this TSF.

#### F.4.3 BasicRZ

##### F.4.3.1 Definition

A pulse class in which each bit period is subdivided into two subperiods. The binary data is carried in the first subperiod. A logic one or High is carried by a pulse of one amplitude, and the logic zero or Low is carried by a pulse of a different amplitude. The second subperiod contains a “no pulse” condition, which is at the same level as the logic zero amplitude.

See Figure F.1 for a typical waveform and Figure F.3 for a graphical model of the TSF.

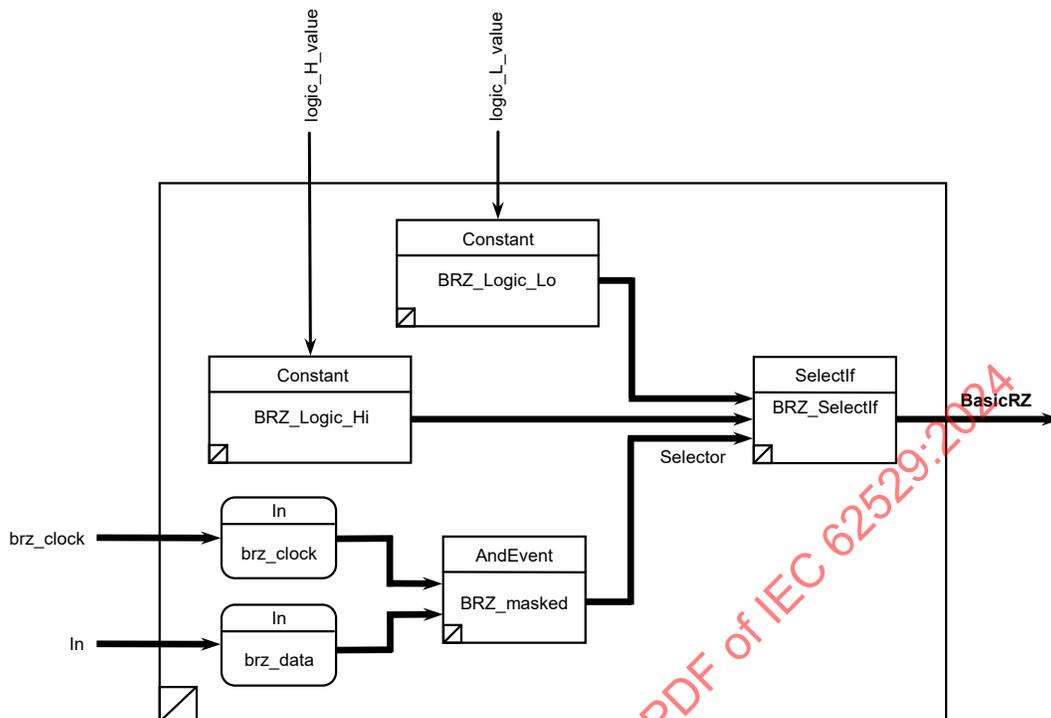


Figure F.3—TSF BasicRZ

#### F.4.3.2 Interface properties

See Table F.3 for details of the TSF BasicRZ interface.

Table F.3—TSF BasicRZ interface

Description	Name	Type	Default	Range
Logic High Value	<code>logic_H_value</code>	Physical	—	—
Logic Low Value	<code>logic_L_value</code>	Physical	—	—
Input Clock	<code>brz_clock</code>	SignalFunction	—	—
Digital Stream Input	<code>brz_data</code> (In)	SignalFunction	—	—

#### F.4.3.3 Notes

The Input Clock needs to have the same period as the data rate from the digital stream.

#### F.4.3.4 Model description

See Table F.4 for details of the TSF BasicRZ model.

**Table F.4—TSF BasicRZ model**

Name	Type	Terminal	Inputs	Output	Formula
BRZ_SelectIf	SelectIf	Signal [Out]	—	BasicRZ	—
		Selector	BRZ_Masked	—	—
		Signal [In]	BRZ_Logic_Lo	—	—
		Signal [In]	BRZ_Logic_Hi	—	—
BRZ_Masked	AndEvent	Signal [Out]	—	BRZ_SelectIf	—
		Signal [In]	brz_clock	—	—
		Signal [In]	brz_data	—	—
BRZ_Logic_Hi	Constant	Signal [Out]	—	BRZ_SelectIf	—
		amplitude	logic_H_value	—	—
BRZ_Logic_Lo	Constant	Signal [Out]	—	BRZ_SelectIf	—
		amplitude	logic_L_value	—	—

#### F.4.3.5 Rules

There are no special rules for this TSF.

#### F.4.4 BiPLLevel

##### F.4.4.1 Definition

A pulse class in which each bit period is subdivided into two subperiods. The binary data is carried by the transition of the signal level during each bit period, i.e., the amplitude is at one level during the first subperiod and at another level during the second subperiod.

A logic one or High is carried by two subperiods in which the first is at the high amplitude level and the second subperiod is at the low (zero) amplitude. A logic zero or Low is carried by two subperiods in which the first is at the low (zero) amplitude level and the second subperiod is at the high amplitude.

See Figure F.1 for a typical waveform and Figure F.4 for a graphical model of the TSF.

