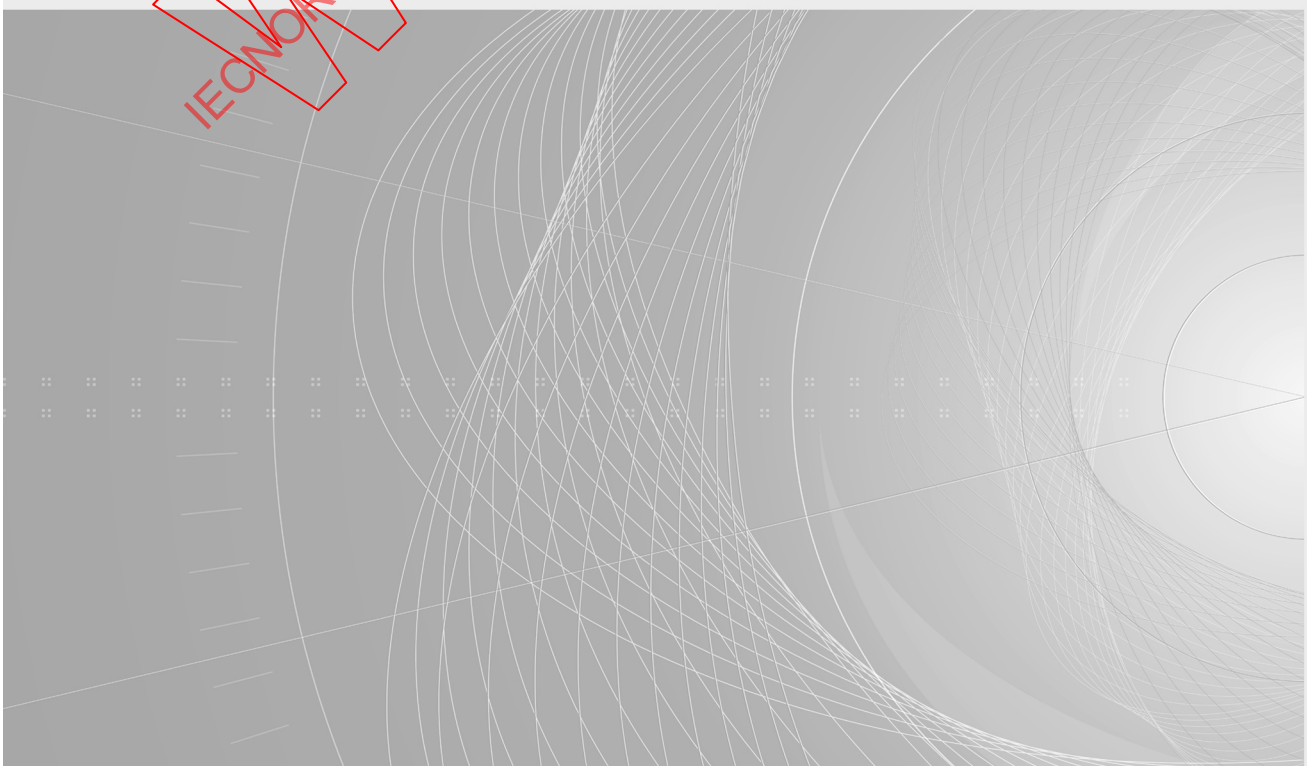
IEC 62529

Edition 1.0 2007-11

INTERNATIONAL STANDARD

IEEE 1641™

Standard for Signal and Test Definition



IEC 62529:2007(E) IEEE Std. 1641-2004

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

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

IEC 62529

Edition 1.0 2007-11

INTERNATIONAL STANDARD

IEEE 1641™

Standard for signal and test definition

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

PRICE CODE XH

ICS 25.040

ISBN 2-8318-9482-4

CONTENTS

FOREWORD....10
IEEE introduction....13

1. Overview....14

1.1 Scope....14
1.2 Purpose....15

2. Definitions, abbreviations, and acronyms....15

2.1 Definitions....15
2.2 Abbreviations and acronyms....17

3. Structure of this standard....18

3.1 Layers....18

4. Signal modeling language (SML) layer....20

5. Basic signal component (BSC) layer....20

5.1 BSC layer base classes....20
5.2 BSCs....21
5.3 SignalFunction template....22

6. Test signal framework (TSF) layer....22

6.1 TSF classes....22
6.2 TSF signals....23

7. Test procedure language (TPL) layer....25

7.1 Goals of the TPL....25
7.2 Elements of the TPL....25
7.3 Use of the TPL....25

Annex A (normative) Signal modeling language (SML)....26

A.1 Use of the SML....26
A.2 Introduction....26
A.3 Physical types....27
A.4 Signal definitions....30
A.5 Pure signals....31
A.5.1 Nonperiodic signals....31
A.5.2 Periodic signals....32
A.6 Pure signal-combining mechanisms....33
A.6.1 Piecewise continuous signals (PCSs)....33
A.6.2 Sum....36
A.6.3 Product....36
A.7 Pure function transformations....36
A.7.1 Fourier transform....37
A.8 Measuring, limiting, and sampling signals....37
A.8.1 Confining parameters to a limit....38
A.8.2 Sampling signals....38
A.9 Digital signals....38

A.9.1 Defining Digital ... 39
A.9.2 Defining DigitalSignal ... 39
A.9.3 Conversion routines ... 40
A.9.4 Patterns ... 41
A.10 Basic component SML ... 42
A.10.1 Source ::SignalFunction ... 42
A.10.2 Conditioner ::SignalFunction ... 44
A.10.3 EventFunction ::SignalFunction ... 47
A.10.4 Sensor ::SignalFunction ... 50
A.10.5 Digital ::SignalFunction ... 50
A.10.6 Connection ::SignalFunction ... 51

Annex B (normative) Basic signal component (BSC) layer ... 53

B.1 BSC layer base classes ... 53
B.2 BSC subclasses ... 53
B.3 Description of a BSC ... 58
B.3.1 Diagrammatic representation of a BSC ... 58
B.3.2 BSC interfaces ... 59
B.3.3 Types of BSCs ... 60
B.3.4 BSC attribute default values ... 61
B.3.5 BSC subclass descriptions ... 61
B.4 Physical class ... 63
B.4.1 Permissible physical types and their units ... 65
B.4.2 Unit prefixes ... 69
B.5 PulseDefns class ... 70
B.5.1 PulseDefn class ... 71
B.6 SignalFunction class ... 71
B.6.1 Source ::SignalFunction ... 72
B.6.2 Conditioner ::SignalFunction ... 80
B.6.3 EventFunction ::SignalFunction ... 94
B.6.4 Sensor ::SignalFunction ... 100
B.6.5 Digital ::SignalFunction ... 106
B.6.6 Connection ::SignalFunction ... 108

Annex C (normative) Dynamic signal descriptions ... 112

C.1 Introduction ... 112
C.2 Basic classes ... 113
C.2.1 ResourceManager ... 113
C.2.2 Signal ... 114
C.2.3 BSCs ... 116
C.3 Dynamic signal goals and use cases ... 118

Annex D (normative) IDL basic components ... 119

D.1 Introduction ... 119
D.2 IDL BSC library ... 119

Annex E (informative) Test signal framework (TSF) for ATLAS ... 161

E.1 Introduction ... 161
E.2 AC_SIGNAL<type: Current|| Power|| Voltage> ... 161
E.2.1 Definition ... 161
E.2.2 Interface properties ... 162

E.2.3 Notes 162
E.2.4 Model description 162
E.2.5 Rules 162
E.2.6 Example 163
E.3 AM_SIGNAL 163
E.3.1 Definition 163
E.3.2 Interface properties 164
E.3.3 Notes 164
E.3.4 Model description 164
E.3.5 Rules 165
E.3.6 Example 165
E.4 DC_SIGNAL<type: Voltage|| Current|| Power> 165
E.4.1 Definition 165
E.4.2 Interface properties 166
E.4.3 Notes 166
E.4.4 Model description 166
E.4.5 Rules 167
E.4.6 Example 167
E.5 DIGITAL_PARALLEL 168
E.5.1 Definition 168
E.5.2 Interface properties 168
E.5.3 Notes 168
E.5.4 Model description 169
E.5.5 Rules 169
E.5.6 Example 169
E.6 DIGITAL_SERIAL 170
E.6.1 Definition 170
E.6.2 Interface properties 171
E.6.3 Notes 171
E.6.4 Model description 171
E.6.5 Rules 171
E.6.6 Example 172
E.7 DME_INTERROGATION 172
E.7.1 Definition 172
E.7.2 Interface properties 173
E.7.3 Notes 173
E.7.4 Model description 174
E.7.5 Rules 164
E.7.6 Example 174
E.8 DME_RESPONSE 175
E.8.1 Definition 175
E.8.2 Interface properties 176
E.8.3 Notes 177
E.8.4 Model description 177
E.8.5 Rules 179
E.8.6 Example 179
E.9 FM_SIGNAL<type: Voltage|| Power|| Current> 180
E.9.1 Definition 180
E.9.2 Interface properties 180
E.9.3 Notes 180
E.9.4 Model description 180
E.9.5 Rules 181
E.9.6 Example 181
E.10 ILS_GLIDE_SLOPE<type: Voltage|| Power> 182

E.10.1 Definition 182
E.10.2 Interface properties 183
E.10.3 Notes 184
E.10.4 Model description 184
E.10.5 Rules 185
E.10.6 Example 185
E.11 ILS_LOCALIZER<type: Power|| Voltage> 186
E.11.1 Definition 186
E.11.2 Interface properties 186
E.11.3 Notes 187
E.11.4 Model description 187
E.11.5 Rules 188
E.11.6 Example 188
E.12 ILS_MARKER 189
E.12.1 Definition 189
E.12.2 Interface properties 190
E.12.3 Notes 190
E.12.4 Model description 190
E.12.5 Rules 191
E.12.6 Example 191
E.13 PM_SIGNAL 191
E.13.1 Definition 191
E.13.2 Interface properties 192
E.13.3 Notes 192
E.13.4 Model description 192
E.13.5 Rules 193
E.13.6 Example 193
E.14 PULSED_AC_SIGNAL<type: Current|| Power|| Voltage> 194
E.14.1 Definition 194
E.14.2 Interface properties 194
E.14.3 Notes 194
E.14.4 Model description 195
E.14.5 Rules 195
E.14.6 Example 195
E.15 PULSED_AC_TRAIN<type: Voltage|| Current|| Power> 196
E.15.1 Definition 196
E.15.2 Interface properties 196
E.15.3 Notes 197
E.15.4 Model description 197
E.15.5 Rules 197
E.15.6 Example 198
E.16 PULSED_DC_SIGNAL<type: Voltage|| Current|| Power> 198
E.16.1 Definition 198
E.16.2 Interface properties 199
E.16.3 Notes 199
E.16.4 Model description 199
E.16.5 Rules 200
E.16.6 Example 200
E.17 PULSED_DC_TRAIN<type: Voltage|| Current|| Power> 201
E.17.1 Definition 201
E.17.2 Interface properties 201
E.17.3 Notes 201
E.17.4 Model description 202
E.17.5 Rules 202

E.17.6 Example 202
E.18 RADAR_RX_SIGNAL 203
E.18.1 Definition 203
E.18.2 Interface properties 203
E.18.3 Notes 204
E.18.4 Model description 204
E.18.5 Rules 205
E.18.6 Example 205
E.19 RADAR_TX_SIGNAL<type: Current|| Voltage|| Power> 206
E.19.1 Definition 206
E.19.2 Interface properties 206
E.19.3 Notes 207
E.19.4 Model description 207
E.19.5 Rules 207
E.19.6 Example 207
E.20 RAMP_SIGNAL<type: Voltage|| Current|| Power> 208
E.20.1 Definition 208
E.20.2 Interface properties 209
E.20.3 Notes 209
E.20.4 Model description 209
E.20.5 Rules 209
E.20.6 Example 210
E.21 RANDOM_NOISE 210
E.21.1 Definition 210
E.21.2 Interface properties 210
E.21.3 Notes 211
E.21.4 Model description 211
E.21.5 Rules 211
E.21.6 Example 211
E.22 RESOLVER 212
E.22.1 Definition 212
E.22.2 Interface properties 213
E.22.3 Notes 213
E.22.4 Model description 213
E.22.5 Rules 214
E.22.6 Example 215
E.23 RS_232 215
E.23.1 Definition 215
E.23.2 Interface properties 215
E.23.3 Notes 216
E.23.4 Model description 216
E.23.5 Rules 216
E.24 SQUARE_WAVE<type: Current|| Voltage|| Power> 216
E.24.1 Definition 216
E.24.2 Interface properties 217
E.24.3 Notes 217
E.24.4 Model description 217
E.24.5 Rules 218
E.24.6 Example 218
E.25 SSR_INTERROGATION<type: Voltage|| Current|| Power> 219
E.25.1 Definition 219
E.25.2 Interface properties 219
E.25.3 Notes 220
E.25.4 Model description 221

E.25.5 Rules 221
E.25.6 Example 221
E.26 SSR_RESPONSE<type: Voltage|| Current|| Power> 222
E.26.1 Definition 222
E.26.2 Interface properties 222
E.26.3 Notes 223
E.26.4 Model description 224
E.26.5 Rules 225
E.26.6 Example 225
E.27 STEP_SIGNAL 226
E.27.1 Definition 226
E.27.2 Interface properties 226
E.27.3 Notes 227
E.27.4 Model description 227
E.27.5 Rules 227
E.27.6 Example 227
E.28 SUP_CAR_SIGNAL 228
E.28.1 Definition 228
E.28.2 Interface properties 228
E.28.3 Notes 229
E.28.4 Model description 229
E.28.5 Rules 230
E.28.6 Example 230
E.29 SYNCHRO 230
E.29.1 Definition 230
E.29.2 Interface properties 231
E.29.3 Notes 232
E.29.4 Model description 232
E.29.5 Rules 233
E.29.6 Example 233
E.30 TACAN 234
E.30.1 Definition 234
E.30.2 Interface properties 235
E.30.3 Notes 236
E.30.4 Model description 236
E.30.5 Rules 238
E.30.6 Example 238
E.31 TRIANGULAR_WAVE_SIGNAL<type: Voltage|| Current|| Power> 239
E.31.1 Definition 239
E.31.2 Interface properties 239
E.31.3 Notes 240
E.31.4 Model description 240
E.31.5 Rules 240
E.31.6 Example 240
E.32 VOR 241
E.32.1 Definition 241
E.32.2 Interface properties 242
E.32.3 Notes 242
E.32.4 Model description 243
E.32.5 Rules 244
E.32.6 Example 244

Annex F (informative) IDL for TSF for ATLAS 245

F.1 Introduction 245
F.2 IDL for TSF for ATLAS library 245

Annex G (normative) Carrier language requirements 265

G.1 Carrier language requirements 265
G.1.1 General requirements 265
G.1.2 Human interface and communication 265
G.2 IDL 265
G.3 Data types 265
G.3.1 Enumeration data type 266
G.3.2 Integer data type 266
G.3.3 Real data type 266
G.3.4 Character data type 267
G.3.5 Boolean data type 267
G.3.6 File data type 267
G.3.7 Array data type 267
G.3.8 Record data type 267
G.3.9 Variables and constants 267
G.4 Data-processing requirements 267
G.4.1 Data manipulation 267
G.4.2 Arithmetic operators 268
G.4.3 Relational operators 268
G.4.4 Logical operators 268
G.4.5 Other operators 269
G.4.6 Mathematical functions 269
G.4.7 File-handling functions 269
G.4.8 Type conversion functions 270
G.4.9 String related functions 270
G.4.10 Other functions 271
G.5 Control structures 271
G.5.1 If 271
G.5.2 Else 271
G.5.3 Case 271
G.5.4 For 271
G.5.5 While 271

Annex H (normative) Test procedure language (TPL) 273

H.1 TPL layer 273
H.2 Elements of the TPL 273
H.3 Structure of test requirements 273
H.4 Carrier language 273
H.5 Signal statements 273
H.5.1 Definition of signal statements 273
H.5.2 Structure of signal statements 274
H.5.3 Syntax of signal statements 274
H.6 Mapping of test statements to carrier language 275
H.7 Test statement definitions 275
H.7.1 Setup statements 275
H.7.2 Reset statement 283
H.7.3 Connect statement 284
H.7.4 Disconnect statement 286
H.7.5 Enable statement 287

H.7.6 Disable statement 288
H.7.7 Read statement 289
H.7.8 Change statement 290
H.7.9 Compare statement 290
H.7.10 Wait_For statement 292
H.8 Elements used in test statement definitions 292
H.8.1 <TSFClass> 292
H.8.2 Attribute-Value groups 293
H.9 Attributes with multiple properties 300
H.9.1 Entering literal data 300
H.9.2 Using arrays of data 302
H.9.3 Acquiring sensor data 303
H.10 Transferring data in digital signals 304
H.10.1 Representation of digital data 304
H.10.2 Transmitting digital data using digital sources 305
H.10.3 Acquiring digital sensor data 307
H.10.4 Bidirectional digital signals 307
H.11 Creating test requirements 308
H.11.1 Creating test statements 308
H.11.2 Use of gate in signal statements 309
H.12 Delimiting TPL statements 310
H.12.1 Introducing a group of one or more TPL statements 310
H.12.2 Indicating end of group of TPL statements 310

Annex I (normative) Extensible markup language (XML) signal descriptions (XSDs) 312

I.1 Introduction 312
I.2 XML signal schema definition 312

Annex J (informative) XML for TSF for ATLAS 350

J.1 Introduction 350
J.2 TSF XML schema 350
J.2.1 Library information (<TSFLibrary> tag) 350
J.2.2 TSF information (<TSF> tag) 350
J.2.3 Interface information (<interface> tag) 351
J.2.4 Model information (<model> tag) 351
J.2.5 XML schema 352
J.3 XML for TSF for ATLAS 353

Annex K (informative) Support for ATLAS nouns and modifiers 398

K.1 STD support for ATLAS signals 398
K.2 STD support for ATLAS nouns 398
K.3 STD support for ATLAS noun modifiers 401
K.3.1 Example of noun modifier supported by combination of BSCs 411
K.3.2 Example of noun modifier supported by a technique 411
K.4 Support for ATLAS extensions 411

Annex L (informative) Bibliography 412

Annex M (informative) List of participants 413

INTERNATIONAL ELECTROTECHNICAL COMMISSION

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

International Standard IEC/IEEE 62529 has been processed through Technical Committee 93: Design automation.

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

IEEE Std	FDIS	Report on voting
1641(2004)	93/251/FDIS	93/262/RVD

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

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

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

IEC/IEEE Dual Logo International Standards

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

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

Use of an IEC/IEEE Dual Logo International Standard is wholly voluntary. The IEC and IEEE disclaim liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEC or IEEE Standard document.

The IEC and IEEE do not warrant or represent the accuracy or content of the material contained herein, and expressly disclaim any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEC/IEEE Dual Logo International Standards documents are supplied "AS IS".

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

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

In publishing and making this document available, the IEC and IEEE are not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Neither the IEC nor IEEE is undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEC/IEEE Dual Logo International Standards or IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations – Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

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

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

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

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

IEEE Standard for

Signal and Test Definition

Sponsor

IEEE Standards Coordinating Committee 20 on Test and Diagnosis for Electronic Systems

Approved 2 February 2005

American National Standards Institute

Approved 23 September 2004

IEEE-SA Standards Board

Abstract: This standard provides the means to define and describe signals used in testing. It also provides a set of common basic signals, mathematically underpinned so that signals can be combined to form complex signals usable across all test platforms.
Keywords: ATE, ATLAS, automatic test equipment, signal definitions, test definitions, test requirements, test signals, unit under test, UUT

IEEE Introduction

This standard is the culmination of a radical review of the Abbreviated Test Language for All Systems (ATLAS) and the requirement to create truly portable test requirements. During the review process, it was determined that it would be impractical to revise the existing ATLAS standard to include the required improvements. The decision was made to formulate a new standard to resolve these issues.

The key feature of the signal and test definition (STD) in this standard is 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 will be the same whatever equipment 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 STD can be used in a programming environment of the user's choice provided that that environment fulfills the minimum requirements stated 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 Test Description Subcommittee, whose intention is to prepare a companion guide to explain how to implement signal definitions and test requirements in conformance with STD.

Notice to users

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: http://standards.ieee.org/reading/ieee/updates/errata/index.html. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: http://standards.ieee.org/reading/ieee/interp/index.html.

Patents

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

STANDARD FOR SIGNAL AND TEST DEFINITION

1. Overview

This standard, also known as the signal and test definition (STD) standard, is the culmination of a radical review of the Abbreviated Test Language for All Systems (ATLAS) test programming language and the requirement to create truly portable test requirements. STD will allow test information to pass more freely between the design, test, and maintenance phases of a project, enabling the same information to be used directly across project phases. This more efficient use of information will lead to reduced life-cycle costs.

STD provides the capability to 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).

Portability is extended beyond that of test specifications by virtue of a layered architecture.

STD defines a collection of objects and their interfaces. These objects describe signal components relevant to test requirements. The STD standard defines how to interconnect these objects using interfaces, through which the objects exchange information, so that a test model may be defined that describes actual test requirements.

Finally, the link to published ATLAS standards (such as IEEE Std 716™-1995 [B9][1]) is preserved in that the user can describe signal operations using very similar test-signal-related keywords. These keywords now have formal definitions. Furthermore, the parameters of the signals themselves also have a rigorous formal behavioral description.

1.1 Scope

This standard provides the means to define and describe signals used in testing. It also provides a set of common basic signals, mathematically underpinned so that signals can be combined to form complex signals usable across all test platforms. The provision of language elements supports test signal descriptions for interoperability.

[1]The numbers in brackets correspond to the numbers of the bibliography in Annex L.

This standard is divided into seven clauses:

— Clause 1 provides an introduction to this standard.
— Clause 2 provides definitions of terms and lists abbreviations.
— Clause 3 describes the structure of the STD standard.
— Clause 4 specifies the signal modeling language (SML).
— Clause 5 specifies the STD basic signal components (BSCs).
— Clause 6 defines the test signal frameworks (TSFs).
— Clause 7 describes the test procedure language (TPL) layer.

This standard also contains the following annexes:

a) Annex A provides the Signals Modeling Language that is used to construct the BSCs and the TSFs.
b) Annex B provides BSC descriptions.
c) Annex C provides dynamic signal model description, states, and state transitions.
d) Annex D provides the interface definition language (IDL) description for the BSCs.
e) Annex E provides a TSF. This framework provides a formal description of signals similar to the signals defined in IEEE Std 716-1995. It also serves to illustrate how complex test signal models can be built up from BSCs.
f) Annex F provides the IDL description for the TSF provided in Annex E.
g) Annex G defines the requirements for a carrier language.
h) Annex H provides the formal TPL description.
i) Annex I provides the extensible markup language (XML) description mapping signal models into XML descriptions.
j) Annex J provides XML description mapping the TSF provided in Annex E into XML descriptions.
k) Annex K provides a description of how ATLAS nouns and noun modifiers are supported by STD.
l) Annex L provides a bibliography of related documents.

1.2 Purpose

The purpose of this standard is to provide a common signal reference for use throughout the life cycle of a unit under test (UUT) or test system. Such a reference will in turn facilitate information transfer, test reuse, and broader application of test information—accessible through commercially available development tools.

2. Definitions, abbreviations, and acronyms

2.1 Definitions

For the purposes of this document, the following terms and definitions apply. *The Authoritative Dictionary of IEEE Standards Terms* [B8] should be referenced for terms not defined in this clause.

2.1.1 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.

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

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

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

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

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

2.1.7 class: A generic set of predefined abstract test objects.

2.1.8 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.

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

2.1.10 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.

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

2.1.12 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**.

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

2.1.14 method: *Syn:* **function**.

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

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

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

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

2.1.19 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 other purpose than as defined for that language.

2.1.20 semantics: A branch of linguistics concerned with meaning. For the test procedure language (TPL), semantics is the connotative meaning of words in an 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.

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

2.1.22 SignalFunction: The name of the base class, for all classes that provide signals.

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

2.1.24 static signal: A signal whose definition does not change over time. All basic signal components (BSCs) and test signal frameworks (TSFs) are static signals.

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

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

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

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

2.1.29 test: An action or group of actions that are performed on a unit under test (UUT) to evaluate its parameter(s) or characteristic(s).

2.1.30 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.

2.1.31 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.

2.1.32 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.

2.1.33 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.

2.1.34 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.

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

2.1.36 value: The quantitative size of a signal attribute.

2.2 Abbreviations and acronyms

ARB	auxiliary reference burst
ARINC	Aeronautical Radio, Inc.[2]
ASCII	American Standard Code for Information Interchange

[2]ARINC provides engineering specifications and services to airlines, airframe constructors, and avionics suppliers.

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
DME	distance measuring equipment
IDL	interface definition language
IFF	identification, friend, or foe
ILS	instrument landing system
MRB	main reference burst
PCS	piecewise continuous signal
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
TPL	test procedure language
TSF	test signal framework
UHF	ultrahigh frequency
UUT	unit under test
VHF	very high frequency
VOR	VHF omnidirectional range
XMD	XML signal definition
XML	extensible markup language

3. Structure of this standard

3.1 Layers

This standard has a layered format depicted below in Figure 1. Each layer and its function are described in this clause. 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.

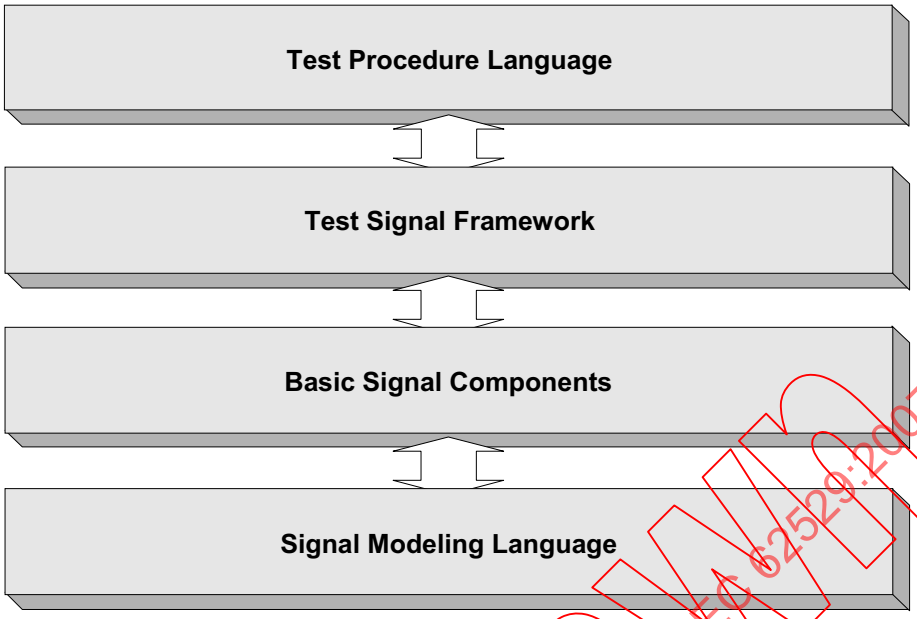


Figure 1—STD layers

The layer model provides additional support features over a purely textual language. 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 definitions, which can be written in any programming language or object-oriented environment. The link to ATLAS and Common/Abbreviated Test Language for All Systems (C/ATLAS) standards is preserved as test requirements are mapped to signals that have formal definitions tied to the TSF and BSC layers.

Throughout all the layers, there exists the concept of signals. Signals are characterized by their type, which may be typeless or may map onto a physical property. Examples of typeless signals are events or generic (such as SUM), and an example of a type that maps onto a physical property is voltage, such as in a Sinusoidal signal of type Voltage. The type of a signal also includes the reference type; most signals encountered in this standard have the reference type Time. Therefore, a Sinusoidal Voltage signal describes how the signal's voltage changes with respect to time, in a sinusoidal manner. The use of types is such that signals can be combined only when they have the same types or when one of the types is typeless or generic.

Extendibility is served by providing the capability to describe new signals formally by creating them from the existing signals in either the TSF or BSC layers.

3.1.1 SML layer

The SML layer provides the mathematical definitions that support the description of BSCs. This mathematical underpinning provides evidence that the signals are defined by BSCs. 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.

3.1.2 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, IDL description, and SML signal definition.

3.1.3 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 definitions made up from other TSF signals and BSCs.

The TSF is the extendibility mechanism that allows the creation of additional signal class definitions. Each TSF class is described by its class name, class types, properties and default values, IDL description, and a static signal model definition.

3.1.4 TPL layer

The TPL layer allows test descriptions (e.g., test requirements, test procedures, or test programs) to be formally described in a textual format by combining STD signals with features that satisfy the carrier language requirements.

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.

4. Signal modeling language (SML) layer

The SML layer allows the definition 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 behaviors that can be combined as necessary to produce the desired signal definition. This clause describes the use of SML to define signals, the measurement of signal parameters, and the conditions that a signal must meet.

The SML provides an exact mathematical definition for each BSC, in terms of dependant and independent variables, by using the de-facto functional programming concepts of Haskell 98. This definition represents the functioning of an entity, which is a requirement for component use and reuse. This is accomplished by giving a formal definition of the syntax and predefined signals. An execution mechanism may be provided 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 *dependant variable*, and the reference type is known as the *independent variable*.

5. Basic signal component (BSC) layer

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

5.1 BSC layer base classes

All BSC classes used to define signals are derived from one of the base classes shown in Table 1. 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
Resource	Creates the BSC objects
PulseDefns	Defines a group of pulses
Physical	Real, dimensioned signal values

5.2 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 (i.e., reference type). 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

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.

NOTE—Annex D and Annex I are normative annexes in that they provide the normative descriptions for the BSCs in IDL and XML, respectively. This does not mean that the BSCs may not be described in other interface languages.[3]

5.3 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 the class name is always 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)

NOTES

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

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

6. Test signal framework (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 supplies an interface definition in IDL and a static signal model description. A TSF library will generally be a collection of domain-specific signal definitions.

If the TSF class is to be used utilizing the XML descriptions in the Require method or TPL setup statement, then the TSF class shall exist in an XML TSF library conforming to Annex J.

6.1 TSF classes

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

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

A TSF class shall define the IDL definition for its class and its attribute interface.

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

— BSCs
— Other TSF classes

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 true:

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

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

6.2 TSF signals

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

— Title, which indicates the syntax of the TSF class name (in upper case characters with underscores).
— 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.

6.2.1 Interface properties table

The interface properties table shall comprise of 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 will 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 must 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 will 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	0–2π 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, Distortion Max 3% would mean distortion must be less than (or equal to) 3%. As such, all such capability attributes are optional.

6.2.2 Model description table

The model description table describes the signal model using a network list format. The model description table shall comprise of 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 will be inputs to other BSCs or TSF signals within the model, and at least one will be the output from the model.

— *Formula*. An optional mathematical definition (or constant value) of the function or input.

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		

6.2.3 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. 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.

7. Test procedure language (TPL) layer

The TPL layer 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 C/ATLAS inasmuch as the TPL uses stylized English signal statements to describe tests and to manipulate signals. It differs from using C/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.

7.1 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.

7.2 Elements of the TPL

The TPL comprises 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.

7.3 Use of the TPL

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

Users must recognize that there must 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.

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 UUT. Occasionally, a user may need to refer to the SML either for the mathematical justification 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 will need to refer to the SML in order to ensure that any new keyword that is introduced into the test methodology is coherent with the existing predefined keywords.

For 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 has been adopted.

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 BSCs

— Definitions for the combining mechanism for piecewise continuous signals (PCSs) and others

As a convention, the reserved words for the SML entities are written in *italics* in the format descriptions.

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

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

Example:

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

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

Example:

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

In other words, a value of type Resistance is written using one of the three type constructor keywords, OHM, KOHM, MOHM, followed by a value of type Float, e.g., 5 k 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 (Eq 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 Eq 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 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 (Eq a, Show a) => Physical a where
>       fromPhysical:: a -> Float
```

Floating-point values may be converted to physical values by prefixing them by 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*.

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 (Eq, 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

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


Table A.1—SML physical types and their units

SML type	Unit	S/A*	10^0 SML	10^3 SML	10^6 SML	10^9 SML	10^{-3} SML	10^{-6} SML	10^{-9} SML	10^{-12} SML
PlaneAngle	radian	rad	RAD				MRAD	URAD		
	degree	°	DEG							
	revolu-tion		REV							
SolidAngle	stera-dian	sr	SR				MSR			
Capaci-tance	farad	F	FD					UFD	NFD	PFD
Charge	cou-lomb	C	C	KC				UC	NC	
Conduc-tance	sie-mens	S	S							
Current	ampere	A	A	KA			MA	UA	NA	

Table A.1—SML physical types and their units *(continued)*

SML type	Unit	S/A*	10^0 SML	10^3 SML	10^6 SML	10^9 SML	10^{-3} SML	10^{-6} SML	10^{-9} SML	10^{-12} SML
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			
	elec-tron-volt	eV	EV	KEV	MEV					
Magnetic-Flux	weber	Wb	WB				MWB			
Magnetic-Flux-Density	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	can-dela per square meter	cd/m^2	NT							
Luminous-Flux	lumen	lm	LM							
Luminous-Intensity	candela	cd	CD							
Mass	kilo-gram	kg	KG				G	MG	UG	
Power	watt	W	W	KW			MW	UW		
Pressure	pascal	Pa	PA	KPA			MPA	UPA		
	milli-bar	mbar	MB							
Resistance	ohm	Ω	OHM	KOHM	MOHM					

Table A.1—SML physical types and their units *(continued)*

SML type	Unit	S/A*	10^0 SML	10^3 SML	10^6 SML	10^9 SML	10^{-3} SML	10^{-6} SML	10^{-9} SML	10^{-12} SML
Tempera-ture	kelvin	K	DEGK							
	deg. Celsius	°C	DEGC							
	deg. Fahren-heit	°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			

*S/A = symbol or abbreviation.

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	S/A*	10^0 SML	10^3 SML	10^6 SML	10^9 SML	10^{-3} SML	10^{-6} SML	10^{-9} SML	10^{-12} SML
Burst-Length	cycle		CYCLE							
	pulse		PULSE							
RatioInOut	decibel	dB	DB							

*S/A = symbol or abbreviation.

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 Fourier.

```
>module Pure where
>import Physical
>import Fourier
>import Complex
>import Random
>infixr 7 |>
```

Any data type 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
>  isNull:: (Physical a, Physical b) => (s a b)-> a -> Bool
```

```
>  mapSignal = mapSignal . toSig
>  mapSigList = map . mapSignal
>  toSig = FunctionRep . mapSignal
>  isNull _ _ = False
```

An instance of a signal can be observed at a specific point in its domain, effectively calling 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

— By a function, or

— As piecewise continuous windows made up of other signal representations, or

— By the value Not Present and detected through the use of the isNull method.

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

A signal representation is itself an instance of the class Signal.

```
>instance Signal SignalRep where
>  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
>  isNull NullRep _ = True
>  isNull (FunctionRep f) _ = False
>  isNull (PieceContRep f) t = isNull f t
```

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.

A.5 Pure signals

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

A.5.1 Nonperiodic signals

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

A.5.1.1 Constant

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

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

A.5.1.2 Linear

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

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

A.5.1.3 Random

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

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

A.5.1.4 Exponential

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

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

where

t is the time interval;

τ is the damping factor.

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

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

A.5.2 Periodic signals

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

A.5.2.1 Sinusoid

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

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

where

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

It has the following form:

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

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

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

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

A.5.2.2 Waveform

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

A waveform has the following form:

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

A.6 Pure signal-combining mechanisms

Clause A.5 established a number of ways to create signals, and this clause presents some mechanisms of combining signals together. Each mechanism will be handled separately.

A.6.1 Piecewise continuous signals (PCSs)

A PCS 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 a fixed amount of time (i.e., a time 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 (Physical a, Physical b) => Event a b=
```

a) The first kind of event is a time event, which is a specified period of time. The window lasts for the duration of time given. It has the following form:

```
>  TimeEvent 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) The second kind of event is a function event. 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) A third kind of event is an active event. The active event is when a signal representation transitions from a NullRep representation to a non-NullRep representation.

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

d) The fourth kind of event is the burst event. The burst event is triggered by a given number of triggers of another defined event. Its form is as follows:

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

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

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

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 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 width of 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 measuring the width and 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 entire PCS). Therefore, a flag is included that determines the zero against the event defining the width of the window and 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.6.1.3 Piecewise continuous functions

A piecewise continuous function 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)) NullRep, [])
>getWindow st t (w:wl) = if t' < et then (t', w, wl)
>                        else getWindow nt t wl
>    where et = eventOccurs e st'
>          wt = if t' < et then t' else et
>          (Window z e s) = w
>          t'  = if z == LocalZero then t-st else t
>          nt  = if z == LocalZero then st+wt else wt
>          st' = if z == LocalZero then 0.0 else st
>
>data PieceCont a b = Windows [FunctionWindow a b]
>instance Signal PieceCont where
>  mapSignal (Windows []) t = mapSignal NullRep t
>  mapSignal (Windows wl) t = (mapSignal s) (toPhysical t')
>      where (t', (Window z e s), wl') = getWindow 0.0 (fromPhysical t) wl
>  toSig = pieceRep
>  isNull (Windows []) t = True
>  isNull (Windows wl) t = (isNull 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 PCS. The form of a PCS is, therefore, as follows:

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

The preceding PCS, 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 PCSs 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

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

```
>sumSig:: (Physical a, Physical b, Signal s, Signal s') =>
>         (s a b) -> (s' a b) -> SignalRep a b
>sumSig f f' =
>  let s1 t = fromPhysical (mapSignal f t)
>      s2 t = fromPhysical (mapSignal f' 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:: (Physical a, Physical b, Signal s) => [ s a b ] -> SignalRep a b
>sumSigList ls = let zero = constant (toPhysical 0.0)
>               in foldr sumSig zero (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' =
>  let f1 t = fromPhysical (mapSignal f t)
>      f2 t = fromPhysical (mapSignal f' 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. It is, therefore, more restricted than other signal combination mechanisms. It takes a number of samples (which is always rounded up to the nearest power of two), the amount of time over which the signal will be sampled, and the signal to be converted. It has the following form:

```
>fourTrans:: (Physical a, Physical a', Physical b)=>
>               Int -> a -> SignalRep a b  -> SignalRep a' b
>fourTrans sam t f =
>  let
>    waveform:: (Physical a, Physical b) => a -> [b] -> SignalRep a b
>    waveform samp ampls =
>      let stepSlope y y' = (/) ((fromPhysical y') - (fromPhysical y))
>                               (fromPhysical samp)
>          makeWin (v,v') = Window LocalZero (TimeEvent (fromPhysical samp))
>                               (linear (stepSlope v v') v)
>          points = ampls ++ (cycle [(toPhysical 0.0)])
>      in pieceRep (Windows (map makeWin (zip points (tail points))))
>    s = 2 ^ ((truncate (logBase 2 ((fromInteger (toInteger sam)) - 1.0))) + 1)
>    si = toPhysical (1.0 / (fromPhysical t))
>    trl = sampleCount (toPhysical 0.0) t s f
>    mc x = (fromPhysical x) :+ 0.0
>    til = map mc trl
>    fil = fft til
>    frl = map magnitude fil
>  in waveform si (map toPhysical frl)
```

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

```
fft, fftinv:: [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 Measuring, 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 always 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. This representation, however, is complicated by the necessity to represent aspects of the behavior of digital signals within the electronic devices that operate on them.

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 unique in that their values do not take on physical values; rather, they take on enumeration values that represent physical values. A definition must 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 List
>import Pure
```

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

— H corresponds to true or 1; it translates to 1 in a control signal (but see below).

— L corresponds to false or 0; it translates to 0 in a control signal (but see below).

— 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 will sink very little current. It represents a signal at a high impedance.

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

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

— &* is the digital function *and*.

— |* is the digital function *or*.

— **notd** is the digital function *not*.

— =* is the digital function *xor* (i.e., equal).

```
>class DigitalOpers a where
>  (&*),(|*),(=*):: a -> a -> a
>  notd:: a -> a

>instance DigitalOpers Digital where
>  x &* y = case x of { L -> L; H -> y; X -> X; Z -> Z }
>  x |* y = case x of { L -> y; H -> H; X -> X; Z -> Z }
>  notd x = case x of { L -> H; H -> L; X -> X; Z -> Z }
>  x =* y = case x of
>             L -> if y == L then H else L
>             H -> if y == H then H else L
>             X -> X
>             Z -> Z
```

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 Time [ Digital ] deriving (Eq, Show)
>instance DigitalOpers DigitalSignal where
>  d1@(Dig t1 l1) &* d2@(Dig t2 l2) =
>    let doAnd (x,y) = x &* y
>        t = if t1 == t2 then t1 else
>              error "Attempting to and two signals with different times"
>    in Dig t (map doAnd (zip l1 l2))
>  d1@(Dig t1 l1) |* d2@(Dig t2 l2) =
```

```
>    let doOr (x,y) = x |* y
>        t = if t1 == t2 then t1 else
>                error "Attempting to or two signals with different times"
>    in Dig t (map doOr (zip l1 l2))
>  notd (Dig t dl) = Dig t (map notd dl)
>  d1@(Dig t1 l1) =* d2@(Dig t2 l2) =
>    let doXor (x,y) = x =* y
>        t = if t1 == t2 then t1 else
>                error "Attempting to xor two signals with different times"
>    in Dig t (map doXor (zip l1 l2))
```

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

```
>str2dig:: Time -> String -> DigitalSignal
>str2dig t s = Dig t (map char2dig (filter (not.isSpace) s)) where
>      char2dig 'L' = L
>      char2dig 'H' = H
>      char2dig 'X' = X
>      char2dig 'Z' = Z
```

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 NullRep 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 b, Signal s) => b -> b -> Time -> (s Time 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 = fromPhysical sampRate
>      val:: Float -> Digital
>      val x = let dv = fromPhysical (mapSignal s (toPhysical x)) - mt
>              in if isNull 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 gives no signal (NullRep).
— A value of L produces a low threshold analog value.
— A value of H produces a high threshold analog value.
— A value of X produces a value that is the mean of the high and low thresholds.