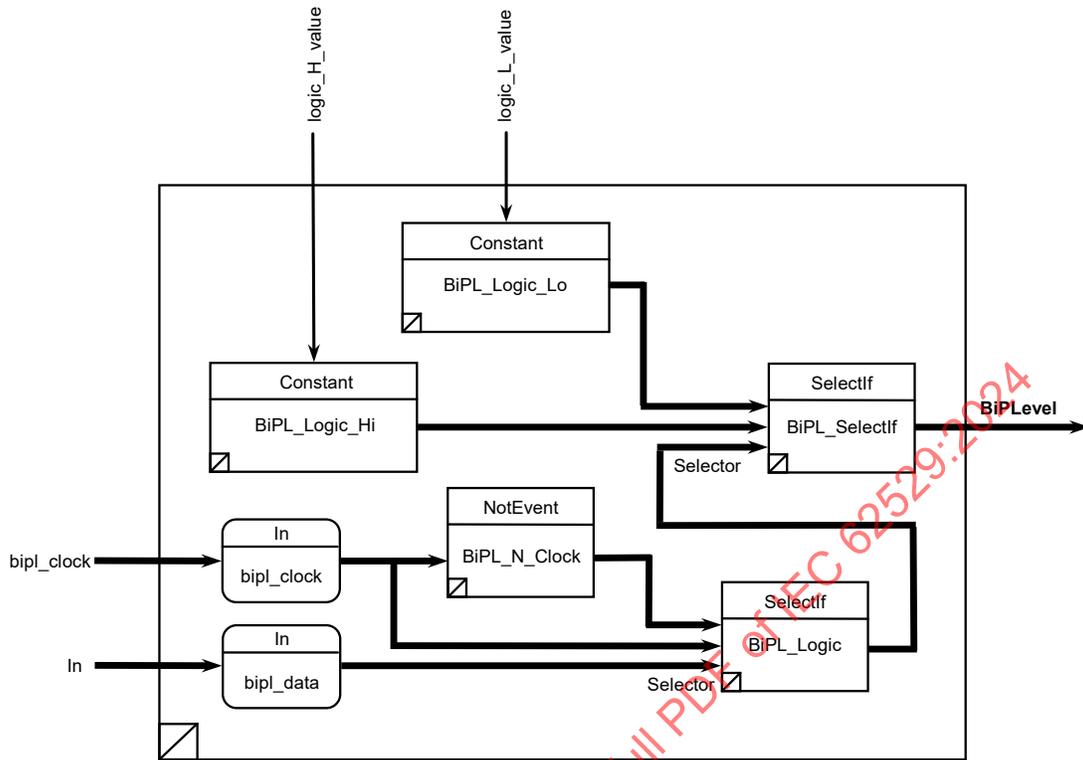


Figure F.4—TSF BiPLLevel

#### F.4.4.2 Interface properties

See Table F.5 for details of the TSF BiPLlevel interface.

Table F.5—TSF BiPLlevel interface

Description	Name	Type	Default	Range
Logic High Value	<code>logic_H_value</code>	Physical	—	—
Logic Low Value	<code>logic_L_value</code>	Physical	—	—
Input Clock	<code>bipl_clock</code>	SignalFunction	—	—
Digital Stream Input	<code>bipl_data</code> (In)	SignalFunction	—	—

#### F.4.4.3 Notes

The Input Clock needs to have the same period as the data rate from the digital stream.

#### F.4.4.4 Model description

See Table F.6 for details of the TSF BiPLlevel model.

**Table F.6—TSF BiPLLevel model**

Name	Type	Terminal	Inputs	Output	Formula
BiPL_SelectIf	SelectIf	Signal [Out]	—	BiPLLevel	—
		Selector	BiPL_Logic	—	—
		Signal [In]	BRZ_Logic_Hi	—	—
		Signal [In]	BRZ_Logic_Lo	—	—
BiPL_Logic	SelectIf	Signal [Out]	—	BiPL_SelectIf	—
		Selector	bipl_data	—	—
		Signal [In]	BiPL_N_Clock	—	—
		Signal [In]	bipl_clock	—	—
BiPL_Logic_Hi	Constant	Signal [Out]	—	BiPL_SelectIf	—
		amplitude	logic_H_value	—	—
BiPL_Logic_Lo	Constant	Signal [Out]	—	BiPL_SelectIf	—
		amplitude	logic_L_value	—	—
BiPL_N_Clock	Not_Event	Signal_Out	—	BiPL_Logic	—
		Signal[In]	bipl_clock	—	—

**F.4.4.5 Rules**

There are no special rules for this TSF.

**F.4.5 BiPSpace**

**F.4.5.1 Definition**

A pulse class in which each bit period is subdivided into two subperiods. A transition occurs at the start of each full bit period. The logic one or High is represented by a second transition at the start of the second subperiod. The logic zero or Low has no second transition, and the level remains unchanged for the whole period.

See Figure F.1 for a typical waveform and Figure F.5 for a graphical model of the TSF.

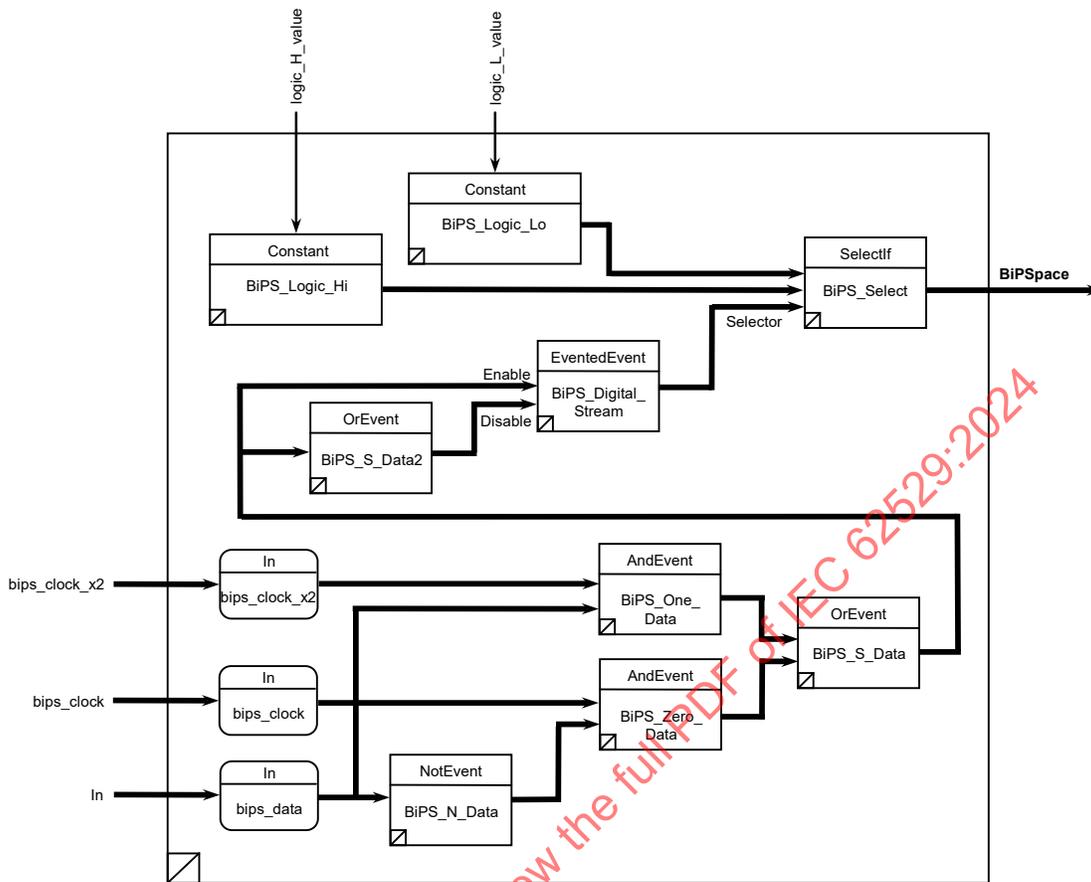


Figure F.5—TSF BiPSpace

F.4.5.2 Interface properties

See Table F.7 for details of the TSF BiPSpace interface.