In addition, when the signal goes to Z, the level of the voltage signal tends to "float" as necessary as follows:

```
>d2a::(Physical b) => b -> b -> DigitalSignal -> (SignalRep Time b)
>d2a lowth highth s@(Dig clock ds) =
>  let mth = toPhysical ((fromPhysical lowth + fromPhysical highth)/2)
>      win s = Window LocalZero (TimeEvent (fromPhysical clock)) s
>      makewin Z = win NullRep
>      makewin H = win (constant highth)
>      makewin L = win (constant lowth)
>      makewin X = win (constant mth)
>  in pieceRep (Windows (map 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 white space is as follows:

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

The definition for a pattern is as follows:

```
>data Pattern = Pattern Time [ String ] deriving (Eq, Show)
```

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

```
>pat2diglist:: Pattern -> [ DigitalSignal ]
>pat2diglist (Pattern t ds) = map (str2dig t) (transpose ds)
```

A.10 Basic component SML

The BSC SML is defined in the module BSC

```
>module BSC where
```

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

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

by the following SML definitions: Source, Conditioner, EventFunction, Sensor, Digital, and Connection.

A.10.1 Source ::SignalFunction

A.10.1.1 NonPeriodic ::Source

A.10.1.1.1 Constant<type: Voltage|| Current|| Power> ::NonPeriodic

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

A.10.1.1.2 Step<type: Voltage|| Current|| Power> ::NonPeriodic

```
> (\startTime amplitude -> --Step startTime amplitude
>      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<type: Voltage|| Current|| Power> ::NonPeriodic

```
> (\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) |>
>     nullWindow
> in  pieceRep wins
> )
```

A.10.1.1.4 Noise<type: Voltage|| Current|| Power> ::NonPeriodic

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

A.10.1.1.5 SingleRamp<type: Voltage|| Current|| Power> ::NonPeriodic

```
> (\amplitude startTime riseTime ->let {
>    ;wins = Window LocalZero (TimeEvent startTime') (constant (toPhysical 0)) |>
>        Window LocalZero (TimeEvent riseTime') (linear (amplitude'/
riseTime') (toPhysical 0)) |>
>        Window LocalZero (TimeEvent inf) (constant amplitude) |>
>        nullWindow
>    ;amplitude' = fromPhysical amplitude
>    ;startTime' = fromPhysical startTime
>    ;riseTime' = fromPhysical riseTime
>    } in pieceRep wins
> )
```

A.10.1.2 Periodic ::Source

A.10.1.2.1 Sinusoid<type: Voltage|| Current|| Power> ::Periodic

```
> (\amplitude frequency phase -> --
Sine_wave {amplitude=amplitude, frequency=frequency, phase_angle=phase}
>      sine amplitude frequency phase
> )
```

A.10.1.2.2 Trapezoid<type: Voltage|| Current|| Power> ::Periodic

```
> (\amplitude period riseTime pulseWidth fallTime -> let
>  period' = fromPhysical period
>  riseTime' = fromPhysical riseTime
>  pulseWidth' = fromPhysical pulseWidth
>  fallTime' = fromPhysical fallTime
>  trapezoid = pieceRep $
>   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 ) |>
>   nullWindow
> in pieceRep $ cycleWindows $  Window LocalZero (TimeEvent period') trapezoid |>
nullWindow
> )
```

A.10.1.2.3 Ramp<type: Voltage|| Current|| Power> ::Periodic

```
> (\amplitude period riseTime -> --Ramp {period=period, rise_time=riseTime,
level=amplitude}
>       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<type: Voltage|| Current|| Power> ::Periodic

```
> (\amplitude period dutyCycle-> --Triangle {period=period, level=amplitude}
>      let per = fromPhysical period
>          v = fromPhysical amplitude
>          qper = per *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<type: Voltage|| Current|| Power> ::Periodic

```
> (\amplitude period dutyCycle -> --Square {period=period, level=amplitude}
>       let per = fromPhysical period
>           lvl = fromPhysical amplitude
>           trans = per * 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<type: Voltage|| Current|| Power> ::Periodic

```
> (\amplitude samplintInt points ->
>    let pts = map ((*) (fromPhysical amplitude)) points in
>        waveform samplintInt (map toPhysical pts)
> )
```

A.10.1.2.7 WaveformStep<type: Voltage|| Current|| Power>::Periodic

```
> (\amplitude samplingInt points ->
>    let pts = map ((*) (fromPhysical amplitude)) points in
>        FunctionRep (\t->cycle (map toPhysical pts) !! floor (fromPhysical t /
(fromPhysical samplingInt)))
> )
```

A.10.2 Conditioner ::SignalFunction

A.10.2.1 Filter ::Conditioner

A.10.2.1.1 BandPass ::Filter

```
> (\interval samples centerFrequency frequencyBand signal -> let { t = toPhysical
(interval);s=samples
>      ;cf = centerFrequency; fb = 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 (1+x) z ++ take (y-x+1) o ++ take (s-2*y-3) 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
>      ;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)/(fromInt s))) (map toPhysical
frl)
> )
```

A.10.2.1.2 LowPass ::Filter

```
> (\interval samples passband signal ->let { t = toPhysical (interval);s=samples
>      ;x = truncate(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)/(fromInt s))) (map toPhysical
frl)
> )
```

A.10.2.1.3 HighPass ::Filter

```
> (\interval samples passband signal -> let { t = toPhysical interval;s=samples
>      ;x = truncate(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)/(fromInt s))) (map toPhysical
frl)
> )
```

A.10.2.1.4 Notch ::Filter

```
> (\interval samples centerFrequency frequencyBand signal -> let { t = toPhysical
(interval);s=samples
>      ;cf = centerFrequency; fb = 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 (1+x) o ++ take (y-x+1) z ++ take (s-2*y-3) 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
>      ;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)/(fromInt s))) (map toPhysical
frl)
> )
```

A.10.2.2 Combiner ::Conditioner

A.10.2.2.1 Sum ::Combiner

```
> (\signals -> foldl1 sumSig ((constant (toPhysical 0)):signals))
```

A.10.2.2.2 Product ::Combiner

```
> (\signals -> foldl1 mulSig signals)
```

A.10.2.2.3 Diff ::Combiner

```
> (\signals -> foldl1 (\x y -
> sumSig x  (negate y)) (signals++[constant (toPhysical 0)]))
```

A.10.2.3 Modulator ::Conditioner

A.10.2.3.1 FM ::Modulator

```
> (\freqDev carAmpP carFreq signal->
>    let {
>    ; phsfnc = mulSig (constant (toPhysical (freqDev*2*pi))) (intSig (16*carFreq) signa
l)
>    ; freqFn = constant (toPhysical carFreq)
>    ; fm = sineFunc (constant carAmpP) freqFn phsfnc
>    } in toSig fm
> )
```

A.10.2.3.2 AM ::Modulator

```
> (\modIndex carrier signal  ->
>    let one = constant (toPhysical 1.0)
>        ;modsig = mulSig (constant (toPhysical modIndex)) signal
>    in mulSig carrier (sumSig one modsig)
> )
```

A.10.2.3.3 PM ::Modulator

```
> (\phaseDev carAmpP carFreq signal ->
>    let {phsfnc = mulSig (constant (toPhysical phaseDev)) signal
>                    ; freqFn = constant (toPhysical carFreq)
>                    ; pm = sineFunc (constant carAmpP) freqFn phsfnc
>                    } in toSig pm
> )
```

A.10.2.4 Transformation ::Conditioner

A.10.2.4.1 SignalDelay ::Transformation

```
> (\delay rate acceleration signal -> let {
>    ;dt t = delay + rate*t + acceleration*t*t/2
>    ;t' t = max 0 (t-(dt t))
>--     ;delaySig s = FunctionRep (\t ->mapSignal s (toPhysical (t' (fromPhysical t))))
>
>    ;delaySig NullRep = NullRep
>     ;delaySig (FunctionRep fn) = FunctionRep (\t -
>fn (toPhysical (t' (fromPhysical t))))
>     ;delaySig (PieceContRep (Windows xs)) = PieceContRep (Windows ((Window GlobalZero
(TimeEvent (t'' 0)) NullRep):(delayWin 0 xs)))
>    ;delayWin gt ((Window GlobalZero  (TimeEvent t) s):xs) = (Window GlobalZero (TimeEv
ent (t'' t)) (delaySig s)):(delayWin t xs)
>    ;delayWin gt ((Window LocalZero (TimeEvent t) s):xs) = (Window LocalZero (TimeEvent
 ((t'' (gt+t))-(t'' gt))) (delaySig s)):(delayWin (gt+t) xs)
>    ;sqe c 0 0 t = 0
>    ;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)
>    } in delaySig signal)
```

A.10.2.4.2 Exponential ::Transformation

```
> mulSig (expc (1.0))
```

A.10.2.4.3 E ::Transformation

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

A.10.2.4.4 Negate ::Transformation

```
> (\signal -> mulSig signal (constant (toPhysical (-1))))
```

A.10.2.4.5 Inverse ::Transformation

```
> (\f ->
>   let f1 t = fromPhysical (mapSignal f t)
>   in FunctionRep (\t -> toPhysical (1.0 / (f1 t)))
> )
```

A.10.2.4.6 PulseTrain ::Transformation

```
> (\repetition pulses ->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.10.2.4.7 Attenuator ::Transformation

```
> (\gain -> mulSig (constant (toPhysical gain)))
```

A.10.2.4.8 Load ::Transformation

```
> id
```

A.10.2.4.9 Limit<type: Voltage|| Current|| Power> ::Transformation

```
> (\lim sig -> FunctionRep (\t->limit (toPhysical (-
lim)) (toPhysical lim) (mapSignal sig t)))
```

A.10.2.4.10 FFT ::Transformation

```
> (\samples interval signal -> fourTrans samples interval (toSig (mulSig (constant
(toPhysical 2)) signal)))
```

A.10.3 EventFunction ::SignalFunction

A.10.3.1 EventSource ::EventFunction

A.10.3.1.1 Clock ::EventSource

```
> (\clock_rate->
>           let {
>    ;per = 0.5 / (fromPhysical clock_rate)
>    ;one = constant (toPhysical 1.0)
```

```
>    ;zer = NullRep
>    ;wins = Window LocalZero (TimeEvent per) one |>
>    Window LocalZero (TimeEvent per) zer |>
>          nullWindow
>    }in pieceRep (cycleWindows wins) )
```

A.10.3.1.2 TimedEvent ::EventSource

```
> (\delay repetition duration period -> let {
>    ;one = constant (toPhysical 1)
>    ;zero = NullRep
>    ;repNX 0 0 ls = cycleWindows ls
>    ;repNX delay 0 ls = Window LocalZero (TimeEvent delay) zero |> cycleWindows ls
>    ;repNX 0 x ls = repNWindows x ls
>    ;repNX delay x ls = Window LocalZero (TimeEvent delay) zero |> repNWindows x ls
>    } in pieceRep (repNX delay repetition
(Window LocalZero (TimeEvent duration) one |>
>                              Window LocalZero (TimeEvent (period-duration)) zero |>
>                              nullWindow))
> )
>
```

A.10.3.1.3 PulsedEvent ::EventSource

```
>(\repetition pulses -> let
>{
>       pt ps = let
>       {
>               pulse (a, b, c) = let
>               {
>                       zero = NullRep
>                       ;sig = constant (toPhysical 1)
>                       ;funcwins = Window LocalZero (TimeEvent (a)) zero :
>                               Window LocalZero (TimeEvent (b)) sig :
>                               []
>               } in funcwins
>               ;xOr NullRep s = s
>               ;xOr s _ = s
>               ;sumEvnts a b = splice xOr a b 0.0
>       }
>       in  Windows $ foldl1 (\a b->splice xOr a b 0.0) (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

```
>(\events -> let
>{
>       ;one = constant (toPhysical 1)
>       ;ebe e d = let
>       {
>               ;enable = toSig e
>               ;disable = toSig d
>               ;wins = Window LocalZero (ActiveEvent enable) NullRep |>
>                       Window LocalZero (ActiveEvent disable) one |> nullWindow
>       }
>               in pieceRep $ cycleWindows wins
>       ;sglEvt e = let
>       {
>               ;wins = Window LocalZero (ActiveEvent (toSig e)) NullRep |>
>                       Window LocalZero (TimeEvent inf) one |> nullWindow
>       }
>               in pieceRep $  wins
>} in case events of
>       (e:[]) -> sglEvt e
```

```
>       (e:es)  ->   foldl1 (ebe) ( events))
```

A.10.3.2.2 EventCount ::EventConditioner

```
> (\count event -> let {
>        ;ec [] _ = [Window LocalZero (TimeEvent inf) NullRep]
>        ;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

```
> (\seed propability event -> let {
>        ;pbe [] _ _ _ = [Window LocalZero (TimeEvent inf) NullRep]
>        ;pbe ((w@(Window z e NullRep)):wl) xs _ _ = w:pbe wl xs True True
>        ;pbe ws (x:xl) True _ = pbe ws xl False (x*100<=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

```
> (\event ->
>           let {
>    ;xNot NullRep _ = constant (toPhysical 1.0)
>    ;xNot _ _ = NullRep
>    } in PieceContRep $ 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

```
> (\events ->
>           let {
>    ;xOr NullRep s = s
>    ;xOr s _ = s
>    } in PieceContRep $ Windows $ foldl1 (\a b->splice xOr a b 0.0)
 (map functionWindows events))
```

A.10.3.2.5.2 XOrEvent ::Logical

```
> (\events ->
>           let {
>    ;xXOr s NullRep = s
>    ;xXOr NullRep s = s
>    ;xXOr _ _ = NullRep
>    } in PieceContRep $ Windows $ foldl1 (\a b-> splice xXOr a b 0.0)
 (map functionWindows events))
```

A.10.3.2.5.3 AndEvent ::Logical

```
> (\events ->
>           let {
>    ;xAnd _ NullRep = NullRep
>    ;xAnd s _ = s
>    } in PieceContRep $ Windows $ foldl1 (\a b->splice xAnd a b 0.0)
 (map functionWindows events))
```

A.10.4 Sensor ::SignalFunction

A.10.4.1 Counter ::Sensor

```
> \points -> length points
```

A.10.4.2 TimeInterval ::Sensor

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

A.10.4.3 Instantaneous<type: Voltage|| Current|| Power|| Frequency|| Resistance|| Capacitance|| Conductance|| Inductance> ::Sensor

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

A.10.4.4 RMS<type: Voltage|| Current> ::Sensor

```
> \points ->  toPhysical $ sqrt $ (foldl (+) 0 $
 map ((\x->x*x).fromPhysical.fst) points) / fromInt (length points)
```

A.10.4.5 Average<type: Voltage|| Current|| Power|| Frequency> ::Sensor

```
> \points -> toPhysical $
 (foldl (+) 0 (map (fromPhysical.fst) points)) / fromInt (length points)
```

A.10.4.6 PeakToPeak<type: Voltage|| Current|| Power|| Frequency> ::Sensor

```
> \points -> let {
>    ;h = foldl1 max $ map (fromPhysical.fst) points
>    ;l = foldl1 min $ map (fromPhysical.fst) points
>    } in toPhysical $ h - l
```

A.10.4.7 Peak<type: Voltage|| Current|| Power|| Frequency> ::Sensor

```
> \points -> toPhysical $ (foldl1 max $ map (fromPhysical.fst) points) -
 (foldl (+) 0 (map (fromPhysical.fst) points)) / fromInt (length points)
```

A.10.4.8 PeakNeg<type: Voltage|| Current|| Power|| Frequency> ::Sensor

```
> \points -> toPhysical $ (foldl1 min $ map (fromPhysical.fst) points) -
 (foldl (+) 0 (map (fromPhysical.fst) points)) / fromInt (length points)
```

A.10.4.9 MaxInstantaneous<type: Voltage|| Current|| Power|| Frequency> ::Sensor

```
> \points -> toPhysical $ foldl1 max $ map (fromPhysical.fst) points
```

A.10.4.10 MinInstantaneous<type: Voltage|| Current|| Power|| Frequency> ::Sensor

```
> \points -> toPhysical $ foldl1 min $ map (fromPhysical.fst) points
```

A.10.4.11 Measure ::Sensor

A.10.5 Digital ::SignalFunction

A.10.5.1 SerialDigital<type: Voltage|| Current> ::Digital

```
> (\d period hv lv ->  d2a lv hv (str2dig period d)  )
```

A.10.5.2 ParallelDigital<type: Voltage|| Current> ::Digital

```
> (\ds period hv lv -> map (d2a lv hv) (pat2diglist $ Pattern period ds) )
```

A.10.6 Connection ::SignalFunction

```
> id
```

A.10.6.1 TwoWire ::Connection

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

A.10.6.2 TwoWireComp ::Connection

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

A.10.6.3 ThreeWireComp ::Connection

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

A.10.6.4 SinglePhase ::Connection

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

A.10.6.5 TwoPhase ::Connection

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

A.10.6.6 ThreePhaseDelta ::Connection

```
> (\xs -> let { f s@(s':s'':s''':[]) =xs
>     ;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 xs)
```

A.10.6.7 ThreePhaseWye ::Connection

```
> (\xs -> let { f s@(s':s'':s''':[]) =xs
>     ;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 xs)
```

A.10.6.8 ThreePhaseSynchro ::Connection

```
> (\xs -> let { f s@(s':s'':s''':[]) =xs
>     ;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 xs)
```

A.10.6.9 FourWireResolver ::Connection

```
> (\xs -> let { f s@(s':s'':s''':s'''':[]) =xs
>     ;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 xs)
```

A.10.6.10 SynchroResolver ::Connection

```
> (\xs -> let { f s@(s':s'':s''':s'''':[]) =xs
>     ;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 xs)
```

A.10.6.11 Series ::Connection

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

A.10.6.12 NonElectrical ::Connection

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

A.10.6.13 DigitalBus ::Connection

```
> id
```

Annex B

(normative)

Basic signal component (BSC) layer

B.1 BSC layer base classes

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

Table B.1—Signal function base classes

Base class	Description
SignalFunction	The base class of all BSCs
Signal	Allows BSCs to exchange information
Resource	Creates the BSC objects
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.

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

Subclasses			Attributes
1st level	2nd level	3rd/4th level	
Source	—	—	—
	NonPeriodic	—	—
		Constant	amplitude
		Step	amplitude startTime
		SingleTrapezoid	amplitude startTime riseTime pulseWidth fallTime

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

Subclasses			Attributes
1st level	**2nd level**	**3rd/4th level**	
Source, *continued*	NonPeriodic, *continued*	Noise	amplitude seed frequency
		SingleRamp	amplitude riseTime startTime
	Periodic	—	—
		Sinusoid	amplitude frequency phase
		Trapezoid	amplitude period riseTime pulseWidth fallTime
		Ramp	amplitude period riseTime
		Triangle	amplitude period dutyCycle
		SquareWave	amplitude period dutyCycle
		WaveformRamp	amplitude period samplingInterval points
		WaveformStep	amplitude period samplingInterval points
Conditioner	—	—	—
	Filter	—	—
		BandPass	centerFrequency frequencyBand
		LowPass	cutoff
		HighPass	cutoff
		Notch	centerFrequency frequencyBand

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

Subclasses			Attributes
1st level	**2nd level**	**3rd/4th level**	
Conditioner, *continued*	Combiner	—	—
		Sum	—
		Product	—
		Diff	—
	Modulator	—	—
		FM	amplitude carrierFrequency frequencyDeviation
		AM	modIndex Carrier
		PM	amplitude carrierFrequency phaseDeviation
	Transformation	—	—
		SignalDelay	acceleration delay rate
		Exponential	dampingFactor
		E	—
		Negate	—
		Inverse	—
		PulseTrain	pulses repetition
		Attenuator	gain
		Load	resistance reactance
		Limit	limit
		FFT	samples interval
EventFunction	—	—	—
	EventSource	—	—
		Clock	clockRate
		TimedEvent	delay duration period repetition
		PulsedEvent	pulses repetition

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

Subclasses				Attributes
1st level	**2nd level**	**3rd/4th level**		
EventFunction, *continued*	EventConditioner	—		—
		EventedEvent		—
		EventCount		count
		ProbabilityEvent		seed probability
		NotEvent		—
		Logical	—	—
			OrEvent	—
			XOrEvent	—
			AndEvent	—
Sensor	—	—		measuredVariable measurement measurements samples count gateTime nominal condition NOGO GO HI LO UL LL
	Counter	—		—
	TimeInterval	—		—
	Instantaneous	—		—
	RMS	—		—
	Average	—		—
	PeakToPeak	—		—
	Peak	—		—
	PeakNeg	—		—
	MaxInstantaneous	—		—
	MinInstantaneous	—		—
	Measure	—		As attribute
Control	(Reserved)	—		—

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

Subclasses			Attributes
1st level	**2nd level**	**3rd/4th level**	
Digital	—	—	—
	SerialDigital	—	data period logic_H_value logic_L_value
	ParallelDigital	—	data period logic_H_value logic_L_value
Connection	—	—	channelWidth
	TwoWire	—	lo hi (channelWidth = 1)
	TwoWireComp	—	true comp (channelWidth = 1)
	ThreeWireComp	—	true comp lo (channelWidth = 1)
	SinglePhase	—	a n (channelWidth = 1)
	TwoPhase	—	a b n (channelWidth = 2)
	ThreePhaseDelta	—	a b c (channelWidth = 3)
	ThreePhaseWye	—	a b c n (channelWidth = 3)
	ThreePhaseSynchro	—	x y z (channelWidth = 3)
	FourWireResolver	—	s1 s2 s3 s4 (channelWidth = 2)

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

Subclasses			Attributes
1st level	**2nd level**	**3rd/4th level**	
Connection, *continued*	SynchroResolver	—	r1 r2 r3 r4 (channelWidth = 1 or channelWidth = 2)
	Series	—	via (channelWidth = 1)
	NonElectrical	—	to from (channelWidth = 1)
	DigitalBus	—	—

B.3 Description of a BSC

This clause 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.

NOTE—The use of **boldface** denotes a class, first-level subclass, property, or interface name in the text of this annex.

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.

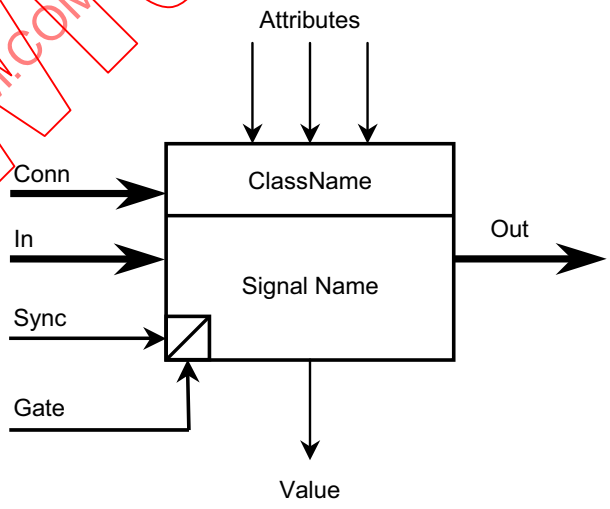


Figure B.1—BSC diagram

In the figure, the following naming conventions are used.

a) ClassName is the name of the class that the template represents, e.g., **Constant**.

b) Signal Name is the name of the specific signal being modeled, e.g., dc signal.

B.3.2 BSC interfaces

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 properties of **In**, **Out,** etc. 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 what happens to their **Signal** properties and attributes. For testing purposes, what happens with signals at the UUT or test interface is of ultimate interest. Where a signal defined by a BSC model passes through a **Connection** subclass, 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 only become a physical entity that can be used for testing 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** refers to the BSC **Signal** whereas the term *signal* 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)**

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

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

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. 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.)

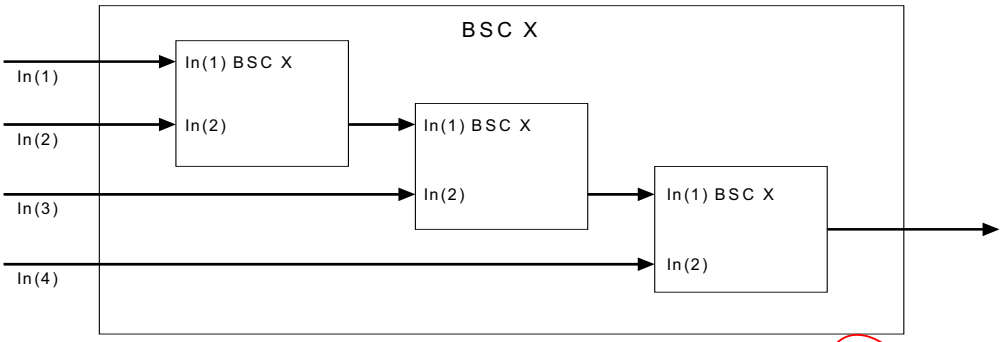


Figure B.2—Multiple inputs semantics

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. For a **Source,** this action would be outputting a signal or not outputting a signal. 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. (See subclass descriptions in B.3.5 for more examples.)

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** connections. By contrast, all **In** signal's states are monitored, and the **In** signal is controlled by the BSC.

B.3.3 Types of BSCs

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

a) Typeless—represents signals such as events, and can be combined with any other signal of any type

b) Generic— A generic type 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 only be used when their input types are all of the same physical type

c) Physical—defines the type of a signal, e.g. 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, comma separated and within brackets.

Example:

— Constant (Voltage, Time)

Where no explicit types are provided, the default type is voltage and the default reference type is time.

Examples:

Constant (Power) is equivalent to **Constant (Power, Time).**
Constant is equivalent to **Constant(Voltage)** and **Constant (Voltage, Time).**

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

Example:

— **WaveFormStep** (**Voltage**, **Frequency**)

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 value provided is meaningful in that the default value 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 will be correct in many instances.

This situation will not be true for most 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 correct. The user shall ensure that the correct value is provided for each use of a BSC.

B.3.5 BSC 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** must 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. **Source**s 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.

B.3.5.2 Subclass description: Conditioner

A **Conditioner** combines and conditions one or more commutative input signals into an associated output signal, based on any attribute values. A **Conditioner** must have at least one **In** signal and will 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 remain 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** must create an **Out Signal** interface and may possess attributes. It may have a **In** signal, will 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 same semantics as *Anding* the **EventFunction** event with the **Gate** event.

Events can originate from either signals or other events; however, events generated from an **EventFunction** do not contain any signal information and cannot be used in place of a signal.

B.3.5.4 Subclass description: Sensor

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

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

Sensors are used both to monitor signals by creating events and to take measurement values.

B.3.5.5 Subclass description: Control

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

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, will support **Gate** and **Sync** events, but does not possess values.

This subclass group is reserved for future use.

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** must 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. **Digitals** possess, but do not use, any **In** signals.

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 attributes identifying the names of the pins through which the signals pass, **Gate** and **Sync**. A **Connection** does not possess any values.

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 through which the physical signal must flow. A signal is only considered an external quantity where it passes through a **Connection**.

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	Pins names and properties
TwoWire	Two wire	hi, lo
TwoWireComp	Two wire complement	true, compl
ThreeWireComp	Three wire complement	true, compl, lo
SinglePhase	Single phase	a, n b, n c, 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
NonElectrical	Nonelectrical	to, from
DigitalBus	Data or address or control bus	width

B.4 Physical class

The **Physical** base class (see Table B.1 in 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**.

Properties

value <string>(default) contains the full textual description (e.g., "RMS 3 V errlmt 100 mV range 1 V to 10 V").
magnitude <real> is the value of the physical type, e.g., 3.0.
units <enum> is the unit symbol of the value, e.g., V, Hz, A.
qualifier <enum> provides different ways of observing the value; contains one of
— trms (true root mean square)
— pk_pk (peak-peak)
— pk (peak)
— pk_pos (positive peak same as peak)
— pk_neg (negative peak)
— av (average)
— inst (instantaneous)
— inst_max (instantaneous maximum value)
— inst_min (instantaneous minimum value)
errlmt <enum:UL,LL>
— magnitude <real> is the value of the UL or LL error limit, e.g., 0.10.
— units <enum> is the unit symbol of the UL or LL errot limit, e.g., pc.

range <enum:MAX,MIN>

— magnitude <real> is the value of the maximum or minimum range, e.g., 10.

— units <enum> is the unit symbol of the maximum or minimum range, e.g., V.

The **value** property is the default property of the **Physical** class and is internally parsed to complete the **magnitude**, **units**, **qualifier**, **errlmt**, and **range** properties. Explicitly changing any **Physical** class properties will also change the default **value** property.

Note—The enumeration value for the **units** property never contains the metric prefix, e.g., 300 mV, and has SI **magnitude** 0.3 and **units** V. The default value for the **qualifier** is determined by the associated signal attribute.

The string format of the **value** property is defined as follows:

```
[<Qualifier>] <numeric expression> [<Unit>]
[(errlmt|±|+-|+) <numeric expression> [<Unit>|pc|%] [-|:|to
<numeric expression> [<Unit>|pc|%]]]
[range [+|-|MAX|MIN]<numeric expression> [<Unit>|pc|%] [-|:|to
<numeric expression> [<Unit>|pc|%]]
```

where

All the occurrences of <Unit> must 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>.

The **errlmt** is always expressed as a relative range to the value either as a **ratio** or ± a fixed amount (e.g., "10 V ± 10 mV" or "10 V ± 0.1%") and represents the uncertainty of the value.

The **range** property is always held as absolute values, identifying the range of values that may be used. The **value** syntax allows for relative range values to be specified (e.g., "10 V **range** ± 1 V" or "10 V **range** 1%" or "10 V **range** 11 V to 9 V") where these values are converted to absolute values.

A **Physical** class object value does not have to be written with any dimensional quantity. The units are taken from the **units** 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 value is not of interest, and any resource selection will not consider the missing property. Changing other property values do not affect other property values, except where there is a need to ensure consistency of dimensions. Specifying an **errlmt** uncertainty of zero implies that any resource selection will choose the best resource available.

For relative values, a single-ended, positive or negative, **errlmt** or **range** 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 **errlmt** ±10 mV," which is the same as "300 mV **errlmt** +10 mV –10 mV." Similarly, "300 mV **errlmt** +10 mV" is the same as "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 dimension (e.g., V) of the physical quantity also carries any attribute **qualifier**, such as pk_pk (e.g., pk_pk 5V).

When assigning physical types to each other, the complete value of the physical type is transferred, e.g., "pk –8 V **errlmt** ±5% **range** –10 V to –5 V."

B.4.1 Permissible physical types and their units

Table B.4 lists the allowed quantities, physical types, unit symbols, and their units.[4]

Table B.4—Physical types

Quantity	Physical type	Unit	Units symbol (See Notes 1, 10)	Mappings	Property or note
Acceleration	Acceleration	meter per second squared	m/s^2		
Admittance	Admittance				See Note 14
Amount of information	AmountOfInformation	byte	B		
Amount of substance	AmountOfSubstance	mole	mol		
Angular acceleration	AngularAcceleration	radian per second squared	rad/s^2		
Angular velocity	AngularSpeed	radian per second	rad/s	Frequency	
Area	Area	square meter	m^2		
Capacitance	Capacitance	farad	F		
Concentration	Concentration	mole per cubic meter	mol/m^3		
Current density	CurrentDensity	ampere per square meter	A/m^2		
Dynamic viscosity	DynamicViscosity	pascal second	Pa·s		
Electric charge	Charge	coulomb	C		
Electric charge density	ElectricChargeDensity	coulomb per cubic meter	C/m^3		
Electric conductance	Conductance	siemens	S	Resistance	See Note 14
Electric current	Current	ampere	A		
Electric field strength	ElectricFieldStrength	volt per meter, newton per coulomb	V/m, N/C		See Note 11
Electric flux density	ElectricFluxDensity	coulomb per square meter	C/m^2		
Electric potential difference	Voltage	volt	V	Power	Load = 50 Ohm
Electric resistance	Resistance	ohm	Ohm	Admittance	See Notes 5, 14

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

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

Quantity	Physical type	Unit	Units symbol (See Notes 1, 10)	Mappings	Property or note
Electromotive force	Voltage	volt	V	Power	Load = 50 Ohm
Energy	Energy	joule, electronvolt	J, eV		
Energy density	EnergyDensity	joule per cubic meter	J/m^3		
Entropy	Entropy	joule per kelvin	J/K		
Exposure	Exposure	coulomb per kilogram	C/kg		
Force	Force	newton	N		
Frequency	Frequency	hertz	Hz	Time	
Heat	Heat	joule	J		
Heat capacity	HeatCapacity	joule per kelvin	J/K		
Heat flux density	HeatFluxDensity	watt per square meter	W/m^2		
Illuminance	Illuminance	lux	lx		
Impedance	Impedance				See Notes 5, 14
Inductance	Inductance	henry	H		
Irradiance	Irradiance	watt per square meter	W/m^2		
Kinematic viscosity	KinematicViscosity	square meter per second	m^2/s		
Length	Distance	meter, inch, foot, mile, nautical mile	m, in, ft, mi, nmi		See Note 13
Luminance	Luminance	candela per square meter	cd/m^2		
Luminous flux	LuminousFlux	lumen	lm		
Luminous intensity	LuminousIntensity	candela	cd		
Magnetic field strength	MagneticFieldStrength	ampere per meter	A/m		
Magnetic flux	MagneticFlux	weber	Wb		
Magnetic flux density	MagneticFluxDensity	tesla	T		
Mass	Mass	kilogram	kg		See Note 2
Mass density	MassDensity	kilogram per square meter	kg/m^2		
Mass Flow	MassFlow	kilogram per second	kg/s		

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

Quantity	Physical type	Unit	Units symbol (See Notes 1, 10)	Mappings	Property or note
Molar energy	MolarEnergy	joule per mole	J/mol		
Molar entropy	MolarEntropy	joule per mole kelvin	J/(mol·K)		
Molar heat capacity	MolarHeatCapacity	joule per mole kelvin	J/(mol·K)		
Moment of force	MomentOfForce	newton meter	N·m		
Moment of inertia	MomentOfInertia	kilogram meter squared	$kg·m^2$		
Momentum	Momentum	kilogram meter per second	kg·m/s		
Permeability	Permeability	henry per meter	H/m		
Permittivity	Permittivity	farad per meter	F/m		
Plane angle	PlaneAngle	radian, degree	rad, °, deg		See Notes 8, 12
Power	Power	watt, decibel watt, decibel milliwatt	W, dBW, dBm, dB(1 mW), dB(1 W)	Voltage	Load = 50 Ohm Also see Notes 3, 4
Power density	PowerDensity	watt per square meter	W/m^2		
Pressure	Pressure	pascal, millibar	Pa, mbar		See Note 9
Radiance	Radiance	watt per square meter steradian	$W/(m^2·sr)$		
Radiant intensity	RadiantIntensity	watt per steradian	W/sr		
Ratio	Ratio	decibel, percent	dB, %, pc		See Notes 3, 7
Reactance	Reactance	ohm	Ohm	Susceptance	See Notes 5, 14
Solid angle	SolidAngle	steradian	sr		
Specific energy	SpecificEnergy	joule per kilogram	J/kg		
Specific entropy	SpecificEntropy	joule per kilogram kelvin	J/(kg·K)		
Specific heat capacity	SpecificHeatCapacity	joule per kilogram kelvin	J/(kg·K)		
Specific volume	SpecificVolume	cubic meter per kilogram	m^3/kg		
Speed	Speed	meter per second	m/s		

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

Quantity	Physical type	Unit	Units symbol (See Notes 1, 10)	Mappings	Property or note
Surface tension	SurfaceTension	newton per meter	N/m		
Susceptance	Susceptance	siemens	S	Reactance	See Note 14
Thermal conductivity	ThermalConductivity	watt per meter kelvin	W/(m·K)		
Thermodynamic temperature	Temperature	Kelvin, degree Celsius, degree Fahrenheit	K, °C, °F, degC, degF		See Notes 3, 6, 13
Time	Time	second, minute, hour	s, min, h		See Notes 3, 12
Volume	Volume	cubic meter, liter	m^3, L		See Note 12
Volume flow	VolumeFlow	liter per second	L/s		

NOTES

1—It is preferred practice to leave one space between the numeric value and the unit symbol when defining a value. (See Note 8 below).

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

3—These units are not used with the SI prefixes.

4—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).

5—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.

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

7—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 (%).

8—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 abbreviation 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.

9—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 1 hPa.

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

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

11—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).

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

13—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.

14—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 *conductance* are reserved for future use as physical type names.

B.4.2 Unit prefixes

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

The μ 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^{-24}	
z	zepto	10^{-21}	
a	atto	10^{-18}	
f	femto	10^{-15}	
p	pico	10^{-12}	
n	nano	10^{-9}	
μ, u	micro	10^{-6}	See Note 1
m	milli	10^{-3}	
c	centi	10^{-2}	See Note 2
d	deci	10^{-1}	See Note 2
h	hecto	10^{+2}	See Note 2
k	kilo	10^{+3}	
M	mega	10^{+6}	
G	giga	10^{+9}	
T	tera	10^{+12}	
P	peta	10^{+15}	
E	exa	10^{+18}	
Z	zetta	10^{+21}	
Y	yotta	10^{+24}	

Table B.5—Metric prefixes *(continued)*

Prefix	Name	Value	Comments
NOTES 1—The preferred symbol for the prefix micro (i.e., μ) 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. 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.5 PulseDefns class

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

The **PulseDefns** class shall also support the *_NewEnum* enumeration property, where IEnumVARIANT and VARIANT are defined in IDL.

Properties

_NewEnum returns an object that supports IEnumVARIANT. This property is not visible to users, but allows native languages to iterate through the **PulseDefns** using the ForEach mechanism.
item (index VARIANT) is of type **PulseDefn**.
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., "(1,0.5,1),(30us,2ms,1)"].

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.

An individual pulse definition is retrieved using the **item** property.

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.

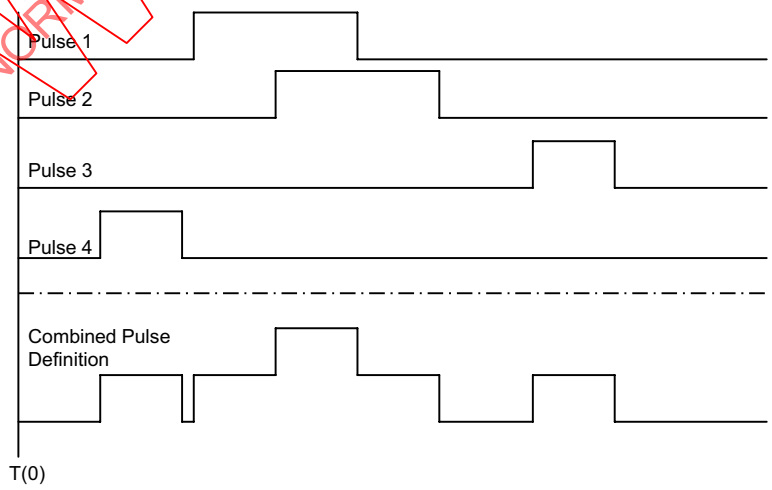


Figure B.3—Pulses using PulseDefns

B.5.1 PulseDefn class

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

Properties

value is of type String (i.e., a string containing the pulse, e.g., “30us, 2ms,1”).
start <Physical> is the point where the pulse occurs. (default 0s)
pulseWidth <Physical> is the duration of the pulse. (default 0s)
levelFactor <Ratio> is the multiplication factor that the pulse applies to an input signal. (default 1.0)
name is of type String and is the name of the pulse.

B.6 SignalFunction class

All BSCs originate from classes derived from the **SignalFunction** base class (see Table B.1 in 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 [**(at=0)**] 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**.

The **Out**, **In**, and **Conn** properties may all contain multiple signals. The parameter at can be used to identify which signal channel is being referenced, e.g., **In**(1). The default parameter value 0 implies all signal channels.

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**.

Subclauses B.6.1 through B.6.6 describe all of the BSCs available to the standard covering **Source**s, **Conditioner**s, **Sensor**s, **EventFunction**s, **Control**, **Digital,** and **Connection**s. 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 alternatives are not exhaustive, but rather indicative of the common BSCs. The BSC can describe any signal based on any physical type listed in Table B.4 (in B.4.1). 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 are as described in B.3 unless explicitly changed in the following text. In general, this behavior is consistent across all BSCs and ensures that BSCs do not invent special meaning for these terms.

B.6.1 Source ::SignalFunction

a) *Definition*—**Source**s 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 **Source**s 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: Voltage|| Current|| Power> ::NonPeriodic

a) *Definition*—A Constant signal retains its given level. See Figure B.4.

b) *Attributes*

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

c) *Description*

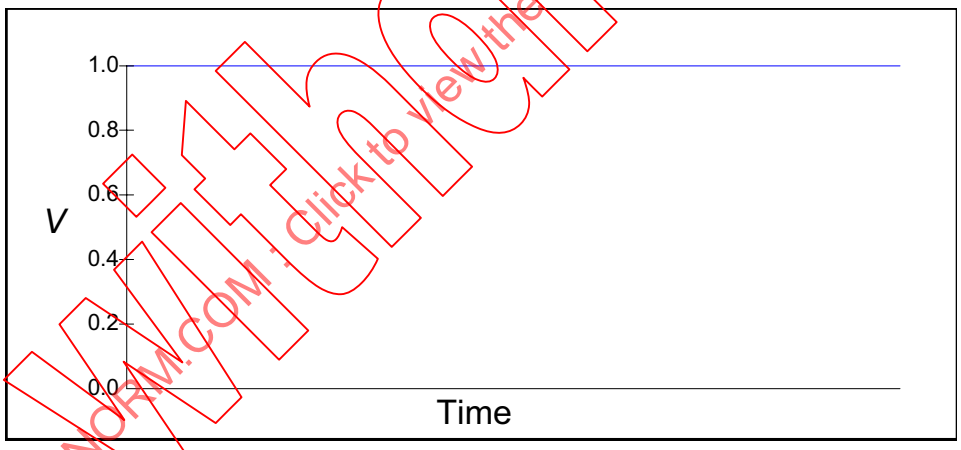


Figure B.4—Constant (amplitude = 1 V)

B.6.1.1.2 Step<type: Voltage|| Current|| Power> ::NonPeriodic

a) *Definition*—A Step signal makes a transition from zero to a given level. See Figure B.5.

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.

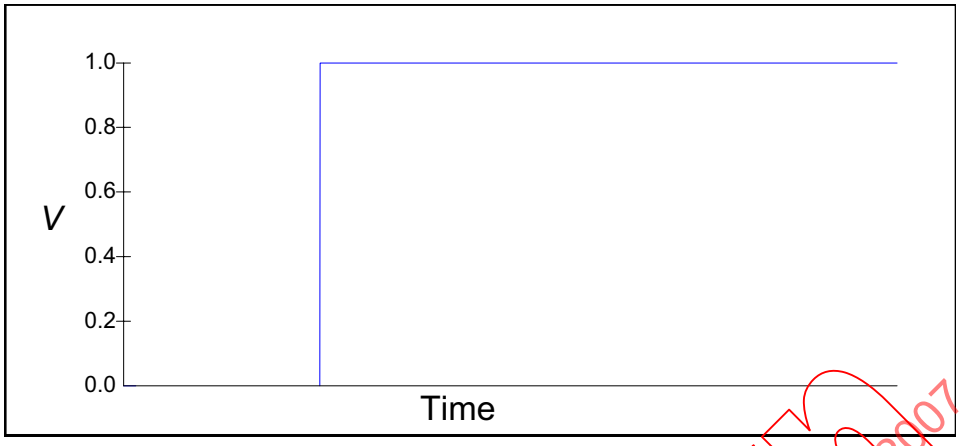


Figure B.5—Step (amplitude = 1 V, startTime = 0.5 s)

B.6.1.1.3 SingleTrapezoid<type: Voltage|| Current|| Power> ::NonPeriodic

a) *Definition*—A SingleTrapezoid is a NonPeriodic signal defined by the geometric trapezoid shape. See Figure B.6.

b) *Attributes*

amplitude <Physical>—value of pulse amplitude	(default = 0)
startTime <Time>—time at which trapezoid starts relative to it 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 trapezoid 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.

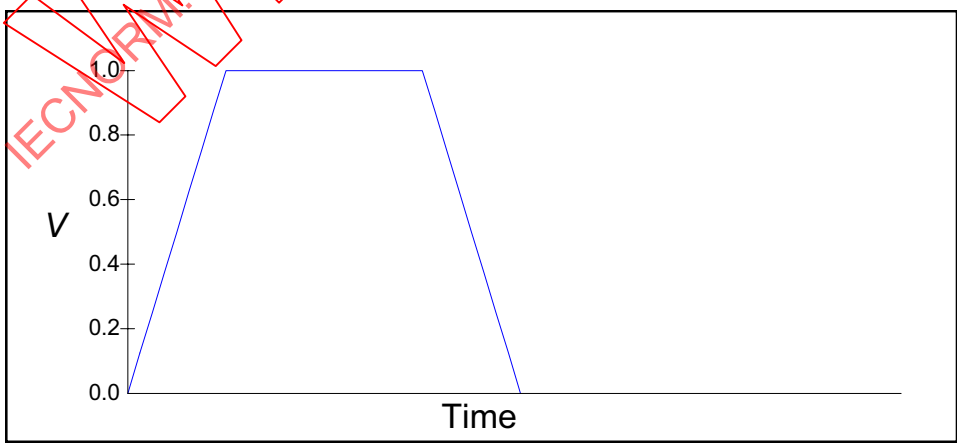


Figure B.6—SingleTrapezoid (amplitude = 1 V)

B.6.1.1.4 Noise<type: Voltage|| Current|| Power> ::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 pseudo-random. Noise occurs across a range of frequencies and can be characterized by amplitude; it may take the form of a **Sensor** or **Source** signal. Pseudo-random 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.7.

b) *Attributes*

amplitude <Physical>—the peak noise amplitude (default = 0)
seed <integer>—used for pseudo-random 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 pseudo-random effect applies only to multiple sequences of the same implementation, and different implementations will give different pseudo-random 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.

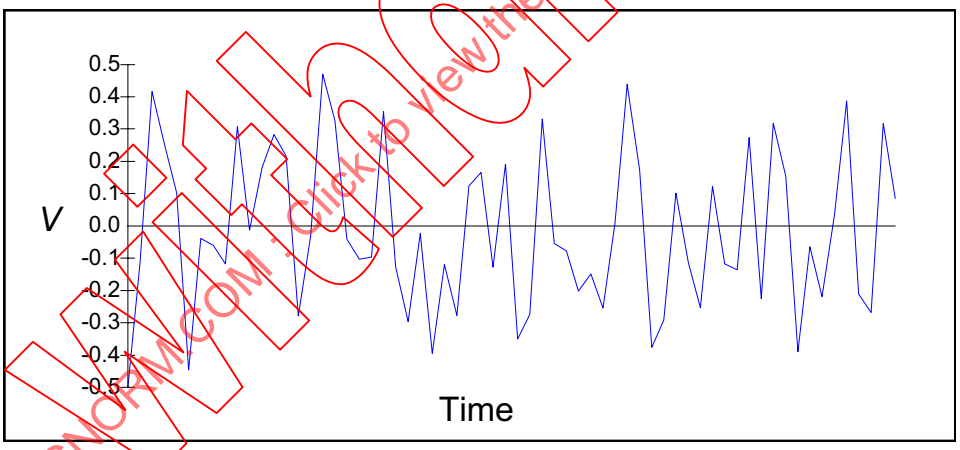


Figure B.7—Noise (seed = 1, amplitude = 0.5 V, frequency = 50 Hz)

B.6.1.1.5 SingleRamp<type: Voltage|| Current|| Power> ::NonPeriodic

a) *Definition*—A SingleRamp represents a linear transition from 0 to the defined amplitude level during a defined time period. See Figure B.8.

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 signal 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.

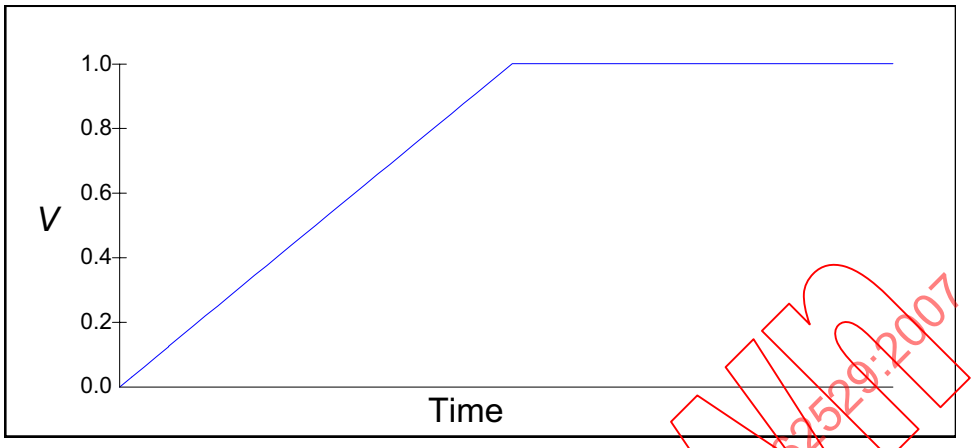


Figure B.8—SingleRamp (amplitude = 1 V, riseTime = 1 s)

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: Voltage|| Current|| Power> ::Periodic

a) *Definition*—A Sinusoid is a signal where the amplitude of the dependent variable is given by the following formula:

$$a = A\sin(\omega t+\varphi)$$

where

A is the amplitude;

ω is $2\pi \times$ frequency;

φ 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.9.

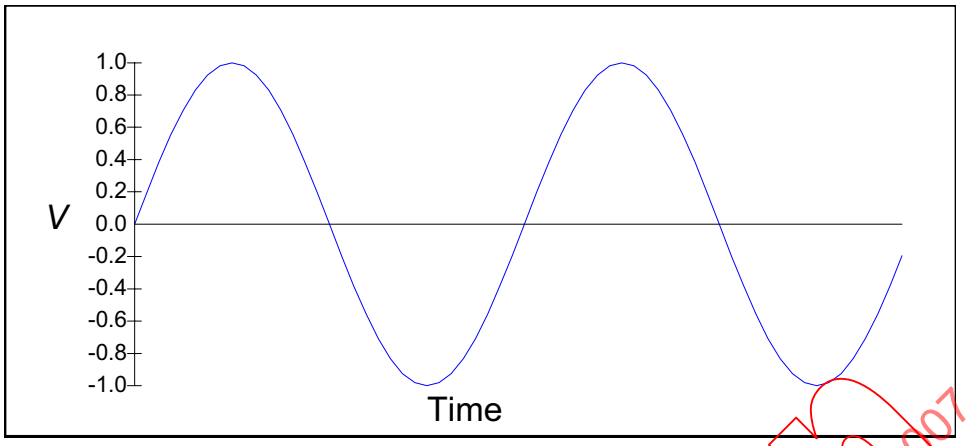


Figure B.9—Sinusoid (amplitude = 1 V, frequency = 30 Hz)

B.6.1.2.2 Trapezoid<type: Voltage|| Current|| Power> ::Periodic

a) *Definition*—A Trapezoid signal 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.10.

b) *Attributes*

amplitude <Physical>—value of pulse amplitude	(default = 0)
period <Time>—time in which the signal repeats itself	(default = 1 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 Trapezoid signal represents the trapezoidal geometric shape. The continuous signal always starts on the rising edge, and the trapezoid is repeated every period even if the trapezoid has not been completed.

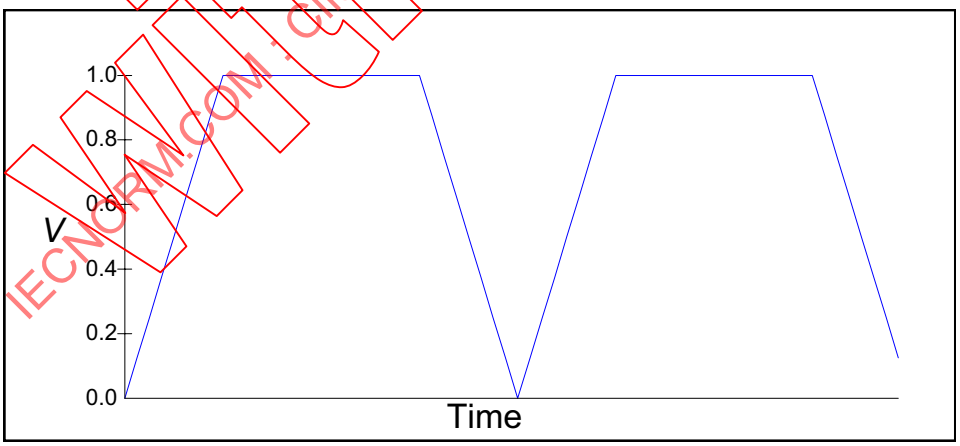


Figure B.10—Trapezoid (amplitude = 1 V, pulseWidth = 0.5 s)

B.6.1.2.3 Ramp<type: Voltage|| Current|| Power> ::Periodic

a) *Definition*—A Ramp signal is a Periodic signal whose instantaneous value varies alternately and linearly between two specified values. Its parameters are defined by its amplitude, the time to rise from 0, and the period. See Figure B.11.

b) *Attributes*

amplitude <Physical>—final level of the signal (default = 0)
period <Time>—time in which signal repeats itself (default = 1 s)
riseTime <Time>—rise time of Ramp signal (default = 1 s)

c) *Description*

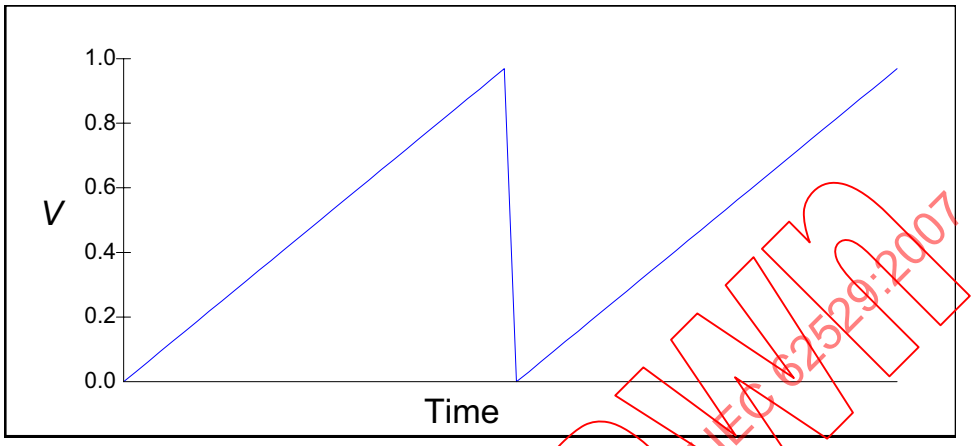


Figure B.11—Ramp (amplitude = 1 V)

B.6.1.2.4 Triangle<type: Voltage|| Current|| Power> ::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.12.

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 signal 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.