Table F.7—TSF BiPSpace interface

Description	Name	Type	Default	Range
Logic High Value	logic_H_value	Physical	—	—
Logic Low Value	logic_L_value	Physical	—	—
Input Clock	bips_clock	SignalFunction	—	—
Input Double Clock	bips_clock_x2	SignalFunction	—	—
Digital Stream Input	bips_data (In)	SignalFunction	—	—

F.4.5.3 Notes

The Input Clock needs to have the same period as the data rate from the digital stream. The Input Double Clock needs to have a period of exactly half the period of the Input Clock.

#### F.4.5.4 Model description

See Table F.8 for details of the TSF BiPSpace model.

**Table F.8—TSF BiPSpace model**

Name	Type	Terminal	Inputs	Output	Formula
BiPS_Select	SelectIf	Signal [Out]	—	BiPSpace	—
		Selector	BiPS_Digital_Stream	—	—
		Signal [In]	BiPS_Logic_Lo	—	—
		Signal [In]	BiPS_Logic_Hi	—	—
BiPS_Digital_Stream	EventedEvent	Signal [Out]	—	BiPS_Select	—
		Signal [In]	BiPS_S_Data	—	—
		Signal [In]	BiPS_S_Data2	—	—
BiPS_Logic_Hi	Constant	Signal [Out]	—	BiPS_Select	—
		amplitude	logic_H_value	—	—
BiPS_Logic_Lo	Constant	Signal [Out]	—	BiPS_Select	—
		amplitude	logic_L_value	—	—
BiPS_S_Data2	OrEvent	Signal [Out]	—	BiPS_Digital_Stream	—
		Signal [In]	BiPS_S_Data	—	—
BiPS_S_Data	OrEvent	Signal [Out]	—	BiPS_Digital_Stream, BiPS_S_Data2	—
		Signal [In]	BiPS_One_Data	—	—
		Signal [In]	BiPS_Zero_Data	—	—
BiPS_Zero_Data	AndEvent	Signal [Out]	—	BiPS_S_Data	—
		Signal [In]	BiPS_N_Data	—	—
		Signal [In]	bips_clock	—	—
BiPS_One_Data	AndEvent	Signal [Out]	—	BiPS_S_Data	—
		Signal [In]	bips_data	—	—
		Signal [In]	bips_clock_x2	—	—
BiPS_N_Data	NotEvent	Signal [Out]	—	BiPS_Zero_Data	—
		Signal [In]	bips_data	—	—

#### F.4.5.5 Rules

There are no special rules for this TSF.

#### F.4.6 BiPMark

##### F.4.6.1 Definition

A pulse class in which each bit period is subdivided into two subperiods. A transition occurs at the start of each full bit period. The logic zero or Low is represented by a second transition at the start of the second subperiod. The logic one or High has no second transition, and the level remains unchanged for the whole period.

See Figure F.1 for a typical waveform and Figure F.7 for a graphical model of the TSF.

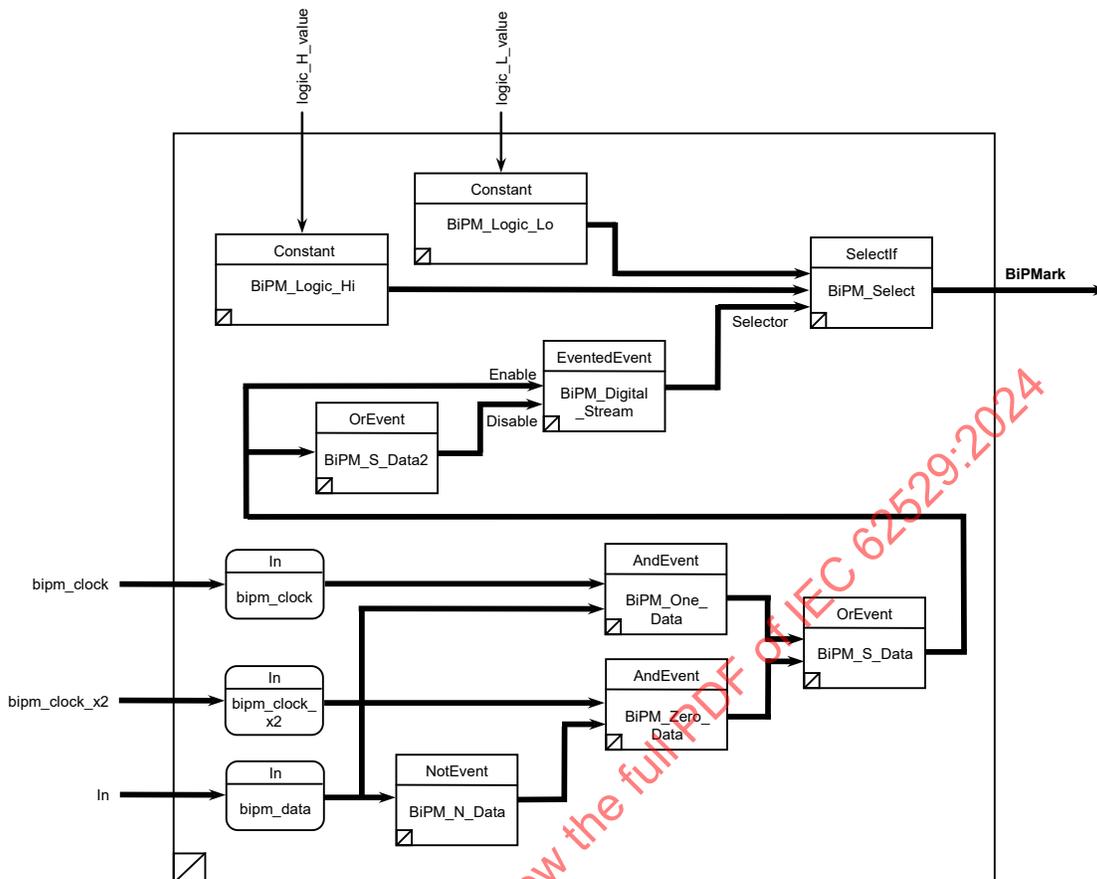


Figure F.6—TSF BiPMark

F.4.6.2 Interface properties

See Table F.9 for details of the TSF BiPMark interface.