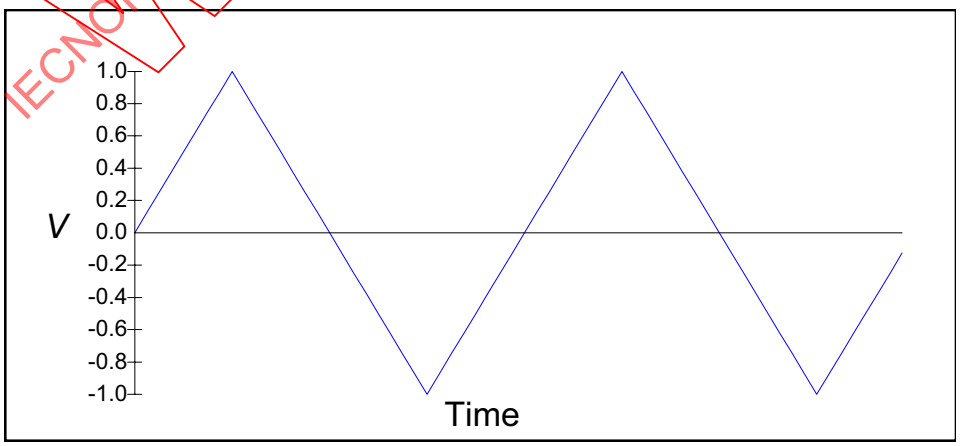


Figure B.12—Triangle (amplitude = 1 V)

B.6.1.2.5 SquareWave<type: Voltage|| Current|| Power> ::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.13.

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.

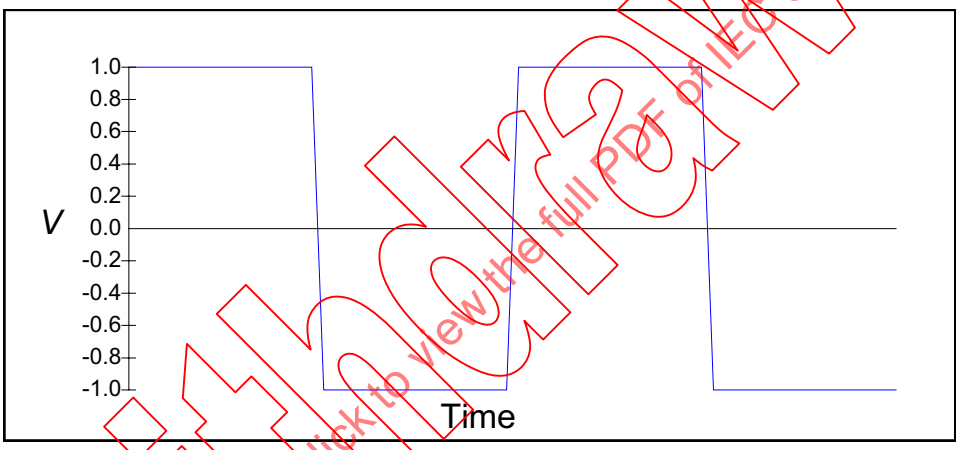


Figure B.13—SquareWave (amplitude = 1 V)

B.6.1.2.6 WaveformRamp<type: Voltage|| Current|| Power> ::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.14.

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 <array of real>—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 non-zero 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.

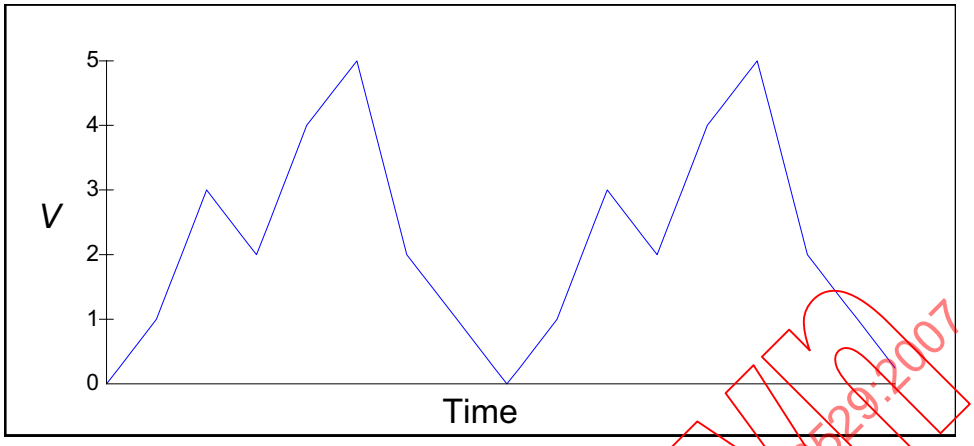


Figure B.14—WaveformRamp (points = [0,1,3,2,4,5,2,1], period = 1 s)

B.6.1.2.7 WaveformStep<type: Voltage|| Current|| Power> ::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.15.

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 <array of real>—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 non-zero 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.

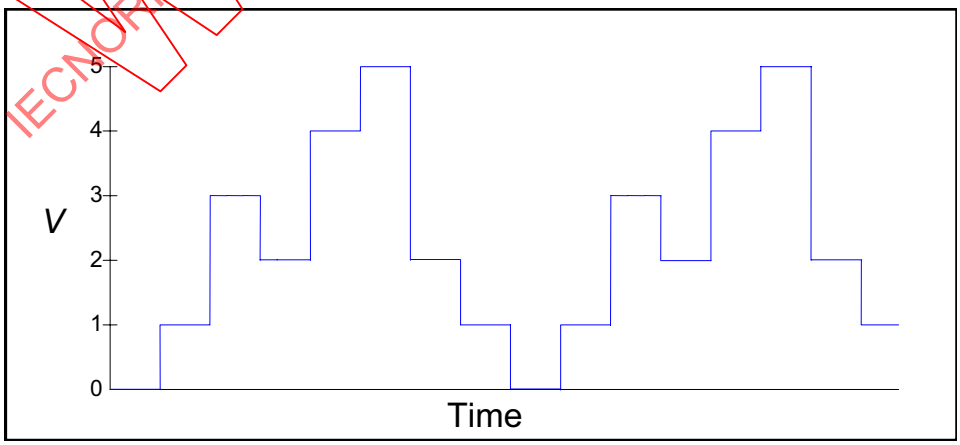


Figure B.15—WaveformStep (points = [0,1,3,2,4,5,2,1], period = 1 s)

B.6.2 Conditioner ::SignalFunction

a) *Definition*—**Conditioner**s 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*—**Conditioner**s act on signals, e.g., **Source**s, but not on events (e.g., **EventFunctions**). **Conditioner**s 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 are defined as pure with instantaneous frequency cutoff across their bandwidths. Practically tests may use the **errlmt** property of the frequency to define the minimum frequency rolloff required.

B.6.2.1.1 BandPass ::Filter

a) *Definition*—A BandPass Filter passes a set of frequencies from an input signal (with equal gain across all its frequencies) and filters out all frequencies outside of the band. See Figure B.16.

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)

c) *Description*—The output (e_{out}) is given as follows:

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

and

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

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.

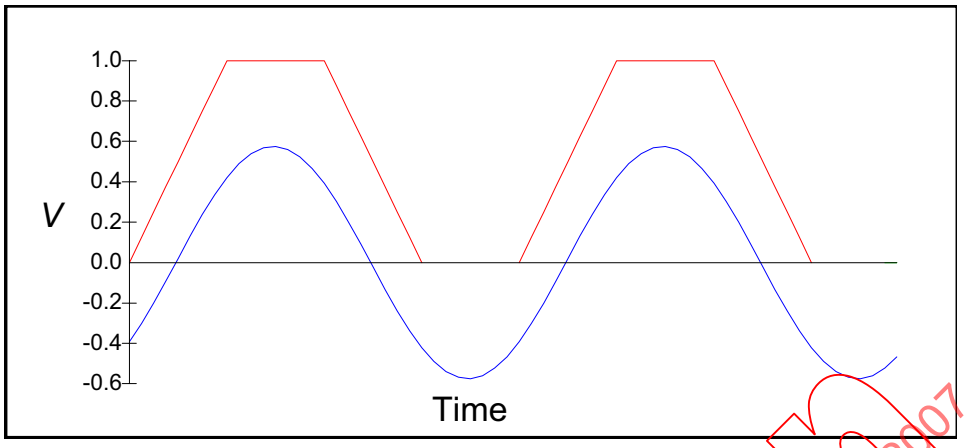


Figure B.16—BandPass (response of Filter to TrapezoidVoltage)

B.6.2.1.2 LowPass ::Filter

a) *Definition*—The LowPass Filter suppresses all frequencies above the cutoff frequency. All frequencies up to and including the cutoff frequency are passed (with equal gain) to the output signal. The LowPass Filter represents a perfect step filter. See Figure B.17.

b) *Attributes*

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

c) *Description*—The output (e_{out}) is given as follows:

$e_{out} = e_{in}$ for $f \leq f_c$

and

$e_{out} = 0$ for $f > f_c$

where

e_{in} is the input;

f is the frequency of the input signal;

f_c is the absolute value of the cutoff frequency.

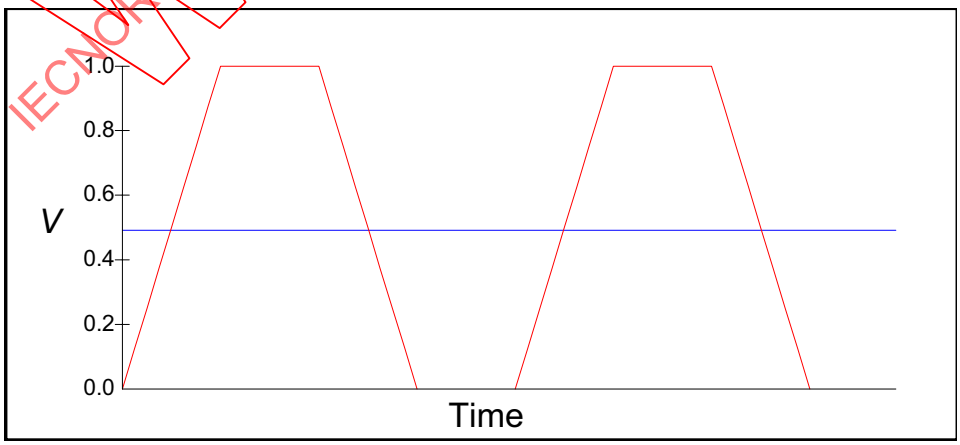


Figure B.17—LowPass (response of Filter to TrapezoidVoltage)

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 represents a perfect step filter. See Figure B.18.

b) *Attributes*

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

c) *Description*—The output (e_{out}) is given as follows:

$$e_{out} = e_{in} \quad \text{for } f > f_c$$

and

$$e_{out} = 0 \quad \text{for } f \leq f_c$$

where

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

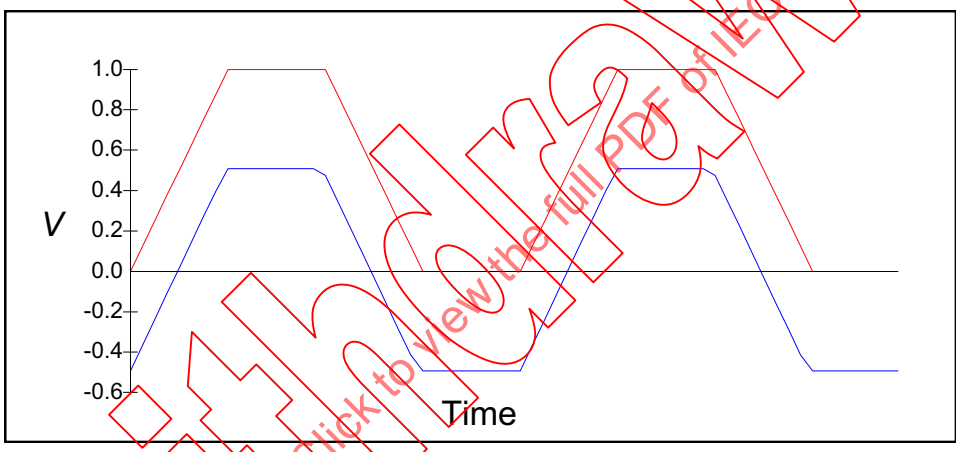


Figure B.18—HighPass (response of Filter to TrapezoidVoltage)

B.6.2.1.4 Notch ::Filter

a) *Definition*—A Notch Filter blocks a set of frequencies from an input signal and passes (with equal gain across all its frequencies) all frequencies outside of the band. See Figure B.19.

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)

c) *Description*—The output (e_{out}) is given as follows:

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

and

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

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.

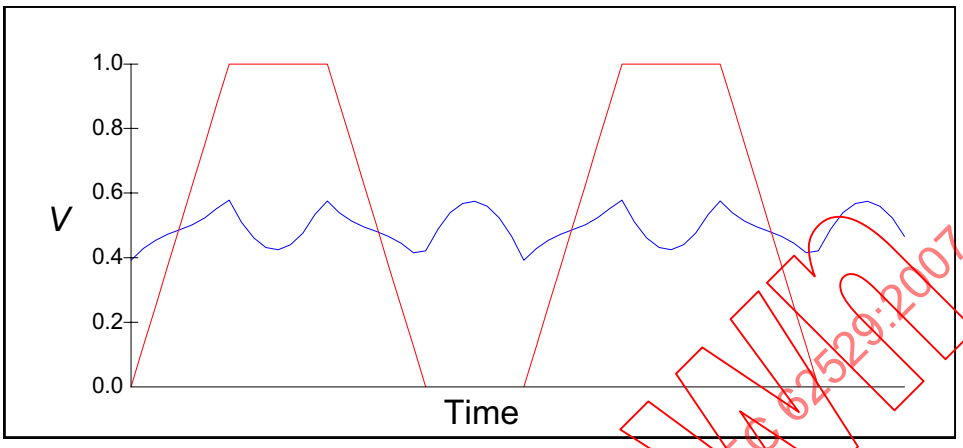


Figure B.19—Notch (response of Filter to TrapezoidVoltage)

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.20 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.

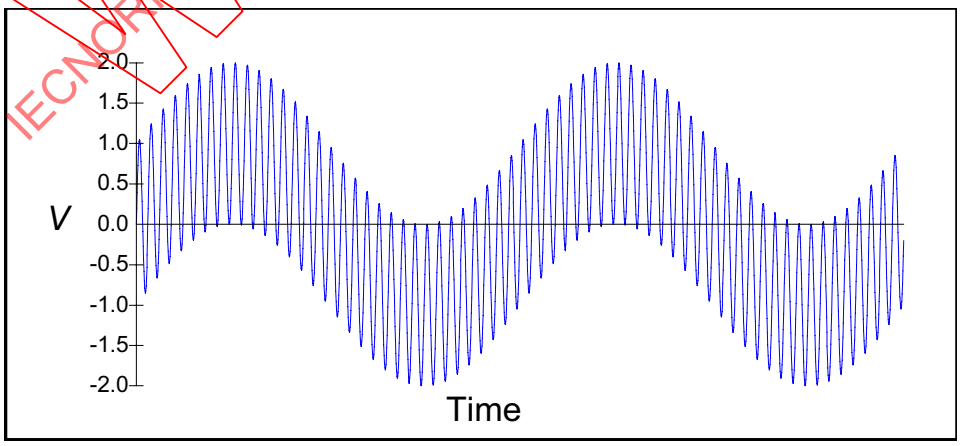


Figure B.20—Sum

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.21 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).

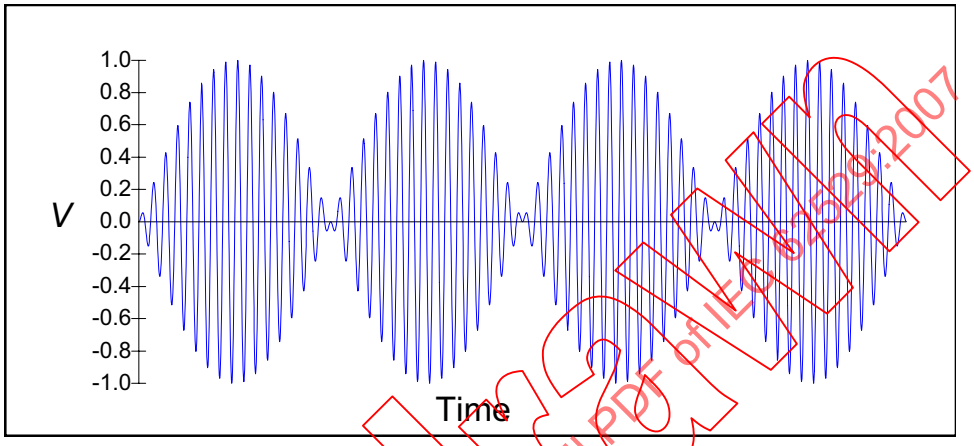


Figure B.21—Product

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.22 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.

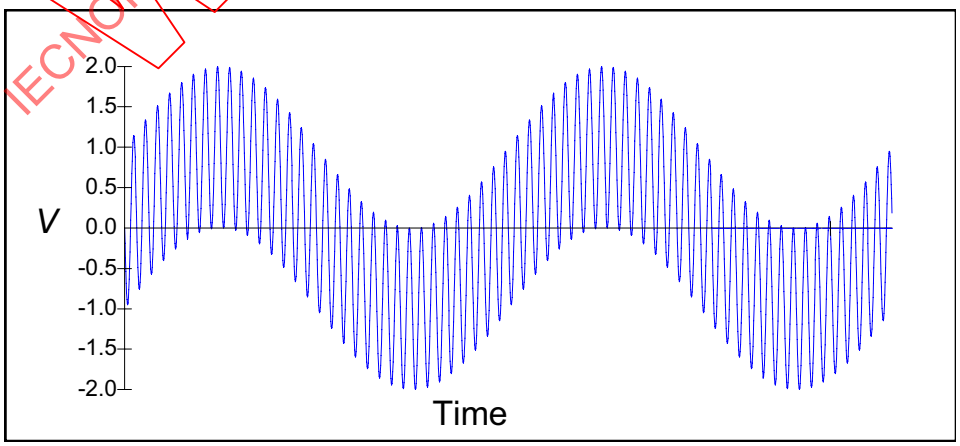


Figure B.22—Diff

B.6.2.3 Modulator ::Conditioner

a) *Definition*—Modulator provides facilities to create a modulated signal where the modulation is proportional to the input signal.

b) *Attributes*—Not applicable

c) *Description*

B.6.2.3.1 FM<type: Voltage|| Power> ::Modulator

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

b) *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)

c) *Description*—The instantaneous frequency of a signal is defined as rate of change of φ ($d\varphi/dt$). For FM, the general solution is given as follows:

$$e = E_c \sin(d\varphi/dt)$$

where

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

where the general solution for $d\varphi/dt = f_c +$ frequencyDeviation $\times m(t)$ is given as follows:

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

and

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

As an example, the output for a FM-modulated cosine waveform is given by the following equation:

$$e = E_c \sin(\omega_c t + m_f \sin \omega_m t)$$

where

ω_c is $2\pi f_c$;
ω_m is $2\pi \times$ modulating frequency;
m_f deviation ratio ($\equiv$ modulation index).

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

Figure B.23 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.

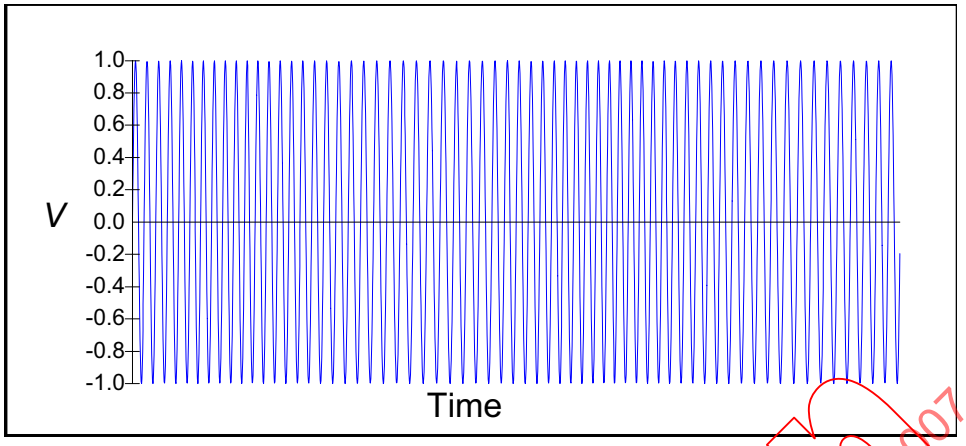


Figure B.23—FM

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 equation for AM signal is given as follows:

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

where

E_c is the carrier amplitude (unmodulated);

$m(t)$ is the modulating signal;

ω_c is 2π × carrier frequency.

As an example, the output for an AM-modulated sinusoid signal is given by the following equation:

$$e = E_c(1 + m_a \sin\omega_m t)\sin\omega_c t$$

where

E_c is the carrier amplitude (unmodulated);

m_a is the depth of modulation (≡ modulation index);

ω_m is 2π × modulating frequency;

ω_c is 2π × carrier frequency.

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

Figure B.24 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.

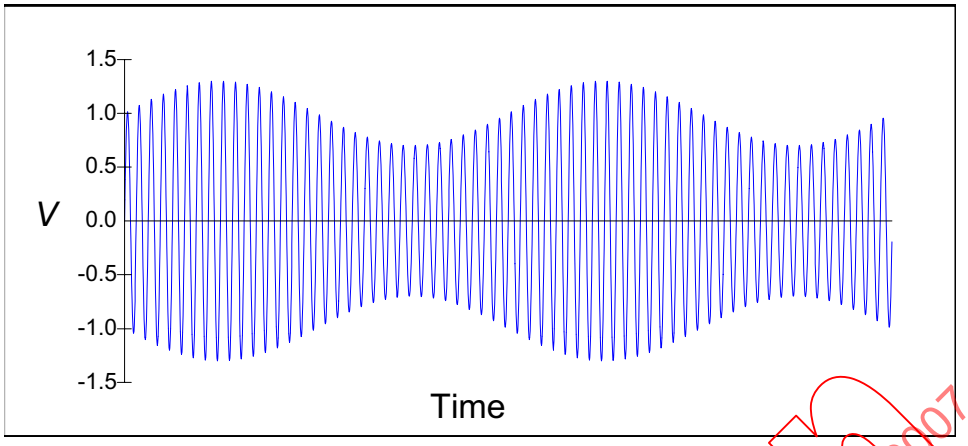


Figure B.24—AM

B.6.2.3.3 PM<type: Voltage|| Power> ::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 = π/4 rad)

c) *Description*—The equation for PM signal is given as follows:

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

where

E_c is the carrier amplitude (unmodulated);
ω_m is 2π × modulating frequency;
$m(t)$ is the modulating signal.

As an example, the output PM-modulated cosine signal is given by the following equation:

$$e = E_c \sin(\omega_c t + \varphi_d \cos \omega_m t)$$

where

E_c is the carrier amplitude (unmodulated);
ω_c is 2π × carrier frequency;
φ_d is phase deviation (≡ modulation index);
ω_m is 2π × 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.25 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.

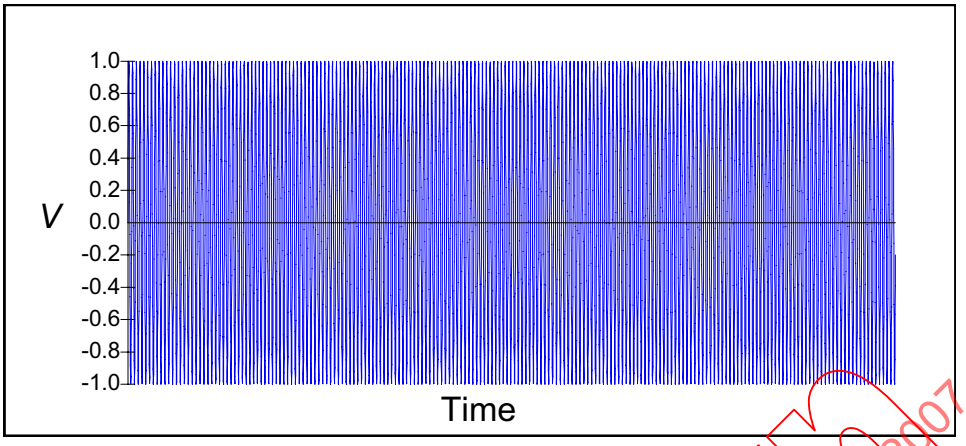


Figure B.25—PM

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* will alter over time (default = 0 s^{-1})
delay <Time>—the fixed delay that signal will be delayed (default = 0 s)
rate <Ratio>—the rate at which the delay will alter over time (default = 0)

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

— Delay signals (*delay*)
— 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 follows:

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

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

A signal delay time that is negative refers to events that will 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), m/(m/s) = s.

— Rate is the velocity at which the target is moving away from the observer, (m/s)/(m/s) = dimensionless.

— Acceleration is the rate at which the velocity of the target (from the observer) is changing, $(m/s^2)/(m/s) = s^{-1}$.

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.26 shows the effect of a SignalDelay on a WaveformRamp. In this example, the delay is 0.1 s, the acceleration is –0.1 s^{-1}, and the rate is –0.1.

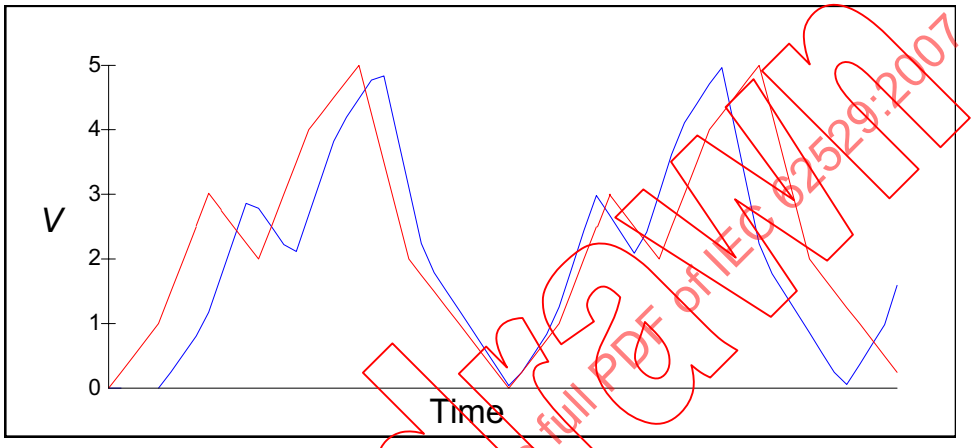


Figure B.26—SignalDelay

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.27.

b) *Attributes*

dampingFactor <real>—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. Exponential is determined by the damping factor, using the expression $e^{-t/\tau}$, where τ is equal to the damping factor.

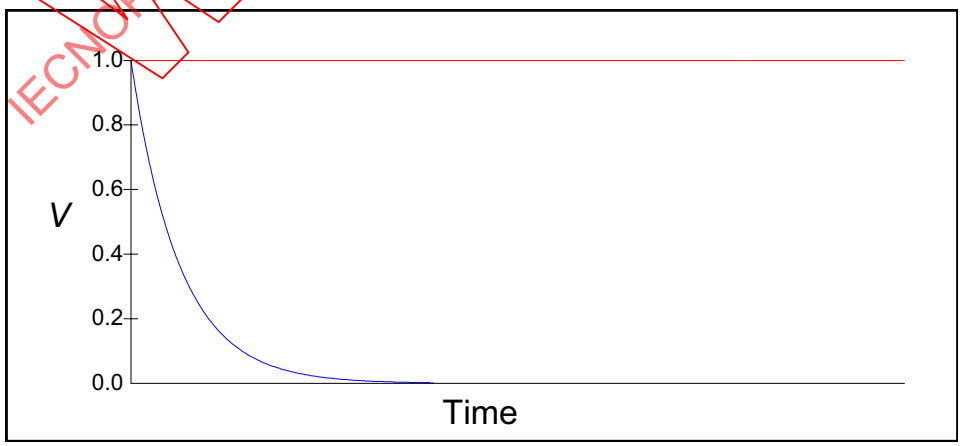


Figure B.27—Exponential (constant amplitude = 1 V)

B.6.2.4.3 E ::Transformation

a) *Definition*—E is an exponential operation on a signal. See Figure B.28.

b) *Attributes*—Not applicable

c) *Description*—The output of the signal may be expressed by the following equation:

$$y = e^{x}$$

where

y is the signal output;
x is the signal input.

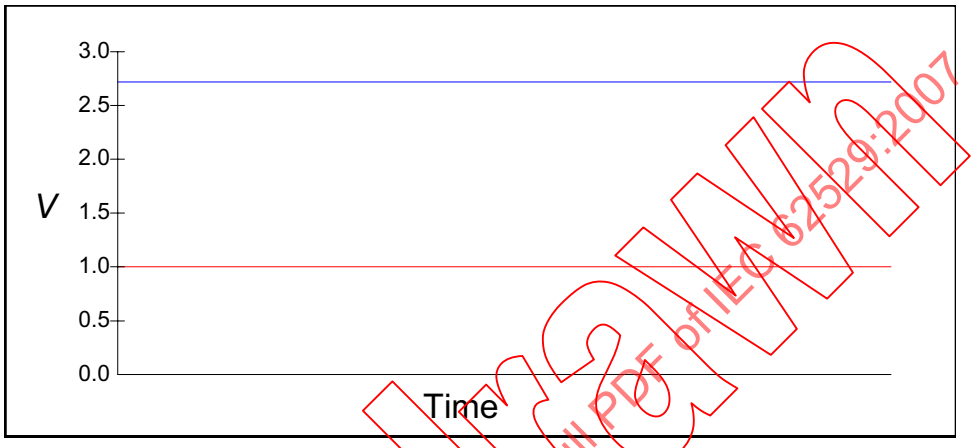


Figure B.28—E (constant amplitude = 1 V)

B.6.2.4.4 Negate ::Transformation

a) *Definition*—Negate modifies a signal so that its amplitude is the negative of the **In** signal amplitude. See Figure B.29.

b) *Attributes*—Not applicable

c) *Description*—The output of the signal may be expressed by the following equation:

$$y = -x$$

where

y is the signal output;
x is the signal input.

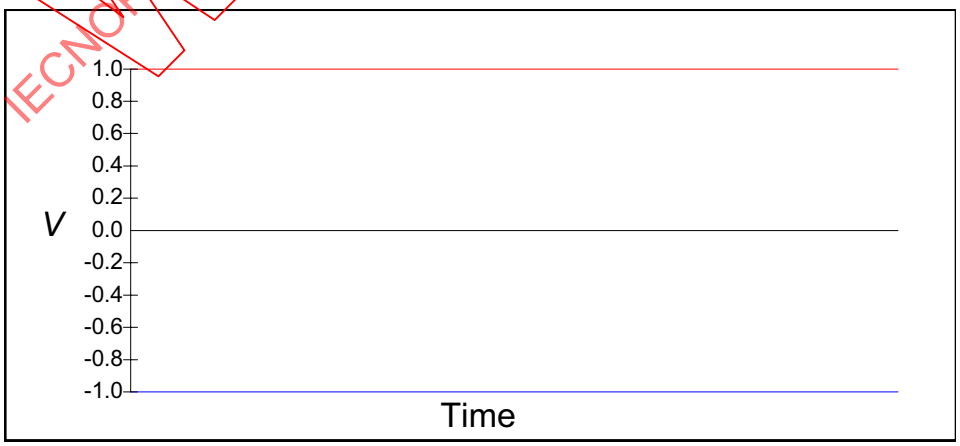


Figure B.29—Negate (constant amplitude = 1 V)

B.6.2.4.5 Inverse ::Transformation

a) *Definition*—Inverse is the mathematical reciprocal of a signal. See Figure B.30.

b) *Attributes*—Not applicable

c) *Description*—The output of the signal may be expressed by the following equation:

$$y = 1/x$$

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.

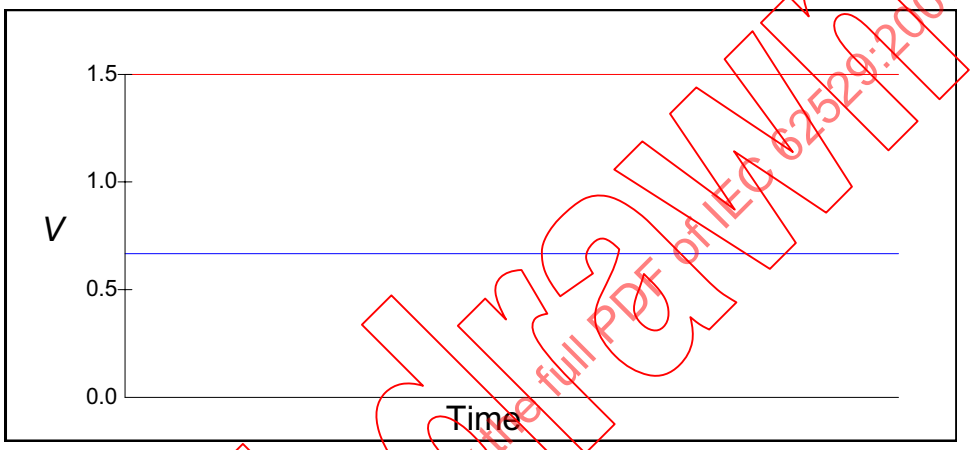


Figure B.30—Inverse (constant amplitude = 1 V)

B.6.2.4.6 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 <integer>—the number of times the list of pulses is output; zero indicates that the sequence is repeated indefinitely (default = 0)

c) *Description*

Figure B.31 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 [(0.2,0.5,0.5),(0.4,0.3,0.5)]. The default repetition value of 0 is assumed.

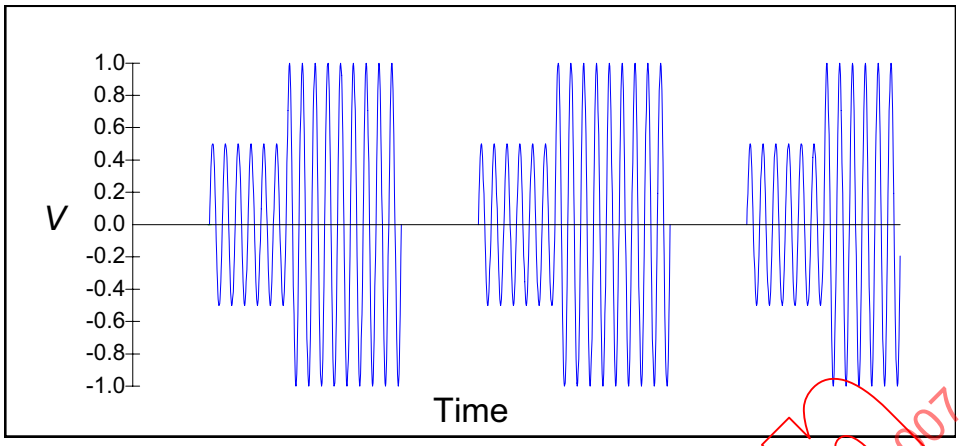


Figure B.31—PulseTrain

B.6.2.4.7 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.32.

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 the following equation:

$$y = mx$$

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 will also be negated, and an ac signal will acquire a 180° phase shift.

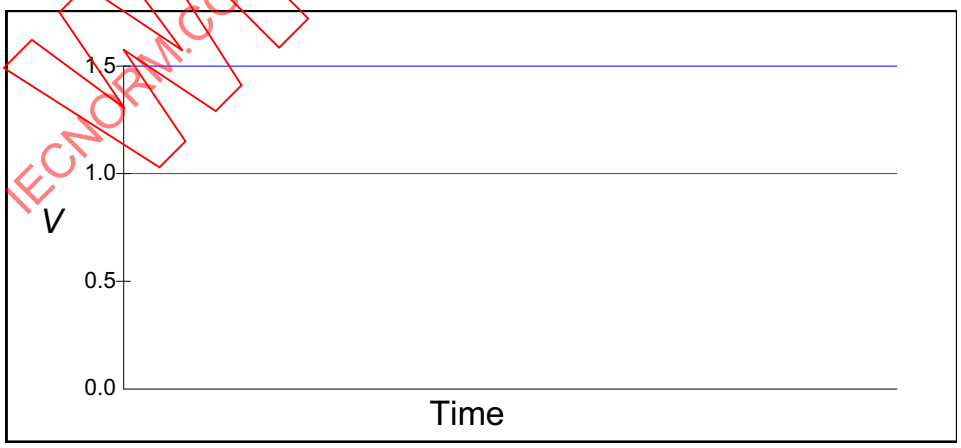


Figure B.32—Attenuator (gain = 1.5, constant amplitude = 1 V)

B.6.2.4.8 Load ::Transformation

a) *Definition*—Load provides an impedance to load a signal. See Figure B.33.

b) *Attributes*

resistance <Resistance>—the impedance (in ohms) of the resistive part of the load(default = 0 Ω)
reactance <Resistance>—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.

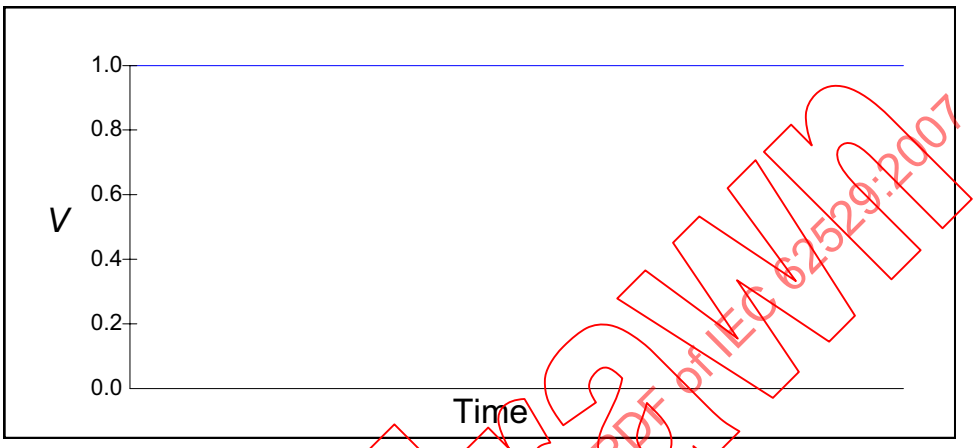


Figure B.33—Load (resistance = 50 Ω, constant amplitude = 1 V)

B.6.2.4.9 Limit<type: Voltage|| Current|| Power> ::Transformation

a) *Definition*—Limit restricts the values of the signal to ± 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.34 shows the effect of a Limit of 0.90 V on a Sinusoid of amplitude 1 V and frequency 30 Hz.

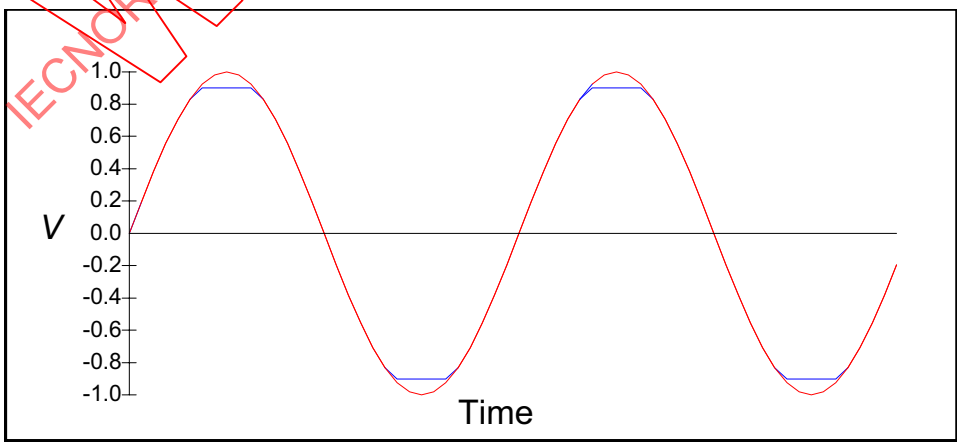


Figure B.34—Limit

B.6.2.4.10 FFT ::Transformation

a) *Definition*—FFT (i.e., Fourier transform) converts time domain signals to frequency domain signals. 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 will be sampled, and the signal to be converted.

b) *Attributes*

samples <Integer>—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 is always the next power of 2.

FFT converts 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/interval, and the axis defined from 0 Hz to the Nyquest frequency defined by half of the sampling frequency (samples/(2*interval)).

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

Figure B.35 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.

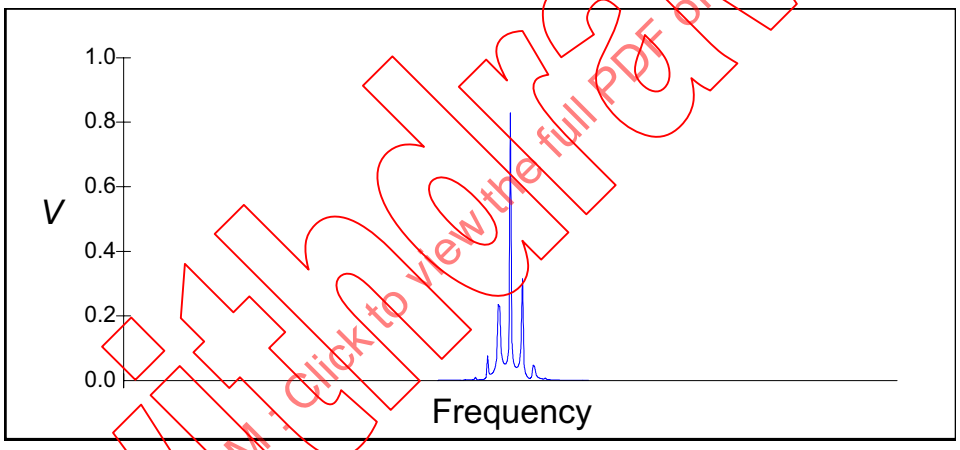


Figure B.35—FFT of PM signal

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, **EventFunction**s 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.36.

b) *Attributes*

clockRate <Frequency>—frequency of the Clock (default = 1 Hz)

c) Description

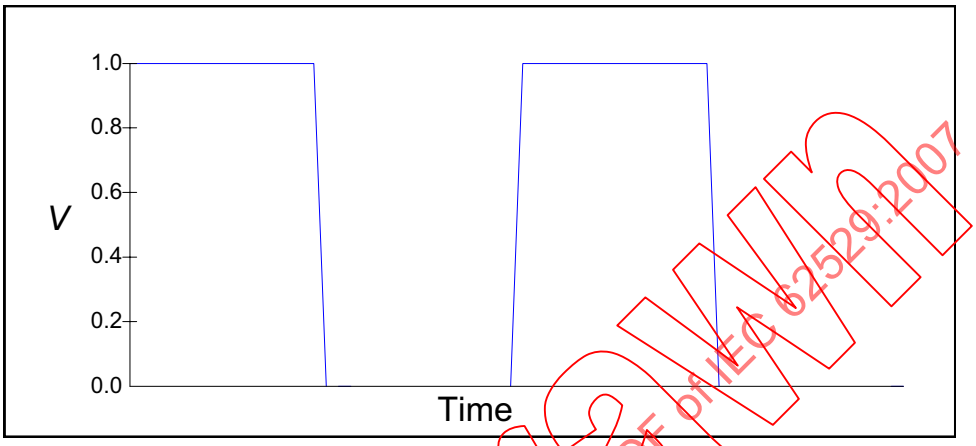


Figure B.36—Clock (clockRate = 20 Hz)

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 (Every), the **Out** event is signaled active at each interval, but never becomes paused. See Figure B.37.

b) *Attributes*

delay <Time>—the delay time before the first event will be 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 <integer>—the number of events to be output; zero indicates that events are generated indefinitely (default = 0)

c) Description

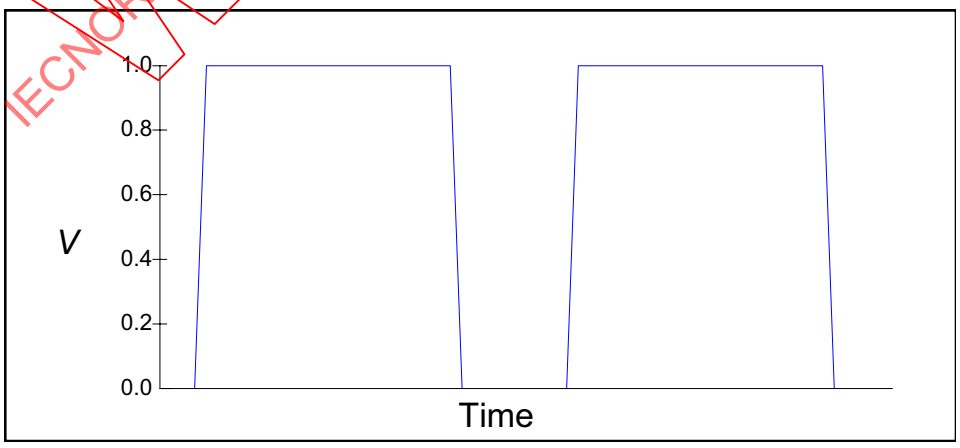


Figure B.37—TimedEvent (delay = 0.1 s, duration = 0.7 s)

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.38.

b) *Attributes*

pulses <PulseDefns>—a list defining the shape of the pulses to be created
repetition <integer>—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.

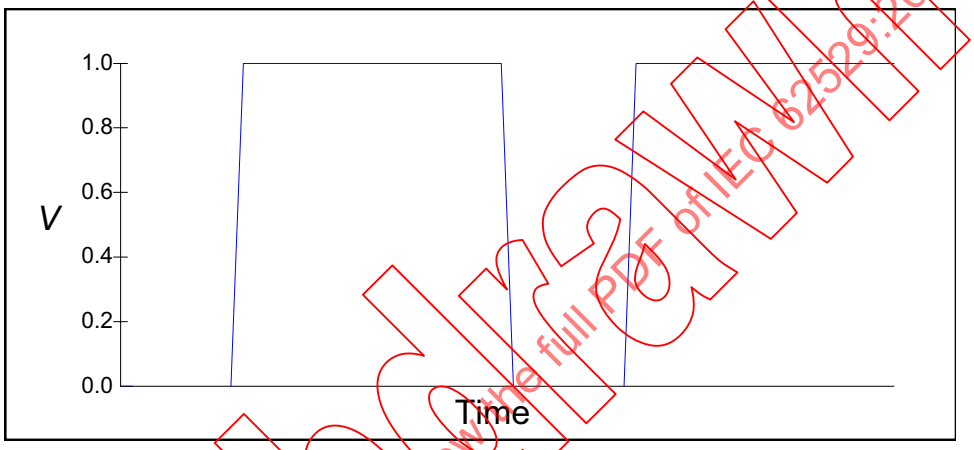


Figure B.38—PulsedEvent (pulses = (0.3,0.7,1))

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 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.39 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.

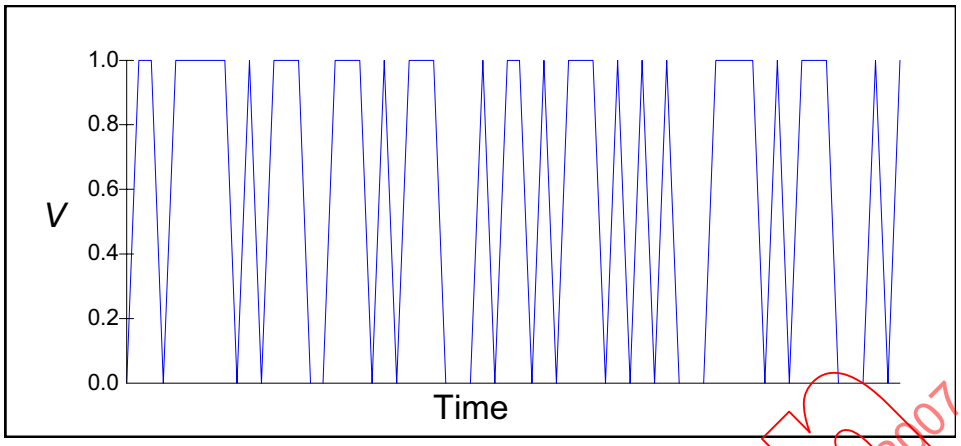


Figure B.39—EventedEvent

B.6.3.2.2 EventCount ::EventConditioner

a) *Definition*—EventCount filters out input events to only allow every **count** input event. See Figure B.40.

b) *Attributes*

count <Integer>—identifies the number of events that must occur before a event is generated (default = 0)

c) *Description*—EventCount counts events and produces an event when **count** events are received. EventCount acts as an event divider in which the divider is dependent on the value of the **count** property.