Table F.9—TSF BiPMark interface

Description	Name	Type	Default	Range
Logic High Value	logic_H_value	Physical	—	—
Logic Low Value	logic_L_value	Physical	—	—
Input Clock	bipm_clock	SignalFunction	—	—
Input Double Clock	bipm_clock_x2	SignalFunction	—	—
Digital Stream Input	bipm_data (In)	SignalFunction	—	—

F.4.6.3 Notes

The Input Clock needs to have the same period as the data rate from the digital stream. The Input Double Clock needs to have a period of exactly half the period of the Input Clock.

#### F.4.6.4 Model description

See Table F.10 for details of the TSF BiPMark model.

**Table F.10—TSF BiPMark model**

Name	Type	Terminal	Inputs	Output	Formula
BiPM_Select	SelectIf	Signal [Out]	—	BiPMark	—
		Selector	BiPM_Digital_Stream	—	—
		Signal [In]	BiPM_Logic_Lo	—	—
		Signal [In]	BiPM_Logic_Hi	—	—
BiPM_Digital_Stream	EventEvent	Signal [Out]	—	BiPM_Select	—
		Signal [In]	BiPM_S_Data	—	—
		Signal [In]	BiPM_S_Data2	—	—
BiPM_Logic_Hi	Constant	Signal [Out]	—	BiPM_Select	—
		amplitude	logic_H_value	—	—
BiPM_Logic_Lo	Constant	Signal [Out]	—	BiPM_Select	—
		amplitude	logic_L_value	—	—
BiPM_S_Data2	OrEvent	Signal [Out]	—	BiPM_Digital_Stream	—
		Signal [In]	BiPM_S_Data	—	—
BiPM_S_Data	OrEvent	Signal [Out]	—	BiPM_Digital_Stream, BiPM_S_Data2	—
		Signal [In]	BiPM_One_Data	—	—
		Signal [In]	BiPM_Zero_Data	—	—
BiPM_Zero_Data	AndEvent	Signal [Out]	—	BiPM_S_Data	—
		Signal [In]	BiPM_N_Data	—	—
		Signal [In]	bipm_clock_x2	—	—
BiPM_One_Data	AndEvent	Signal [Out]	—	BiPM_S_Data	—
		Signal [In]	bipm_clock	—	—
		Signal [In]	bipm_data	—	—
BiPS_N_Data	NotEvent	Signal [Out]	—	BiPS_Zero_Data	—
		Signal [In]	bipm_data	—	—

#### F.4.6.5 Rules

There are no special rules for this TSF.

#### F.4.7 RZBipolar

##### F.4.7.1 Definition

A pulse class in which each bit period is subdivided into two subperiods. Three signal levels are used, and each bit is represented by the level during the first subperiod. The logic one or High is represented by the first subperiod at a specified amplitude followed by the second subperiod at zero amplitude. The logic zero or Low is represented by the first subperiod at second (normally negative) amplitude followed by the second subperiod at zero amplitude.

See Figure F.1 for a typical waveform and Figure F.7 for a graphical model of the TSF.

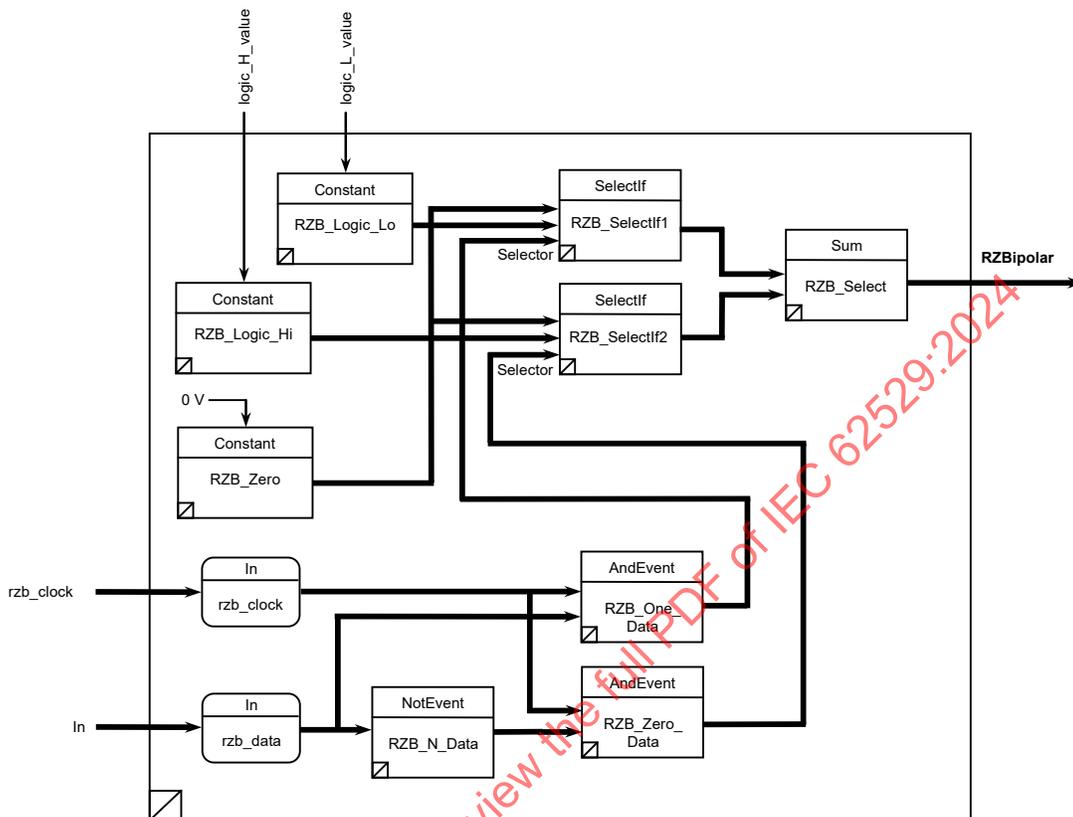


Figure F.7—TSF RZBipolar

#### F.4.7.2 Interface properties

See Table F.11 for details of the TSF RZBipolar interface.

Table F.11—TSF RZBipolar interface

Description	Name	Type	Default	Range
Logic High Value	<code>logic_H_value</code>	Physical	—	—
Logic Low Value	<code>logic_L_value</code>	Physical	—	—
Input Clock	<code>rzb_clock</code>	SignalFunction	—	—
Digital Stream Input	<code>rzb_data</code> (In)	SignalFunction	—	—

#### F.4.7.3 Notes

The Input Clock needs to have the same period as the data rate from the digital stream.

#### F.4.7.4 Model description

See Table F.12 for details of the TSF RZBipolar model.

**Table F.12—TSF RZBipolar model**

Name	Type	Terminal	Inputs	Output	Formula
RZB_Select	Sum	Signal [Out]	—	RZBipolar	—
		Signal [In]	RZB_SelectIf_1	—	—
		Signal [In]	RZB_SelectIf_2	—	—
RZB_SelectIf_2	SelectIf	Signal [Out]	—	RZB_Select	—
		Selector	RZB_Zero_Data	—	—
		Signal [In]	RZB_Zero	—	—
		Signal [In]	RZB_Logic_Lo	—	—
RZB_SelectIf_1	SelectIf	Signal [Out]	—	RZB_Select	—
		Selector	RZB_One_Data	—	—
		Signal [In]	RZB_Zero	—	—
		Signal [In]	RZB_Logic_Hi	—	—
RZB_Zero_Data	AndEvent	Signal [Out]	—	RZB_SelectIf_2	—
		Signal [In]	RZB_N_Data	—	—
			rz_b_clock	—	—
RZB_One_Data	AndEvent	Signal [Out]	—	RZB_SelectIf_1	—
		Signal [In]	rz_b_clock	—	—
		Signal [In]	rz_b_data	—	—
RZB_Logic_Hi	Constant	Signal [Out]	—	RZB_SelectIf_1	—
		amplitude	logic_H_value	—	—
RZB_Logic_Lo	Constant	Signal [Out]	—	RZB_SelectIf_2	—
		amplitude	logic_L_value	—	—
RZB_Zero	Constant	Signal [Out]	—	RZB_SelectIf_1, RZB_SelectIf_2	—
		amplitude	0	—	—
RZB_N_Data	NotEvent	Signal [Out]	—	BiPS_Zero_Data	—
		Signal [In]	rz_b_data	—	—

#### F.4.7.5 Rules

There are no special rules for this TSF.

#### F.4.8 RZPulse

##### F.4.8.1 Definition

This pulse class is very similar to BasicRZ as described in F.4.3, but with a duty cycle of 25%.

See Figure F.1 for a typical waveform and Figure F.8 for a graphical model of the TSF.

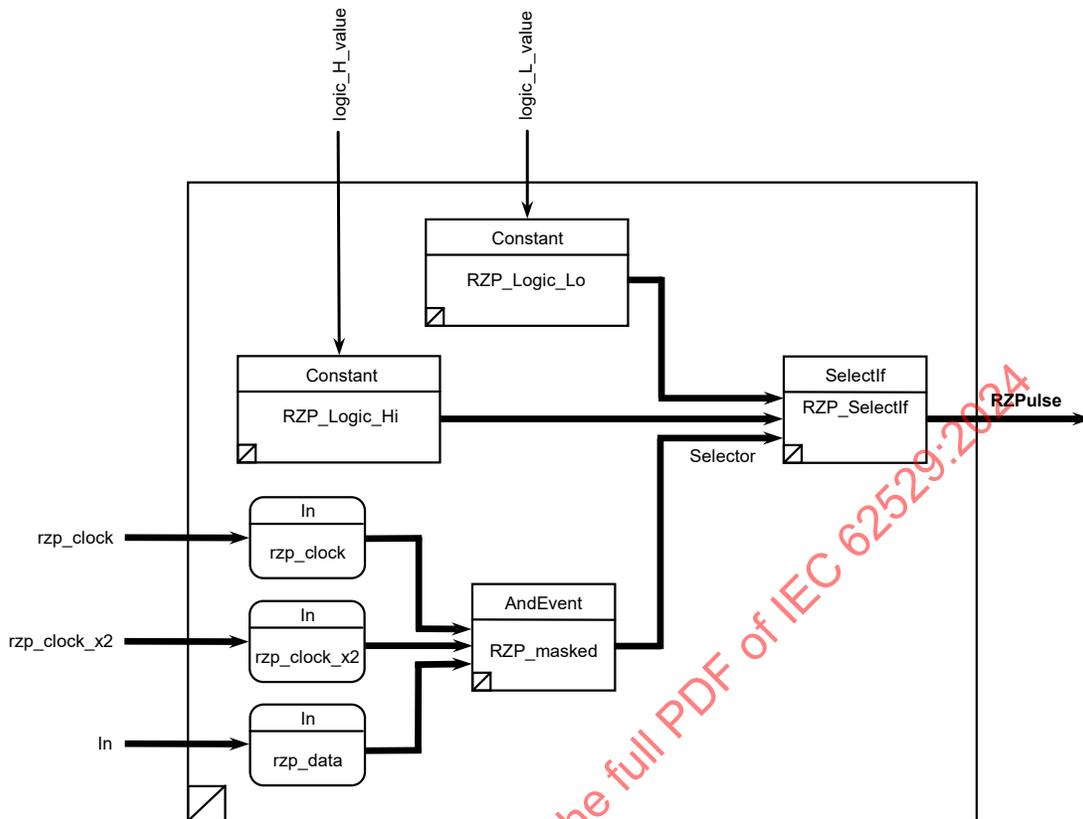


Figure F.8—TSF RZPulse

**F.4.8.2 Interface properties**

See Table F.13 for details of the TSF RZPulse interface.

Table F.13—TSF RZPulse interface

Description	Name	Type	Default	Range
Logic High Value	<code>logic_H_value</code>	Physical	—	—
Logic Low Value	<code>logic_L_value</code>	Physical	—	—
Input Clock	<code>rzp_clock</code>	SignalFunction	—	—
Input Double Clock	<code>rzp_clock_x2</code>	SignalFunction	—	—
Digital Stream Input	<code>rzp_data</code> (In)	SignalFunction	—	—

**F.4.8.3 Notes**

The Input Clock needs to have the same period as the data rate from the digital stream. The Input Double Clock needs to have a period of exactly half the period of the Input Clock.

#### F.4.8.4 Model description

See Table F.14 for details of the TSF RZPulse model.

**Table F.14—TSF RZPulse model**

Name	Type	Terminal	Inputs	Output	Formula
RZP_SelectIf	SelectIf	Signal [Out]	—	RZPulse	—
		Selector	RZP_Masked	—	—
		Signal [In]	RZP_Logic_Lo	—	—
		Signal [In]	RZP_Logic_Hi	—	—
RZP_Masked	AndEvent	Signal [Out]	—	RZP_SelectIf	—
		Signal [In]	rzp_clock	—	—
		Signal [In]	rzp_clock_x2	—	—
		Signal [In]	rzp_data	—	—
RZP_Logic_Hi	Constant	Signal [Out]	—	RZP_SelectIf	—
		amplitude	logic_H_value	—	—
RZP_Logic_Lo	Constant	Signal [Out]	—	RZP_SelectIf	—
		amplitude	logic_L_value	—	—

#### F.4.8.5 Rules

There are no special rules for this TSF.