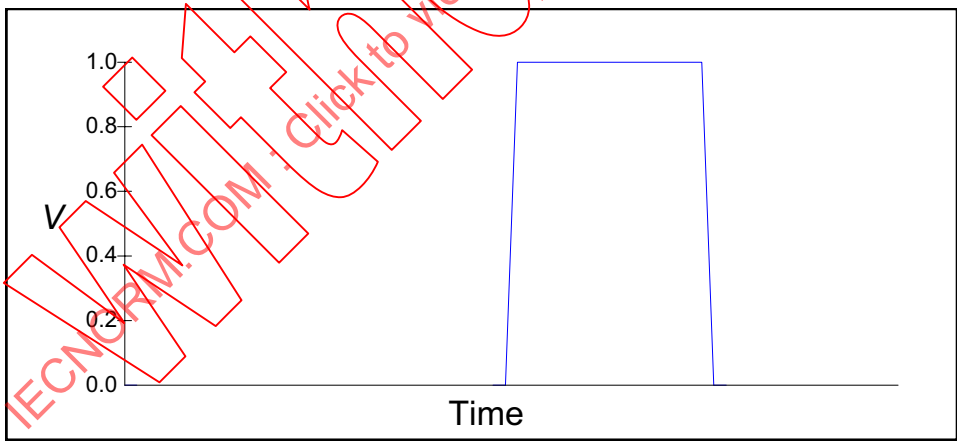


Figure B.40—EventCount (count = 1)

B.6.3.2.3 ProbabilityEvent ::EventConditioner

a) *Definition*—ProbabilityEvent generates an event stream based upon the event stream present at the **In** event. It will replicate the same timing information, but will randomly suppress **In** pulses. Conceptually, 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.41.

b) *Attributes*

seed <integer>—for pseudo-random probabilities (default = 0)

probability <Ratio>—value for comparison with random number (default = 50%)

c) *Description*—ProbabiltyEvent 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.

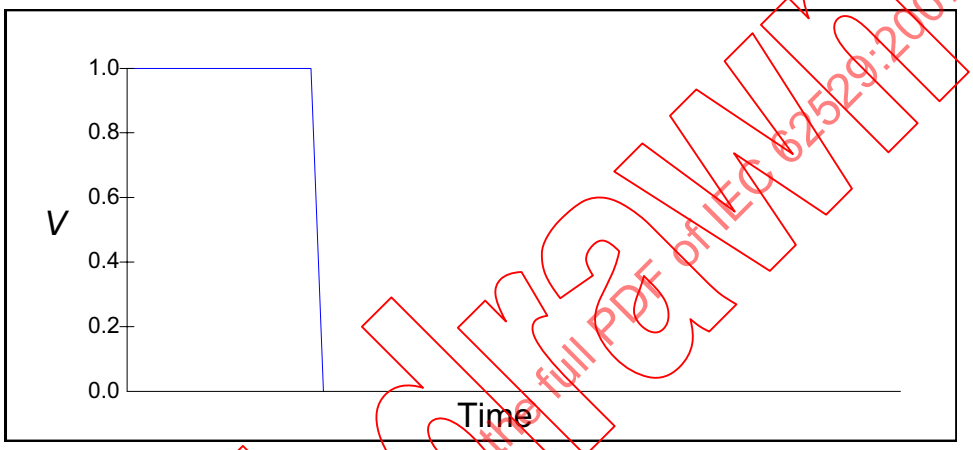


Figure B.41—ProbabilityEvent

B.6.3.2.4 NotEvent ::EventConditioner

a) *Definition*—NotEvent is active when the **In** signal is not active. See Figure B.42.

b) *Attributes*—Not applicable

c) *Description*

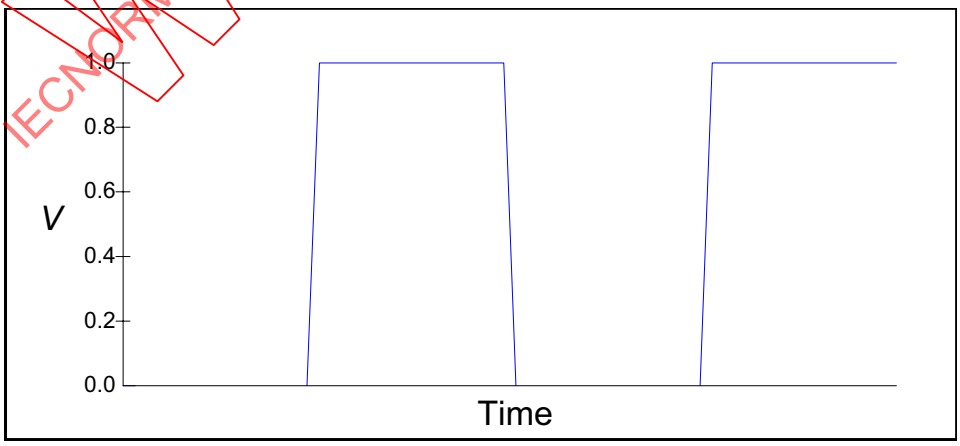


Figure B.42—NotEvent

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.43 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.

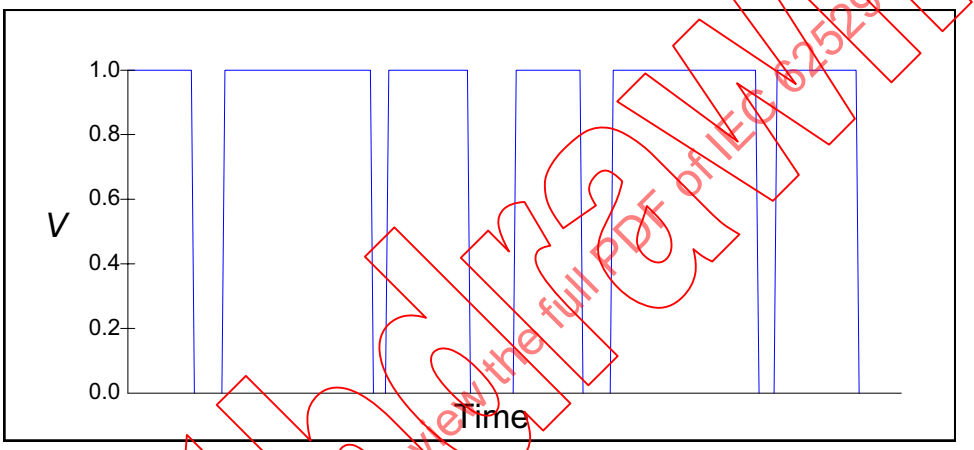


Figure B.43—OrEvent

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.44 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.

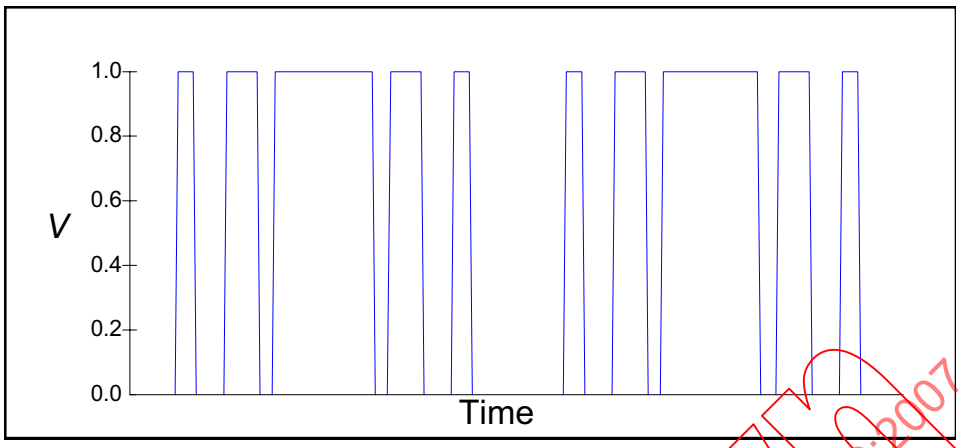


Figure B.44—XOrEvent

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.45 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.

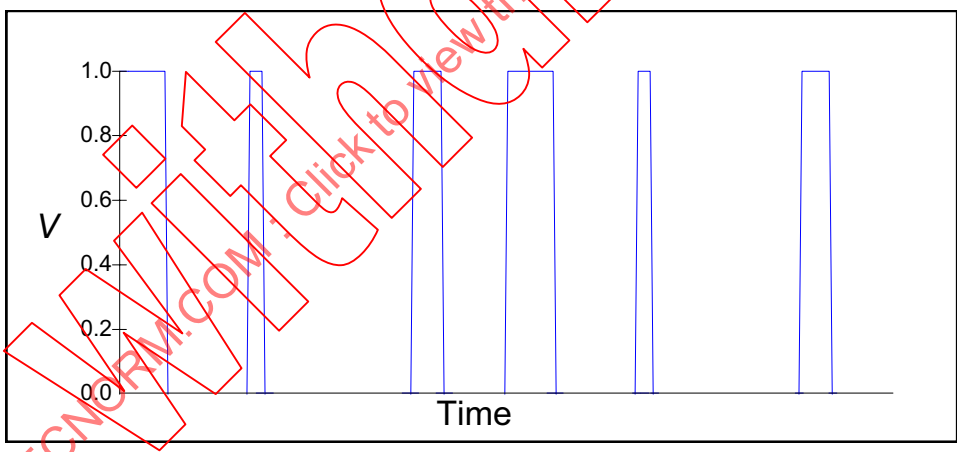


Figure B.45—AndEvent

B.6.4 Sensor ::SignalFunction

a) *Definition*—**Sensor**s allow signals 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 measured

measurements <any attribute type)>—read-only, array of measurements made
samples <Integer>—number of consecutive measurements to be made; zero indicates no measurement is to be taken and indicates the **Sensor** is acting as a monitor only
count < Integer >—read-only number of measurements currently made
gateTime <real>—continuous range of independent variable (Time) over which measurement is made
nominal <Physical>—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
condition <enumCondition>—test made between measurement and nominal value
NOGO <Boolean>—read-only flag indicating last measurement of pass/fail status; if no measurement is taken, GO is false
GO <Boolean>—read-only flag indicating last measurement of pass/fail status; if no measurement is taken, NOGO is false
HI <Boolean>—read-only flag indicating last measurement of high/low status; if no measurement is taken, HI is false
LO <Boolean>—read-only flag indicating last measurement of high/low status; if no measurement is taken, LO is false
UL <Physical>—upper limit value against which condition is checked
LL <Physical>—lower limit value against which condition is checked

c) *Description*—**Sensor**s are used to measure physical characteristics of signals, which can then be read back through measurement values. **Sensor**s are primarily used to take measurements such as root mean square (rms) or average. **Sensor**s may also monitor a signal and create an event signal that can be used by other BSCs. The **Out** signal of a **Sensor** is active while the monitor condition is being met.

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

A **Sensor** can take many measurements. The number of measurements required is the attribute "samples." Zero indicates that the measurement values are never required and the **Sensor** is a monitor only. The number of measurements currently taken is held in the read-only attribute "count."

The attribute gateTime defines the window over which 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** event is provided, the measurement value is evaluated whenever the input signal becomes active. When a **Gate** event is provided, 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 allocated, whenever the **Gate** signal 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 only operational while it is taking measurements. When the number of measurements in the attribute "samples" have been made, the **Sensor** calls **Change** (see Annex C) on the input signal. A monitor where the attribute "samples" is zero never calls **Change** on the input.

B.6.4.1 Counter ::Sensor

a) *Definition*—For all **Sensor**s, 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*

B.6.4.2 TimeInterval ::Sensor

a) *Definition*—TimeInterval measures the time 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 time counter is set to 0 every time the **In** or **Sync** signal becomes active.

If no **Gate** is present (i.e., unassigned), the time interval is the time between consecutive **In** events going active. If the **Gate** event is present (i.e., allocated), the time interval is recorded when the **Gate** event becomes active. The timer is reset when the **In** event goes active.

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

B.6.4.3 Instantaneous<type: Voltage|| Current|| Power|| Frequency|| Resistance|| Capacitance|| Conductance|| Inductance> ::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.

Figure B.46 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.

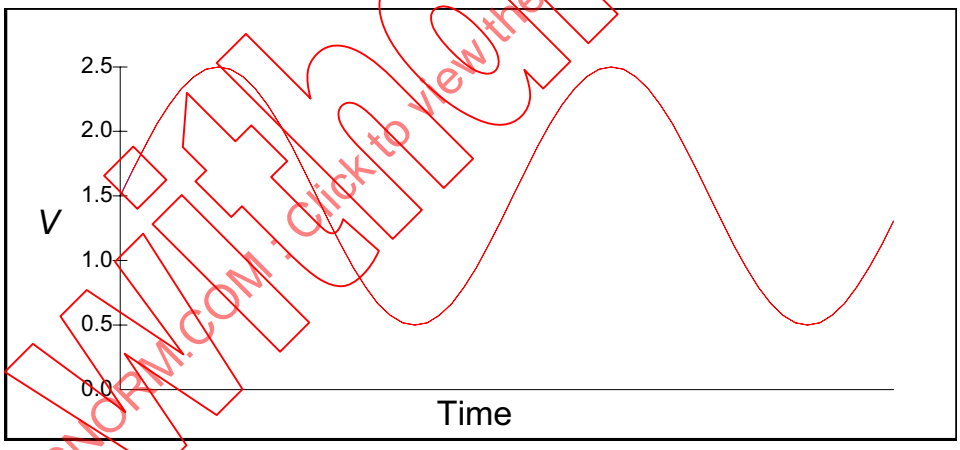


Figure B.46—Instantaneous

B.6.4.4 RMS<type: Voltage|| Current> ::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.

Figure B.47 shows the rms value of the sum of two signals; a sinusoid of amplitude = 1 V with frequency = 1 Hz and a constant of amplitude = 1.5 V.

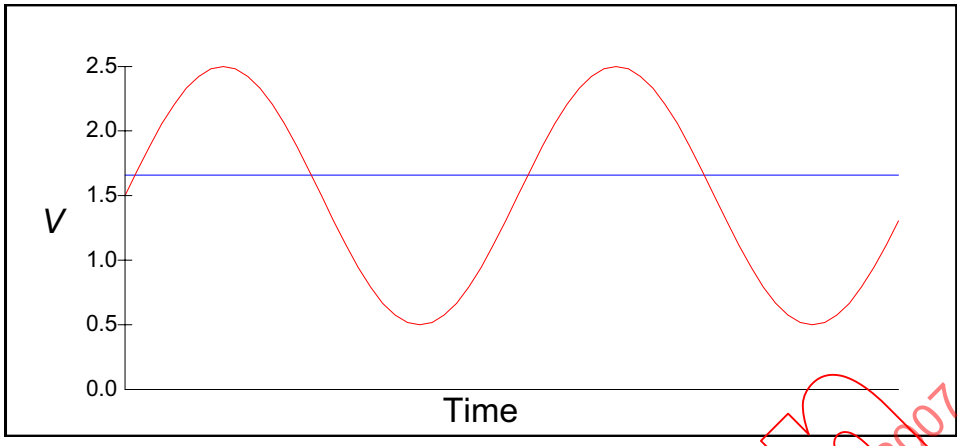


Figure B.47—RMS

B.6.4.5 Average<type: Voltage|| Current|| Power|| Frequency> ::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

Figure B.48 shows the average value of the sum of two signals, a sinusoid of amplitude = 1 V with frequency = 1 Hz and a constant of amplitude = 1.5 V.

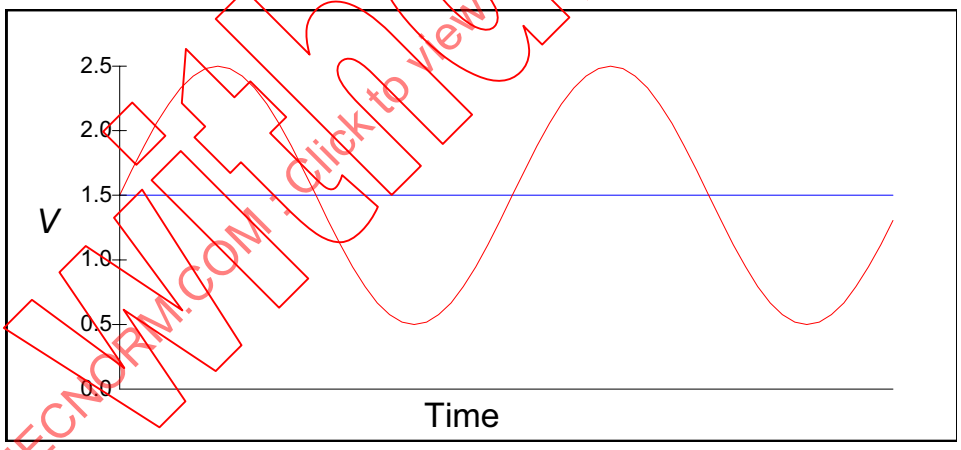


Figure B.48—Average

B.6.4.6 PeakToPeak<type: Voltage|| Current|| Power|| Frequency> ::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*

Figure B.49 shows the peak-to-peak value of the sum of two signals; a sinusoid of amplitude = 1 V with frequency = 1 Hz and a constant of amplitude = 1.5 V.

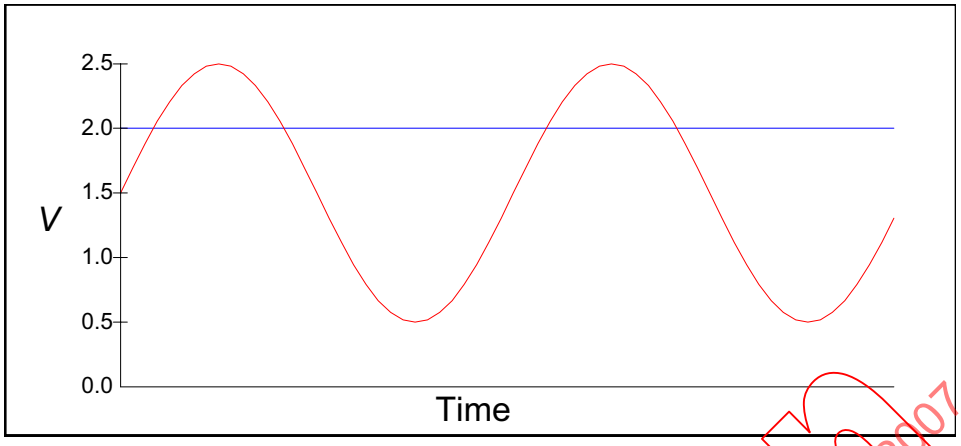


Figure B.49—PeakToPeak

B.6.4.7 Peak<type: Voltage|| Current|| Power|| Frequency> ::Sensor

a) *Definition*—Peak is the value obtained by subtracting the mean value from the maximum measurement of the signal during the gate time.

b) *Attributes*—Not applicable

c) *Description*

Figure B.50 shows the peak value of the sum of two signals; a sinusoid of amplitude = 1 V with frequency = 1 Hz and a constant of amplitude = 1.5 V.

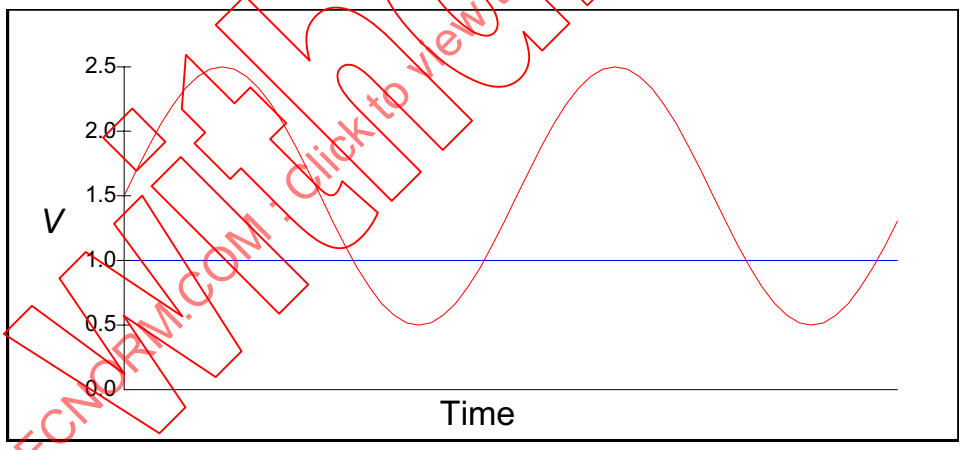


Figure B.50—Peak

B.6.4.8 PeakNeg<type: Voltage|| Current|| Power|| Frequency> ::Sensor

a) *Definition*—PeakNeg measurement is the value obtained by subtracting the mean value from the minimum measurement of the signal during the gate time.

b) *Attributes*—Not applicable

c) *Description*

Figure B.51 shows the peak negative value of the sum of two signals; a sinusoid of amplitude = 1 V with frequency = 1 Hz and a constant of amplitude = 1.5 V.

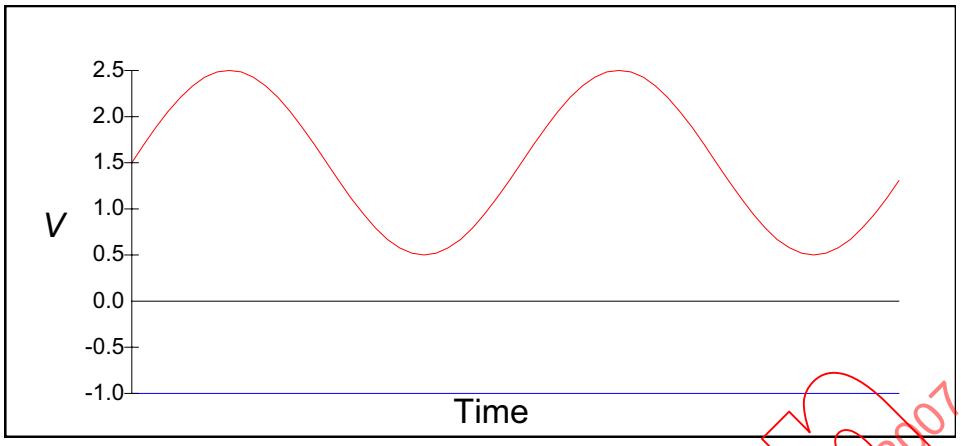


Figure B.51—PeakNeg

B.6.4.9 MaxInstantaneous<type: Voltage|| Current|| Power|| Frequency> ::Sensor

a) *Definition*—MaxInstantaneous measurement is the maximum measurement of the signal during the gate time.

b) *Attributes*—Not applicable

c) *Description*

Figure B.52 shows the maximum instantaneous value of the sum of two signals; a sinusoid of amplitude = 1 V with frequency = 1 Hz and a constant of amplitude = 1.5 V.

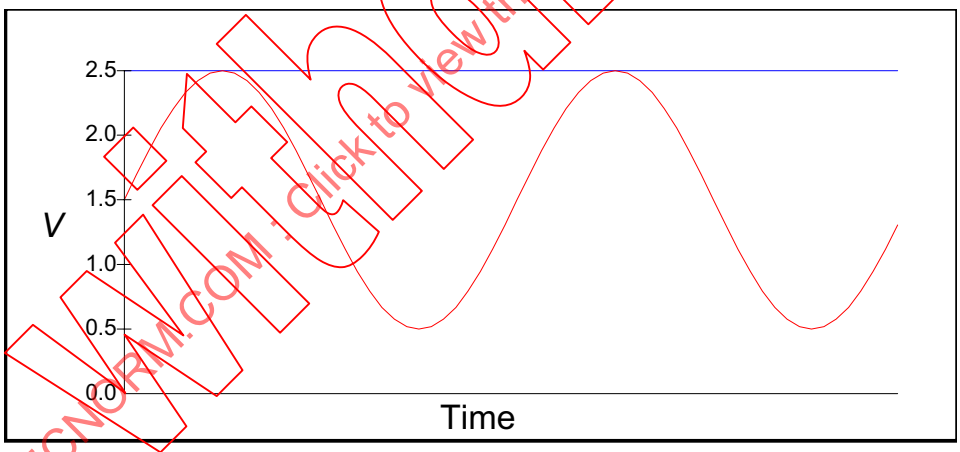


Figure B.52—MaxInstantaneous

B.6.4.10 MinInstantaneous<type: Voltage|| Current|| Power|| Frequency> ::Sensor

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

b) *Attributes*—Not applicable

c) *Description*

Figure B.53 shows the minimum instantaneous value of the sum of two signals; a sinusoid of amplitude = 1 V with frequency = 1 Hz and a constant of amplitude = 1.5 V.

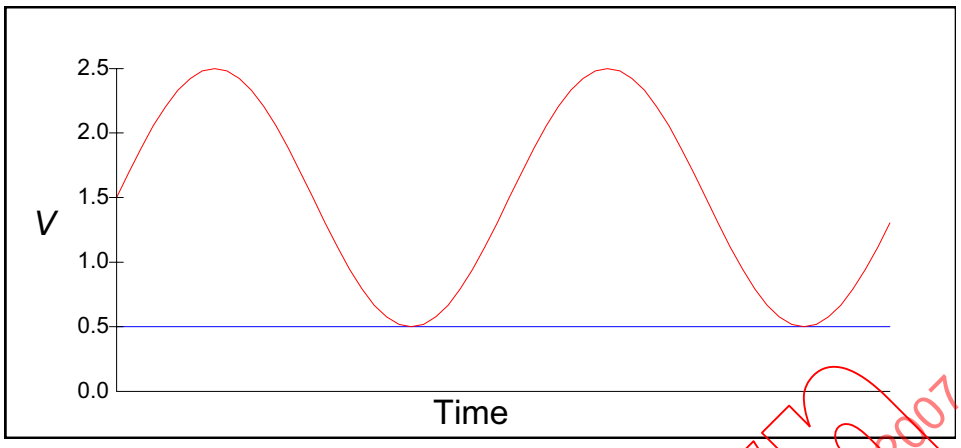


Figure B.53—MinInstantaneous

B.6.4.11 Measure ::Sensor

a) *Definition*—Measure measures any control attribute, as opposed to a capability attribute, of a BSC or TSF.

b) *Attributes*

As <SignalFunction>—reference to a Signal representing signal model of input signal
attribute <SignalFunction Property>—Attribute of the signal that is to be measured

c) *Description*—This subclass provides the ability to measure any control attribute for any TSF or BSC.

The measurement is the value of the attribute that provides the minimum rms value of the difference between the input signal and the signal As.

B.6.5 Digital ::SignalFunction

a) *Definition*—**Digital** is a 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.). This representation, however, is complicated by the necessity to represent aspects of the behavior of digital signals within the electronic devices that operate on them.

b) *Attributes*—Not applicable

c) *Description*—**Digital** may have a number of representations. This standard defines two: digital signal and control signal, respectively. 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.6.5.1 SerialDigital<type: Voltage|| Current> ::Digital

a) *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 in a character string.

b) *Attributes*

data <Character String>—string containing characters H, L, Z, X, which identify digital state
period <Time>—digital clock rate

logic_H_value <Physical>—analog logic high (1)

logic_L_value <Physical>—analog logic low (0)

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

— H logic high (or logic 1)

— L logic low (or logic 0)

— Z high impedance (absence of logic signal)

— X unknown or indeterminate logic level

Figure B.54 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.

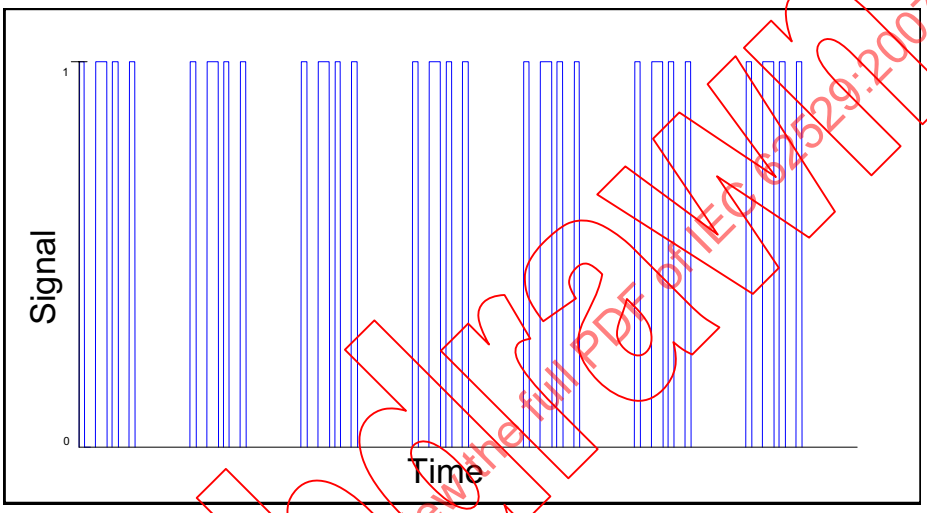


Figure B.54—SerialDigital

B.6.5.2 ParallelDigital<type: Voltage|| Current> ::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 <Array of Character String>—each string String containing characters H, L, Z, X, which identify digital state

period <Time>—digital clock rate

logic_H_value <Physical>—analog logic high (1)

logic_L_value <Physical>—analog logic low (0)

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

— H logic high (or logic 1)

— L logic low (or logic 0)

— Z high impedance (absence of logic signal)

— X unknown or indeterminate logic level

Figure B.55 shows a parallel digital sequence where data = [HLHLZH, LHLHHL, LLHHLZ] is sent with a period of 30 µs.

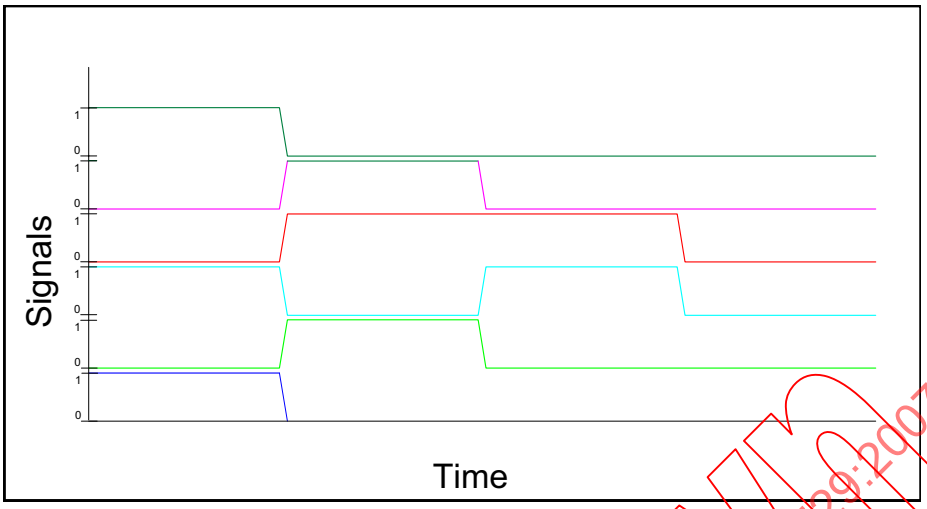


Figure B.55—ParallelDigital

B.6.6 Connection ::SignalFunction

a) *Definition*—**Connection** is the base class that collects signals into multiple channels.

b) *Attributes*

channelWidth <int>—maximum number of channels allowed to be connected (default = 0)

c) *Description*—**Connection**s 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.

B.6.6.1 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*

hi <Character String>
lo <Character String>

c) *Description*

B.6.6.2 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*

true <Character String>
comp <Character String>

c) *Description*

B.6.6.3 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*

true <Character String>

comp <Character String>

lo<Character String>

c) *Description*

B.6.6.4 SinglePhase ::Connection

d) *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.

e) *Attributes*

a <Character String>

n <Character String>

f) *Description*

B.6.6.5 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*

a <Character String>

b <Character String>

n <Character String>

c) *Description*

B.6.6.6 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*

a <Character String>

b <Character String>

c <Character String>

c) *Description*

B.6.6.7 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*

n <Character String>

a <Character String>

b <Character String>

c <Character String>

c) *Description*

B.6.6.8 ThreePhaseSynchro ::Connection

a) *Definition*—ThreePhaseSyncro is a three-wire connection for use with the three-stator outputs of a Synchro.

b) *Attributes*

x <Character String>
y <Character String>
z <Character String>

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.6.6.9 FourWireResolver ::Connection

a) *Definition*—FourWireResolver is a four-wire connection for use with the four-stator terminals of a Resolver.

b) *Attributes*

s1 <Character String>
s2 <Character String>
s3 <Character String>
s4 <Character String>

c) *Description*—Terminals s1 and s3 are used for the sine output, and terminals s2 and s4 are used for the cosine output.

B.6.6.10 SynchroResolver ::Connection

a) *Definition*—SynchroResolver consists of up to four connections for use with the rotor terminals of a Synchro or Resolver.

b) *Attributes*

r1 <Character String>
r2 <Character String>
r3 <Character String>
r4 <Character String>

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.

B.6.6.11 Series ::Connection

a) *Definition*—A Series connection "via" is used when only one connection is required at the test subject.

b) *Attributes*

via <Character String>

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.6.6.12 NonElectrical ::Connection

a) *Definition*—NonElectrical is a connection for use with nonelectrical signals (such as the connection of fluids and gasses).

b) *Attributes*

to <Character String>
from <Character String>

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.6.6.13 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*—Not applicable

c) *Description*—The number of parallel connections is specified by the <width> of the signal. Only active (i.e., data-carrying) signals are considered. Ground or return connections that may be required by the mechanical interfacing are not considered.

Annex C

(normative)

Dynamic signal descriptions

C.1 Introduction

This annex describes the dynamic interactions of BSCs in signal models and TSF components and what happens when they are programmed in any native carrier language.

Signal descriptions can use both static and dynamic signal definitions. The STD standard defines all actions available through the control interface (i.e., IDL) and ensures all the actions 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 is always 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 will be in-deterministic because the actual software notification of the **Signal** state change is only guaranteed to happen after the event.

In order that multiple BSCs can describe a single signal without ambiguity, the STD 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 active without the other cannot happen. The problem is that, in a dynamic signal model, the user can start either component, that is, start the sinusoidal component or the exponential decay component. Rather than attempt to forbid certain control actions, the STD standard defines its dynamic behavior to ensure all cases are semantically described. This standard ensures that starting any component within a signal model will start the whole signal model and, therefore, enable the signal.

Starting different components may lead to transient differences even though the stable signals will be the same.

The STD 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. In these cases, the STD standard does not ensure such signals have a useful purpose; it ensures only that their behavior is deterministic.

The TPL provides a subset of the more traditional behavior expected from signals for test purposes.

Unlike static signals, dynamic signal descriptions can have their definitions changed, 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, for use within a native carrier program.

The **ResourceManager** is the only directly createable class object specified by the STD standard. The minimum number of **ResourceManager**s that a valid system can have is one.

The use of different **ResourceManager**s within a native carrier program allows concurrent support of different test environments, e.g., an intermix of **ResourceManager**s that represent simulation environment and ATE subsystem environments all within the same test program.

C.2.1.1 Method

Require(SignalDescriptor [,UniqueId]) – The **Require** method provides the signal components or signal component models.

- **SignalDescriptor** is a string with the name of a signal class (e.g., BSC, TSF) or contains an XML static signal model description as prescribed in Annex I.
- **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>”

Example 1—signal class name

```
Set myAM = Std.Require('AM_SIGNAL')
      myAM.car_ampl = "100kHz"
      myAM.car_freq = "1V"
      myAM.mod_freq = "660Hz"
      myAM.mod_ampl = "0.5V"
```

Example 2—XML static signal model description

```
Set myAM = Std.Require(
'<Signal out=amSig>' &
  '<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 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 by BSCs 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 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 *Active* 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.
Active—The *Active* signal is active and exists as a signal or gated event stream. An *Active* **Signal** is measurable and available for use.

C.2.2.2 Methods

Stop([timeout=0])—The **Stop** method resets, disconnects, or turns off any *Paused* or *Active* **Signal** and thereby frees any associated signal resources. Following a successful **Stop**, the state of the **Signal** will become *Stopped*.
Run([timeout=0])—The **Run** method setups, starts, connects, or turns on a *Stopped* signal. Following a successful **Run**, the state of the **Signal** is *Paused* and subsequently becomes *Active*. The **Run** method on an *Active* or *Paused* **Signal** will 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 will 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.

C.2.2.3 Comments

The **Signal** class is provided to define interactions between different BSCs and between BSCs and 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 through **Signal** method calls

An event represents information at a discrete point in time. It has no time duration and as such represents a time singularity. An event is when a **Signal**'s state changes.

A signal is based on an event and has characteristics that are additional to any event information. A signal represents a real physical entity, e.g., the current flowing through a wire. Any signal can be used as an event to initiate some activity.

The **Sync** reference considers the event occurrence when the state of the **Sync Signal** becomes *Active*.

The **Gate** reference considers the event gated *Active* while the event 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 *Active*. 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 *Active* states.

The **Signal** interface also provides an enumeration interface so that a native code program can enumerate through all of a **Signal**'s allocated BSCs.

Example:

```
For Each sf in SinusoidWave.Out
      'sf is an allocated Basic Signal Component
Next
```

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.1.

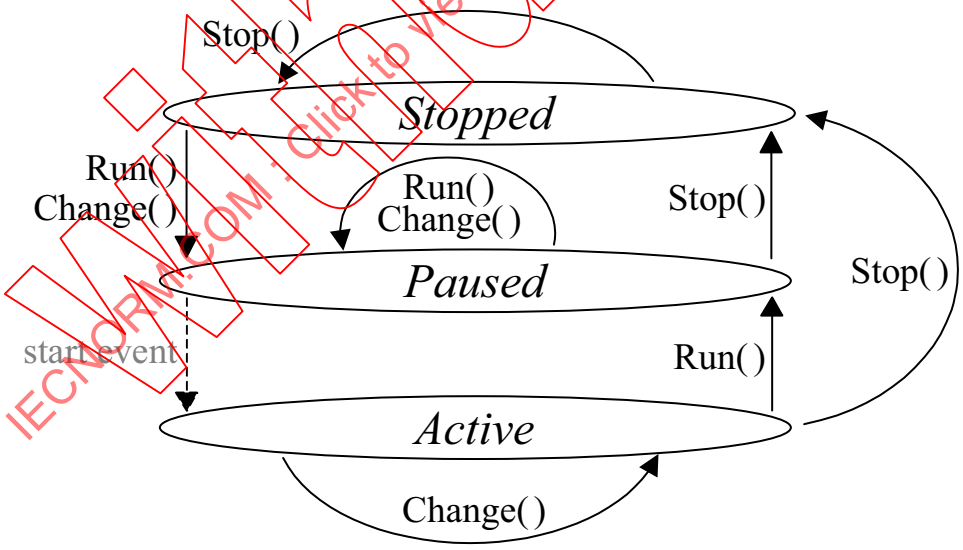


Figure C.1—Signal state changes

For example, when calling the **Run** method, the state of a **Signal** may never become *Active* if the **Sync** event never happens. The reason is that **Run()** tells the BSC associated with a **Signal** that an active signal is required. This knowledge in turn causes the BSC to call **Run()** on all its inputs and then on all its events and returns with the **Signal** in the *Paused* or *Active* state. When the method returns with the **Signal** in the *Paused* state, the BSC is waiting for some internal event to happen. When this expected event occurs, the signal

becomes *Active*. **Run()** does not cause a **Signal** state to become *Active*, but it indicates that an active 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 active.

C.2.3 BSCs

The BSC operation is controlled both through its **Out Signal** interface methods and the state of any **In, Gate**, and **Sync Signals**, providing a two-way control mechanism.

The general behavior of a BSC is that the **Out Signal** state reflects the **In Signal** reference state.

- 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 *Active*, the output **Signal** state is either *Active* or, if a **Sync** referenced is assigned, *Paused* awaiting a **Sync** event.

All BSCs have two input event references called **Sync** and **Gate**. These events affect the behavior of a BSC as follows:

- **Sync** unassigned - restarted when the **In Signal** enters the *Active* state.
- **Sync** assigned - restarted when the **Sync** event enters the *Active* state while an **In** Signal is *Active*.
- **Gate** unassigned - is operational while the **In Signal** is in the *Active* state.
- **Gate** assigned - is operational while the **Gate** and **In** event is in the *Active* state, so that the signal's characteristics are available only while the **Gate** event is gated on.

When a BSC or signal restarts, it repeats its operation from time zero.

Any of the method calls, **Stop()**, **Change()**, or **Run()**, made on the **Out Signal** interface of the BSC will affect the BSC behavior. The BSC will 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 Signal**s.

Run—The **Out Signal** proceeds to the *Paused* state; and the BSC calls **Run()** on all its assigned **In Signal**s 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—Causes the **Signal**'s associated BSC to go on to its next internal function settings and calls **Change()** on all its input **Signal**s.

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 will 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.2, the => symbol should be read as "enters the state," e.g., **In** => *Paused* is read "**In** enters the state *Paused*."

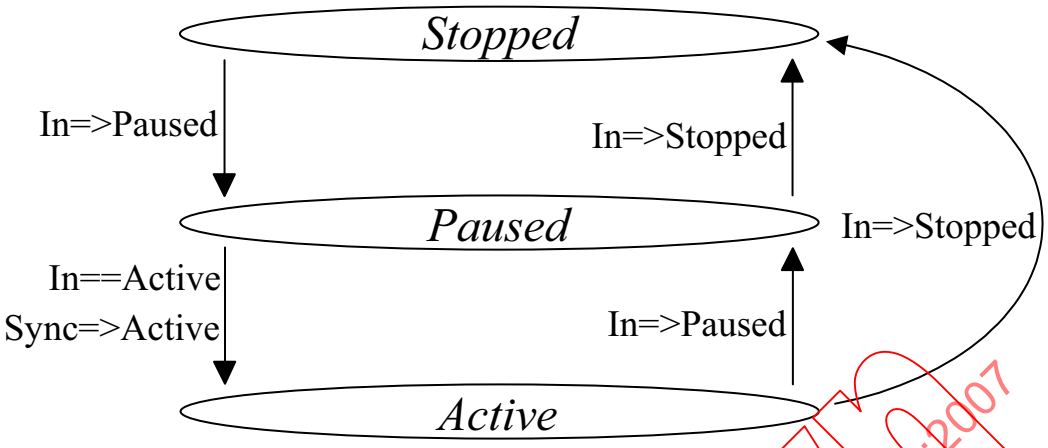


Figure C.2—In (Event) state changes

When the BSC's **Signal** becomes *Paused* if **Sync** event is unassigned, then the **Signal** represents a continuous signal, and the **Signal** immediately becomes *Active*. If a **Sync** event is assigned, it represents a synchronized signal, and the **Signal** becomes *Paused* and is then driven *Active* by the **Sync** event.

When a **Signal** proceeds to a state, it enters each state in turn, e.g., *Active -> 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 will be immediately destroyed.
- When a BSC is destroyed and prior to its destruction, any **Out Signal**s of the BSCs have the **Stop** method called, and the BSC waits until they have all become *Stopped*. **In Signal**s 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 *Active* and BSCs properties are altered, these properties represent its next signal settings that will take effect when the **Out Signal Change** method is next called.
- If the **Sync** reference is unassigned, the **Out Signal** shall enter *Active* from *Paused* immediately.
- If the **Sync** reference is assigned, then the **Out Signal** shall enter *Active* from *Paused* when the **Sync** event becomes *Active*. If the **Sync** event becomes *Active* again while the **Out Signal** is *Active*, then the **Out Signal** is synchronized to this **Sync** event, i.e., it starts again from time zero (T0). Once *Active*, the **Out Signal** state is not affected by the **Sync** state.
- Once the **Out Signal** is *Active*, the BSCs are operational only while the **Gate** reference event is *Active* (gated on). In other words, once triggered, the **Out Signal** enters the *Active* state, but the signal is only present 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 *Active* from *Paused*, the real signal is activated, and the output **Signal** state becomes *Active*.
- When the **Out Signal** enters *Paused* from *Active*, 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.

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 interface so that a native code program can enumerate though all of the **Out Signal**s of a **SignalFunction**. In most cases, there is only one **Out Signal** to enumerate through.

Example:

```
For Each s in Sinusoid
      s.Run 's is the Out Signal
Next
```

C.3 Dynamic signal goals and use cases

The use of BSC or TSF components within the system is identical. The STD 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 of the TSF within the same model. This equivalence ensures 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 will activate every **Signal** within the model so that the whole signal model including any **Gate**/**Sync** events becomes *Active*.

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 subsignal 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.

Annex D

(normative)

IDL basic components

D.1 Introduction

The following 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 BSCs regardless of which carrier environment is use, 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 ensure 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.

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

```
[
    uuid(C4AECBF1-FDF5-11D4-842F-00010214C4DC),
    version(1.0),
    helpstring("STD 1.0 Type Library")
]
library STDBSC
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    //helper to forward declare interfaces and provide typedefs
    #define INTDECL( i )    interface I##i; typedef I##i* i

    INTDECL( Quantity );
    INTDECL( Physical );
    INTDECL( Signal );
    INTDECL( SignalFunction );
    INTDECL( PulseDefn );
    INTDECL( PulseDefns );

    //Base or Derived Physical Types
    typedef Physical Admitance;
    typedef Physical AmountOfInformation;
    typedef Physical AmountOfSubstance;
    typedef Physical Capacitance;
    typedef Physical Charge;
    typedef Physical Conductance;
    typedef Physical Current;
    typedef Physical Voltage;
    typedef Physical Resistance;
    typedef Physical Energy;
    typedef Physical Force;
    typedef Physical Frequency;
    typedef Physical Heat;
    typedef Physical Illuminance;
```

```
    typedef Physical Impedance;
    typedef Physical Inductance;
    typedef Physical Distance;
    typedef Physical LuminousFlux;
    typedef Physical LuminousIntensity;
    typedef Physical MagneticFlux;
    typedef Physical MagneticFluxDensity;
    typedef Physical Mass;
    typedef Physical PlaneAngle;
    typedef Physical Power;
    typedef Physical Pressure;
    typedef Physical Ratio;
    typedef Physical Reactance;
    typedef Physical SolidAngle;
    typedef Physical Susceptance;
    typedef Physical Temperature;
    typedef Physical Time;

    //Compound Types
    typedef Physical Acceleration;
    typedef Physical AngularAcceleration;
    typedef Physical AngularSpeed;
    typedef Physical Area;
    typedef Physical Concentration;
    typedef Physical CurrentDensity;
    typedef Physical DynamicViscosity;
    typedef Physical ElectricChargeDensity;
    typedef Physical ElectricFieldStrength;
    typedef Physical ElectricFluxDensity;
    typedef Physical EnergyDensity;
    typedef Physical Entropy;
    typedef Physical Exposure;
    typedef Physical HeatCapacity;
    typedef Physical HeatFluxDensity;
    typedef Physical Irradiance;
    typedef Physical KinematicViscosity;
    typedef Physical Luminance;
    typedef Physical MagneticFieldStrength;
    typedef Physical MassDensity;
    typedef Physical MassFlow;
    typedef Physical MolarEnergy;
    typedef Physical MolarEntropy;
    typedef Physical MolarHeatCapacity;
    typedef Physical MomentOfForce;
    typedef Physical MomentOfInertia;
    typedef Physical Momentum;
    typedef Physical Permeability;
    typedef Physical Permittivity;
    typedef Physical PowerDensity;
    typedef Physical Radiance;
    typedef Physical RadiantIntensity;
    typedef Physical SpecificEnergy;
    typedef Physical SpecificEntropy;
    typedef Physical SpecificHeatCapacity;
    typedef Physical SpecificVolumn;
    typedef Physical Speed;
    typedef Physical SurfaceTension;
    typedef Physical ThermalConductivity;
    typedef Physical Volume;
    typedef Physical VolumeFlow;

    //custom attribute to represent default value of this property
    #define GUID_DEFAULTVALUE    A6A997E7-94F8-4be2-8366-1814FF70EAFE


//Quantity
    [
        object,
        uuid(E27AA290-461E-11d6-849D-00010214C4DC),
        dual,
        helpstring("IQuantity Interface"),
        pointer_default(unique)
```

```
    ]
    interface IQuantity : IDispatch
    {
        [id(DISPID_VALUE), propget, helpstring("property Quantity value")]
            HRESULT value( [out, retval] VARIANT *pVal);
        [id(DISPID_VALUE), propput, helpstring("property Quantity value")]
            HRESULT value( [in] VARIANT newVal);
        [id(1), propget, helpstring("property magnitude")]
            HRESULT magnitude( [out,retval] double *pVal) ;
        [id(1), propput, helpstring("property magnitude")]
            HRESULT magnitude( [in] double newVal) ;
        [id(2), propget, helpstring("property Unit String")]
            HRESULT units( [out,retval] BSTR *pVal);
        [id(2), propput, helpstring("property Unit String")]
            HRESULT units( [in] BSTR newVal);
    };