#### F.4.9 AMI

##### F.4.9.1 Definition

The alternate mark inversion (AMI) pulse class requires three amplitude levels be defined. In this TSF, only two of the amplitudes are specified as the inverted logic one or High level is assumed to be of the same amplitude as the noninverted logic one or High but with the opposite sign. The logic one amplitudes are offset from the amplitude specified for the logic zero or Low.

See Figure F.1 for a typical waveform and Figure F.9 for a graphical model of the TSF.

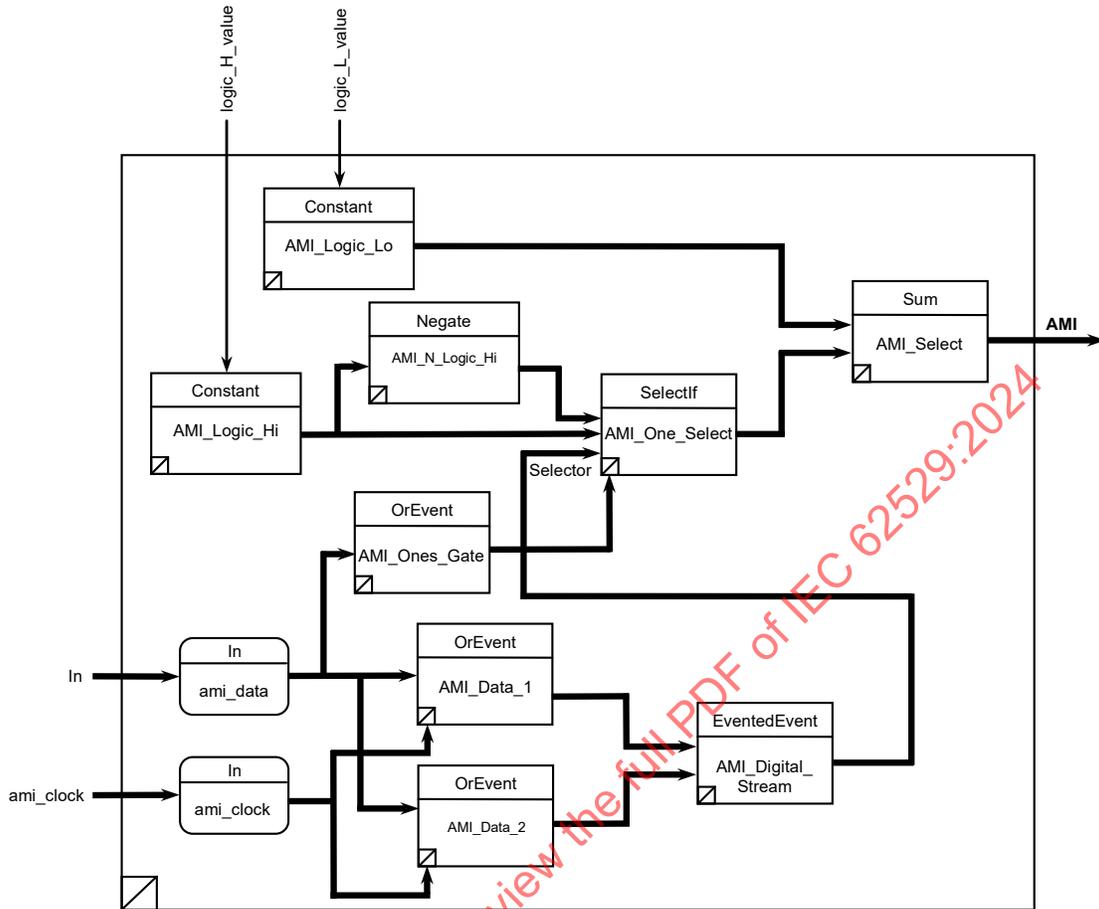


Figure F.9—TSF AMI

#### F.4.9.2 Interface properties

See Table F.15 for details of the TSF AMI interface.

Table F.15—TSF AMI interface

Description	Name	Type	Default	Range
Logic High Value	logic_H_value	Physical	—	—
Logic Low Value	logic_L_value	Physical	—	—
Input Clock	ami_clock	SignalFunction	—	—
Digital Stream Input	ami_data (In)	SignalFunction	—	—

#### F.4.9.3 Notes

There are no special notes for this TSF.

#### F.4.9.4 Model description

See Table F.16 for details of the TSF AMI model.

**Table F.16—TSF AMI model**

Name	Type	Terminal	Inputs	Output	Formula
AMI_Select	Sum	Signal [Out]	—	AMI	—
		Signal [In]	AMI_One_Select	—	—
		Signal [In]	AMI_Logic_Lo	—	—
AMI_Logic_Lo	Constant	Signal [Out]	—	AMI_Select	—
		amplitude	logic_L_value	—	—
AMI_One_Select	SelectIf	Signal [Out]	—	AMI_Select	—
		Selector	AMI_Digital_Stream	—	—
		Signal [In]	AMI_N_Logic_Hi	—	—
		Signal [In]	AMI_Logic_Hi	—	—
		Gate [In]	AMI_Data_1	—	—
AMI_Digital_Stream	EventedEvent	Signal [Out]	—	AMI_One_Select	—
		Signal [In]	AMI_Data_1	—	—
		Signal [In]	AMI_Data_2	—	—
AMI_N_Logic_Hi	Negate	Signal [Out]	—	AMI_One_Select	—
		Signal [In]	AMI_Logic_Hi	—	—
AMI_Logic_Hi	Constant	Signal [Out]	—	AMI_N_Logic_Hi	—
		amplitude	logic_H_value	—	—
AMI_Data_2	OrEvent	Signal [Out]	—	AMI_Digital_Stream	—
		Signal [In]	ami_data	—	—
AMI_Data_1	OrEvent	Signal [Out]	—	AMI_Digital_Stream, AMI_One_Select	—
		Signal [In]	ami_data	—	—

#### F.4.9.5 Rules

There are no special rules for this TSF.

### F.5 DTIF

#### F.5.1 Definition

The DTIF TSF represents a simple TSF and contains only the “location” of the DTIF files and a definition of how they shall be used. This TSF allows for the generation of a complete digital signal stream containing all the digital patterns and necessary timing information.

This TSF provides support for digital data files defined in IEEE Std 1445–1998 [B15].

See Figure F.10 for a graphical model of the TSF.

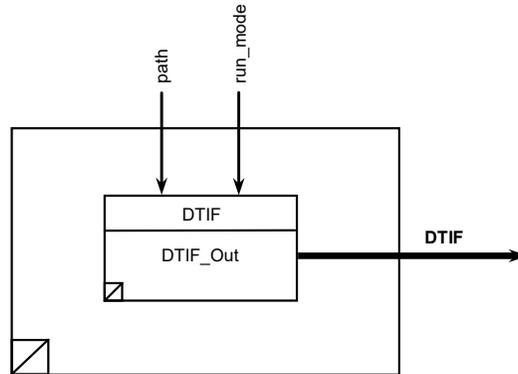


Figure F.10—TSF DTIF

### F.5.2 Interface properties

See Table F.17 for details of the TSF DTIF interface.

Table F.17—TSF DTIF interface

Description	Name	Type	Default	Range
Path to data file	path	string	C:\TPS\UUT\DTIF	
Run mode	run_mode	enumeration	Go_Nogo	Go_Nogo   Fault_Dictionary   Guided_Probe   GP_FD

### F.5.3 Notes

The output from the DTIF TSF represents the complete digital signal for all channels. The method of using a DTIF TSF would be as follows:

- Add a Pulse Class to convert digital stream into the logic logic levels.
- Add a Digital Pins connector to identify the unit under test (UUT) pins used for the signals.

### F.5.4 Model description

See Table F.18 for details of the TSF DTIF model.

Table F.18—TSF DTIF model

Name	Type	Terminal	Inputs	Output	Formula
DTIF_Out	DTIF	Signal [Out]	—	DTIF	—
		path	path	—	—
		run_mode	run_mode	—	—

### F.5.5 Rules

There are no special rules for this TSF.

## **Annex G**

(normative)

### **Carrier language requirements**

#### **G.1 Carrier language requirements**

This annex describes the requirements for a suitable carrier language for mapping data and objects on to the signal and test definition (STD) methodology. The requirements address data definition, objects and structures.

##### **G.1.1 General requirements**

The carrier language shall run on a host system (in a compiled form if necessary). It may be supported by an operating system (according to the requirements of the carrier language and host system). Test statements written in the carrier language shall use the signal TPL statements and/or may use the runtime signal interfaces to control signal behavior.

##### **G.2 Interface definition language (IDL)**

The carrier language shall support the IDL or associated TypeLibrary as defined in the Distributed Computing Environment (DCE) Specifications [B5].

##### **G.3 Datatypes**

The datatypes defined in G.3 shall support the datatypes defined in the IDL. Table G.1 shows the relationship between the datatypes required, their IDL names, and the carrier language datatype that supports them.

**Table G.1—Datatypes used in STD**

Name used in STD	Reference	IDL mapping	Supported by XML (See NOTE)
any	Datatypes	VARIANT	#any
boolean	Boolean datatype	VARIANT_BOOL	std:boolean
digitalString	Digital datatype	BSTR	std:digitalString
double	Real datatype IEEE Std 754™-2008 [B14] (binary64 format)	double	std:double
enumCondition	Enumeration datatype	enumCondition	std:enumCondition
enumMeasuredVariable	Enumeration datatype	enumMeasuredVariable	std:enumMeasuredVariable
enumPulseClass	Enumeration datatype	enumPulseClass	std:enumPulseClass
int	Integer datatype $-2^{31}$ to $-(2^{31}-1)$	long	std:int
list_Physical	See Annex B	SAFEARRAY(Physical)	std:list_Physical
list_any	Datatypes & Array datatype	SAFEARRAY(VARIANT)	std:list_any
list_boolean	Boolean datatype & Array datatype	SAFEARRAY(VARIANT_BOOL)	std:list_boolean
list_double	Real datatype & Array datatype	SAFEARRAY(double)	std:list_double
list_int	Integer datatype & Array datatype	SAFEARRAY(long)	std:list_int
list_string	ASCII datatype & Array datatype	SAFEARRAY(BSTR)	std:list_string
Physical <sup>a</sup>	see Annex B	Physical	std:Physical
pinString	Connector pin datatype	BSTR	std:pinString
PulseDefn	See Annex B	PulseDefn	std:PulseDefn
PulseDefns	See Annex B	PulseDefns	std:PulseDefns
string	ASCII datatype	BSTR	std:string
<enumerationList>	User defined list of enumeration values	long	xs:string
NOTE—The namespace prefix “std” used in the “Supported by XML” column represents the namespace of the STD basic signal component (BSC) extensible markup language (XML) Schema.			

<sup>a</sup>The numeric value of any physical type shall be supported by the real datatype (double).

### G.3.1 Enumeration datatype

The carrier language shall allow the user to define and label an enumeration datatype, which includes a list of specified enumeration values. Each enumeration value may comprise a string of one or more characters.

The enumeration value is then referenced (in the IDL interface) as an integer value corresponding to its position in the enumeration list.

### G.3.2 Integer datatype

To establish, label, and identify decimal integer numbers. A decimal integer number is written as a string of characters beginning with an optional plus or minus sign, followed by one or more digits (0 through 9).

The coded representation of the int type data shall be in the range of at least  $-2\,147\,483\,648$  to  $+2\,147\,483\,647$  (i.e., supported by a minimum of 32 bits).

### G.3.3 Real datatype

#### G.3.3.1 General

To establish, label, and identify decimal numbers. Numbers may be written in a fixed point or floating point format. IEEE Std 754-2008 [B14] defines the recommended formats for storing numbers used as real datatypes.

For single-precision real numbers, the minimum precision of the coded representation of real data shall be at least 6 significant digits over a magnitude range of  $\pm (1-2^{-24}) \times 2^{128}$ , which is approximately equal to  $\pm 3.4028235 \times 10^{38}$  (IEEE 754 format binary32).

For double-precision real numbers, the precision of the coded representation of real data shall be at least 15 significant digits over a magnitude range of  $\pm ((1-(1/2)^{53}) \times 2^{1024})$ , which is approximately equal to  $\pm 1.7976931348623157 \times 10^{308}$  (IEEE 754 format binary64).

#### G.3.3.2 Fixed-point number

A decimal fixed-point number is written as a string of characters beginning with an optional plus or minus sign, followed by one or more digits (0 through 9), optionally followed by a decimal point and one or more digits (0 through 9).

#### G.3.3.3 Decimal floating-point number

A decimal floating point number is written in the form  $\pm n.mE\pm p$ , where n, m, and p are numerical strings consisting of one or more digits (0 through 9).