//Physical
    [
        object,
        uuid(C4AECBFE-FDF5-11D4-842F-00010214C4DC),
        dual,
        helpstring("IPhysical Interface"),
        pointer_default(unique)
    ]
    interface IPhysical : IQuantity
    {
        enum {PHYSICAL_BASE=16};

        typedef enum _ERRLMT_TYPE {UL=0, LL} ERRLMT_TYPE;
        typedef enum _RANGE_TYPE {MAX=0, MIN} RANGE_TYPE;
        typedef enum _QUALIFIER_TYPE {trms=0, pk_pk, pk, pk_pos=pk, pk_neg, av, inst,
inst_max, inst_min } QUALIFIER_TYPE;

        [id(PHYSICAL_BASE+0), propget, helpstring("property qualifier")]
            HRESULT qualifier([out,retval] QUALIFIER_TYPE *pRel);
        [id(PHYSICAL_BASE+0), propput, helpstring("property qualifier")]
            HRESULT qualifier([in] QUALIFIER_TYPE newVal);
        [id(PHYSICAL_BASE+1), propget, helpstring("property errlmt value")]
           HRESULT errlmt([in, defaultvalue(UL)] ERRLMT_TYPE index, [out,retval] Quantity
*pVal);
        [id(PHYSICAL_BASE+2), propget, helpstring("property range value")]
            HRESULT range([in, defaultvalue(MAX)] RANGE_TYPE index, [out,retval] Quantity
*pVal);
        [id(PHYSICAL_BASE+3), propget, helpstring("property load")]
            HRESULT load([out,retval] Quantity *pVal);
    };

//ISignal
    [
        object,
        uuid(C4AECC00-FDF5-11D4-842F-00010214C4DC),
        dual,
        helpstring("Signal Interface"),
        pointer_default(unique)
    ]
    interface ISignal : IDispatch
    {
        typedef enum _SIGNAL_STATE {STOPPED=0, PAUSED, ACTIVE} SIGNAL_STATE;

        [propget, id(DISPID_NEWENUM), restricted, helpstring("Enumerator")]
            HRESULT _NewEnum([out, retval] IUnknown** ppUnk);
        [propget, id(DISPID_VALUE), helpstring("property SignalFunction")]
            HRESULT SignalFunction([out, retval] SignalFunction* pVal);
        [propget, id(1), helpstring("property State"), bindable, requestedit]
            HRESULT state([out, retval] SIGNAL_STATE *pVal);
        [id(2), helpstring("method Stop")] HRESULT Stop([in, defaultvalue(0)] long
timeOut);
        [id(3), helpstring("method Run")] HRESULT Run([in, defaultvalue(0)] long timeOut);
        [id(4), helpstring("method Change")] HRESULT Change([in, defaultvalue(0)] long
timeOut);
```

```
    };


//ISignalFunction
    [
        object,
        uuid(C4AECC03-FDF5-11D4-842F-00010214C4DC),
        dual,
        helpstring("ISignalFunction Interface"),
        pointer_default(unique)
    ]
    interface ISignalFunction : IDispatch
    {
        enum {SIGNALFUNCTION_BASE=0};

        [propget, id(DISPID_NEWENUM), restricted, helpstring("Enumerator")]
            HRESULT _NewEnum([out, retval] IUnknown** ppUnk);
        [propget, id(DISPID_VALUE), helpstring("property name")]
            HRESULT name([out, retval] BSTR *pVal);
        [propget, id(1), helpstring("property Out")]
            HRESULT Out([in, defaultvalue(0)] long at, [out, retval] Signal *pVal);
        [propget, id(2), helpstring("property In")]
            HRESULT In([in, defaultvalue(0)] long at, [out, retval] Signal *pVal);
        [propputref, id(2), helpstring("property In")]
            HRESULT In([in, defaultvalue(0)] long at, [in] Signal newVal);
        [propget, id(3), helpstring("property Sync")]
            HRESULT Sync([out, retval] Signal *pVal);
        [propputref, id(3), helpstring("property Sync")]
            HRESULT Sync([in] Signal newVal);
        [propget, id(4), helpstring("property Gate")]
            HRESULT Gate([out, retval] Signal *pVal);
        [propputref, id(4), helpstring("property Gate")]
            HRESULT Gate([in] Signal newVal);
        [propget, id(5), helpstring("property Conn")]
            HRESULT Conn([in, defaultvalue(0)] long at, [out, retval] Signal *pVal);
        [propputref, id(5), helpstring("property Conn")]
            HRESULT Conn([in, defaultvalue(0)] long at, [in] Signal newVal);
    };


//PulseDefn
    [
        object,
        uuid(A7C6235F-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IPulseDefn Interface"),
        pointer_default(unique)
    ]
    interface IPulseDefn : IDispatch
    {
        [propget, id(DISPID_VALUE), helpstring("pulse data")]
            HRESULT value([out, retval] BSTR *pVal);
        [propput, id(DISPID_VALUE), helpstring("pulse data")]
            HRESULT value([in] BSTR newVal);
        [propget, id(1), helpstring("start of pulse")]
            HRESULT start([out, retval, custom(GUID_DEFAULTVALUE, "0s")] Physical *pVal);
        [propget, id(2), helpstring("width of pulse")]
            HRESULT pulseWidth([out, retval, custom(GUID_DEFAULTVALUE, "0s")] Physical
*pVal);
        [propput, id(3), helpstring("level multiplication factor of pulse")]
            HRESULT levelFactor([in] double newVal);
        [propget, id(3), helpstring("level multiplication factor of pulse")]
            HRESULT levelFactor([out, retval] double *pVal);
        [propget, id(4), helpstring("name of pulse")]
            HRESULT name([out, retval] BSTR *pVal);
        [propput, id(4), helpstring("name of pulse")]
            HRESULT name([in] BSTR newVal);
    };


//PulseDefns
    [
```

```
        object,
        uuid(39540EDD-C51F-4077-A4E3-4555AE6E928F),
        dual,
        helpstring("IPulseDefns Interface"),
        pointer_default(unique)
    ]
    interface IPulseDefns : IDispatch
    {
        [propget, id(DISPID_NEWENUM), restricted, helpstring("Enumerator")]
            HRESULT _NewEnum([out, retval] IUnknown** ppUnk);
        [propget, id(DISPID_VALUE), helpstring("Get item from collection")]
            HRESULT item([in] VARIANT varItem, [out, retval] PulseDefn* pVal);
        [propget, id(1), helpstring("Get count of items in collection")]
            HRESULT count([out, retval] long* pVal);
        [propget, id(2), helpstring("All pulses")]
            HRESULT value([out, retval] BSTR* pVal);
        [propput, id(2), helpstring("All pulses")]
            HRESULT value([in] BSTR newVal);
        [id(3), helpstring("Add a new pulse to collection")]
            HRESULT Add([in, defaultvalue("")] BSTR bstrName, [out, retval] PulseDefn*
pVal);
        [id(4), helpstring("Remove a pulse from collection")]
            HRESULT Remove([in] VARIANT varItem);
    };


//ResourceManager
    [
        object,
        uuid(DC87F8D2-FE62-11D4-842F-00010214C4DC),
        dual,
        helpstring("ResourceManager Interface"),
        pointer_default(unique)
    ]
    interface IResourceManager : IDispatch
    {
        [id(1), helpstring("method Require - obtains a SignalFunction Interface")]
            HRESULT Require([in] BSTR SignalDescriptor,
                [in, defaultvalue(0)] VARIANT UniqueId,
                [out, retval] SignalFunction *pVal);
    };

    [
        uuid(DC87F8D5-FE62-11D4-842F-00010214C4DC),
        helpstring("ResourceManager Class")
    ]
    coclass ResourceManager
    {
        [default] interface IResourceManager;
    };


//Source
    [
        object,
        uuid(A7C622A5-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ISource Interface"),
        pointer_default(unique)
    ]
    interface ISource : ISignalFunction
    {
        enum {SOURCE_BASE=(SIGNALFUNCTION_BASE+256)};
    };
    [
        uuid(A7C622A5-C6E6-11D5-8476-00010214C4D0),
        helpstring("Source class"),
        noncreatable
    ]
    coclass Source
    {
        [default] interface ISource;
```

```
    };


//NonPeriodic
    [
        object,
        uuid(A7C622A4-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("INonPeriodic Interface"),
        pointer_default(unique)
    ]
    interface INonPeriodic : ISource
    {
        enum {NONPERIODIC_BASE=(SOURCE_BASE+256)};
    };
    [
        uuid(A7C622A4-C6E6-11D5-8476-00010214C4D0),
        helpstring("NonPeriodic class"),
        noncreatable
    ]
    coclass NonPeriodic
    {
        [default] interface INonPeriodic;
    };


//Constant<type>
    [
        object,
        uuid(A7C62302-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IConstant Interface"),
        pointer_default(unique)
    ]
    interface IConstant : INonPeriodic
    {
        enum {CONSTANT_BASE=(NONPERIODIC_BASE+256)};

        [propget, id(CONSTANT_BASE+1), helpstring("the level of the signal.")]
            HRESULT amplitude([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(CONSTANT_BASE+1), helpstring("the level of the signal.")]
            HRESULT amplitude([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
    };
    [
        uuid(A7C62302-C6E6-11D5-8476-00010214C4D0),
        helpstring("Constant class"),
        noncreatable
    ]
    coclass Constant
    {
        [default] interface IConstant;
    };


//Step<type>
    [
        object,
        uuid(A7C623B1-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IStep Interface"),
        pointer_default(unique)
    ]
    interface IStep : INonPeriodic
    {
        enum {STEP_BASE=(NONPERIODIC_BASE+256)};

        [propget, id(STEP_BASE+1), helpstring("final value of step signal")]
            HRESULT amplitude([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(STEP_BASE+1), helpstring("final value of step signal")]
            HRESULT amplitude([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
```

```
        [propget, id(STEP_BASE+2), helpstring("defines when the step transition starts")]
            HRESULT startTime([out, retval, custom(GUID_DEFAULTVALUE, "0.5s")] Time *pVal);
        [propputref, id(STEP_BASE+2), helpstring("defines when the step transition
starts")]
            HRESULT startTime([in, custom(GUID_DEFAULTVALUE, "0.5s")] Time newVal);
    };
    [
        uuid(A7C623B1-C6E6-11D5-8476-00010214C4D0),
        helpstring("Step class"),
        noncreatable
    ]
    coclass Step
    {
        [default] interface IStep;
    };


//SingleTrapezoid<type>
    [
        object,
        uuid(A7C623EF-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ISingleTrapezoid Interface"),
        pointer_default(unique)
    ]
    interface ISingleTrapezoid : INonPeriodic
    {
        enum {SINGLETRAPEZOID_BASE=(NONPERIODIC_BASE+256)};

        [propget, id(SINGLETRAPEZOID_BASE+1), helpstring("value of pulse amplitude")]
            HRESULT amplitude([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(SINGLETRAPEZOID_BASE+1), helpstring("value of pulse amplitude")]
            HRESULT amplitude([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(SINGLETRAPEZOID_BASE+2), helpstring("time at which trapezoid starts
relative to it initialization")]
            HRESULT startTime([out, retval, custom(GUID_DEFAULTVALUE, "0s")] Time *pVal);
       [propputref, id(SINGLETRAPEZOID_BASE+2), helpstring("time at which trapezoid starts
relative to it initialization")]
            HRESULT startTime([in, custom(GUID_DEFAULTVALUE, "0s")] Time newVal);
        [propget, id(SINGLETRAPEZOID_BASE+3), helpstring("time taken to reach amplitude")]
            HRESULT riseTime([out, retval, custom(GUID_DEFAULTVALUE, "0.25s")] Time *pVal);
        [propputref, id(SINGLETRAPEZOID_BASE+3), helpstring("time taken to reach
amplitude")]
            HRESULT riseTime([in, custom(GUID_DEFAULTVALUE, "0.25s")] Time newVal);
        [propget, id(SINGLETRAPEZOID_BASE+4), helpstring("time that trapezoid is stable at
amplitude")]
            HRESULT pulseWidth([out, retval, custom(GUID_DEFAULTVALUE, "0.5s")] Time
*pVal);
        [propputref, id(SINGLETRAPEZOID_BASE+4), helpstring("time that trapezoid is stable
at amplitude")]
            HRESULT pulseWidth([in, custom(GUID_DEFAULTVALUE, "0.5s")] Time newVal);
        [propget, id(SINGLETRAPEZOID_BASE+5), helpstring("Time taken to fall back to
transient state")]
            HRESULT fallTime([out, retval, custom(GUID_DEFAULTVALUE, "0.25s")] Time *pVal);
        [propputref, id(SINGLETRAPEZOID_BASE+5), helpstring("Time taken to fall back to
transient state")]
            HRESULT fallTime([in, custom(GUID_DEFAULTVALUE, "0.25s")] Time newVal);
    };
    [
        uuid(A7C623EF-C6E6-11D5-8476-00010214C4D0),
        helpstring("SingleTrapezoid class"),
        noncreatable
    ]
    coclass SingleTrapezoid
    {
        [default] interface ISingleTrapezoid;
    };


//Noise<type>
    [
```

```
        object,
        uuid(A7C622A3-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("INoise Interface"),
        pointer_default(unique)
    ]
    interface INoise : INonPeriodic
    {
        enum {NOISE_BASE=(NONPERIODIC_BASE+256)};

        [propget, id(NOISE_BASE+1), helpstring("property amplitude")]
            HRESULT amplitude([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(NOISE_BASE+1), helpstring("property amplitude")]
            HRESULT amplitude([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(NOISE_BASE+2), helpstring("used for pseudo-random noise")]
            HRESULT seed([out, retval, custom(GUID_DEFAULTVALUE, "0")] long *pVal);
        [propput, id(NOISE_BASE+2), helpstring("used for pseudo-random noise")]
            HRESULT seed([in, custom(GUID_DEFAULTVALUE, "0")] long newVal);
        [propget, id(NOISE_BASE+3), helpstring("upper bound frequency bandwidth for
transient disturbances")]
            HRESULT frequency([out, retval, custom(GUID_DEFAULTVALUE, "50 Hz")] Frequency
*pVal);
        [propputref, id(NOISE_BASE+3), helpstring("upper bound frequency bandwidth for
transient disturbances")]
            HRESULT frequency([in, custom(GUID_DEFAULTVALUE, "50 Hz")] Frequency newVal);
    };
    [
        uuid(A7C622A3-C6E6-11D5-8476-00010214C4D0),
        helpstring("Noise class"),
        noncreatable
    ]
    coclass Noise
    {
        [default] interface INoise;
    };


//SingleRamp<type>
    [
        object,
        uuid(A7C623C4-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ISingleRamp Interface"),
        pointer_default(unique)
    ]
    interface ISingleRamp : INonPeriodic
    {
        enum {SINGLERAMP_BASE=(NONPERIODIC_BASE+256)};

        [propget, id(SINGLERAMP_BASE+1), helpstring("final value of ramp signal")]
            HRESULT amplitude([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(SINGLERAMP_BASE+1), helpstring("final value of ramp signal")]
            HRESULT amplitude([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(SINGLERAMP_BASE+2), helpstring("time for signal to reach final
value")]
            HRESULT riseTime([out, retval, custom(GUID_DEFAULTVALUE, "1s")] Time *pVal);
        [propputref, id(SINGLERAMP_BASE+2), helpstring("time for signal to reach final
value")]
            HRESULT riseTime([in, custom(GUID_DEFAULTVALUE, "1s")] Time newVal);
        [propget, id(SINGLERAMP_BASE+3), helpstring("defines when the step transition
starts")]
            HRESULT startTime([out, retval, custom(GUID_DEFAULTVALUE, "0s")] Time *pVal);
        [propputref, id(SINGLERAMP_BASE+3), helpstring("defines when the step transition
starts")]
            HRESULT startTime([in, custom(GUID_DEFAULTVALUE, "0s")] Time newVal);
    };
    [
        uuid(A7C623C4-C6E6-11D5-8476-00010214C4D0),
        helpstring("SingleRamp class"),
        noncreatable
```

```
    ]
    coclass SingleRamp
    {
        [default] interface ISingleRamp;
    };


//Periodic
    [
        object,
        uuid(A7C622AB-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IPeriodic Interface"),
        pointer_default(unique)
    ]
    interface IPeriodic : ISource
    {
        enum {PERIODIC_BASE=(SOURCE_BASE+256)};
    };
    [
        uuid(A7C622AB-C6E6-11D5-8476-00010214C4D0),
        helpstring("Periodic class"),
        noncreatable
    ]
    coclass Periodic
    {
        [default] interface IPeriodic;
    };


//Sinusoid<type>
    [
        object,
        uuid(6A29F1AF-1395-4D6F-95EC-9BFAD61E6F5C),
        dual,
        helpstring("ISinusoid Interface"),
        pointer_default(unique)
    ]
    interface ISinusoid : IPeriodic
    {
        enum {SINUSOID_BASE=(PERIODIC_BASE+256)};

        [propget, id(SINUSOID_BASE+1), helpstring("Amplitude")]
            HRESULT amplitude([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(SINUSOID_BASE+1), helpstring("Amplitude")]
            HRESULT amplitude([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(SINUSOID_BASE+2), helpstring("Frequency")]
            HRESULT frequency([out, retval, custom(GUID_DEFAULTVALUE, "1Hz")] Frequency
*pVal);
        [propputref, id(SINUSOID_BASE+2), helpstring("Frequency")]
            HRESULT frequency([in, custom(GUID_DEFAULTVALUE, "1Hz")] Frequency newVal);
        [propget, id(SINUSOID_BASE+3), helpstring("initial phase angle")]
            HRESULT phase([out, retval, custom(GUID_DEFAULTVALUE, "0 rad")] PlaneAngle
*pVal);
        [propputref, id(SINUSOID_BASE+3), helpstring("initial phase angle")]
            HRESULT phase([in, custom(GUID_DEFAULTVALUE, "0 rad")] PlaneAngle newVal);
    };
    [
        uuid(6A29F1AF-1395-4D6F-95EC-9BFAD61E6F50),
        helpstring("Sinusoid class"),
        noncreatable
    ]
    coclass Sinusoid
    {
        [default] interface ISinusoid;
    };


//Trapezoid<type>
    [
        object,
```

```
        uuid(A7C623D5-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ITrapezoid Interface"),
        pointer_default(unique)
    ]
    interface ITrapezoid : IPeriodic
    {
        enum {TRAPEZOID_BASE=(PERIODIC_BASE+256)};

        [propget, id(TRAPEZOID_BASE+1), helpstring("value of pulse amplitude")]
            HRESULT amplitude([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(TRAPEZOID_BASE+1), helpstring("value of pulse amplitude")]
            HRESULT amplitude([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(TRAPEZOID_BASE+2), helpstring("time in which the signal repeats
itself")]
            HRESULT period([out, retval, custom(GUID_DEFAULTVALUE, "1 s")] Time *pVal);
        [propputref, id(TRAPEZOID_BASE+2), helpstring("time in which the signal repeats
itself")]
            HRESULT period([in, custom(GUID_DEFAULTVALUE, "1 s")] Time newVal);
        [propget, id(TRAPEZOID_BASE+3), helpstring("time taken to reach amplitude")]
            HRESULT riseTime([out, retval, custom(GUID_DEFAULTVALUE, "0.25 s")] Time
*pVal);
        [propputref, id(TRAPEZOID_BASE+3), helpstring("time taken to reach amplitude")]
            HRESULT riseTime([in, custom(GUID_DEFAULTVALUE, "0.25 s")] Time newVal);
        [propget, id(TRAPEZOID_BASE+4), helpstring("time that trapezoid is stable at
amplitude")]
            HRESULT pulseWidth([out, retval, custom(GUID_DEFAULTVALUE, "0.5 s")] Time
*pVal);
        [propputref, id(TRAPEZOID_BASE+4), helpstring("time that trapezoid is stable at
amplitude")]
            HRESULT pulseWidth([in, custom(GUID_DEFAULTVALUE, "0.5 s")] Time newVal);
        [propget, id(TRAPEZOID_BASE+5), helpstring("Time taken to fall back to transiant
state")]
            HRESULT fallTime([out, retval, custom(GUID_DEFAULTVALUE, "0.25 s")] Time
*pVal);
       [propputref, id(TRAPEZOID_BASE+5), helpstring("Time taken to fall back to transiant
state")]
            HRESULT fallTime([in, custom(GUID_DEFAULTVALUE, "0.25 s")] Time newVal);
    };
    [
        uuid(A7C623D5-C6E6-11D5-8476-00010214C4D0),
        helpstring("Trapezoid class"),
        noncreatable
    ]
    coclass Trapezoid
    {
        [default] interface ITrapezoid;
    };


//Ramp<type>
    [
        object,
        uuid(A7C622B3-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IRamp Interface"),
        pointer_default(unique)
    ]
    interface IRamp : IPeriodic
    {
        enum {RAMP_BASE=(PERIODIC_BASE+256)};

        [propget, id(RAMP_BASE+1), helpstring("final level of the signal")]
            HRESULT amplitude([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(RAMP_BASE+1), helpstring("final level of the signal")]
            HRESULT amplitude([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(RAMP_BASE+2), helpstring("time is which signal repeats itself")]
            HRESULT period([out, retval, custom(GUID_DEFAULTVALUE, "1s")] Time *pVal);
        [propputref, id(RAMP_BASE+2), helpstring("time is which signal repeats itself")]
            HRESULT period([in, custom(GUID_DEFAULTVALUE, "1s")] Time newVal);
```

```
        [propget, id(RAMP_BASE+3), helpstring("Rise Time of ramp signal")]
            HRESULT riseTime([out, retval, custom(GUID_DEFAULTVALUE, "1s")] Time *pVal);
        [propputref, id(RAMP_BASE+3), helpstring("Rise Time of ramp signal")]
            HRESULT riseTime([in, custom(GUID_DEFAULTVALUE, "1s")] Time newVal);
    };
    [
        uuid(A7C622B3-C6E6-11D5-8476-00010214C4D0),
        helpstring("Ramp class"),
        noncreatable
    ]
    coclass Ramp
    {
        [default] interface IRamp;
    };


//Triangle<type>
    [
        object,
        uuid(A7C622AA-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ITriangle Interface"),
        pointer_default(unique)
    ]
    interface ITriangle : IPeriodic
    {
        enum {TRIANGLE_BASE=(PERIODIC_BASE+256)};

       [propget, id(TRIANGLE_BASE+1), helpstring("maximum amplitude level of the signal")]

            HRESULT amplitude([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(TRIANGLE_BASE+1), helpstring("maximum amplitude level of the
signal")]
            HRESULT amplitude([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(TRIANGLE_BASE+2), helpstring("time in which signal repeats itself")]
            HRESULT period([out, retval, custom(GUID_DEFAULTVALUE, "1s")] Time *pVal);
        [propputref, id(TRIANGLE_BASE+2), helpstring("time in which signal repeats
itself")]
            HRESULT period([in, custom(GUID_DEFAULTVALUE, "1s")] Time newVal);
        [propget, id(TRIANGLE_BASE+3), helpstring("ratio between time taken to increase
from its minimum to its maximum value and the time for one period    ")]
           HRESULT dutyCycle([out, retval, custom(GUID_DEFAULTVALUE, "50%")] Ratio *pVal);
       [propputref, id(TRIANGLE_BASE+3), helpstring("ratio between time taken to increase
from its minimum to its maximum value and the time for one period    ")]
            HRESULT dutyCycle([in, custom(GUID_DEFAULTVALUE, "50%")] Ratio newVal);
    };
    [
        uuid(A7C622AA-C6E6-11D5-8476-00010214C4D0),
        helpstring("Triangle class"),
        noncreatable
    ]
    coclass Triangle
    {
        [default] interface ITriangle;
    };


//SquareWave<type>
    [
        object,
        uuid(A7C62419-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ISquareWave Interface"),
        pointer_default(unique)
    ]
    interface ISquareWave : IPeriodic
    {
        enum {SQUAREWAVE_BASE=(PERIODIC_BASE+256)};

        [propget, id(SQUAREWAVE_BASE+1), helpstring("Amplitude of signal")]
```

```
            HRESULT amplitude([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(SQUAREWAVE_BASE+1), helpstring("Amplitude of signal")]
            HRESULT amplitude([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(SQUAREWAVE_BASE+2), helpstring("period of signal")]
            HRESULT period([out, retval, custom(GUID_DEFAULTVALUE, "1s")] Time *pVal);
        [propputref, id(SQUAREWAVE_BASE+2), helpstring("period of signal")]
            HRESULT period([in, custom(GUID_DEFAULTVALUE, "1s")] Time newVal);
        [propget, id(SQUAREWAVE_BASE+3), helpstring("duty cycle of the wave")]
           HRESULT dutyCycle([out, retval, custom(GUID_DEFAULTVALUE, "50%")] Ratio *pVal);
        [propputref, id(SQUAREWAVE_BASE+3), helpstring("duty cycle of the wave")]
            HRESULT dutyCycle([in, custom(GUID_DEFAULTVALUE, "50%")] Ratio newVal);
    };
    [
        uuid(A7C62419-C6E6-11D5-8476-00010214C4D0),
        helpstring("SquareWave class"),
        noncreatable
    ]
    coclass SquareWave
    {
        [default] interface ISquareWave;
    };


//WaveformRamp<type>
    [
        object,
        uuid(A7C62478-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IWaveformRamp Interface"),
        pointer_default(unique)
    ]
    interface IWaveformRamp : IPeriodic
    {
        enum {WAVEFORMRAMP_BASE=(PERIODIC_BASE+256)};

       [propget, id(WAVEFORMRAMP_BASE+1), helpstring("amplitude of the output signal where
the level factor (in points) ")]
            HRESULT amplitude([out, retval, custom(GUID_DEFAULTVALUE, "1")] Physical
*pVal);
        [propputref, id(WAVEFORMRAMP_BASE+1), helpstring("amplitude of the output signal
where the level factor (in points) ")]
            HRESULT amplitude([in, custom(GUID_DEFAULTVALUE, "1")] Physical newVal);
        [propget, id(WAVEFORMRAMP_BASE+2), helpstring("the time between each sequence
")]
            HRESULT period([out, retval, custom(GUID_DEFAULTVALUE, "1s")] Time *pVal);
       [propputref, id(WAVEFORMRAMP_BASE+2), helpstring("the time between each sequence
")]
            HRESULT period([in, custom(GUID_DEFAULTVALUE, "1s")] Time newVal);
       [propget, id(WAVEFORMRAMP_BASE+3), helpstring("the time between successive (points)
outputs     ")]
            HRESULT samplingInterval([out, retval, custom(GUID_DEFAULTVALUE, "0")] Time
*pVal);
        [propputref, id(WAVEFORMRAMP_BASE+3), helpstring("the time between successive
(points) outputs     ")]
            HRESULT samplingInterval([in, custom(GUID_DEFAULTVALUE, "0")] Time newVal);
        [propget, id(WAVEFORMRAMP_BASE+4), helpstring("level factor of each waveform
sample")]
            HRESULT points([out, retval] SAFEARRAY(VARIANT) *pVal);
        [propput, id(WAVEFORMRAMP_BASE+4), helpstring("level factor of each waveform
sample")]
            HRESULT points([in] SAFEARRAY(VARIANT) newVal);
    };
    [
        uuid(A7C62478-C6E6-11D5-8476-00010214C4D0),
        helpstring("WaveformRamp class"),
        noncreatable
    ]
    coclass WaveformRamp
    {
        [default] interface IWaveformRamp;
    };
```

```
//WaveformStep<type>
    [
        object,
        uuid(A7C62463-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IWaveformStep Interface"),
        pointer_default(unique)
    ]
    interface IWaveformStep : IPeriodic
    {
        enum {WAVEFORMSTEP_BASE=(PERIODIC_BASE+256)};

       [propget, id(WAVEFORMSTEP_BASE+1), helpstring("amplitude of the output signal where
the level factor (in points) ")]
            HRESULT amplitude([out, retval, custom(GUID_DEFAULTVALUE, "1")] Physical
*pVal);
        [propputref, id(WAVEFORMSTEP_BASE+1), helpstring("amplitude of the output signal
where the level factor (in points) ")]
            HRESULT amplitude([in, custom(GUID_DEFAULTVALUE, "1")] Physical newVal);
        [propget, id(WAVEFORMSTEP_BASE+2), helpstring("the time between each sequence")]
            HRESULT period([out, retval, custom(GUID_DEFAULTVALUE, "1s")] Time *pVal);
       [propputref, id(WAVEFORMSTEP_BASE+2), helpstring("the time between each sequence")]

            HRESULT period([in, custom(GUID_DEFAULTVALUE, "1s")] Time newVal);
       [propget, id(WAVEFORMSTEP_BASE+3), helpstring("the time between successive (points)
outputs    ")]
            HRESULT samplingInterval([out, retval, custom(GUID_DEFAULTVALUE, "0")] Time
*pVal);
        [propputref, id(WAVEFORMSTEP_BASE+3), helpstring("the time between successive
(points) outputs    ")]
            HRESULT samplingInterval([in, custom(GUID_DEFAULTVALUE, "0")] Time newVal);
      [propget, id(WAVEFORMSTEP_BASE+4), helpstring("level factor of each waveform sample
")]
            HRESULT points([out, retval] SAFEARRAY(VARIANT) *pVal);
      [propput, id(WAVEFORMSTEP_BASE+4), helpstring("level factor of each waveform sample
")]
            HRESULT points([in] SAFEARRAY(VARIANT) newVal);
    };
    [
        uuid(A7C62463-C6E6-11D5-8476-00010214C4D0),
        helpstring("WaveformStep class"),
        noncreatable
    ]
    coclass WaveformStep
    {
        [default] interface IWaveformStep;
    };


//Conditioner
    [
        object,
        uuid(A7C622FD-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IConditioner Interface"),
        pointer_default(unique)
    ]
    interface IConditioner : ISignalFunction
    {
        enum {CONDITIONER_BASE=(SIGNALFUNCTION_BASE+256)};
    };
    [
        uuid(A7C622FD-C6E6-11D5-8476-00010214C4D0),
        helpstring("Conditioner class"),
        noncreatable
    ]
    coclass Conditioner
    {
        [default] interface IConditioner;
    };
```

```
//Filter
    [
        object,
        uuid(A7C6253F-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IFilter Interface"),
        pointer_default(unique)
    ]
    interface IFilter : IConditioner
    {
        enum {FILTER_BASE=(CONDITIONER_BASE+256)};
    };
    [
        uuid(A7C6253F-C6E6-11D5-8476-00010214C4D0),
        helpstring("Filter class"),
        noncreatable
    ]
    coclass Filter
    {
        [default] interface IFilter;
    };


//BandPass
    [
        object,
        uuid(A7C6253E-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IBandPass Interface"),
        pointer_default(unique)
    ]
    interface IBandPass : IFilter
    {
        enum {BANDPASS_BASE=(FILTER_BASE+256)};

       [propget, id(BANDPASS_BASE+1), helpstring("Center frequency of the filter's band")]

          HRESULT centerFrequency([out, retval, custom(GUID_DEFAULTVALUE, "0")] Frequency
*pVal);
        [propputref, id(BANDPASS_BASE+1), helpstring("Center frequency of the filter's
band")]
           HRESULT centerFrequency([in, custom(GUID_DEFAULTVALUE, "0")] Frequency newVal);
        [propget, id(BANDPASS_BASE+2), helpstring("Bandwidth of filter. Zero implies
narrowest band ")]
            HRESULT frequencyBand([out, retval, custom(GUID_DEFAULTVALUE, "0")] Frequency
*pVal);
        [propputref, id(BANDPASS_BASE+2), helpstring("Bandwidth of filter. Zero implies
narrowest band ")]
            HRESULT frequencyBand([in, custom(GUID_DEFAULTVALUE, "0")] Frequency newVal);
    };
    [
        uuid(A7C6253E-C6E6-11D5-8476-00010214C4D0),
        helpstring("BandPass class"),
        noncreatable
    ]
    coclass BandPass
    {
        [default] interface IBandPass;
    };


//LowPass
    [
        object,
        uuid(D9FE1DA9-C6E8-11D5-8476-00010214C4DC),
        dual,
        helpstring("ILowPass Interface"),
        pointer_default(unique)
    ]
    interface ILowPass : IFilter
```

```
    {
        enum {LOWPASS_BASE=(FILTER_BASE+256)};

       [propget, id(LOWPASS_BASE+1), helpstring("Cut off frequency filter. Zero implies DC
offset")]
            HRESULT cutoff([out, retval, custom(GUID_DEFAULTVALUE, "0Hz")] Frequency
*pVal);
       [propputref, id(LOWPASS_BASE+1), helpstring("Cut off frequency filter. Zero implies
DC offset")]
            HRESULT cutoff([in, custom(GUID_DEFAULTVALUE, "0Hz")] Frequency newVal);
    };
    [
        uuid(D9FE1DA9-C6E8-11D5-8476-00010214C4D0),
        helpstring("LowPass class"),
        noncreatable
    ]
    coclass LowPass
    {
        [default] interface ILowPass;
    };


//HighPass
    [
        object,
        uuid(D9FE1DAC-C6E8-11D5-8476-00010214C4DC),
        dual,
        helpstring("IHighPass Interface"),
        pointer_default(unique)
    ]
    interface IHighPass : IFilter
    {
        enum {HIGHPASS_BASE=(FILTER_BASE+256)};

       [propget, id(HIGHPASS_BASE+1), helpstring("Start frequency of filter. Zero implies
AC Coupled")]
            HRESULT cutoff([out, retval, custom(GUID_DEFAULTVALUE, "0Hz")] Frequency
*pVal);
        [propputref, id(HIGHPASS_BASE+1), helpstring("Start frequency of filter. Zero
implies AC Coupled")]
            HRESULT cutoff([in, custom(GUID_DEFAULTVALUE, "0Hz")] Frequency newVal);
    };
    [
        uuid(D9FE1DAC-C6E8-11D5-8476-00010214C4D0),
        helpstring("HighPass class"),
        noncreatable
    ]
    coclass HighPass
    {
        [default] interface IHighPass;
    };


//Notch
    [
        object,
        uuid(D9FE1DAF-C6E8-11D5-8476-00010214C4DC),
        dual,
        helpstring("INotch Interface"),
        pointer_default(unique)
    ]
    interface INotch : IFilter
    {
        enum {NOTCH_BASE=(FILTER_BASE+256)};

       [propget, id(NOTCH_BASE+1), helpstring("Center frrequency of the filter's notch")]
            HRESULT centerFrequency([out, retval, custom(GUID_DEFAULTVALUE, "0Hz")]
Frequency *pVal);
        [propputref, id(NOTCH_BASE+1), helpstring("Center frrequency of the filter's
notch")]
            HRESULT centerFrequency([in, custom(GUID_DEFAULTVALUE, "0Hz")] Frequency
newVal);
```

```
        [propget, id(NOTCH_BASE+2), helpstring("Frequency band of filter. Zero implies
minimum band")]
           HRESULT frequencyBand([out, retval, custom(GUID_DEFAULTVALUE, "0Hz")] Frequency
*pVal);
        [propputref, id(NOTCH_BASE+2), helpstring("Frequency band of filter. Zero implies
minimum band")]
            HRESULT frequencyBand([in, custom(GUID_DEFAULTVALUE, "0Hz")] Frequency newVal);
    };
    [
        uuid(D9FE1DAF-C6E8-11D5-8476-00010214C4D0),
        helpstring("Notch class"),
        noncreatable
    ]
    coclass Notch
    {
        [default] interface INotch;
    };


//Combiner
    [
        object,
        uuid(A7C622FC-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ICombiner Interface"),
        pointer_default(unique)
    ]
    interface ICombiner : IConditioner
    {
        enum {COMBINER_BASE=(CONDITIONER_BASE+256)};
    };
    [
        uuid(A7C622FC-C6E6-11D5-8476-00010214C4D0),
        helpstring("Combiner class"),
        noncreatable
    ]
    coclass Combiner
    {
        [default] interface ICombiner;
    };


//Sum
    [
        object,
        uuid(A7C622FB-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ISum Interface"),
        pointer_default(unique)
    ]
    interface ISum : ICombiner
    {
        enum {SUM_BASE=(COMBINER_BASE+256)};
    };
    [
        uuid(A7C622FB-C6E6-11D5-8476-00010214C4D0),
        helpstring("Sum class"),
        noncreatable
    ]
    coclass Sum
    {
        [default] interface ISum;
    };


//Product
    [
        object,
        uuid(A7C62353-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IProduct Interface"),
        pointer_default(unique)
```

```
    ]
    interface IProduct : ICombiner
    {
        enum {PRODUCT_BASE=(COMBINER_BASE+256)};
    };
    [
        uuid(A7C62353-C6E6-11D5-8476-00010214C4D0),
        helpstring("Product class"),
        noncreatable
    ]
    coclass Product
    {
        [default] interface IProduct;
    };


//Diff
    [
        object,
        uuid(A7C62357-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IDiff Interface"),
        pointer_default(unique)
    ]
    interface IDiff : ICombiner
    {
        enum {DIFF_BASE=(COMBINER_BASE+256)};
    };
    [
        uuid(A7C62357-C6E6-11D5-8476-00010214C4D0),
        helpstring("Diff class"),
        noncreatable
    ]
    coclass Diff
    {
        [default] interface IDiff;
    };


//Modulator
    [
        object,
        uuid(A7C6231A-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IModulator Interface"),
        pointer_default(unique)
    ]
    interface IModulator : IConditioner
    {
        enum {MODULATOR_BASE=(CONDITIONER_BASE+256)};
    };
    [
        uuid(A7C6231A-C6E6-11D5-8476-00010214C4D0),
        helpstring("Modulator class"),
        noncreatable
    ]
    coclass Modulator
    {
        [default] interface IModulator;
    };


//FM
    [
        object,
        uuid(A7C62431-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IFM Interface"),
        pointer_default(unique)
    ]
    interface IFM : IModulator
    {
```

```
        enum {FM_BASE=(MODULATOR_BASE+256)};

        [propget, id(FM_BASE+1), helpstring("Amplitude of  Sinusoidal Carrier wave")]
            HRESULT amplitude([out, retval, custom(GUID_DEFAULTVALUE, "1")] Physical
*pVal);
        [propputref, id(FM_BASE+1), helpstring("Amplitude of  Sinusoidal Carrier wave")]
            HRESULT amplitude([in, custom(GUID_DEFAULTVALUE, "1")] Physical newVal);
        [propget, id(FM_BASE+2), helpstring("Frequency of Sinusoidal Carrier Wave")]
            HRESULT carrierFrequency([out, retval, custom(GUID_DEFAULTVALUE, "1kHz")]
Frequency *pVal);
        [propputref, id(FM_BASE+2), helpstring("Frequency of Sinusoidal Carrier Wave")]
            HRESULT carrierFrequency([in, custom(GUID_DEFAULTVALUE, "1kHz")] Frequency
newVal);
        [propget, id(FM_BASE+3), helpstring("Frequency deviation")]
            HRESULT frequencyDeviation([out, retval, custom(GUID_DEFAULTVALUE, "100Hz")]
Frequency *pVal);
        [propputref, id(FM_BASE+3), helpstring("Frequency deviation")]
            HRESULT frequencyDeviation([in, custom(GUID_DEFAULTVALUE, "100Hz")] Frequency
newVal);
    };
    [
        uuid(A7C62431-C6E6-11D5-8476-00010214C4D0),
        helpstring("FM class"),
        noncreatable
    ]
    coclass FM
    {
        [default] interface IFM;
    };


//AM
    [
        object,
        uuid(A7C62319-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IAM Interface"),
        pointer_default(unique)
    ]
    interface IAM : IModulator
    {
        enum {AM_BASE=(MODULATOR_BASE+256)};

        [propget, id(AM_BASE+1), helpstring("Modulation Index")]
            HRESULT modIndex([out, retval, custom(GUID_DEFAULTVALUE, "0.3")] Ratio *pVal);
        [propputref, id(AM_BASE+1), helpstring("Modulation Index")]
            HRESULT modIndex([in, custom(GUID_DEFAULTVALUE, "0.3")] Ratio newVal);
        [propget, id(AM_BASE+2), helpstring("property Carrier")]
            HRESULT Carrier([out, retval] Signal *pVal);
        [propputref, id(AM_BASE+2), helpstring("property Carrier")]
            HRESULT Carrier([in] Signal newVal);
    };
    [
        uuid(A7C62319-C6E6-11D5-8476-00010214C4D0),
        helpstring("AM class"),
        noncreatable
    ]
    coclass AM
    {
        [default] interface IAM;
    };


//PM
    [
        object,
        uuid(C945E4B5-354D-46CE-B7B3-7C37B177BDD4),
        dual,
        helpstring("IPM Interface"),
        pointer_default(unique)
    ]
    interface IPM : IModulator
```

```
    {
        enum {PM_BASE=(MODULATOR_BASE+256)};

        [propget, id(PM_BASE+1), helpstring("Amplitude of  Sinusoidal Carrier wave")]
            HRESULT amplitude([out, retval, custom(GUID_DEFAULTVALUE, "1")] Physical
*pVal);
        [propputref, id(PM_BASE+1), helpstring("Amplitude of  Sinusoidal Carrier wave")]
            HRESULT amplitude([in, custom(GUID_DEFAULTVALUE, "1")] Physical newVal);
        [propget, id(PM_BASE+2), helpstring("Frequency of Sinusoidal Carrier Wave")]
            HRESULT carrierFrequency([out, retval, custom(GUID_DEFAULTVALUE, "1kHz")]
Frequency *pVal);
        [propputref, id(PM_BASE+2), helpstring("Frequency of Sinusoidal Carrier Wave")]
            HRESULT carrierFrequency([in, custom(GUID_DEFAULTVALUE, "1kHz")] Frequency
newVal);
        [propget, id(PM_BASE+3), helpstring("Phase deviation")]
            HRESULT phaseDeviation([out, retval, custom(GUID_DEFAULTVALUE, "(pi/4)")]
PlaneAngle *pVal);
        [propputref, id(PM_BASE+3), helpstring("Phase deviation")]
            HRESULT phaseDeviation([in, custom(GUID_DEFAULTVALUE, "(pi/4)")] PlaneAngle
newVal);
    };
    [
        uuid(C945E4B5-354D-46CE-B7B3-7C37B177BDD0),
        helpstring("PM class"),
        noncreatable
    ]
    coclass PM
    {
        [default] interface IPM;
    };


//Transformation
    [
        object,
        uuid(A7C6236B-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ITransformation Interface"),
        pointer_default(unique)
    ]
    interface ITransformation : IConditioner
    {
        enum {TRANSFORMATION_BASE=(CONDITIONER_BASE+256)};
    };
    [
        uuid(A7C6236B-C6E6-11D5-8476-00010214C4D0),
        helpstring("Transformation class"),
        noncreatable
    ]
    coclass Transformation
    {
        [default] interface ITransformation;
    };


//SignalDelay
    [
        object,
        uuid(A7C624A1-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ISignalDelay Interface"),
        pointer_default(unique)
    ]
    interface ISignalDelay : ITransformation
    {
        enum {SIGNALDELAY_BASE=(TRANSFORMATION_BASE+256)};

       [propget, id(SIGNALDELAY_BASE+1), helpstring("the rate at which the Rate will alter
over time (Acceleration)")]
            HRESULT acceleration([out, retval, custom(GUID_DEFAULTVALUE, "0")] Frequency
*pVal);
```

```
        [propputref, id(SIGNALDELAY_BASE+1), helpstring("the rate at which the Rate will
alter over time (Acceleration)")]
            HRESULT acceleration([in, custom(GUID_DEFAULTVALUE, "0")] Frequency newVal);
        [propget, id(SIGNALDELAY_BASE+2), helpstring("Fixed delay that signal will be
delayed (Distance)")]
            HRESULT delay([out, retval, custom(GUID_DEFAULTVALUE, "0")] Time *pVal);
        [propputref, id(SIGNALDELAY_BASE+2), helpstring("Fixed delay that signal will be
delayed (Distance)")]
            HRESULT delay([in, custom(GUID_DEFAULTVALUE, "0")] Time newVal);
        [propget, id(SIGNALDELAY_BASE+3), helpstring("the rate at which the Delay will
alter over time (Speed)")]
            HRESULT rate([out, retval, custom(GUID_DEFAULTVALUE, "0")] Ratio *pVal);
        [propputref, id(SIGNALDELAY_BASE+3), helpstring("the rate at which the Delay will
alter over time (Speed)")]
            HRESULT rate([in, custom(GUID_DEFAULTVALUE, "0")] Ratio newVal);
    };
    [
        uuid(A7C624A1-C6E6-11D5-8476-00010214C4D0),
        helpstring("SignalDelay class"),
        noncreatable
    ]
    coclass SignalDelay
    {
        [default] interface ISignalDelay;
    };


//Exponential
    [
        object,
        uuid(A7C624AB-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IExponential Interface"),
        pointer_default(unique)
    ]
    interface IExponential : ITransformation
    {
        enum {EXPONENTIAL_BASE=(TRANSFORMATION_BASE+256)};

        [propget, id(EXPONENTIAL_BASE+1), helpstring("value of damping factor")]
            HRESULT dampingFactor([out, retval, custom(GUID_DEFAULTVALUE, "1.0")] double
*pVal);
        [propput, id(EXPONENTIAL_BASE+1), helpstring("value of damping factor")]
            HRESULT dampingFactor([in, custom(GUID_DEFAULTVALUE, "1.0")] double newVal);
    };
    [
        uuid(A7C624AB-C6E6-11D5-8476-00010214C4D0),
        helpstring("Exponential class"),
        noncreatable
    ]
    coclass Exponential
    {
        [default] interface IExponential;
    };


//E
    [
        object,
        uuid(A7C624C0-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IE Interface"),
        pointer_default(unique)
    ]
    interface IE : ITransformation
    {
        enum {E_BASE=(TRANSFORMATION_BASE+256)};
    };
    [
        uuid(A7C624C0-C6E6-11D5-8476-00010214C4D0),
        helpstring("E class"),
        noncreatable
```

```
    ]
    coclass E
    {
        [default] interface IE;
    };


//Negate
    [
        object,
        uuid(A7C62377-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("INegate Interface"),
        pointer_default(unique)
    ]
    interface INegate : ITransformation
    {
        enum {NEGATE_BASE=(TRANSFORMATION_BASE+256)};
    };
    [
        uuid(A7C62377-C6E6-11D5-8476-00010214C4D0),
        helpstring("Negate class"),
        noncreatable
    ]
    coclass Negate
    {
        [default] interface INegate;
    };


//Inverse
    [
        object,
        uuid(A7C624C3-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IInverse Interface"),
        pointer_default(unique)
    ]
    interface IInverse : ITransformation
    {
        enum {INVERSE_BASE=(TRANSFORMATION_BASE+256)};
    };
    [
        uuid(A7C624C3-C6E6-11D5-8476-00010214C4D0),
        helpstring("Inverse class"),
        noncreatable
    ]
    coclass Inverse
    {
        [default] interface IInverse;
    };


//PulseTrain
    [
        object,
        uuid(A7C6236A-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IPulseTrain Interface"),
        pointer_default(unique)
    ]
    interface IPulseTrain : ITransformation
    {
        enum {PULSETRAIN_BASE=(TRANSFORMATION_BASE+256)};

        [propget, id(PULSETRAIN_BASE+1), helpstring("list of pulses")]
            HRESULT pulses([out, retval] PulseDefns *pVal);
        [propputref, id(PULSETRAIN_BASE+1), helpstring("list of pulses")]
            HRESULT pulses([in] PulseDefns newVal);
        [propget, id(PULSETRAIN_BASE+2), helpstring("property repetition")]
            HRESULT repetition([out, retval, custom(GUID_DEFAULTVALUE, "0")] long *pVal);
        [propput, id(PULSETRAIN_BASE+2), helpstring("property repetition")]
```

```
            HRESULT repetition([in, custom(GUID_DEFAULTVALUE, "0")] long newVal);
    };
    [
        uuid(A7C6236A-C6E6-11D5-8476-00010214C4D0),
        helpstring("PulseTrain class"),
        noncreatable
    ]
    coclass PulseTrain
    {
        [default] interface IPulseTrain;
    };