### G.3.4 Character datatype

To establish, label, and identify one ASCII 8-bit character or a string of ASCII 8-bit characters (ASCII).

To establish, label, and identify one ASCII 16-bit character or a string of ASCII 16-bit characters (UNICODE).

### G.3.5 Boolean datatype

To establish, label, and identify the data values TRUE or FALSE.

### G.3.6 Digital datatype

To establish, label, and identify digital data.

Characters are used to represent the digital signals as follows:

- H logic high (or logic 1)
- h logic high (or logic 1)
- 1 logic high (or logic 1)
- L logic low (or logic 0)
- l logic low (or logic 0)
- 0 logic low (or logic 0)
- Z high impedance (absence of logic signal)
- X unknown or indeterminate logic level
- , delimiter between blocks
- ; delimiter between blocks

Full details of use with BSCs are provided in Annex B.

The lowercase h (high) and l (low) characters are available for use with Response data. Care shall be exercised so that the lowercase L (l) is not confused with the numeric character 1. When Stimulus and Response data are included in the same context, it is recommended that the numeric characters (0 and 1) are not used.

### G.3.7 Connector pin string datatype

To establish, label, and identify physical connector pins. These pins are normally the unit under test (UUT) pins.

A pin name shall be a contiguous string of characters, which may include alphanumeric, hyphen, and underscore characters. Pin names may not include a comma, semicolon, or whitespace character (namely, space, new-line, carriage-return, line-feed, and tab).

Multiple pin names are delimited by one or more whitespace, comma, or semicolon characters.

Full details are provided in Annex B.

### G.3.8 File datatype

To establish, label, and identify a collection of data items. Each data item has a position in the file.

The distance between two subsequent data items is one space. A file may not be nested within another file.

### G.3.9 Array datatype

To establish, label, and identify either a one-dimensional or a multidimensional ordered collection of data elements of the same type. An array can have any number of dimensions that are identified by indices that are bounded by upper and lower limits. All the elements in an array shall be addressed by their indices.

### G.3.10 Record datatype

To establish, label, and identify a collection of data elements that need not be of the same type or structure. Individual fields in a record shall be addressable by a name. Any type except “file” may be specified.

### G.3.11 Objects, Variables and constants

To establish, label, and identify objects, variables and constants. A unique datatype shall be assigned to an established objects, variable or constant. A facility for initializing objects and variables shall be provided.

Any created object, variable or constant shall be accessible at the structural level where it is established and at any inner nested structural level.

### G.3.12 XML datatypes supported by this standard

Table G.2 provides a list of W3C XML datatypes supported by this standard. These are listed together with the IDL name and the variant type name.

**Table G.2—XML datatypes supported by STD**

XSD (Soap) Type	IDL	Variant type
xs:anyURI	BSTR	VT_BSTR
xs:base64Binary	SAFEARRAY (unsigned char)	VT_ARRAY  VT_UI1
xs:boolean	VARIANT_BOOL	VT_BOOL
xs:byte	byte	VT_I1
xs:date	DATE	VT_DATE
xs:dateTime	DATE	VT_DATE
xs:decimal	DECIMAL	VT_DECIMAL
xs:double	double	VT_R8
xs:duration	BSTR	VT_BSTR
xs:ENTITIES	BSTR	VT_BSTR
xs:ENTITY	BSTR	VT_BSTR
xs:float	float	VT_R4
xs:gDay	BSTR	VT_BSTR
xs:gMonthDay	BSTR	VT_BSTR
xs:gYear	BSTR	VT_BSTR
xs:gYearMonth	BSTR	VT_BSTR
xs:hexBinary	BSTR	VT_BSTR
xs:ID	BSTR	VT_BSTR
xs:IDREF	BSTR	VT_BSTR
xs:IDREFS	BSTR	VT_BSTR
xs:int	long	VT_I4

**Table G.2—XML datatypes supported by STD**

<b>XSD (Soap) Type</b>	<b>IDL</b>	<b>Variant type</b>
xs:integer	DECIMAL	VT_DECIMAL
xs:language	BSTR	VT_BSTR
xs:long	DECIMAL	VT_DECIMAL
xs:gMonth	BSTR	VT_BSTR
xs:Name	BSTR	VT_BSTR
xs:NCName	BSTR	VT_BSTR
xs:negativeInteger	DECIMAL	VT_DECIMAL
xs:NMTOKEN	BSTR	VT_BSTR
xs:NMTOKENS	BSTR	VT_BSTR
xs:nonNegativeInteger	DECIMAL	VT_DECIMAL
xs:nonPositiveInteger	DECIMAL	VT_DECIMAL
xs:normalizedString	BSTR	VT_BSTR
xs:NOTATION	BSTR	VT_BSTR
xs:positiveInteger	DECIMAL	VT_DECIMAL
xs:QName	BSTR	VT_BSTR
xs:short	short	VT_I2
xs:string	BSTR	VT_BSTR
xs:time	DATE	VT_DATE
xs:token	BSTR	VT_BSTR
xs:unsignedByte	unsigned byte	VT_UI1
xs:unsignedInt	unsigned long	VT_UI4
xs:unsignedLong	DECIMAL	VT_DECIMAL
xs:unsignedShort	unsigned short	VT_UI2

## Annex H

(normative)

### Test procedure language (TPL)

#### H.1 TPL layer

The TPL layer provides a mechanism for users who want to specify test requirements in a textual format and provides a subset of the more traditional behavior expected from signals for test purposes.

#### H.2 Elements of the TPL

The TPL comprises three elements:

- a) Signal statements that are used to configure, manipulate, control, and measure signals
- b) A carrier language that is a programming language in which the signals statements can be written, sequenced, observed, and generally supported
- c) Global variable flags, e.g., GO/NOGO

#### H.3 Structure of test requirements

A test requirement written using the TPL can include the following elements:

- a) Pragmas, such as native language includes, defines, and signal and test definition (STD) imports
- b) User declarations of variables and functions
- c) Program flow statements
- d) User-defined function calls
- e) Input-output statements
- f) TPL signal statements

#### H.4 Carrier language

The carrier language may be any programming language. It provides data definition, processing, and control structures in which the signal statements of the TPL may be written, embedded and compiled, or translated.

To facilitate portability of test requirements between different carrier languages without extensive manual recoding, a TPL test requirement shall not include any carrier language constructs beyond the constructs identified in the carrier language requirements detailed in Annex G.

#### H.5 Signal statements

The signal statements can be used to specify the signal test operations to be conducted on a unit under test (UUT).