//Attenuator
    [
        object,
        uuid(A7C62562-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IAttenuator Interface"),
        pointer_default(unique)
    ]
    interface IAttenuator : ITransformation
    {
        enum {ATTENUATOR_BASE=(TRANSFORMATION_BASE+256)};

        [propget, id(ATTENUATOR_BASE+1), helpstring("property gain")]
            HRESULT gain([out, retval, custom(GUID_DEFAULTVALUE, "1.0")] Ratio *pVal);
        [propputref, id(ATTENUATOR_BASE+1), helpstring("property gain")]
            HRESULT gain([in, custom(GUID_DEFAULTVALUE, "1.0")] Ratio newVal);
    };
    [
        uuid(A7C62562-C6E6-11D5-8476-00010214C4D0),
        helpstring("Attenuator class"),
        noncreatable
    ]
    coclass Attenuator
    {
        [default] interface IAttenuator;
    };


//Load
    [
        object,
        uuid(A7C62564-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ILoad Interface"),
        pointer_default(unique)
    ]
    interface ILoad : ITransformation
    {
        enum {LOAD_BASE=(TRANSFORMATION_BASE+256)};

        [propget, id(LOAD_BASE+1), helpstring("property resistance")]
            HRESULT resistance([out, retval, custom(GUID_DEFAULTVALUE, "0 Ohm")] Resistance
*pVal);
        [propputref, id(LOAD_BASE+1), helpstring("property resistance")]
            HRESULT resistance([in, custom(GUID_DEFAULTVALUE, "0 Ohm")] Resistance newVal);
        [propget, id(LOAD_BASE+2), helpstring("property reactance")]
            HRESULT reactance([out, retval, custom(GUID_DEFAULTVALUE, "0 Ohm")] Resistance
*pVal);
        [propputref, id(LOAD_BASE+2), helpstring("property reactance")]
            HRESULT reactance([in, custom(GUID_DEFAULTVALUE, "0 Ohm")] Resistance newVal);
    };
    [
        uuid(A7C62564-C6E6-11D5-8476-00010214C4D0),
        helpstring("Load class"),
        noncreatable
    ]
    coclass Load
    {
        [default] interface ILoad;
```

```
    };


//Limit<type>
    [
        object,
        uuid(2A88B0B8-D480-11D5-957C-0080C8BCAFAC),
        dual,
        helpstring("ILimit Interface"),
        pointer_default(unique)
    ]
    interface ILimit : ITransformation
    {
        enum {LIMIT_BASE=(TRANSFORMATION_BASE+256)};

        [propget, id(LIMIT_BASE+1), helpstring("limits the absolute value of the signal to
+/- limit value")]
            HRESULT limit([out, retval, custom(GUID_DEFAULTVALUE, "1")] Physical *pVal);
        [propputref, id(LIMIT_BASE+1), helpstring("limits the absolute value of the signal
to +/- limit value")]
            HRESULT limit([in, custom(GUID_DEFAULTVALUE, "1")] Physical newVal);
    };
    [
        uuid(2A88B0B8-D480-11D5-957C-0080C8BCAFA0),
        helpstring("Limit class"),
        noncreatable
    ]
    coclass Limit
    {
        [default] interface ILimit;
    };


//FFT
    [
        object,
        uuid(A7C624BA-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IFFT Interface"),
        pointer_default(unique)
    ]
    interface IFFT : ITransformation
    {
        enum {FFT_BASE=(TRANSFORMATION_BASE+256)};

        [propget, id(FFT_BASE+1), helpstring("property samples")]
            HRESULT samples([out, retval, custom(GUID_DEFAULTVALUE, "1023")] long *pVal);
        [propput, id(FFT_BASE+1), helpstring("property samples")]
            HRESULT samples([in, custom(GUID_DEFAULTVALUE, "1023")] long newVal);
        [propget, id(FFT_BASE+2), helpstring("property interval")]
            HRESULT interval([out, retval, custom(GUID_DEFAULTVALUE, "1s")] Time *pVal);
        [propputref, id(FFT_BASE+2), helpstring("property interval")]
            HRESULT interval([in, custom(GUID_DEFAULTVALUE, "1s")] Time newVal);
    };
    [
        uuid(A7C624BA-C6E6-11D5-8476-00010214C4D0),
        helpstring("FFT class"),
        noncreatable
    ]
    coclass FFT
    {
        [default] interface IFFT;
    };


//EventFunction
    [
        object,
        uuid(EAC83656-D5D0-11D5-957C-0080C8BCAFAC),
        dual,
        helpstring("IEventFunction Interface"),
        pointer_default(unique)
```

```
    ]
    interface IEventFunction : ISignalFunction
    {
        enum {EVENTFUNCTION_BASE=(SIGNALFUNCTION_BASE+256)};
    };
    [
        uuid(EAC83656-D5D0-11D5-957C-0080C8BCAFA0),
        helpstring("EventFunction class"),
        noncreatable
    ]
    coclass EventFunction
    {
        [default] interface IEventFunction;
    };


//EventSource
    [
        object,
        uuid(EAC83658-D5D0-11D5-957C-0080C8BCAFAC),
        dual,
        helpstring("IEventSource Interface"),
        pointer_default(unique)
    ]
    interface IEventSource : IEventFunction
    {
        enum {EVENTSOURCE_BASE=(EVENTFUNCTION_BASE+256)};
    };
    [
        uuid(EAC83658-D5D0-11D5-957C-0080C8BCAFA0),
        helpstring("EventSource class"),
        noncreatable
    ]
    coclass EventSource
    {
        [default] interface IEventSource;
    };


//Clock
    [
        object,
        uuid(A7C622BB-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IClock Interface"),
        pointer_default(unique)
    ]
    interface IClock : IEventSource
    {
        enum {CLOCK_BASE=(EVENTSOURCE_BASE+256)};

        [propget, id(CLOCK_BASE+1), helpstring("Frequency of the clock.")]
            HRESULT clockRate([out, retval, custom(GUID_DEFAULTVALUE, "1Hz")] Frequency
*pVal);
        [propputref, id(CLOCK_BASE+1), helpstring("Frequency of the clock.")]
            HRESULT clockRate([in, custom(GUID_DEFAULTVALUE, "1Hz")] Frequency newVal);
    };
    [
        uuid(A7C622BB-C6E6-11D5-8476-00010214C4D0),
        helpstring("Clock class"),
        noncreatable
    ]
    coclass Clock
    {
        [default] interface IClock;
    };


//TimedEvent
    [
        object,
        uuid(A7C622E1-C6E6-11D5-8476-00010214C4DC),
```

```
        dual,
        helpstring("ITimedEvent Interface"),
        pointer_default(unique)
    ]
    interface ITimedEvent : IEventSource
    {
        enum {TIMEDEVENT_BASE=(EVENTSOURCE_BASE+256)};

       [propget, id(TIMEDEVENT_BASE+1), helpstring("the delay time before the first event
will be start")]
            HRESULT delay([out, retval, custom(GUID_DEFAULTVALUE, "0s")] Time *pVal);
        [propputref, id(TIMEDEVENT_BASE+1), helpstring("the delay time before the first
event will be start")]
            HRESULT delay([in, custom(GUID_DEFAULTVALUE, "0s")] Time newVal);
        [propget, id(TIMEDEVENT_BASE+2), helpstring("the duration that each event is
active")]
            HRESULT duration([out, retval, custom(GUID_DEFAULTVALUE, "1s")] Time *pVal);
        [propputref, id(TIMEDEVENT_BASE+2), helpstring("the duration that each event is
active")]
            HRESULT duration([in, custom(GUID_DEFAULTVALUE, "1s")] Time newVal);
        [propget, id(TIMEDEVENT_BASE+3), helpstring("the time interval for each event")]
            HRESULT period([out, retval, custom(GUID_DEFAULTVALUE, "1s")] Time *pVal);
      [propputref, id(TIMEDEVENT_BASE+3), helpstring("the time interval for each event")]

            HRESULT period([in, custom(GUID_DEFAULTVALUE, "1s")] Time newVal);
        [propget, id(TIMEDEVENT_BASE+4), helpstring("property repetition")]
            HRESULT repetition([out, retval, custom(GUID_DEFAULTVALUE, "0")] long *pVal);
        [propput, id(TIMEDEVENT_BASE+4), helpstring("property repetition")]
            HRESULT repetition([in, custom(GUID_DEFAULTVALUE, "0")] long newVal);
    };
    [
        uuid(A7C622E1-C6E6-11D5-8476-00010214C4D0),
        helpstring("TimedEvent class"),
        noncreatable
    ]
    coclass TimedEvent
    {
        [default] interface ITimedEvent;
    };


//PulsedEvent
    [
        object,
        uuid(A7C622D1-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IPulsedEvent Interface"),
        pointer_default(unique)
    ]
    interface IPulsedEvent : IEventSource
    {
        enum {PULSEDEVENT_BASE=(EVENTSOURCE_BASE+256)};

        [propget, id(PULSEDEVENT_BASE+1), helpstring("list of pulses")]
            HRESULT pulses([out, retval] PulseDefns *pVal);
        [propputref, id(PULSEDEVENT_BASE+1), helpstring("list of pulses")]
            HRESULT pulses([in] PulseDefns newVal);
        [propget, id(PULSEDEVENT_BASE+2), helpstring("property repetition")]
            HRESULT repetition([out, retval, custom(GUID_DEFAULTVALUE, "0")] long *pVal);
        [propput, id(PULSEDEVENT_BASE+2), helpstring("property repetition")]
            HRESULT repetition([in, custom(GUID_DEFAULTVALUE, "0")] long newVal);
    };
    [
        uuid(A7C622D1-C6E6-11D5-8476-00010214C4D0),
        helpstring("PulsedEvent class"),
        noncreatable
    ]
    coclass PulsedEvent
    {
        [default] interface IPulsedEvent;
    };
```

```
//EventConditioner
    [
        object,
        uuid(A7C622BC-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IEventConditioner Interface"),
        pointer_default(unique)
    ]
    interface IEventConditioner : IEventFunction
    {
        enum {EVENTCONDITIONER_BASE=(EVENTFUNCTION_BASE+256)};
    };
    [
        uuid(A7C622BC-C6E6-11D5-8476-00010214C4D0),
        helpstring("EventConditioner class"),
        noncreatable
    ]
    coclass EventConditioner
    {
        [default] interface IEventConditioner;
    };


//EventedEvent
    [
        object,
        uuid(A7C625A5-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IEventedEvent Interface"),
        pointer_default(unique)
    ]
    interface IEventedEvent : IEventConditioner
    {
        enum {EVENTEDEVENT_BASE=(EVENTCONDITIONER_BASE+256)};
    };
    [
        uuid(A7C625A5-C6E6-11D5-8476-00010214C4D0),
        helpstring("EventedEvent class"),
        noncreatable
    ]
    coclass EventedEvent
    {
        [default] interface IEventedEvent;
    };


//EventCount
    [
        object,
        uuid(51C0083E-2DD3-4D49-8B39-F915B8E4AB2A),
        dual,
        helpstring("IEventCount Interface"),
        pointer_default(unique)
    ]
    interface IEventCount : IEventConditioner
    {
        enum {EVENTCOUNT_BASE=(EVENTCONDITIONER_BASE+256)};

        [propget, id(EVENTCOUNT_BASE+1), helpstring("Identifies the number of events that
must happen before a event is generated")]
            HRESULT count([out, retval, custom(GUID_DEFAULTVALUE, "0")] long *pVal);
        [propput, id(EVENTCOUNT_BASE+1), helpstring("Identifies the number of events that
must happen before a event is generated")]
            HRESULT count([in, custom(GUID_DEFAULTVALUE, "0")] long newVal);
    };
    [
        uuid(51C0083E-2DD3-4D49-8B39-F915B8E4AB20),
        helpstring("EventCount class"),
        noncreatable
    ]
    coclass EventCount
```

```
    {
        [default] interface IEventCount;
    };


//ProbabilityEvent
    [
        object,
        uuid(D9FE1DA4-C6E8-11D5-8476-00010214C4DC),
        dual,
        helpstring("IProbabilityEvent Interface"),
        pointer_default(unique)
    ]
    interface IProbabilityEvent : IEventConditioner
    {
        enum {PROBABILITYEVENT_BASE=(EVENTCONDITIONER_BASE+256)};

        [propget, id(PROBABILITYEVENT_BASE+1), helpstring("property seed")]
            HRESULT seed([out, retval, custom(GUID_DEFAULTVALUE, "0")] long *pVal);
        [propput, id(PROBABILITYEVENT_BASE+1), helpstring("property seed")]
            HRESULT seed([in, custom(GUID_DEFAULTVALUE, "0")] long newVal);
        [propget, id(PROBABILITYEVENT_BASE+2), helpstring("property probability")]
            HRESULT probability([out, retval, custom(GUID_DEFAULTVALUE, "50")] Ratio
*pVal);
        [propputref, id(PROBABILITYEVENT_BASE+2), helpstring("property probability")]
            HRESULT probability([in, custom(GUID_DEFAULTVALUE, "50")] Ratio newVal);
    };
    [
        uuid(D9FE1DA4-C6E8-11D5-8476-00010214C4D0),
        helpstring("ProbabilityEvent class"),
        noncreatable
    ]
    coclass ProbabilityEvent
    {
        [default] interface IProbabilityEvent;
    };


//NotEvent
    [
        object,
        uuid(A7C622CB-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("INotEvent Interface"),
        pointer_default(unique)
    ]
    interface INotEvent : IEventConditioner
    {
        enum {NOTEVENT_BASE=(EVENTCONDITIONER_BASE+256)};
    };
    [
        uuid(A7C622CB-C6E6-11D5-8476-00010214C4D0),
        helpstring("NotEvent class"),
        noncreatable
    ]
    coclass NotEvent
    {
        [default] interface INotEvent;
    };


//Logical
    [
        object,
        uuid(A7C622C2-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ILogical Interface"),
        pointer_default(unique)
    ]
    interface ILogical : IEventConditioner
    {
        enum {LOGICAL_BASE=(EVENTCONDITIONER_BASE+256)};
```

```
    };
    [
        uuid(A7C622C2-C6E6-11D5-8476-00010214C4D0),
        helpstring("Logical class"),
        noncreatable
    ]
    coclass Logical
    {
        [default] interface ILogical;
    };


//OrEvent
    [
        object,
        uuid(A7C622CE-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IOrEvent Interface"),
        pointer_default(unique)
    ]
    interface IOrEvent : ILogical
    {
        enum {OREVENT_BASE=(LOGICAL_BASE+256)};
    };
    [
        uuid(A7C622CE-C6E6-11D5-8476-00010214C4D0),
        helpstring("OrEvent class"),
        noncreatable
    ]
    coclass OrEvent
    {
        [default] interface IOrEvent;
    };


//XOrEvent
    [
        object,
        uuid(A7C622C1-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IXOrEvent Interface"),
        pointer_default(unique)
    ]
    interface IXOrEvent : ILogical
    {
        enum {XOREVENT_BASE=(LOGICAL_BASE+256)};
    };
    [
        uuid(A7C622C1-C6E6-11D5-8476-00010214C4D0),
        helpstring("XOrEvent class"),
        noncreatable
    ]
    coclass XOrEvent
    {
        [default] interface IXOrEvent;
    };


//AndEvent
    [
        object,
        uuid(A7C622C5-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IAndEvent Interface"),
        pointer_default(unique)
    ]
    interface IAndEvent : ILogical
    {
        enum {ANDEVENT_BASE=(LOGICAL_BASE+256)};
    };
    [
        uuid(A7C622C5-C6E6-11D5-8476-00010214C4D0),
```

```
        helpstring("AndEvent class"),
        noncreatable
    ]
    coclass AndEvent
    {
        [default] interface IAndEvent;
    };


//Sensor
    [
        object,
        uuid(A7C6236F-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ISensor Interface"),
        pointer_default(unique)
    ]
    interface ISensor : ISignalFunction
    {
        enum {SENSOR_BASE=(SIGNALFUNCTION_BASE+256)};

        typedef enum _enumMeasuredVariable {DEPENDENT=0,INDEPENDENT} enumMeasuredVariable;
        typedef enum _enumCondition {NONE=0,GT,GE,LE,LT,EQ,NE} enumCondition;

        [propget, id(SENSOR_BASE+1), helpstring("Whether the measurement made is of the
dependent or independent variable. ")]
            HRESULT measuredVariable([out, retval, custom(GUID_DEFAULTVALUE, "DEPENDENT")]
enumMeasuredVariable *pVal);
        [propput, id(SENSOR_BASE+1), helpstring("Whether the measurement made is of the
dependent or independent variable. ")]
            HRESULT measuredVariable([in, custom(GUID_DEFAULTVALUE, "DEPENDENT")]
enumMeasuredVariable newVal);
        [propget, id(SENSOR_BASE+2), helpstring("Current value measured")]
            HRESULT measurement([out, retval, custom(GUID_DEFAULTVALUE, "0")] VARIANT
*pVal);
        [propget, id(SENSOR_BASE+3), helpstring("Array of measurements made")]
            HRESULT measurements([out, retval] SAFEARRAY(VARIANT) *pVal);
        [propget, id(SENSOR_BASE+4), helpstring("Number of consecutive measurement to be
made. Zero indicates no measurement to be taken and indicates the Sensor is acting as a
monitor only")]
            HRESULT samples([out, retval, custom(GUID_DEFAULTVALUE, "0")] long *pVal);
        [propput, id(SENSOR_BASE+4), helpstring("Number of consecutive measurement to be
made. Zero indicates no measurement to be taken and indicates the Sensor is acting as a
monitor only")]
            HRESULT samples([in, custom(GUID_DEFAULTVALUE, "0")] long newVal);
        [propget, id(SENSOR_BASE+5), helpstring("Readonly number of measurements currently
made")]
            HRESULT count([out, retval, custom(GUID_DEFAULTVALUE, "0")] long *pVal);
        [propget, id(SENSOR_BASE+6), helpstring("Continuous range of independent variable
(Time) over which measurement is made.")]
            HRESULT gateTime([out, retval, custom(GUID_DEFAULTVALUE, "1")] double *pVal);
        [propput, id(SENSOR_BASE+6), helpstring("Continuous range of independent variable
(Time) over which measurement is made.")]
            HRESULT gateTime([in, custom(GUID_DEFAULTVALUE, "1")] double newVal);
        [propget, id(SENSOR_BASE+7), helpstring("Value against which any condition is
checked. This can be either an absolute value (5V) or a ratio value (50%) representing the
percentage value between the low-peak and high-peak values.")]
            HRESULT nominal([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical *pVal);
        [propputref, id(SENSOR_BASE+7), helpstring("Value against which any condition is
checked. This can be either an absolute value (5V) or a ratio value (50%) representing the
percentage value between the low-peak and high-peak values.")]
            HRESULT nominal([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(SENSOR_BASE+8), helpstring("Test made between measurement and nominal
value")]
            HRESULT condition([out, retval, custom(GUID_DEFAULTVALUE, "NONE")]
enumCondition *pVal);
        [propput, id(SENSOR_BASE+8), helpstring("Test made between measurement and nominal
value")]
            HRESULT condition([in, custom(GUID_DEFAULTVALUE, "NONE")] enumCondition
newVal);
        [propget, id(SENSOR_BASE+9), helpstring("Read Only flag indicating last measurement
Pass/Fail Status. If no measurement is taken GO is False")]
```

```
            HRESULT GO([out, retval, custom(GUID_DEFAULTVALUE, "false")] VARIANT_BOOL
*pVal);
        [propget, id(SENSOR_BASE+10), helpstring("Read Only flag indicating last
measurement Pass/Fail Status If no measurement is taken GO is False.")]
            HRESULT NOGO([out, retval, custom(GUID_DEFAULTVALUE, "false")] VARIANT_BOOL
*pVal);
        [propget, id(SENSOR_BASE+11), helpstring("Read Only flag indicating the last
measurement High/Low Status.  If no measurement is taken HI is False")]
            HRESULT HI([out, retval, custom(GUID_DEFAULTVALUE, "false")] VARIANT_BOOL
*pVal);
        [propget, id(SENSOR_BASE+12), helpstring("Read Only flag indicating the last
measurement High/Low Status.  If no measurement is taken LO is False")]
            HRESULT LO([out, retval, custom(GUID_DEFAULTVALUE, "false")] VARIANT_BOOL
*pVal);
       [propget, id(SENSOR_BASE+13), helpstring("Upper Limit value against which condition
is checked")]
            HRESULT UL([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical *pVal);
        [propputref, id(SENSOR_BASE+13), helpstring("Upper Limit value against which
condition is checked")]
            HRESULT UL([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
       [propget, id(SENSOR_BASE+14), helpstring("Lower Limit value against which condition
is checked")]
            HRESULT LL([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical *pVal);
        [propputref, id(SENSOR_BASE+14), helpstring("Lower Limit value against which
condition is checked")]
            HRESULT LL([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
    };
    [
        uuid(A7C6236F-C6E6-11D5-8476-00010214C4D0),
        helpstring("Sensor class"),
        noncreatable
    ]
    coclass Sensor
    {
        [default] interface ISensor;
    };


//Counter
    [
        object,
        uuid(AAA29431-80F5-499F-89F8-FCA91420C15D),
        dual,
        helpstring("ICounter Interface"),
        pointer_default(unique)
    ]
    interface ICounter : ISensor
    {
        enum {COUNTER_BASE=(SENSOR_BASE+256)};
    };
    [
        uuid(AAA29431-80F5-499F-89F8-FCA91420C150),
        helpstring("Counter class"),
        noncreatable
    ]
    coclass Counter
    {
        [default] interface ICounter;
    };


//TimeInterval
    [
        object,
        uuid(A7C62516-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ITimeInterval Interface"),
        pointer_default(unique)
    ]
    interface ITimeInterval : ISensor
    {
        enum {TIMEINTERVAL_BASE=(SENSOR_BASE+256)};
```

```
    };
    [
        uuid(A7C62516-C6E6-11D5-8476-00010214C4D0),
        helpstring("TimeInterval class"),
        noncreatable
    ]
    coclass TimeInterval
    {
        [default] interface ITimeInterval;
    };


//Instantaneous<type>
    [
        object,
        uuid(A7C6236E-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IInstantaneous Interface"),
        pointer_default(unique)
    ]
    interface IInstantaneous : ISensor
    {
        enum {INSTANTANEOUS_BASE=(SENSOR_BASE+256)};
    };
    [
        uuid(A7C6236E-C6E6-11D5-8476-00010214C4D0),
        helpstring("Instantaneous class"),
        noncreatable
    ]
    coclass Instantaneous
    {
        [default] interface IInstantaneous;
    };


//RMS<type>
    [
        object,
        uuid(A7C624FF-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IRMS Interface"),
        pointer_default(unique)
    ]
    interface IRMS : ISensor
    {
        enum {RMS_BASE=(SENSOR_BASE+256)};
    };
    [
        uuid(A7C624FF-C6E6-11D5-8476-00010214C4D0),
        helpstring("RMS class"),
        noncreatable
    ]
    coclass RMS
    {
        [default] interface IRMS;
    };


//Average<type>
    [
        object,
        uuid(A7C624FA-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IAverage Interface"),
        pointer_default(unique)
    ]
    interface IAverage : ISensor
    {
        enum {AVERAGE_BASE=(SENSOR_BASE+256)};
    };
    [
        uuid(A7C624FA-C6E6-11D5-8476-00010214C4D0),
```

```
        helpstring("Average class"),
        noncreatable
    ]
    coclass Average
    {
        [default] interface IAverage;
    };


//PeakToPeak<type>
    [
        object,
        uuid(A7C624F7-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IPeakToPeak Interface"),
        pointer_default(unique)
    ]
    interface IPeakToPeak : ISensor
    {
        enum {PEAKTOPEAK_BASE=(SENSOR_BASE+256)};
    };
    [
        uuid(A7C624F7-C6E6-11D5-8476-00010214C4D0),
        helpstring("PeakToPeak class"),
        noncreatable
    ]
    coclass PeakToPeak
    {
        [default] interface IPeakToPeak;
    };


//Peak<type>
    [
        object,
        uuid(5CF2B379-D60F-43BB-9098-A368B2F65B97),
        dual,
        helpstring("IPeak Interface"),
        pointer_default(unique)
    ]
    interface IPeak : ISensor
    {
        enum {PEAK_BASE=(SENSOR_BASE+256)};
    };
    [
        uuid(5CF2B379-D60F-43BB-9098-A368B2F65B90),
        helpstring("Peak class"),
        noncreatable
    ]
    coclass Peak
    {
        [default] interface IPeak;
    };


//PeakNeg<type>
    [
        object,
        uuid(57DE7809-EDE2-4E3D-8D6D-B77948290628),
        dual,
        helpstring("IPeakNeg Interface"),
        pointer_default(unique)
    ]
    interface IPeakNeg : ISensor
    {
        enum {PEAKNEG_BASE=(SENSOR_BASE+256)};
    };
    [
        uuid(57DE7809-EDE2-4E3D-8D6D-B77948290620),
        helpstring("PeakNeg class"),
        noncreatable
    ]
```

```
    coclass PeakNeg
    {
        [default] interface IPeakNeg;
    };


//MaxInstantaneous<type>
    [
        object,
        uuid(CE0FF197-D039-4489-BC03-2B193B1FEB8B),
        dual,
        helpstring("IMaxInstantaneous Interface"),
        pointer_default(unique)
    ]
    interface IMaxInstantaneous : ISensor
    {
        enum {MAXINSTANTANEOUS_BASE=(SENSOR_BASE+256)};
    };
    [
        uuid(CE0FF197-D039-4489-BC03-2B193B1FEB80),
        helpstring("MaxInstantaneous class"),
        noncreatable
    ]
    coclass MaxInstantaneous
    {
        [default] interface IMaxInstantaneous;
    };


//MinInstantaneous<type>
    [
        object,
        uuid(5680A87C-3277-4516-BAC3-FDBAE686A3F6),
        dual,
        helpstring("IMinInstantaneous Interface"),
        pointer_default(unique)
    ]
    interface IMinInstantaneous : ISensor
    {
        enum {MININSTANTANEOUS_BASE=(SENSOR_BASE+256)};
    };
    [
        uuid(5680A87C-3277-4516-BAC3-FDBAE686A3F0),
        helpstring("MinInstantaneous class"),
        noncreatable
    ]
    coclass MinInstantaneous
    {
        [default] interface IMinInstantaneous;
    };


//Measure
    [
        object,
        uuid(81FF15A8-5B6F-491E-88FF-1783E7462AC8),
        dual,
        helpstring("IMeasure Interface"),
        pointer_default(unique)
    ]
    interface IMeasure : ISensor
    {
        enum {MEASURE_BASE=(SENSOR_BASE+256)};

       [propget, id(MEASURE_BASE+1), helpstring("reference to a Signal representing signal
model of input signal")]
            HRESULT As([out, retval] Signal *pVal);
       [propput, id(MEASURE_BASE+1), helpstring("reference to a Signal representing signal
model of input signal")]
            HRESULT As([in] Signal newVal);
        [propputref, id(MEASURE_BASE+1), helpstring("reference to a Signal representing
signal model of input signal")]
```

```
            HRESULT As([in] Signal newVal);
        [propget, id(MEASURE_BASE+2), helpstring("Attribute of the signal that is to be
measured")]
            HRESULT attribute([out, retval] BSTR *pVal);
        [propput, id(MEASURE_BASE+2), helpstring("Attribute of the signal that is to be
measured")]
            HRESULT attribute([in] BSTR newVal);
    };
    [
        uuid(81FF15A8-5B6F-491E-88FF-1783E7462AC0),
        helpstring("Measure class"),
        noncreatable
    ]
    coclass Measure
    {
        [default] interface IMeasure;
    };


//Digital
    [
        object,
        uuid(A7C62519-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IDigital Interface"),
        pointer_default(unique)
    ]
    interface IDigital : ISignalFunction
    {
        enum {DIGITAL_BASE=(SIGNALFUNCTION_BASE+256)};
    };
    [
        uuid(A7C62519-C6E6-11D5-8476-00010214C4D0),
        helpstring("Digital class"),
        noncreatable
    ]
    coclass Digital
    {
        [default] interface IDigital;
    };


//SerialDigital
    [
        object,
        uuid(2C187B71-ABCF-4121-87AD-2426FDEA6895),
        dual,
        helpstring("ISerialDigital Interface"),
        pointer_default(unique)
    ]
    interface ISerialDigital : IDigital
    {
        enum {SERIALDIGITAL_BASE=(DIGITAL_BASE+256)};

        [propget, id(SERIALDIGITAL_BASE+1), helpstring("String containing characters 'H',
'L', 'Z', 'X' identifying digital state")]
            HRESULT data([out, retval] BSTR *pVal);
        [propput, id(SERIALDIGITAL_BASE+1), helpstring("String containing characters 'H',
'L', 'Z', 'X' identifying digital state")]
            HRESULT data([in] BSTR newVal);
        [propget, id(SERIALDIGITAL_BASE+2), helpstring("Digital Clock Rate")]
            HRESULT period([out, retval, custom(GUID_DEFAULTVALUE, "1")] Time *pVal);
        [propputref, id(SERIALDIGITAL_BASE+2), helpstring("Digital Clock Rate")]
            HRESULT period([in, custom(GUID_DEFAULTVALUE, "1")] Time newVal);
        [propget, id(SERIALDIGITAL_BASE+3), helpstring("Analog Logic High (1)")]
            HRESULT logic_H_value([out, retval, custom(GUID_DEFAULTVALUE, "1")] Voltage
*pVal);
        [propputref, id(SERIALDIGITAL_BASE+3), helpstring("Analog Logic High (1)")]
            HRESULT logic_H_value([in, custom(GUID_DEFAULTVALUE, "1")] Voltage newVal);
        [propget, id(SERIALDIGITAL_BASE+4), helpstring("Analog Logic Low (0)")]
            HRESULT logic_L_value([out, retval, custom(GUID_DEFAULTVALUE, "0")] Voltage
*pVal);
```

```
        [propputref, id(SERIALDIGITAL_BASE+4), helpstring("Analog Logic Low (0)")]
            HRESULT logic_L_value([in, custom(GUID_DEFAULTVALUE, "0")] Voltage newVal);
    };
    [
        uuid(2C187B71-ABCF-4121-87AD-2426FDEA6890),
        helpstring("SerialDigital class"),
        noncreatable
    ]
    coclass SerialDigital
    {
        [default] interface ISerialDigital;
    };


//ParallelDigital
    [
        object,
        uuid(48C1DF77-F924-4311-B5E9-8829AD677242),
        dual,
        helpstring("IParallelDigital Interface"),
        pointer_default(unique)
    ]
    interface IParallelDigital : IDigital
    {
        enum {PARALLELDIGITAL_BASE=(DIGITAL_BASE+256)};

        [propget, id(PARALLELDIGITAL_BASE+1), helpstring("where each string String
containing characters 'H', 'L', 'Z', 'X' identifying digital state")]
            HRESULT data([out, retval] BSTR *pVal);
        [propput, id(PARALLELDIGITAL_BASE+1), helpstring("where each string String
containing characters 'H', 'L', 'Z', 'X' identifying digital state")]
            HRESULT data([in] BSTR newVal);
        [propget, id(PARALLELDIGITAL_BASE+2), helpstring("Digital Clock Rate")]
            HRESULT period([out, retval, custom(GUID_DEFAULTVALUE, "1")] Time *pVal);
        [propputref, id(PARALLELDIGITAL_BASE+2), helpstring("Digital Clock Rate")]
            HRESULT period([in, custom(GUID_DEFAULTVALUE, "1")] Time newVal);
        [propget, id(PARALLELDIGITAL_BASE+3), helpstring("Analog Logic High (1)")]
            HRESULT logic_H_value([out, retval, custom(GUID_DEFAULTVALUE, "1")] Voltage
*pVal);
        [propputref, id(PARALLELDIGITAL_BASE+3), helpstring("Analog Logic High (1)")]
            HRESULT logic_H_value([in, custom(GUID_DEFAULTVALUE, "1")] Voltage newVal);
        [propget, id(PARALLELDIGITAL_BASE+4), helpstring("Analog Logic low (0)")]
            HRESULT logic_L_value([out, retval, custom(GUID_DEFAULTVALUE, "0")] Voltage
*pVal);
        [propputref, id(PARALLELDIGITAL_BASE+4), helpstring("Analog Logic low (0)")]
            HRESULT logic_L_value([in, custom(GUID_DEFAULTVALUE, "0")] Voltage newVal);
    };
    [
        uuid(48C1DF77-F924-4311-B5E9-8829AD677240),
        helpstring("ParallelDigital class"),
        noncreatable
    ]
    coclass ParallelDigital
    {
        [default] interface IParallelDigital;
    };


//Connection
    [
        object,
        uuid(A7C624CE-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IConnection Interface"),
        pointer_default(unique)
    ]
    interface IConnection : ISignalFunction
    {
        enum {CONNECTION_BASE=(SIGNALFUNCTION_BASE+256)};

       [propget, id(CONNECTION_BASE+1), helpstring("maximum number of channels allowed to
be connected")]
```

```
            HRESULT channelWidth([out, retval, custom(GUID_DEFAULTVALUE, "0")] int *pVal);
        [propput, id(CONNECTION_BASE+1), helpstring("maximum number of channels allowed to
be connected")]
            HRESULT channelWidth([in, custom(GUID_DEFAULTVALUE, "0")] int newVal);
    };
    [
        uuid(A7C624CE-C6E6-11D5-8476-00010214C4D0),
        helpstring("Connection class"),
        noncreatable
    ]
    coclass Connection
    {
        [default] interface IConnection;
    };


//TwoWire
    [
        object,
        uuid(A7C6256B-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ITwoWire Interface"),
        pointer_default(unique)
    ]
    interface ITwoWire : IConnection
    {
        enum {TWOWIRE_BASE=(CONNECTION_BASE+256)};

        [propget, id(TWOWIRE_BASE+1), helpstring("property lo")]
            HRESULT lo([out, retval] BSTR *pVal);
        [propput, id(TWOWIRE_BASE+1), helpstring("property lo")]
            HRESULT lo([in] BSTR newVal);
        [propget, id(TWOWIRE_BASE+2), helpstring("property hi")]
            HRESULT hi([out, retval] BSTR *pVal);
        [propput, id(TWOWIRE_BASE+2), helpstring("property hi")]
            HRESULT hi([in] BSTR newVal);
    };
    [
        uuid(A7C6256B-C6E6-11D5-8476-00010214C4D0),
        helpstring("TwoWire class"),
        noncreatable
    ]
    coclass TwoWire
    {
        [default] interface ITwoWire;
    };


//TwoWireComp
    [
        object,
        uuid(A7C625D0-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ITwoWireComp Interface"),
        pointer_default(unique)
    ]
    interface ITwoWireComp : IConnection
    {
        enum {TWOWIRECOMP_BASE=(CONNECTION_BASE+256)};

        [propget, id(TWOWIRECOMP_BASE+1), helpstring("property true")]
            HRESULT true([out, retval] BSTR *pVal);
        [propput, id(TWOWIRECOMP_BASE+1), helpstring("property true")]
            HRESULT true([in] BSTR newVal);
        [propget, id(TWOWIRECOMP_BASE+2), helpstring("property comp")]
            HRESULT comp([out, retval] BSTR *pVal);
        [propput, id(TWOWIRECOMP_BASE+2), helpstring("property comp")]
            HRESULT comp([in] BSTR newVal);
    };
    [
        uuid(A7C625D0-C6E6-11D5-8476-00010214C4D0),
        helpstring("TwoWireComp class"),
```

```
        noncreatable
    ]
    coclass TwoWireComp
    {
        [default] interface ITwoWireComp;
    };


//ThreeWireComp
    [
        object,
        uuid(A7C625CC-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IThreeWireComp Interface"),
        pointer_default(unique)
    ]
    interface IThreeWireComp : IConnection
    {
        enum {THREEWIRECOMP_BASE=(CONNECTION_BASE+256)};

        [propget, id(THREEWIRECOMP_BASE+1), helpstring("property true")]
            HRESULT true([out, retval] BSTR *pVal);
        [propput, id(THREEWIRECOMP_BASE+1), helpstring("property true")]
            HRESULT true([in] BSTR newVal);
        [propget, id(THREEWIRECOMP_BASE+2), helpstring("property comp")]
            HRESULT comp([out, retval] BSTR *pVal);
        [propput, id(THREEWIRECOMP_BASE+2), helpstring("property comp")]
            HRESULT comp([in] BSTR newVal);
    };
    [
        uuid(A7C625CC-C6E6-11D5-8476-00010214C4D0),
        helpstring("ThreeWireComp class"),
        noncreatable
    ]
    coclass ThreeWireComp
    {
        [default] interface IThreeWireComp;
    };


//SinglePhase
    [
        object,
        uuid(A7C625C7-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ISinglePhase Interface"),
        pointer_default(unique)
    ]
    interface ISinglePhase : IConnection
    {
        enum {SINGLEPHASE_BASE=(CONNECTION_BASE+256)};

        [propget, id(SINGLEPHASE_BASE+1), helpstring("property n")]
            HRESULT n([out, retval] BSTR *pVal);
        [propput, id(SINGLEPHASE_BASE+1), helpstring("property n")]
            HRESULT n([in] BSTR newVal);
        [propget, id(SINGLEPHASE_BASE+2), helpstring("property a")]
            HRESULT a([out, retval] BSTR *pVal);
        [propput, id(SINGLEPHASE_BASE+2), helpstring("property a")]
            HRESULT a([in] BSTR newVal);
    };
    [
        uuid(A7C625C7-C6E6-11D5-8476-00010214C4D0),
        helpstring("SinglePhase class"),
        noncreatable
    ]
    coclass SinglePhase
    {
        [default] interface ISinglePhase;
    };
```

```
//TwoPhase
    [
        object,
        uuid(A7C625C3-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ITwoPhase Interface"),
        pointer_default(unique)
    ]
    interface ITwoPhase : IConnection
    {
        enum {TWOPHASE_BASE=(CONNECTION_BASE+256)};

        [propget, id(TWOPHASE_BASE+1), helpstring("property n")]
            HRESULT n([out, retval] BSTR *pVal);
        [propput, id(TWOPHASE_BASE+1), helpstring("property n")]
            HRESULT n([in] BSTR newVal);
        [propget, id(TWOPHASE_BASE+2), helpstring("property a")]
            HRESULT a([out, retval] BSTR *pVal);
        [propput, id(TWOPHASE_BASE+2), helpstring("property a")]
            HRESULT a([in] BSTR newVal);
        [propget, id(TWOPHASE_BASE+3), helpstring("property b")]
            HRESULT b([out, retval] BSTR *pVal);
        [propput, id(TWOPHASE_BASE+3), helpstring("property b")]
            HRESULT b([in] BSTR newVal);
    };
    [
        uuid(A7C625C3-C6E6-11D5-8476-00010214C4D0),
        helpstring("TwoPhase class"),
        noncreatable
    ]
    coclass TwoPhase
    {
        [default] interface ITwoPhase;
    };


//ThreePhaseDelta
    [
        object,
        uuid(A7C625BF-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IThreePhaseDelta Interface"),
        pointer_default(unique)
    ]
    interface IThreePhaseDelta : IConnection
    {
        enum {THREEPHASEDELTA_BASE=(CONNECTION_BASE+256)};

        [propget, id(THREEPHASEDELTA_BASE+1), helpstring("property a")]
            HRESULT a([out, retval] BSTR *pVal);
        [propput, id(THREEPHASEDELTA_BASE+1), helpstring("property a")]
            HRESULT a([in] BSTR newVal);
        [propget, id(THREEPHASEDELTA_BASE+2), helpstring("property b")]
            HRESULT b([out, retval] BSTR *pVal);
        [propput, id(THREEPHASEDELTA_BASE+2), helpstring("property b")]
            HRESULT b([in] BSTR newVal);
        [propget, id(THREEPHASEDELTA_BASE+3), helpstring("property c")]
            HRESULT c([out, retval] BSTR *pVal);
        [propput, id(THREEPHASEDELTA_BASE+3), helpstring("property c")]
            HRESULT c([in] BSTR newVal);
    };
    [
        uuid(A7C625BF-C6E6-11D5-8476-00010214C4D0),
        helpstring("ThreePhaseDelta class"),
        noncreatable
    ]
    coclass ThreePhaseDelta
    {
        [default] interface IThreePhaseDelta;
    };
```

```
//ThreePhaseWye
    [
        object,
        uuid(A7C625BA-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IThreePhaseWye Interface"),
        pointer_default(unique)
    ]
    interface IThreePhaseWye : IConnection
    {
        enum {THREEPHASEWYE_BASE=(CONNECTION_BASE+256)};

        [propget, id(THREEPHASEWYE_BASE+1), helpstring("property n")]
            HRESULT n([out, retval] BSTR *pVal);
        [propput, id(THREEPHASEWYE_BASE+1), helpstring("property n")]
            HRESULT n([in] BSTR newVal);
        [propget, id(THREEPHASEWYE_BASE+2), helpstring("property a")]
            HRESULT a([out, retval] BSTR *pVal);
        [propput, id(THREEPHASEWYE_BASE+2), helpstring("property a")]
            HRESULT a([in] BSTR newVal);
        [propget, id(THREEPHASEWYE_BASE+3), helpstring("property b")]
            HRESULT b([out, retval] BSTR *pVal);
        [propput, id(THREEPHASEWYE_BASE+3), helpstring("property b")]
            HRESULT b([in] BSTR newVal);
        [propget, id(THREEPHASEWYE_BASE+4), helpstring("property c")]
            HRESULT c([out, retval] BSTR *pVal);
        [propput, id(THREEPHASEWYE_BASE+4), helpstring("property c")]
            HRESULT c([in] BSTR newVal);
    };
    [
        uuid(A7C625BA-C6E6-11D5-8476-00010214C4D0),
        helpstring("ThreePhaseWye class"),
        noncreatable
    ]
    coclass ThreePhaseWye
    {
        [default] interface IThreePhaseWye;
    };


//ThreePhaseSynchro
    [
        object,
        uuid(A7C625B6-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IThreePhaseSynchro Interface"),
        pointer_default(unique)
    ]
    interface IThreePhaseSynchro : IConnection
    {
        enum {THREEPHASESYNCHRO_BASE=(CONNECTION_BASE+256)};

        [propget, id(THREEPHASESYNCHRO_BASE+1), helpstring("property x")]
            HRESULT x([out, retval] BSTR *pVal);
        [propput, id(THREEPHASESYNCHRO_BASE+1), helpstring("property x")]
            HRESULT x([in] BSTR newVal);
        [propget, id(THREEPHASESYNCHRO_BASE+2), helpstring("property y")]
            HRESULT y([out, retval] BSTR *pVal);
        [propput, id(THREEPHASESYNCHRO_BASE+2), helpstring("property y")]
            HRESULT y([in] BSTR newVal);
        [propget, id(THREEPHASESYNCHRO_BASE+3), helpstring("property z")]
            HRESULT z([out, retval] BSTR *pVal);
        [propput, id(THREEPHASESYNCHRO_BASE+3), helpstring("property z")]
            HRESULT z([in] BSTR newVal);
    };
    [
        uuid(A7C625B6-C6E6-11D5-8476-00010214C4D0),
        helpstring("ThreePhaseSynchro class"),
        noncreatable
    ]
    coclass ThreePhaseSynchro
    {
```

```
        [default] interface IThreePhaseSynchro;
    };


//FourWireResolver
    [
        object,
        uuid(A7C625B1-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("IFourWireResolver Interface"),
        pointer_default(unique)
    ]
    interface IFourWireResolver : IConnection
    {
        enum {FOURWIRERESOLVER_BASE=(CONNECTION_BASE+256)};

        [propget, id(FOURWIRERESOLVER_BASE+1), helpstring("property s1")]
            HRESULT s1([out, retval] BSTR *pVal);
        [propput, id(FOURWIRERESOLVER_BASE+1), helpstring("property s1")]
            HRESULT s1([in] BSTR newVal);
        [propget, id(FOURWIRERESOLVER_BASE+2), helpstring("property s2")]
            HRESULT s2([out, retval] BSTR *pVal);
        [propput, id(FOURWIRERESOLVER_BASE+2), helpstring("property s2")]
            HRESULT s2([in] BSTR newVal);
        [propget, id(FOURWIRERESOLVER_BASE+3), helpstring("property s3")]
            HRESULT s3([out, retval] BSTR *pVal);
        [propput, id(FOURWIRERESOLVER_BASE+3), helpstring("property s3")]
            HRESULT s3([in] BSTR newVal);
        [propget, id(FOURWIRERESOLVER_BASE+4), helpstring("property s4")]
            HRESULT s4([out, retval] BSTR *pVal);
        [propput, id(FOURWIRERESOLVER_BASE+4), helpstring("property s4")]
            HRESULT s4([in] BSTR newVal);
    };
    [
        uuid(A7C625B1-C6E6-11D5-8476-00010214C4DC),
        helpstring("FourWireResolver class"),
        noncreatable
    ]
    coclass FourWireResolver
    {
        [default] interface IFourWireResolver;
    };


//SynchroResolver
    [
        object,
        uuid(A7C625AC-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ISynchroResolver Interface"),
        pointer_default(unique)
    ]
    interface ISynchroResolver : IConnection
    {
        enum {SYNCHRORESOLVER_BASE=(CONNECTION_BASE+256)};

        [propget, id(SYNCHRORESOLVER_BASE+1), helpstring("property r1")]
            HRESULT r1([out, retval] BSTR *pVal);
        [propput, id(SYNCHRORESOLVER_BASE+1), helpstring("property r1")]
            HRESULT r1([in] BSTR newVal);
        [propget, id(SYNCHRORESOLVER_BASE+2), helpstring("property r2")]
            HRESULT r2([out, retval] BSTR *pVal);
        [propput, id(SYNCHRORESOLVER_BASE+2), helpstring("property r2")]
            HRESULT r2([in] BSTR newVal);
        [propget, id(SYNCHRORESOLVER_BASE+3), helpstring("property r3")]
            HRESULT r3([out, retval] BSTR *pVal);
        [propput, id(SYNCHRORESOLVER_BASE+3), helpstring("property r3")]
            HRESULT r3([in] BSTR newVal);
        [propget, id(SYNCHRORESOLVER_BASE+4), helpstring("property r4")]
            HRESULT r4([out, retval] BSTR *pVal);
        [propput, id(SYNCHRORESOLVER_BASE+4), helpstring("property r4")]
            HRESULT r4([in] BSTR newVal);
```

```
    };
    [
        uuid(A7C625AC-C6E6-11D5-8476-00010214C4D0),
        helpstring("SynchroResolver class"),
        noncreatable
    ]
    coclass SynchroResolver
    {
        [default] interface ISynchroResolver;
    };


//Series
    [
        object,
        uuid(A7C625AA-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("ISeries Interface"),
        pointer_default(unique)
    ]
    interface ISeries : IConnection
    {
        enum {SERIES_BASE=(CONNECTION_BASE+256)};

        [propget, id(SERIES_BASE+1), helpstring("property via")]
            HRESULT via([out, retval] BSTR *pVal);
        [propput, id(SERIES_BASE+1), helpstring("property via")]
            HRESULT via([in] BSTR newVal);
    };
    [
        uuid(A7C625AA-C6E6-11D5-8476-00010214C4D0),
        helpstring("Series class"),
        noncreatable
    ]
    coclass Series
    {
        [default] interface ISeries;
    };


//NonElectrical
    [
        object,
        uuid(A7C625A7-C6E6-11D5-8476-00010214C4DC),
        dual,
        helpstring("INonElectrical Interface"),
        pointer_default(unique)
    ]
    interface INonElectrical : IConnection
    {
        enum {NONELECTRICAL_BASE=(CONNECTION_BASE+256)};

        [propget, id(NONELECTRICAL_BASE+1), helpstring("property to")]
            HRESULT to([out, retval] BSTR *pVal);
        [propput, id(NONELECTRICAL_BASE+1), helpstring("property to")]
            HRESULT to([in] BSTR newVal);
        [propget, id(NONELECTRICAL_BASE+2), helpstring("property from")]
            HRESULT from([out, retval] BSTR *pVal);
        [propput, id(NONELECTRICAL_BASE+2), helpstring("property from")]
            HRESULT from([in] BSTR newVal);
    };
    [
        uuid(A7C625A7-C6E6-11D5-8476-00010214C4D0),
        helpstring("NonElectrical class"),
        noncreatable
    ]
    coclass NonElectrical
    {
        [default] interface INonElectrical;
    };
```

```
//DigitalBus
    [
        object,
        uuid(76E5A6AC-C37F-4ECE-A19D-0E83F2E7E42F),
        dual,
        helpstring("IDigitalBus Interface"),
        pointer_default(unique)
    ]
    interface IDigitalBus : IConnection
    {
        enum {DIGITALBUS_BASE=(CONNECTION_BASE+256)};
    };
    [
        uuid(76E5A6AC-C37F-4ECE-A19D-0E83F2E7E420),
        helpstring("DigitalBus class"),
        noncreatable
    ]
    coclass DigitalBus
    {
        [default] interface IDigitalBus;
    };


};
```

Annex E

(informative)

Test signal framework (TSF) for ATLAS

E.1 Introduction

This annex provides a TSF representing most of the signals defined in IEEE Std 716-1995. 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 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 value are omitted.

Where examples are given, their static signal description is provided using the XML defined in Annex J.

E.2 AC_SIGNAL<type: Current|| Power|| Voltage>

E.2.1 Definition

A sinusoidal time-varying electrical signal. See Figure E.1.

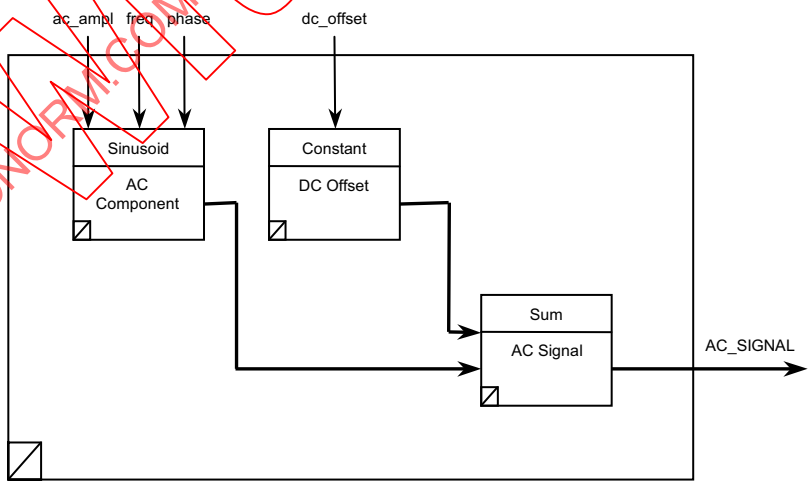


Figure E.1—TSF AC_SIGNAL

E.2.2 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 – 2 rad

E.2.3 Notes

E.2.4 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.2.5 Rules

For this signal, the allowable types are voltage, current, and power. All types must be consistent. Thus for example, if ac signal amplitude is specified in volts, then the dc offset must also be specified in volts.

E.2.6 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" />
```

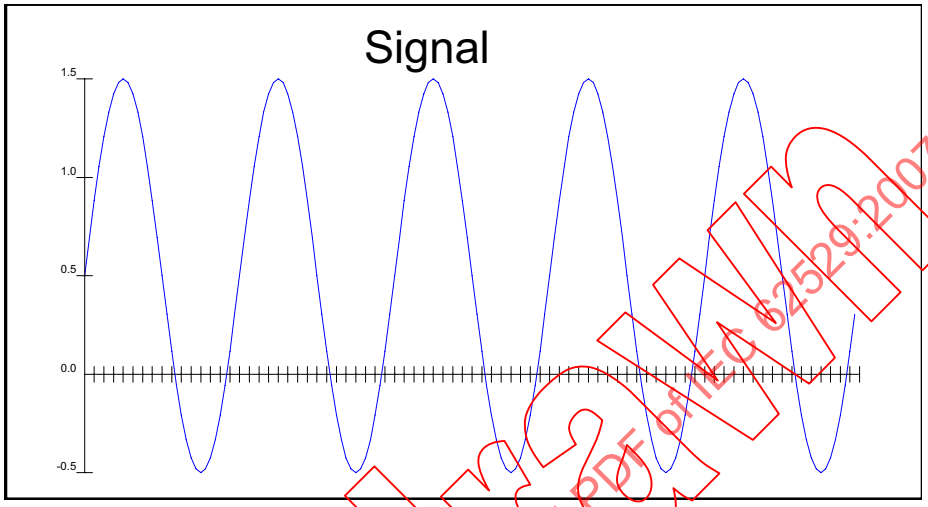


Figure E.2—AC_SIGNAL example

E.3 AM_SIGNAL

E.3.1 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.

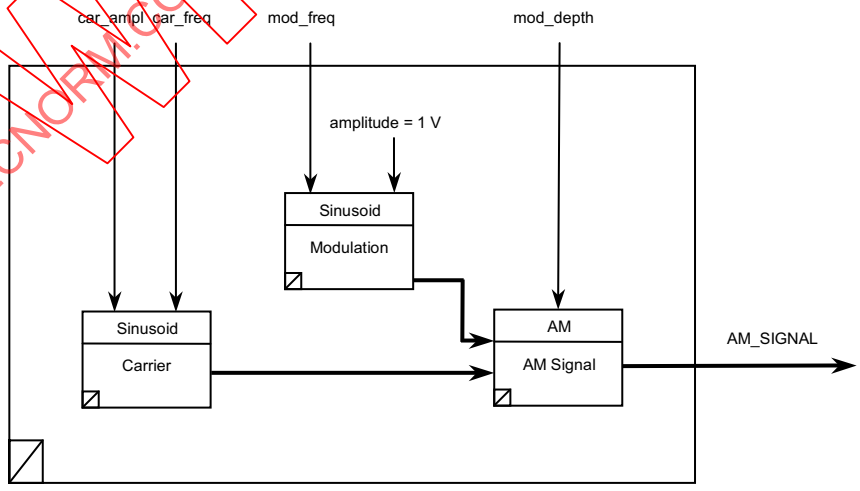


Figure E.3—TSF AM_SIGNAL

E.3.2 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

E.3.3 Notes

An interface for modulation amplitude is not required. The value of modulation depth provides all the necessary information required for this model (see E.3.5).

E.3.4 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			1 V (see note)
		frequency	mod_freq		
		phase			0 rad
NOTE—The BSC requires a unity value for the amplitude of the modulating signal.					

E.3.5 Rules

The output is given by the following equation:

$$e = E_c(1 + m_a \sin(\omega_m t))\sin(\omega_c t)$$

where

E_c is the carrier amplitude (unmodulated);

m_a is the depth of modulation (≡ modulation index);

ω_m is 2π × modulating frequency;

ω_c is 2π × carrier frequency.

E.3.6 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" />
```

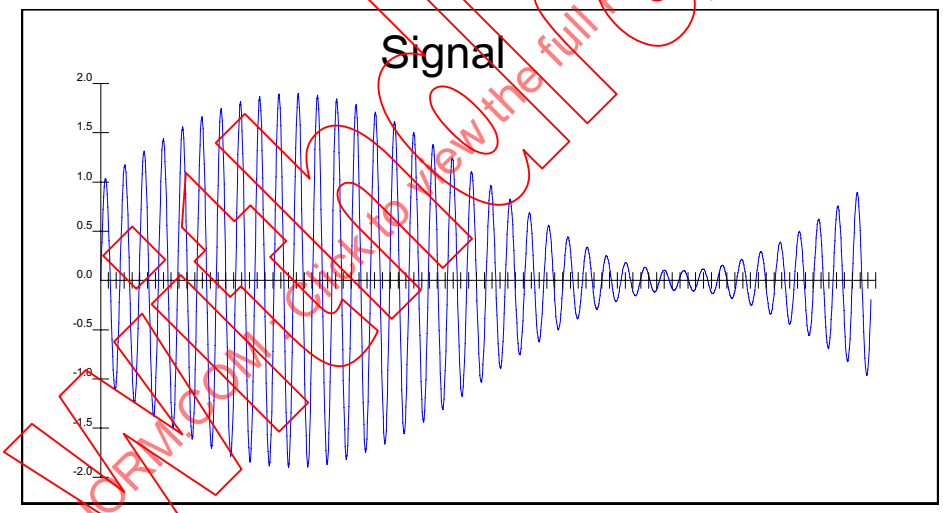


Figure E.4—AM_SIGNAL example

E.4 DC_SIGNAL<type: Voltage|| Current|| Power>

E.4.1 Definition

An unvarying electrical signal with an optional ac component. See Figure E.5.

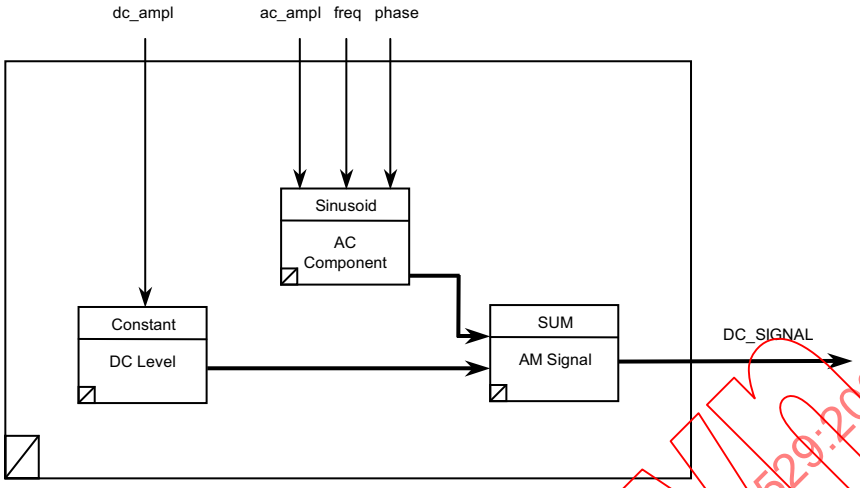


Figure E.5—TSF DC_SIGNAL

E.4.2 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 – 2 rad

E.4.3 Notes

E.4.4 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		
AC Component	Sinusoid	Signal [Out]		DC Signal	
		amplitude	ac_ampl		
		frequency	Freq		
		phase	Phase		

E.4.5 Rules

For this signal, the allowable types are voltage, current, and power. All types must be consistent. Thus for example, if dc level is specified in volts, then the ac component amplitude **must** also be specified in volts.

E.4.6 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" />
```

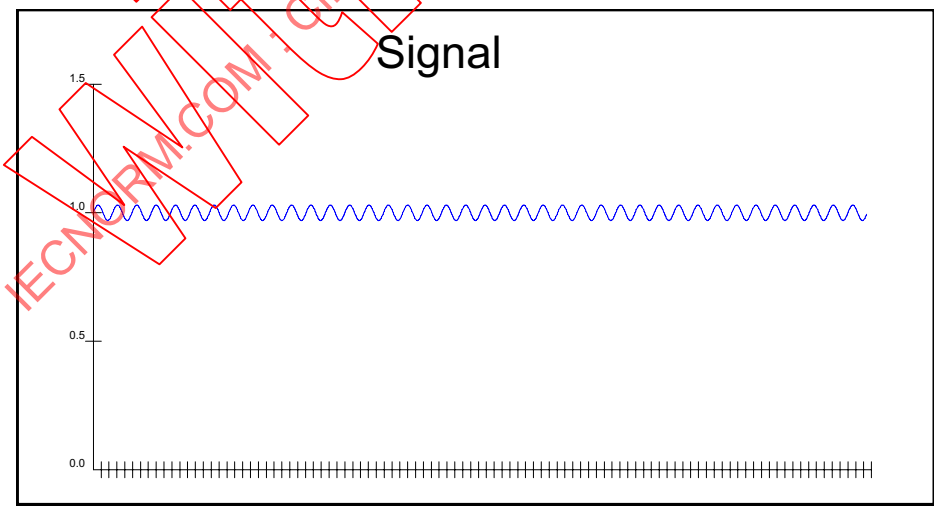


Figure E.6—DC_SIGNAL example

E.5 DIGITAL_PARALLEL

E.5.1 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.

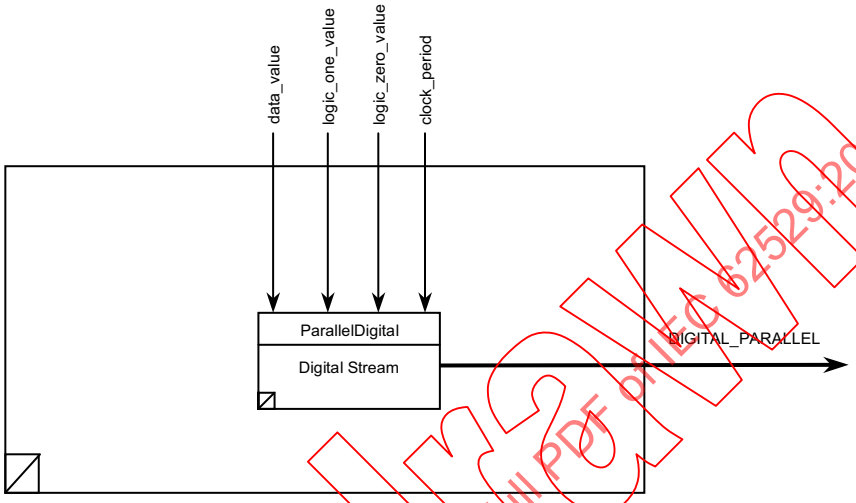


Figure E.7—TSF DIGITAL_PARALLEL

E.5.2 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	Array of Character String		H|L|Z|X
Clock period	clock_period	Time		
Logic One level	logic_one_value	Voltage		
Logic Zero level	logic_zero_value	Voltage		

E.5.3 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.5.4 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.5.5 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 will be available when the signal is used in a measurement.

E.5.6 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"
/>
```

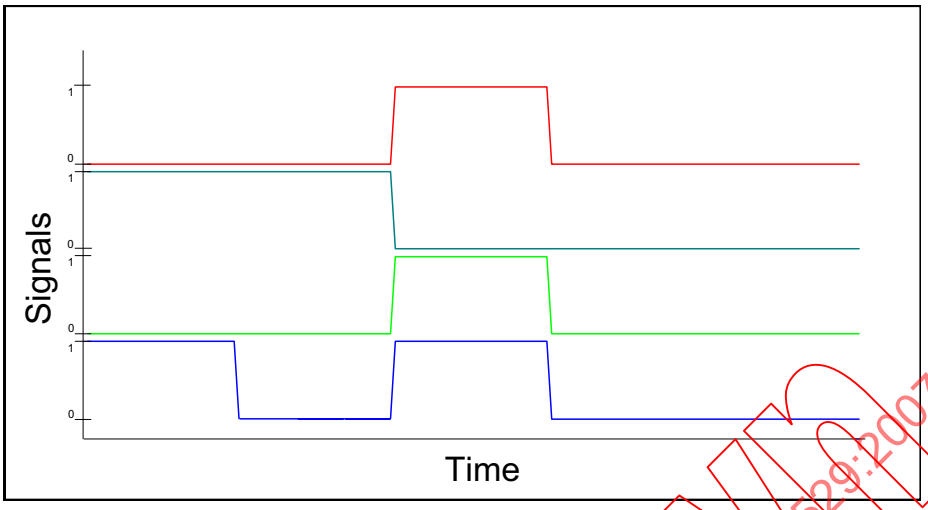


Figure E.8—DIGITAL_PARALLEL example

E.6 DIGITAL_SERIAL

E.6.1 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.

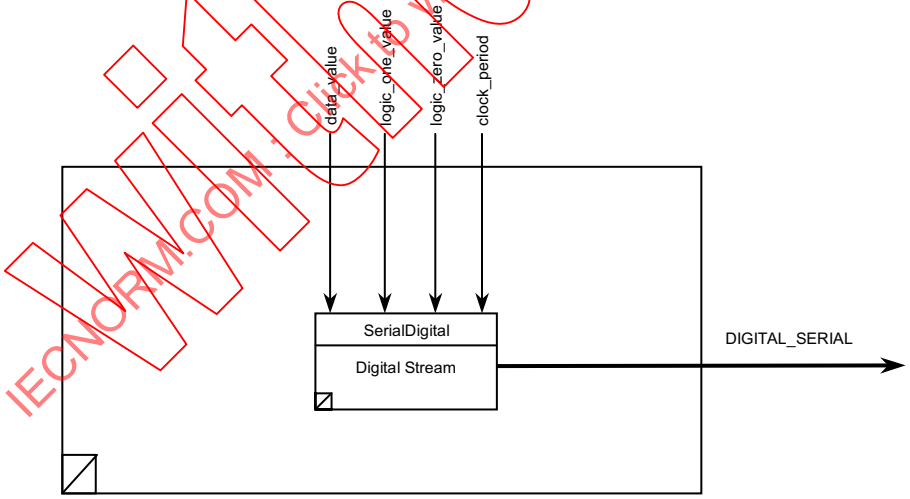


Figure E.9—TSF DIGITAL_SERIAL

E.6.2 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	Character String		H|L|Z|X
Clock period	clock_period	Time		
Logic One level	logic_one_value	Voltage		
Logic Zero level	logic_zero_value	Voltage		

E.6.3 Notes

The default condition for clock period (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.6.4 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
Digital 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.6.5 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 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 serial connections. Data received via the serial connections will be available when the signal is used in a measurement.

E.6.6 Example

See Figure E.10 for an example of DIGITAL_SERIAL.

XML Static Signal Description:

```
<DIGITAL_SERIAL data_value="LZHLHHLLHHHLLZL" clock_period="1 us"
logic_one_value=”3.6 V” logic_zero_value=”-2.6 V” />
```

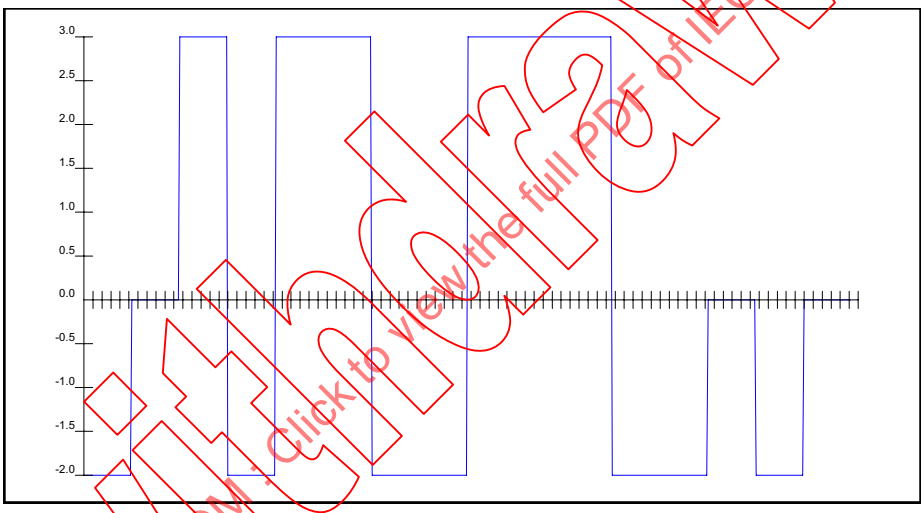


Figure E.10—DIGITAL_SERIAL example

E.7 DME_INTERROGATION

E.7.1 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.11.

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 µs (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 ultrahigh 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, making 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 always of 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.

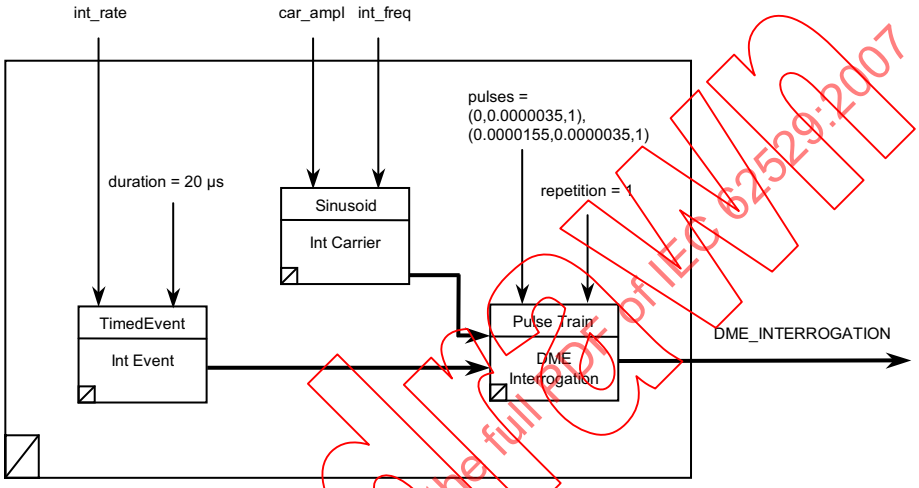


Figure E.11—TSF DME_INTERROGATION

E.7.2 Interface properties

See Table E.11 for details of the TSF DME_INTERROGATION interface.

Table E.11—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.7.3 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.7.4 Model description

See Table E.12 for details of the TSF DME_INTERROGATON model.

Table E.12—TSF DME_INTERROGATION model

Name	Type	Terminal	Inputs	Output	Formula
DME Interrogation	PulseTrain	Signal [Out]		DME_ INTERRO-GATION	
		pulses			(0,0.0000035,1), (0.0000155,0.0000035,1)
		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.7.5 Rules

E.7.6 Example

See Figure E.12 for an example of DME_INTERROGATION.

XML Static Signal Description:

```
<DME_INTERROGATION name="DME_INTERROGATION6" int_freq="1050 MHz"
int_rate="150 Hz" />
```

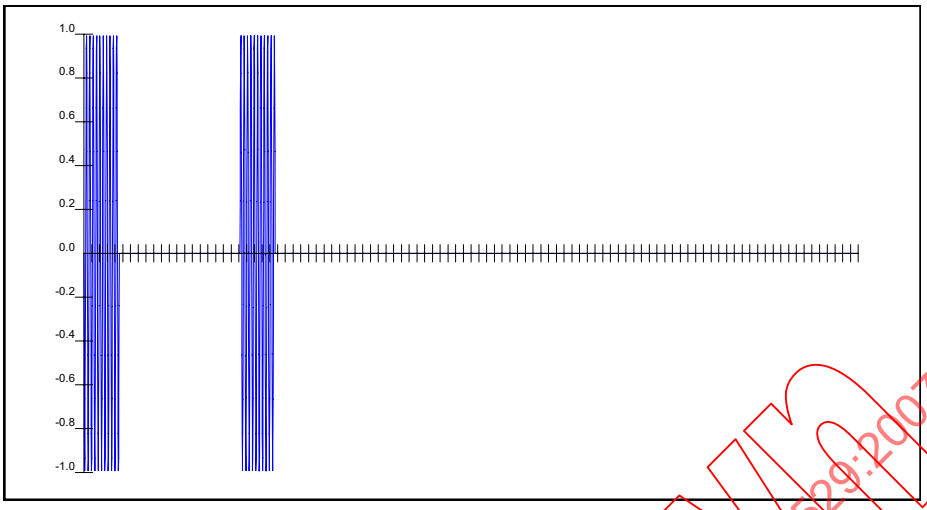


Figure E.12—DME_INTERROGATION example

E.8 DME_RESPONSE

E.8.1 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 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 µs (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, making 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 always of 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.

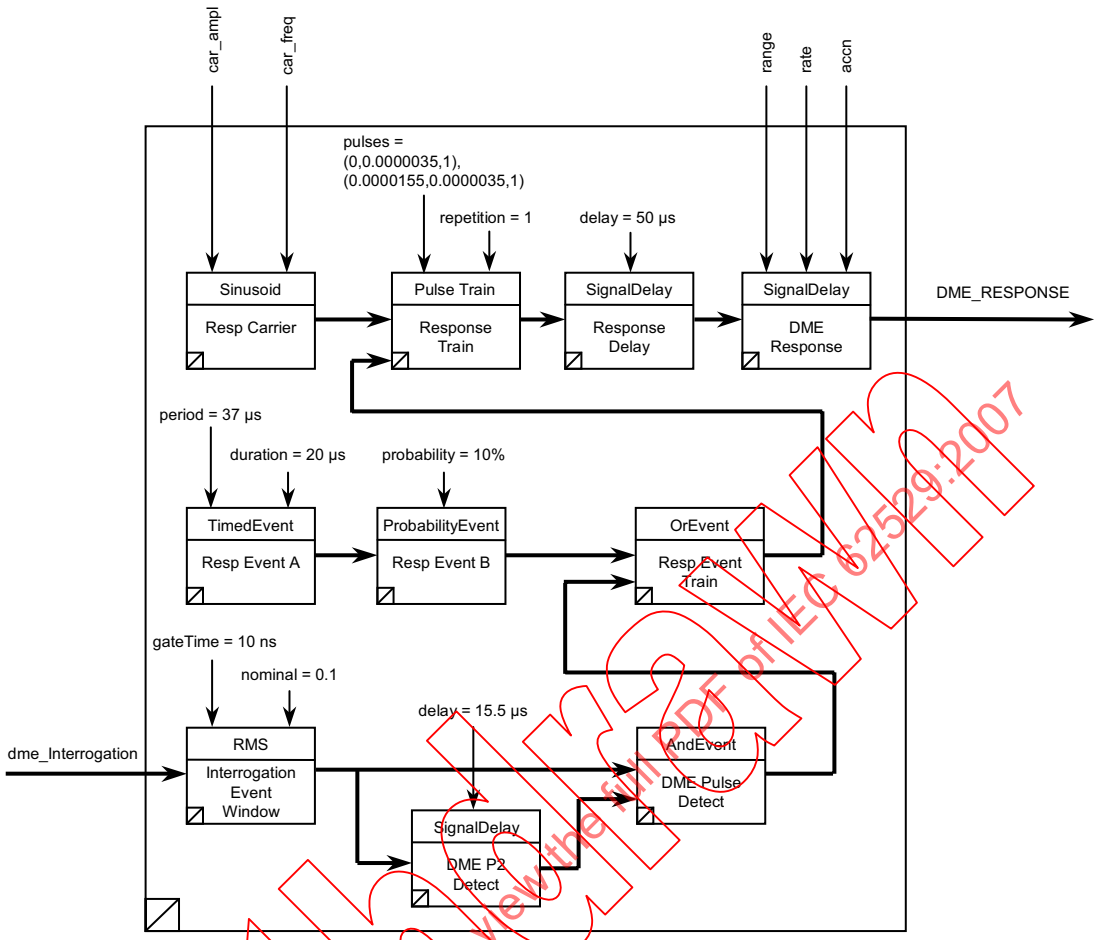


Figure E.13—TSF DME_RESPONSE

E.8.2 Interface properties

See Table E.13 for details of the TSF DME_RESPONSE interface.

Table E.13—TSF DME_RESPONSE interface

Description	Name	Type	Default	Range
Transponder Frequency	resp_freq	Frequency	962 MHz	962 MHz – 1213 MHz
Carrier Amplitude	car_ampl	Voltage		
Slant Range	range	Distance	0 m	
Range Rate	rate	Speed	0 m/s	
Rate of Change of Range Rate	accn	Acceleration	0 m/s^2	
DME Interrogation signal	dme_Interrogation	SignalFunction		

E.8.3 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 480 km (300 mi) up to an altitude of 23 000 m (75 000 ft). The delay range quoted will allow for a transponder transmission range of approximately 640 km (400 mi) and its lower value is 0 m (the default 50 µs usually allowed from receipt of an interrogator signal to the transponder response within the transponder itself). These values must not 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.8.4 Model description

See Table E.14 for details of the DME_RESPONSE model.

Table E.14—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%
		Signal [In]	Response Train		
Response Train	PulseTrain	Signal [Out]		Response Delay	
		pulses			(0,0.0000035,1), (0.0000155,0.0000035,1)
		repetition			1
		Signal [In]	Resp Carrier		
		Sync[In]	Resp Event Train		
Resp Carrier	Sinusoid	Signal [Out]		Response Train	
		amplitude	car_ampl		
		frequency	resp_freq		

Table E.14—TSF DME_RESPONSE model *(continued)*

Name	Type	Terminal	Inputs	Output	Formula
		phase			0 rad
Resp Event Train	OrEvent	Event [Out]		Response Train	
		Signal [In]	Resp Event B		
		Signal [In]	DME Pulse Detect		
Resp Event B	Probabili-tyEvent	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		
Interroga-tion Event Window	RMS	[Out]		DME Pulse Detect, DME P2 Detect	
		measurement			
		sample			
		count			
		gateTime			1.0e-8

Table E.14—TSF DME_RESPONSE model *(continued)*

Name	Type	Terminal	Inputs	Output	Formula
		nominal			0.1
		condition			GE
		NOGO			
		GO			
		UL			
		LL			
		Signal [In]	dme_ Interrogation		

E.8.5 Rules

E.8.6 Example

See Figure E.14 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" />
```

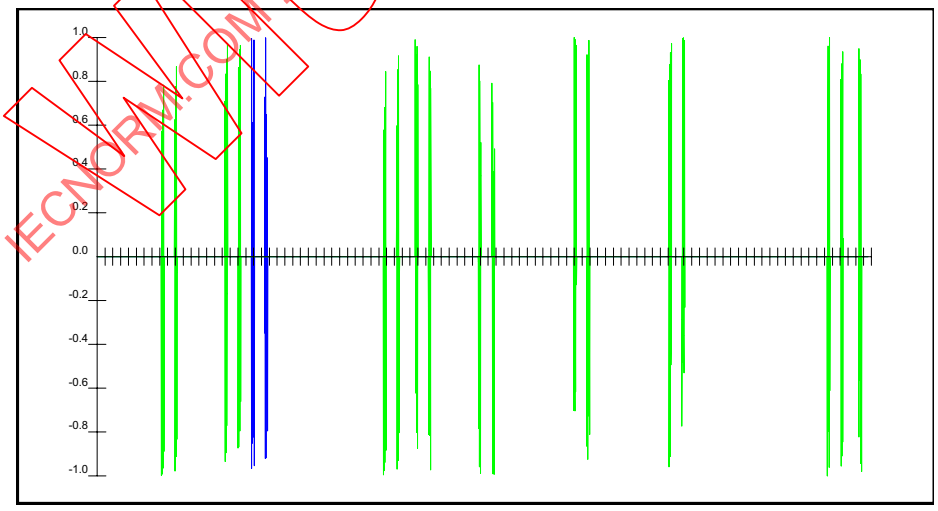


Figure E.14—DME_RESPONSE example

E.9 FM_SIGNAL<type: Voltage|| Power|| Current>

E.9.1 Definition

A continuous sinusoidal wave (carrier) generated when the frequency of one wave is varied in accordance with the amplitude of another wave (modulating). See Figure E.15.

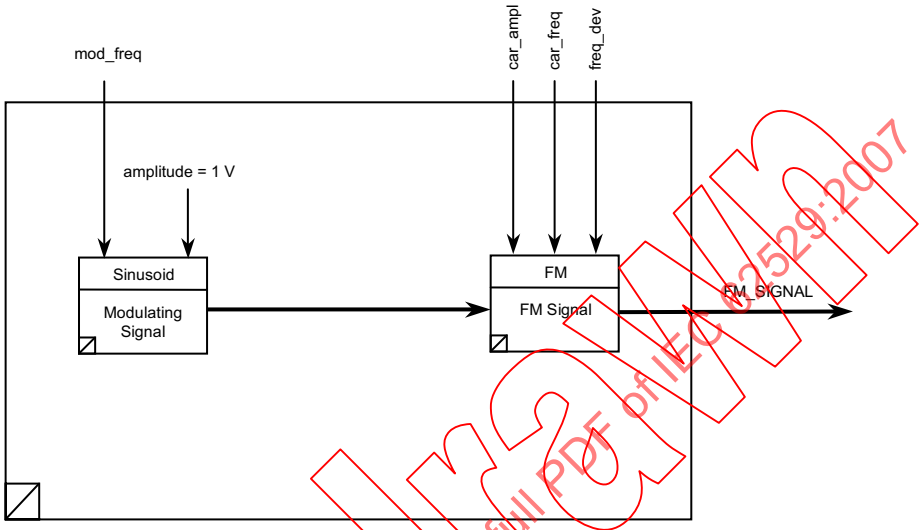


Figure E.15—TSF FM_SIGNAL

E.9.2 Interface properties

See Table E.15 for details of the TSF FM_SIGNAL interface.

Table E.15—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		

E.9.3 Notes

E.9.4 Model description

See Table E.16 for details of the TSF FM_SIGNAL model.

Table E.16—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			1 V (see note)
		frequency	mod_freq		
		phase			0 rad
NOTE—The BSC requires a unity value for the amplitude of the modulating signal.					

E.9.5 Rules

The output is given by the following equation:

$e = E_c \sin(\omega_c t + m_f \sin(\omega_m t))$

where

E_c is the carrier amplitude (unmodulated);

ω_c is 2π × carrier frequency;

m_f deviation ratio (≡ modulation index);

ω_m is 2π × modulating frequency.

E.9.6 Example

See Figure E.16 for an example of FM_SIGNAL.

XML Static Signal Description:

```
<FM_SIGNAL name="FM_SIGNAL9" car_freq="100kHz"
freq_dev="10kHz"mod_freq="1200Hz" />
```

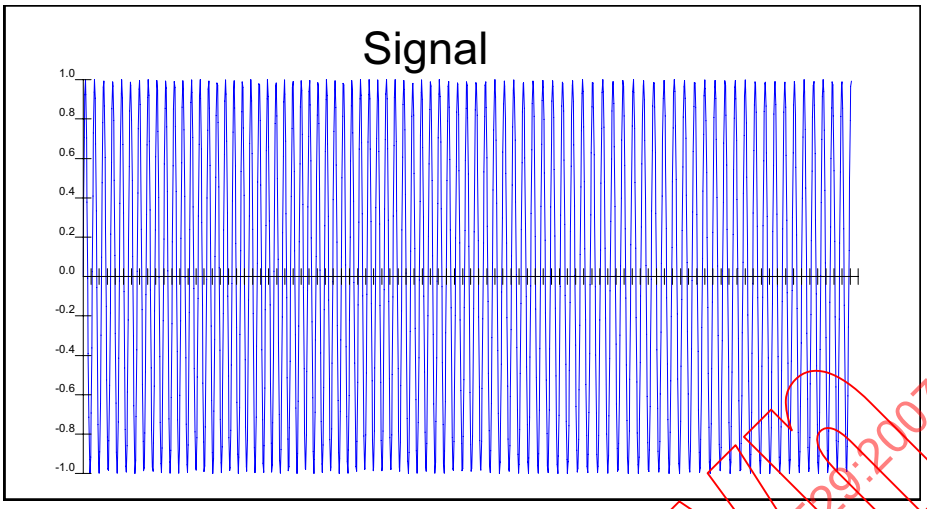


Figure E.16—FM_SIGNAL example

E.10 ILS_GLIDE_SLOPE<type: Voltage|| Power>

E.10.1 Definition

The glide slope is 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. This 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 will treat them as a single-carrier signal with two modulating tones. If the aircraft is positioned off the glide slope, the ILS receiver will detect one signal as stronger than the other. As a result, the demodulated amplitude (or apparent depth of modulation) of one tone will be greater than the tone of the other. If the receiver is exactly on the glide slope, it will 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 will be the difference in amplitude of the tones. See Figure E.17.

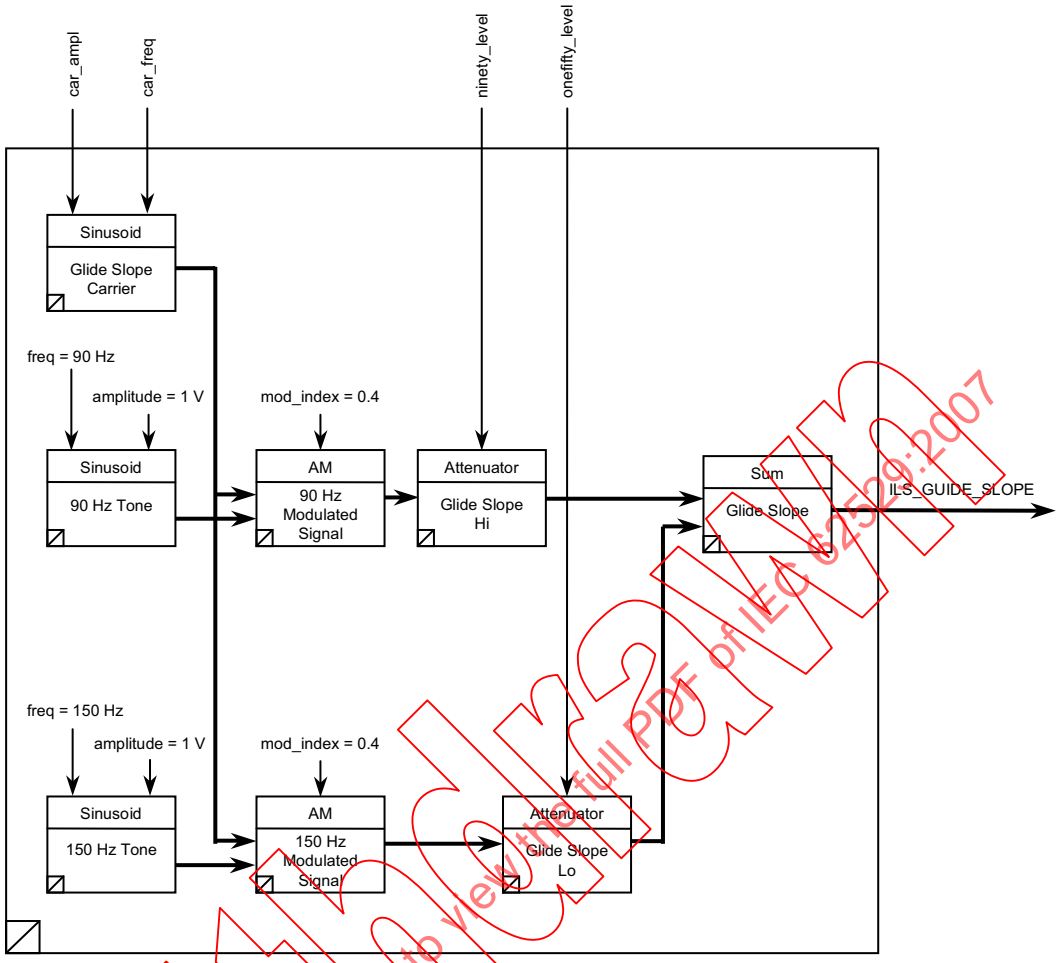


Figure E.17—TSF ILS_GLIDE_SLOPE

E.10.2 Interface properties

See Table E.17 for details of the TSF ILS_GLIDE_SLOPE interface.

Table E.17—TSF ILS_GLIDE_SLOPE interface

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Physical	2 mV	
Frequency	car_freq	Frequency	328.6 MHz	328.6–335.4 MHz
150 Hz attenuation depth	onefifty_level	Ratio	1	0–1
90 Hz attenuation depth	ninety_level	Ratio	1	0–1

E.10.3 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.10.4 Model description

See Table E.18 for details of the TSF ILS_GLIDE_SLOPE model.

Table E.18—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]	90 Hz Modulated Signal		
Glide Slope Lo	Attenuator	Signal [Out]		Glide Slope	
		Gain	onefifty_level		
		Signal [In]	150 Hz Modulated Signal		
90 Hz Modulated Signal	AM	Signal [Out]		Glide Slope Hi	
		modIndex			0.4
		Carrier [In]	Glide Slope Carrier		
		Signal [In]	90 Hz Tone		
150 Hz Modulated Signal	AM	Signal [Out]		Glide Slope Lo	
		modIndex			0.4
		Carrier [In]	Glide Slope Carrier		
		Signal [In]	150 Hz Tone		
Glide Slope Carrier	Sinusoid	Signal [Out]		150 Hz Modulated Signal, 90 Hz Modulated Signal	
		amplitude	car_ampl		
		frequency	car_freq		

Table E.18—TSF ILS_GLIDE_SLOPE model *(continued)*

Name	Type	Terminal	Inputs	Output	Formula
		Phase			0 rad
90Hz Tone	Sinusoid	Signal [Out]		90 Hz Modulated Signal	
		amplitude			1 V (see note)
		frequency			90 Hz
		Phase			0 rad
150Hz Tone	Sinusoid	Signal [Out]		150 Hz Modulated Signal	
		amplitude			1 V (see note)
		frequency			150 Hz
		Phase			0 rad
NOTE—The BSC requires a unity value for the amplitude of the modulating signal.					

E.10.5 Rules

For this signal, the allowable types for carrier amplitudes are voltage and power.

E.10.6 Example

See Figure E.18 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" />
```

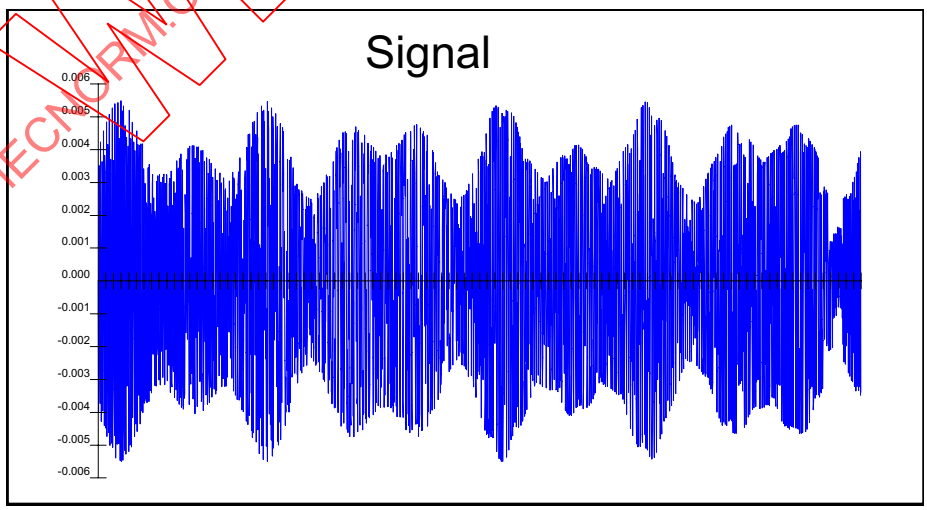


Figure E.18—ILS_GLIDE_SLOPE example

E.11 ILS_LOCALIZER<type: Power|| Voltage>

E.11.1 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.19.

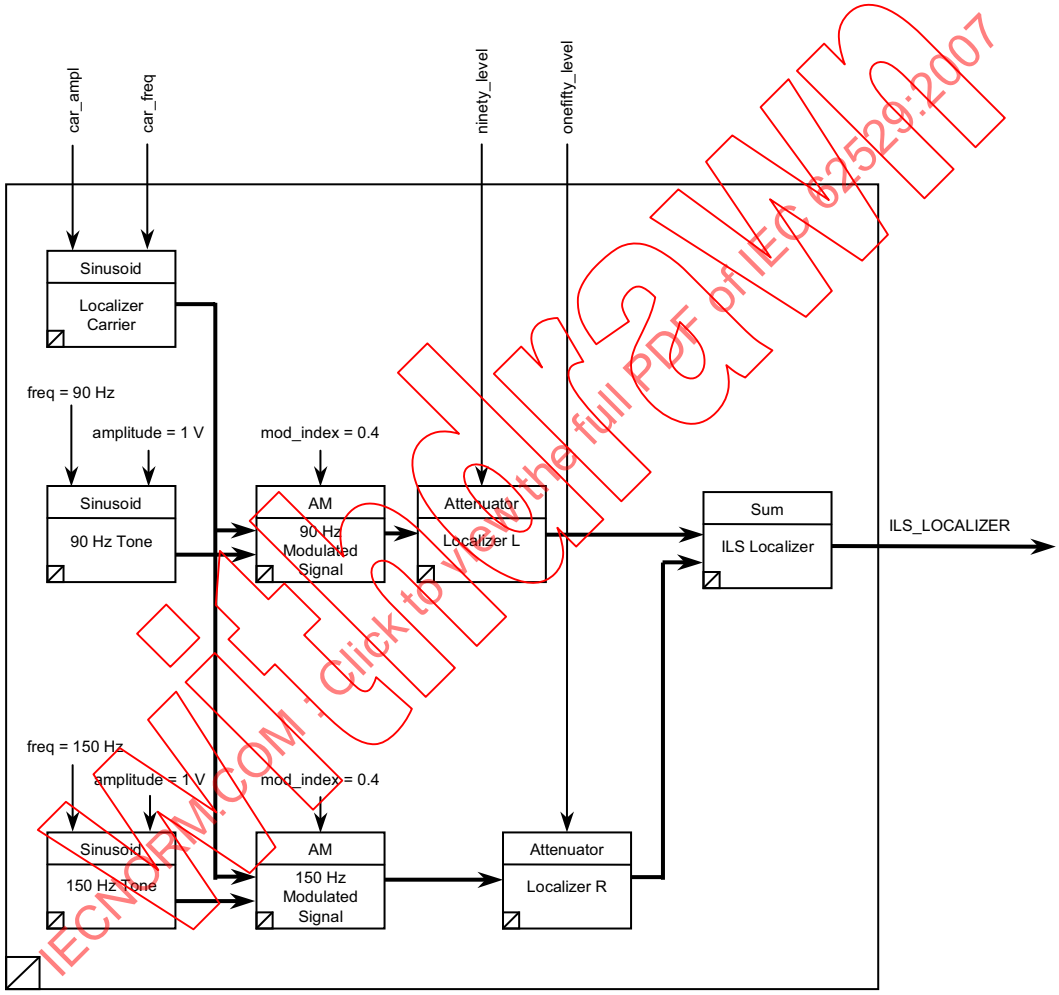


Figure E.19—TSF ILS_LOCALIZER

E.11.2 Interface properties

See Table E.19 for details of the TSF ILS_LOCALIZER interface.

Table E.19—TSF ILS_LOCALIZER interface

Description	Name	Type	Default	Range
Carrier Amplitude	car_ampl	Physical	2 mW	
Carrier frequency	car_freq	Frequency	108.1 MHz	108.1–111.9 MHz
150Hz attenuation depth	onefifty_level	Ratio	1	0–1
90Hz attenuation depth	ninety_level	Ratio	1	0–1

E.11.3 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.11.4 Model description

See Table E.20 for details of the TSF ILS_LOCALIZER model.

Table E.20—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]	150 Hz Modulated Signal		
Localizer L	Attenuator	Signal [Out]		ILS Localizer	
		gain	ninety_level		
		Signal [In]	90 Hz Modulated Signal		
150 Hz Modulated Signal	AM	Signal [Out]		Localizer R	
		modIndex			0.2
		Carrier [In]	Localizer Carrier		
		Signal [In]	150 Hz Tone		
90 Hz Modulated Signal	AM	Signal [Out]		Localizer L	

Table E.20—TSF ILS_LOCALIZER model *(continued)*

Name	Type	Terminal	Inputs	Output	Formula
		modIndex			0.2
		Carrier [In]	Localizer Carrier		
		Signal [In]	90 Hz Tone		
150Hz Tone	Sinusoid	Signal [Out]		150 Hz Modulated Signal	
		amplitude			1 V (see note)
		frequency			150 Hz
		phase			0 rad
90Hz Tone	Sinusoid	Signal [Out]		90 Hz Modulated Signal	
		amplitude			1 V (see note)
		frequency			90 Hz
		phase			0 rad
Localizer Carrier	Sinusoid	Signal [Out]		90 Hz Modulated Signal, 150 Hz 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.11.5 Rules

For this signal, the allowable types for carrier amplitudes are voltage and power.

E.11.6 Example

See Figure E.20 for an example of ILS_LOCALIZER.

XML Static Signal Description:

```
<ILS_LOCALIZER name="ILS_LOCALIZER6" ninety_level="0.9"
 onefifty_level="1.1" />
```

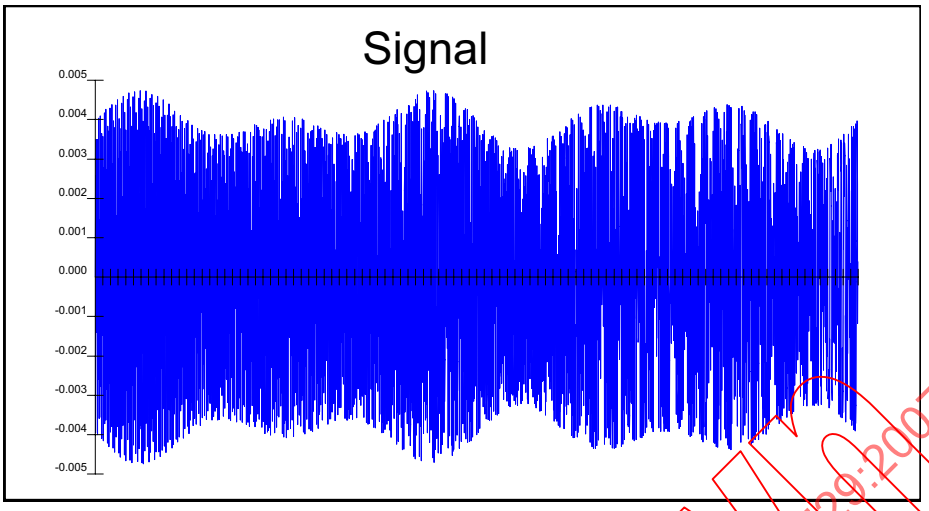


Figure E.20—ILS_LOCALIZER example

E.12 ILS_MARKER

E.12.1 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 6 km to 11 km (4 mi to 7 mi) 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 1150 m (3500 ft) 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 30 m (100 ft) 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 30 m (100 ft) and 400 m (1300 ft) visibility. See Figure E.21.

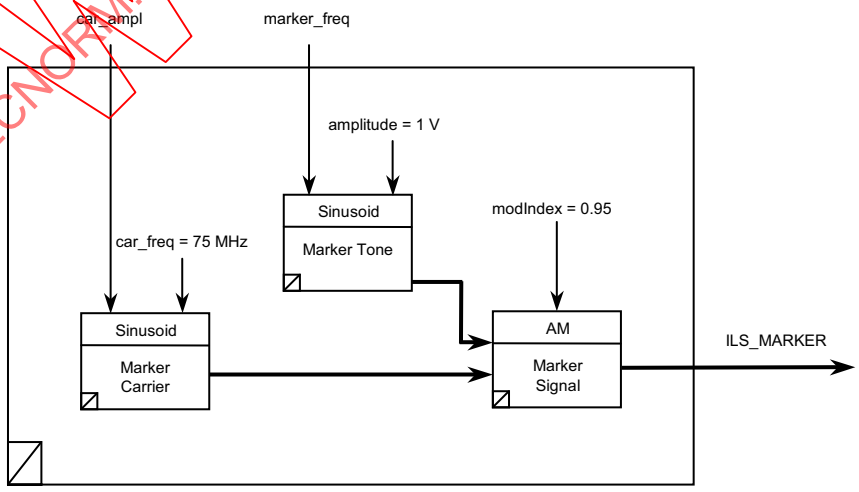


Figure E.21—TSF ILS_MARKER

E.12.2 Interface properties

See Table E.21 for details of the TSF ILS_MARKER interface.

Table E.21—TSF ILS_MARKER interface

Description	Name	Type	Default	Range
Marker Frequency	marker_freq	Frequency	400 Hz	400 Hz | 1.3 kHz | 3 kHz
Carrier Frequency	car_ampl	Power	2 mW	

E.12.3 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.12.4 Model description

See Table E.22 for details of the TSF ILS_MARKER model.

Table E.22—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 V (see note)
		frequency	marker_freq		
		phase			0 rad
NOTE—The BSC requires a unity value for the amplitude of the modulating signal.					

E.12.5 Rules

For this signal, the carrier amplitudes can be expressed only in terms of power.

E.12.6 Example

See Figure E.22 for an example of ILS_MARKER.

XML Static Signal Description:

```
<ILS_MARKER name="ILS_MARKER5" />
```

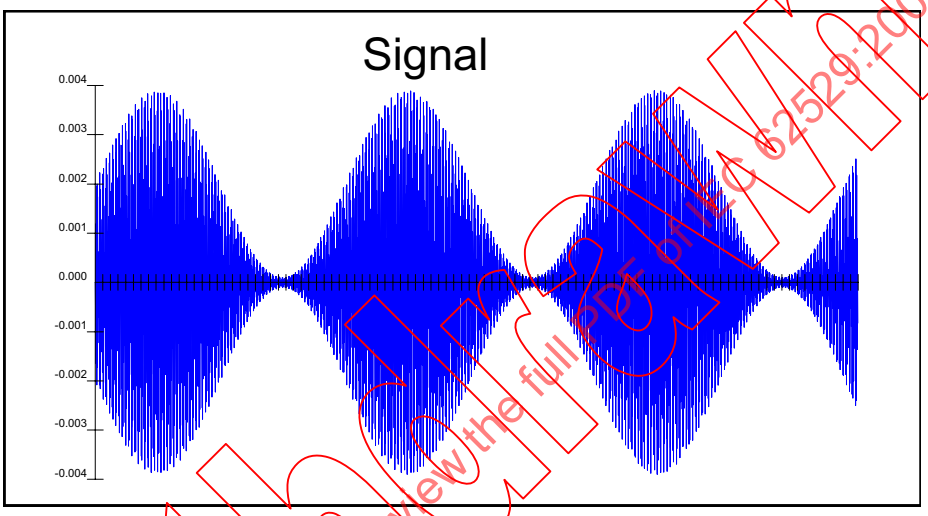


Figure E.22—ILS_MARKER example

E.13 PM_SIGNAL

E.13.1 Definition

A continuous sinusoidal wave (carrier) whose phase is varied in accordance with the amplitude of another wave. See Figure E.23.

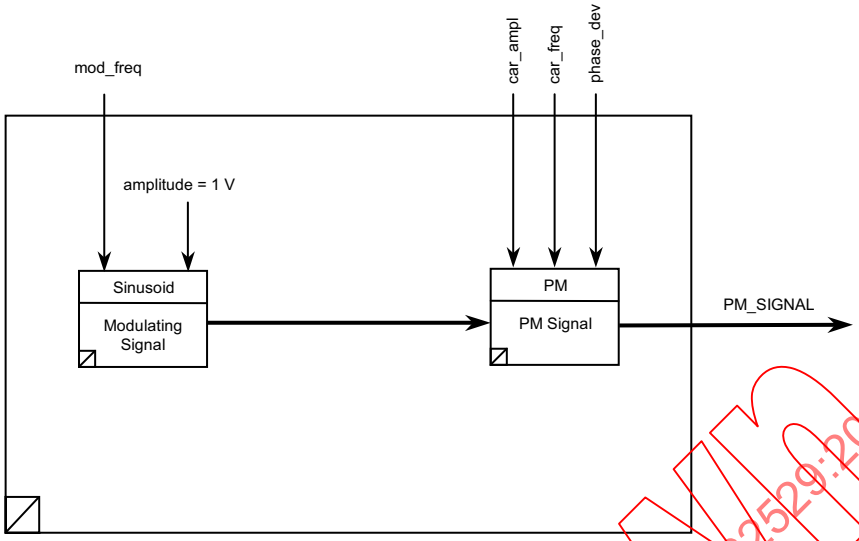


Figure E.23—TSF PM_SIGNAL

E.13.2 Interface properties

See Table E.23 for details of the TSF PM_SIGNAL interface.

Table E.23—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		

E.13.3 Notes

E.13.4 Model description

See Table E.24 for details of the TSF PM_SIGNAL model.

Table E.24—TSF PM_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
PM Signal	PM	Signal [Out]		PM_SIGNAL	
		amplitude	car_ampl		

Table E.24—TSF PM_SIGNAL model *(continued)*

Name	Type	Terminal	Inputs	Output	Formula
		carrierFrequency	car_freq		
		phaseDeviation	phase_dev		
		Signal [In]	Modulating Signal		
Modulating Signal	Sinusoid	Signal [Out]		PM Signal	
		amplitude			1 V (see note)
		frequency	mod_freq		
		phase			0 rad
NOTE—The BSC requires a unity value for the amplitude of the modulating signal.					

E.13.5 Rules

E.13.6 Example

See Figure E.24 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" />
```

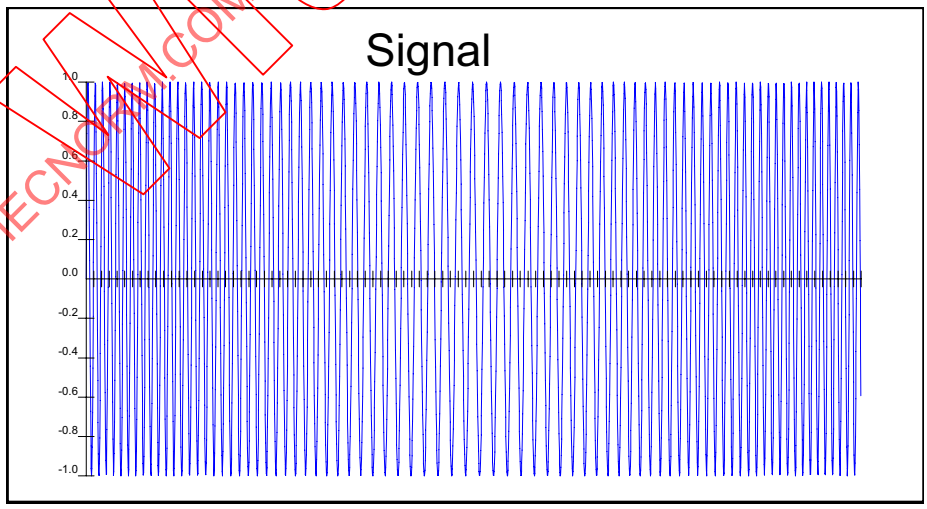


Figure E.24—PM_SIGNAL example

E.14 PULSED_AC_SIGNAL<type: Current|| Power|| Voltage>

E.14.1 Definition

A signal characterized by short duration periods of (sinusoidal) ac electrical potential. See Figure E.25.

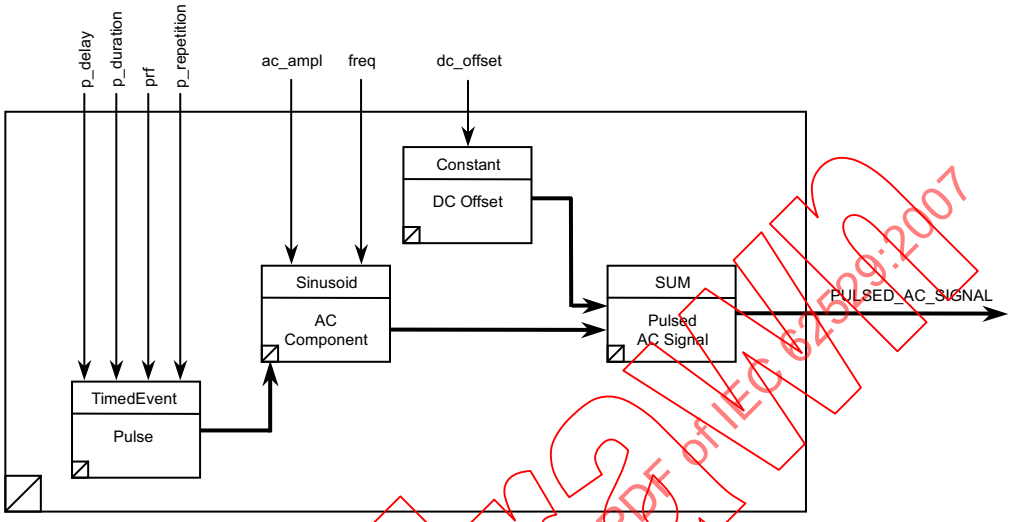


Figure E.25—TSF PULSED_AC_SIGNAL

E.14.2 Interface properties

See Table E.25 for details of the TSF PULSED_AC_SIGNAL interface.

Table E.25—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	Numeric–Integer	0	

E.14.3 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 the pulses (only) have a dc offset.

E.14.4 Model description

See Table E.26 for details of the TSF PULSED_AC_SIGNAL model.

Table E.26—TSF PULSED_AC_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
Pulsed AC Signal	Sum	Signal [Out]		PULSED_AC_SIGNAL	
		Signal [In]	DC Offset		
		Signal [In]	AC Component		
AC Component	Sinusoid	Signal [Out]		Pulsed AC Signal	
		amplitude	ac_ampl		
		frequency	freq		
		phase			0 rad
		Gate [In]	Pulse		
DC Offset	Constant	Signal [Out]		Pulsed AC Signal	
		amplitude	dc_offset		
Pulse	TimedEvent	Event [Out]		AC Component	
		delay	p_delay		
		duration	p_duration		
		period			1/prf
		repetition	p_repetition		

E.14.5 Rules

For this signal, the allowable types are voltage, current, and power. All types must be consistent. Thus, for example, if the ac signal amplitude is specified in volts, then the dc offset **must** also be specified in volts.

E.14.6 Example

See Figure E.26 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" />
```

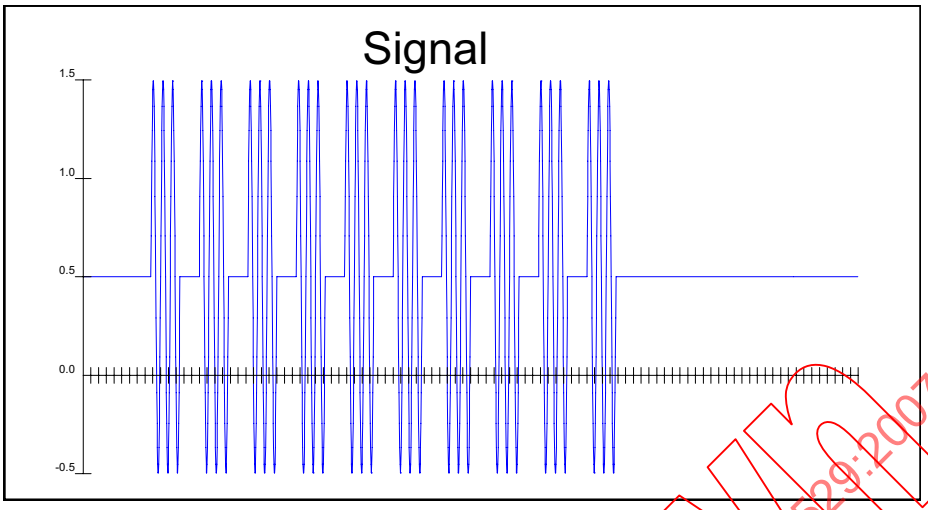


Figure E.26—PULSED_AC_SIGNAL example

E.15 PULSED_AC_TRAIN<type: Voltage|| Current|| Power>

E.15.1 Definition

A signal, characterized by a train of pulses of sinusoidal electrical ac activity with different durations and amplitudes. See Figure E.27.

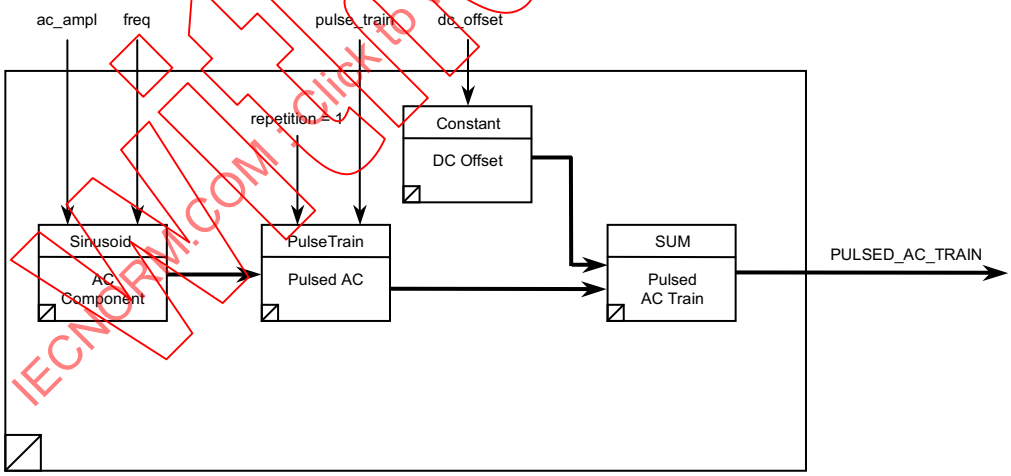


Figure E.27—TSF PULSED_AC_TRAIN

E.15.2 Interface properties

See Table E.27 for details of the TSF PULSED_AC_TRAIN interface.

Table E.27—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.15.3 Notes

This model represents a pulsed ac train with a permanent dc offset. An alternative model may be created where the pulses (only) have a dc offset.

E.15.4 Model description

See Table E.28 for details of the TSF PULSED_AC_TRAIN model.

Table E.28—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]	DC Offset		
Pulsed AC	PulseTrain	Signal [Out]		Pulsed AC Train	
		pulses	pulse_train		
		repetition			1
		Signal [In]	AC Component		
DC Offset	Constant	Signal [Out]		Pulsed AC Train	
		amplitude	dc_offset		
AC Component	Sinusoid	Signal [Out]		Pulsed AC	
		amplitude	ac_ampl		
		frequency	freq		
		phase			0 rad

E.15.5 Rules

For this signal, the allowable types are voltage, current, and power. All types must be consistent. Thus, for example, if ac amplitude is specified in volts, then the dc offset **must** also be specified in volts.

E.15.6 Example

See Figure E.28 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)" />
```

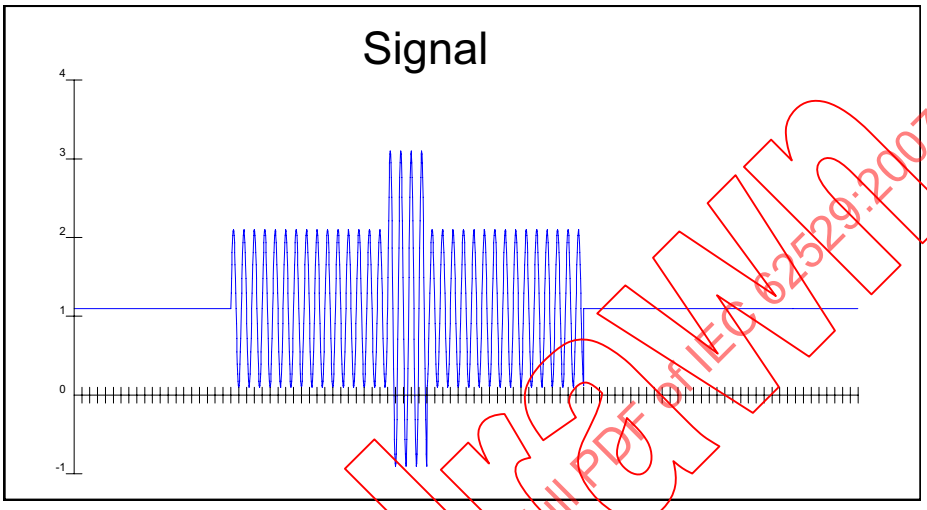


Figure E.28—PULSED_AC_TRAIN example

E.16 PULSED_DC_SIGNAL<type: Voltage|| Current|| Power>

E.16.1 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.29.

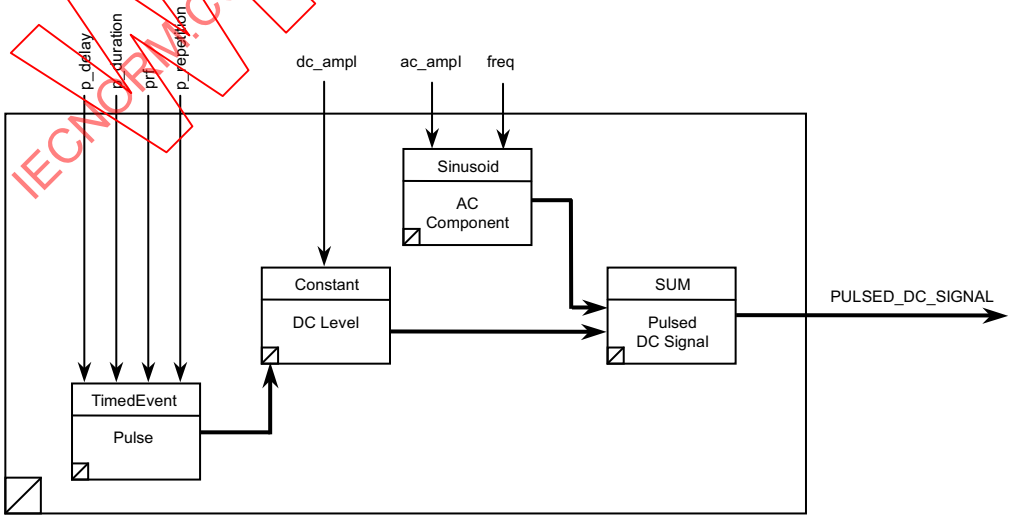


Figure E.29—TSF PULSED_DC_SIGNAL

E.16.2 Interface properties

See Table E.29 for details of the TSF PULSED_DC_SIGNAL interface.

Table E.29—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	Numeric–Integer	0	

E.16.3 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 the pulses (only) had an ac component.

E.16.4 Model description

See Table E.30 for details of the TSF PULSED_DC_SIGNAL model.

Table E.30—TSF PULSED_DC_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
Pulsed DC Signal	Sum	Signal [Out]		PULSED_DC_SIGNAL	
		Signal [In]	DC Level		
		Signal [In]	AC Component		
DC Level	Constant	Signal [Out]		Pulsed DC Signal	
		amplitude	dc_ampl		
		Gate[In]	Pulse		
AC Component	Sinusoid	Signal [Out]		Pulsed DC Signal	
		amplitude	ac_ampl		
		frequency	freq		
		phase			0 rad

Table E.30—TSF PULSED_DC_SIGNAL model *(continued)*

Name	Type	Terminal	Inputs	Output	Formula
Pulse	TimedEvent	Event [Out]		DC Level	
		delay	p_delay		
		duration	p_duration		
		period			1/prf
		repetition	p_repetition		

E.16.5 Rules

For this signal, the allowable types are voltage, current, and power. All types must be consistent. Thus, for example, if a dc level is specified in volts, then the ac component amplitude **must** also be specified in volts.

E.16.6 Example

See Figure E.30 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" />
```

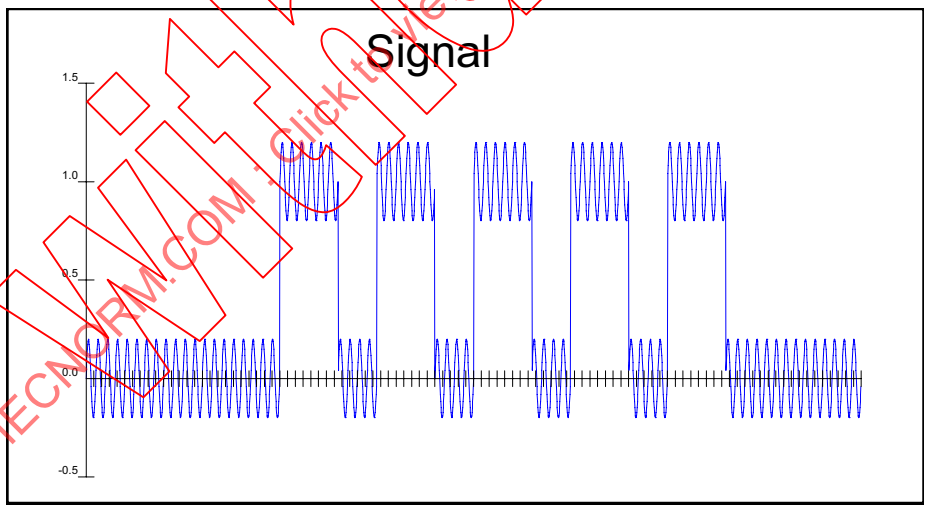


Figure E.30—PULSED_DC_SIGNAL example

E.17 PULSED_DC_TRAIN<type: Voltage|| Current|| Power>

E.17.1 Definition

A signal, characterized by a train of different, short duration periods of dc electrical activity. See Figure E.31.

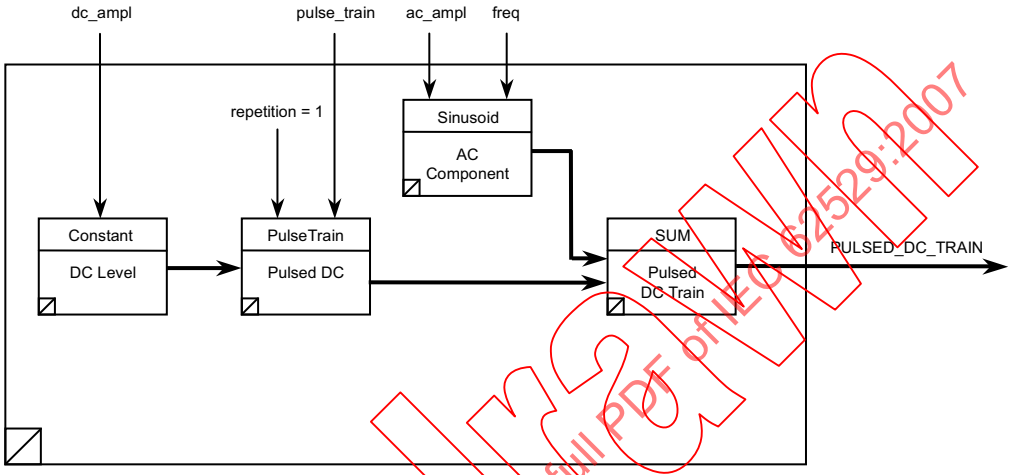


Figure E.31—TSF PULSED_DC_TRAIN

E.17.2 Interface properties

See Table E.31 for details of the TSF PULSED_DC_TRAIN interface.

Table E.31—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.17.3 Notes

For this signal, the allowable types are voltage, current, and power. All types must be consistent. Thus, for example, if dc level is specified in volts, then the ac component amplitude **must** 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 the pulses (only) had an ac component.

E.17.4 Model description

See Table E.32 for details of the TSF PULSED_DC_TRAIN model.

Table E.32—TSF PULSED_DC_TRAIN model

Name	Type	Terminal	Inputs	Output	Formula
Pulsed DC Train	Sum	Signal [Out]		PULSED_DC_TRAIN	
		Signal [In]	Pulsed DC		
		Signal [In]	AC Component		
Pulsed DC	PulseTrain	Signal [Out]		Pulsed DC Train	
		pulses	pulse_train		
		repetition			1
		Signal [In]	DC Level		
AC Component	Sinusoid	Signal [Out]		Pulsed DC Train	
		amplitude	ac_ampl		
		frequency	freq		
		phase			0 rad
DC Level	Constant	Signal [Out]		Pulsed DC	
		amplitude	dc_ampl		

E.17.5 Rules

E.17.6 Example

See Figure E.32 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)" />
```

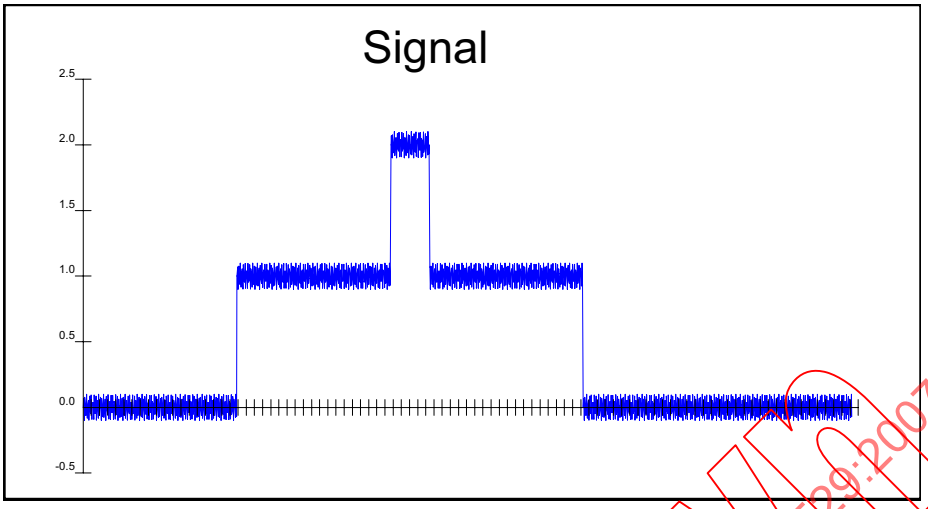


Figure E.32—PULSED_DC_TRAIN example

E.18 RADAR_RX_SIGNAL

E.18.1 Definition

An appropriate delayed signal response to an input radar signal. See Figure E.33.

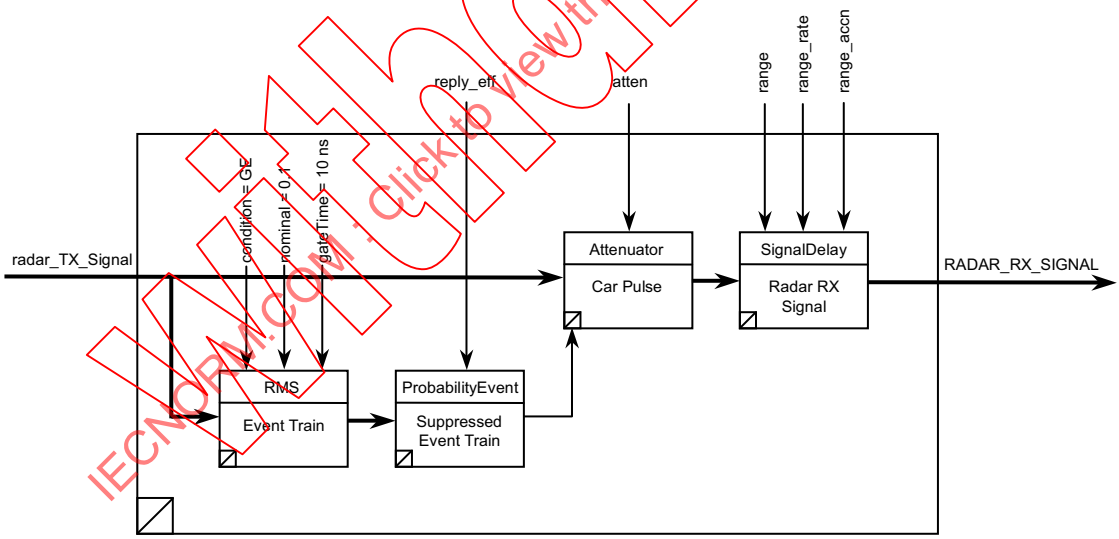


Figure E.33—TSF RADAR_RX_SIGNAL

E.18.2 Interface properties

See Table E.33 for details of the TSF RADAR_TX_SIGNAL interface.

Table E.33—TSF RADAR_RX_SIGNAL interface

Description	Name	Type	Default	Range
Atten	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.18.3 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 must 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.18.4 Model description

See Table E.34 for details of the TSF RADAR_RX_SIGNAL model.

Table E.34—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		

Table E.34—TSF RADAR_RX_SIGNAL model *(continued)*

Name	Type	Terminal	Inputs	Output	Formula
		Gate[In]	Suppressed Event Train		
Suppressed Event Train	ProbabilityEvent	Event [Out]		Car Pulse	
		seed			0
		probability	reply_eff		
		Signal [In]	Event Train		
Event Train	RMS	[Out]		Suppressed Event Train	
		measurement			
		sample			
		count			
		gateTime			1.0e-8
		nominal			0.1
		condition			GE
		NOGO			
		GO			
		UL			
		LL			
		Signal [In]	radar_TX_Signal		

E.18.5 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.18.6 Example

See Figure E.34 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" />
```

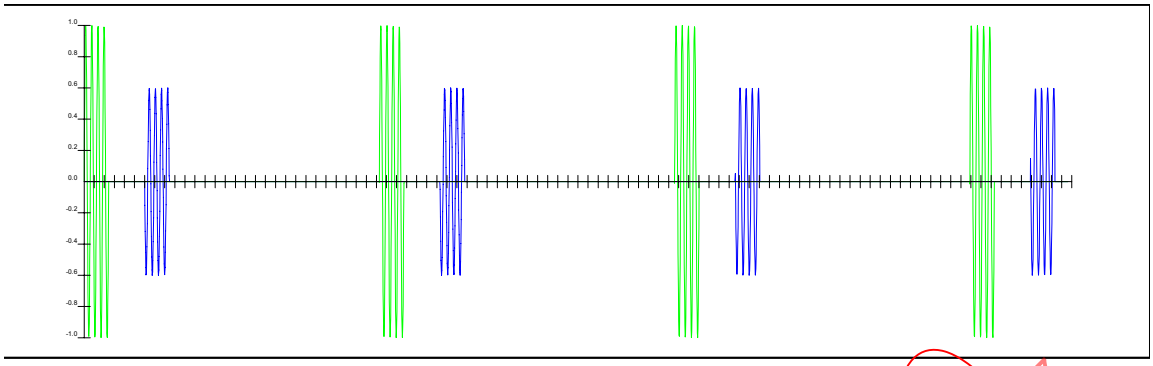


Figure E.34—RADAR_RX_SIGNAL example

E.19 RADAR_TX_SIGNAL<type: Current|| Voltage|| Power>

E.19.1 Definition

A pulsed ac signal used as a reference for received radar signals (i.e., Radar_RX_Signal). See Figure E.35.

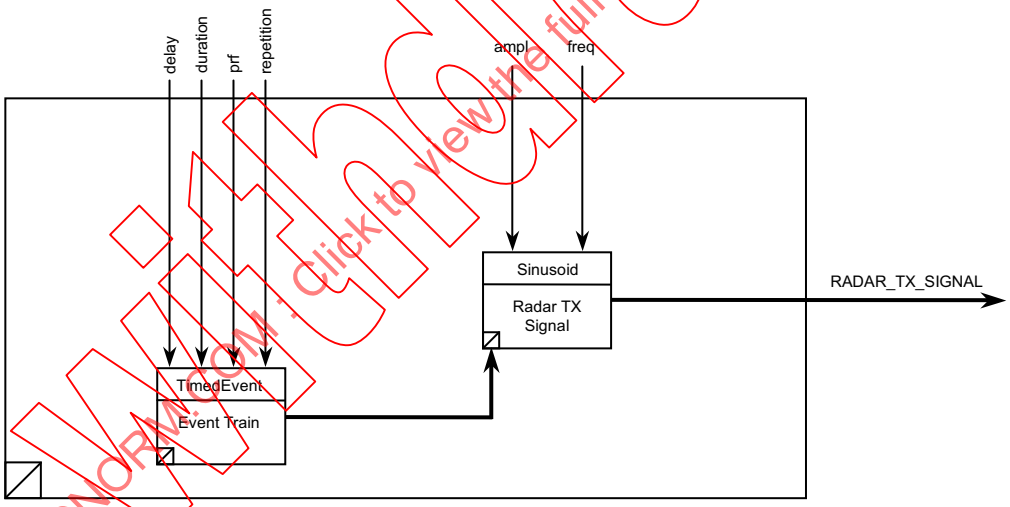


Figure E.35—TSF RADAR_TX_SIGNAL

E.19.2 Interface properties

See Table E.35 for details of the TSF RADE_TX_SIGNAL interface.

Table E.35—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	Numeric–Integer	0	

E.19.3 Notes

Default condition (where repetition = 0) is for continuously repeating pulses

E.19.4 Model description

See Table E.36 for details of the TSF RADAR_TX_SIGNAL model.

Table E.36—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]	Event Train		
Event Train	TimedEvent	Event [Out]		RADAR TX Signal	
		delay	delay		
		duration	duration		
		period			1/prf
		repetition	repetition		

E.19.5 Rules

For this signal, the allowable types are voltage, current, and power.

E.19.6 Example

See Figure E.36 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" />
```

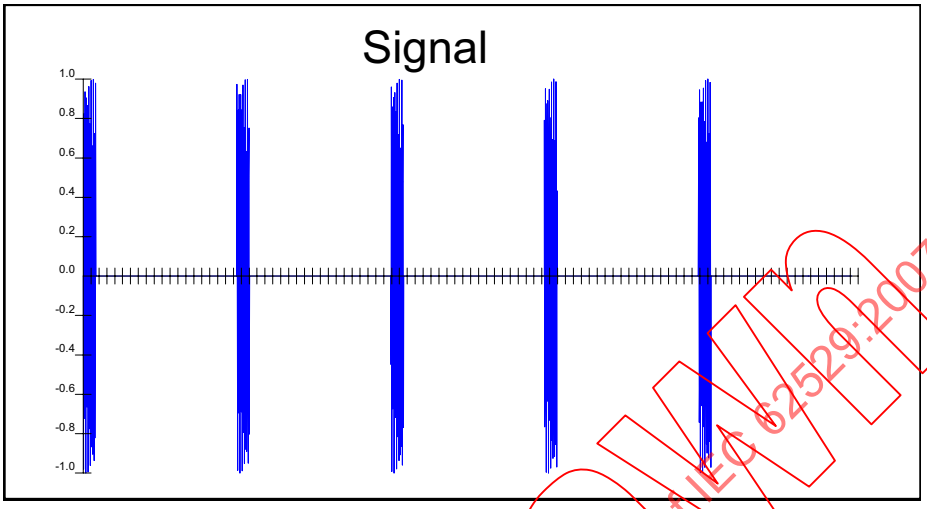


Figure E.36—RADAR_TX_SIGNAL example

E.20 RAMP_SIGNAL<type: Voltage|| Current|| Power>

E.20.1 Definition

A periodic wave whose instantaneous value varies alternately and linearly between two specified values (i.e., initial and alternate). See Figure E.37.

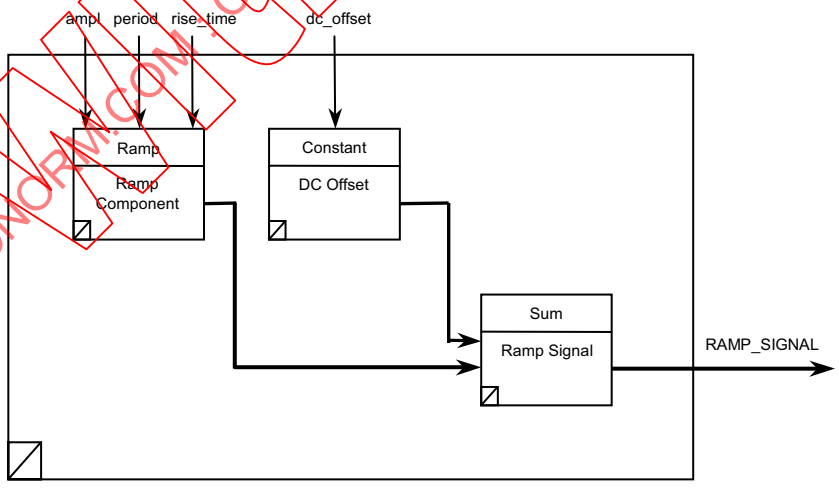


Figure E.37—TSF RAMP_SIGNAL

E.20.2 Interface properties

See Table E.37 for details of the TSF RAMP_SIGNAL interface.

Table E.37—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.20.3 Notes

E.20.4 Model description

See Table E.38 for details of the TSF RAMP_SIGNAL model.

Table E.38—TSF RAMP_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
Ramp Signal	Sum	Signal [Out]		RAMP_SIGNAL	
		Signal [In]	Ramp Component		
		Signal [In]	DC Offset		
Ramp Component	Ramp	Signal [Out]		Ramp Signal	
		amplitude	ampl		
		period	period		
		riseTime	rise_time		
DC Offset	Constant	Signal [Out]		Ramp Signal	
		amplitude	dc_offset		

E.20.5 Rules

For this signal, the allowable types are voltage, current, and power. All types must be consistent. Thus, for example, if the ramp signal amplitude is specified in volts, then the dc offset **must** also be specified in volts.

E.20.6 Example

See Figure E.38 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" />
```

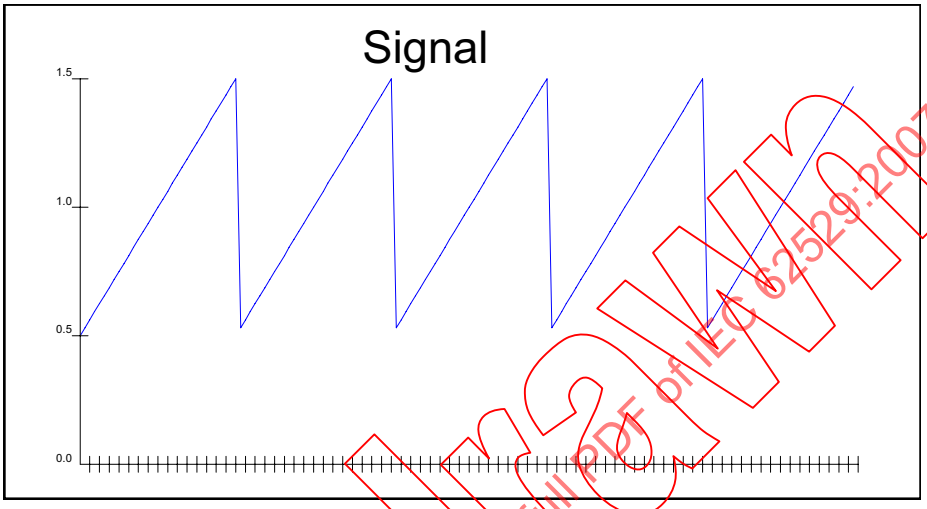


Figure E.38—RAMP_SIGNAL example

E.21 RANDOM_NOISE

E.21.1 Definition

Transient disturbances occurring unpredictably, except in a statistical sense. See Figure E.39.

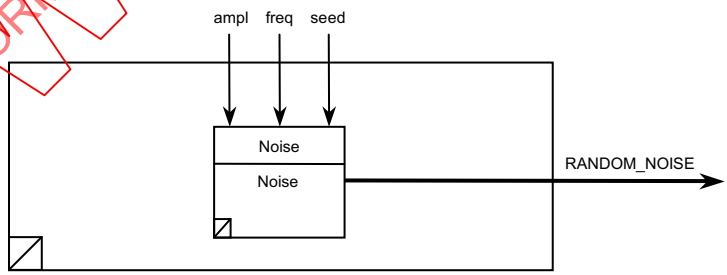


Figure E.39—TSF RANDOM_NOISE

E.21.2 Interface properties

See Table E.39 for details of the TSF RANDOM_NOISE interface.

Table E.39—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	Integer	0	

E.21.3 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 pseudo-random 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.21.4 Model description

See Table E.40 for details of the TSF RANDOM_NOISE model.

Table E.40—TSF RANDOM_NOISE model

Name	Type	Terminal	Inputs	Output	Formula
Noise	Noise	Signal [Out]		RANDOM_NOISE	
		amplitude	Ampl		
		seed	Seed		
		frequency	Freq		

E.21.5 Rules

For this signal, the allowable types are voltage and power.

E.21.6 Example

See Figure E.40 for an example of RANDOM_NOISE.

XML Static Signal Description:

```
<RANDOM_NOISE ampl="100 mV" freq="500 Hz" seed="0" />
```

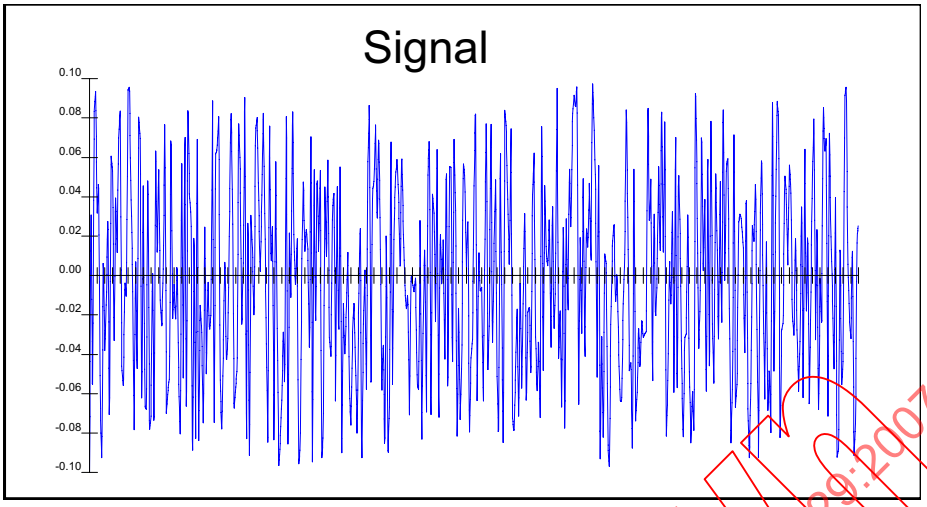


Figure E.40—RANDOM_NOISE example

E.22 RESOLVER

E.22.1 Definition

Two ac sine wave voltages whose relationships of amplitude represent the rotation of a shaft position of an electromechanical transducer. See Figure E.41.

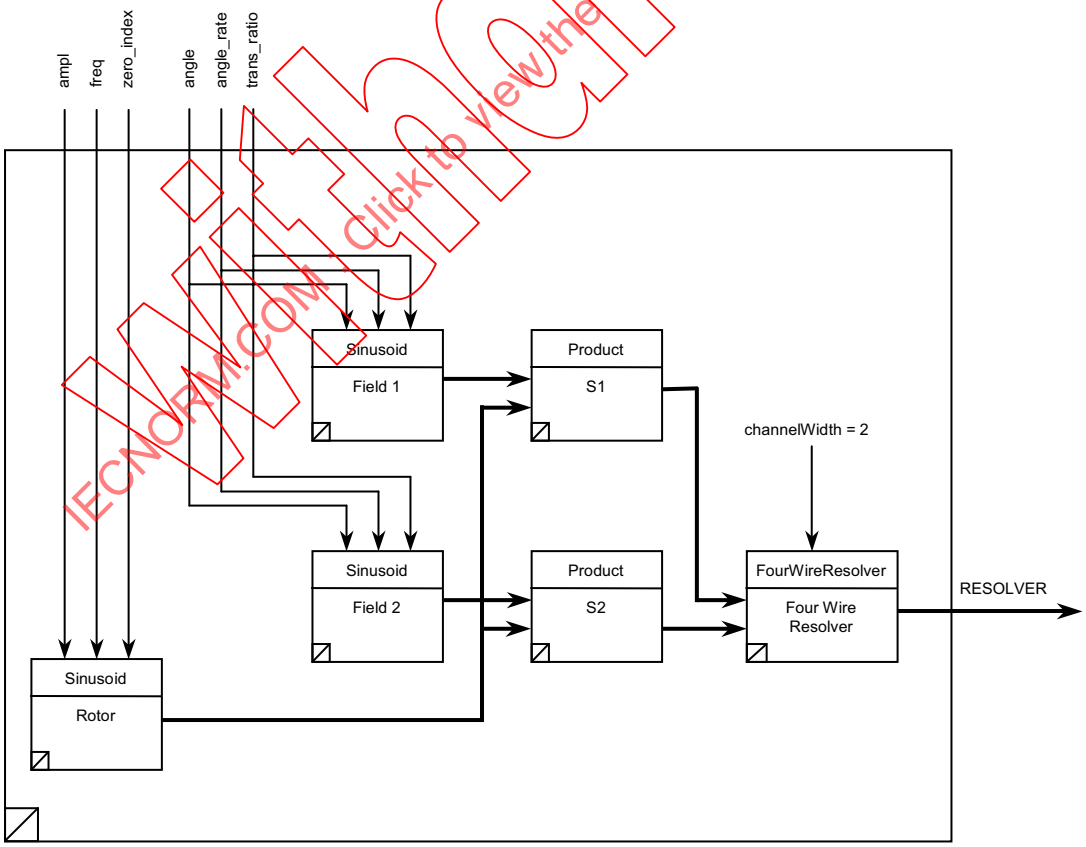


Figure E.41—TSF RESOLVER

E.22.2 Interface properties

See Table E.41 for details of the TSF RESOLVER interface.

Table E.41—TSF RESOLVER interface

Description	Name	Type	Default	Range
Shaft angle	angle	PlaneAngle	0	
Reference amplitude	ampl	Voltage	26 V	26–119 V
Reference frequency	freq	Frequency	400 Hz	30 Hz—54 kHz
Zero index	zero_index	PlaneAngle	0 rad	0–2 rad
Shaft angle rate	angle_rate	Frequency	0 Hz	
Transformer Ratio	trans_ratio	Ratio	1	

E.22.3 Notes

This model does not consider the effects of angular velocity of the rotor and the quadrature voltages generated in the secondaries.

E.22.4 Model description

See Table E.42 for details of the TSF RESOLVER model.

Table E.42—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]		TwoPhase	
		Signal [In]	Rotor		
		Signal [In]	Field 1		
S2	Product	Signal [Out]		TwoPhase	
		Signal [In]	Rotor		
		Signal [In]	Field 2		
Rotor	Sinusoid	Signal [Out]		S1 S2	
		amplitude	Ampl		

Table E.42—TSF RESOLVER model *(continued)*

Name	Type	Terminal	Inputs	Output	Formula
		frequency	Freq		
		phase	zero_index		
Field 1	Sinusoid	Signal [Out]		S1	
		amplitude			trans_ratio
		frequency			(angle_rate)
		phase	Angle		
Field 2	Sinusoid	Signal [Out]		S1	
		amplitude			trans_ratio
		frequency			(angle_rate)
		phase			angle+ /2

E.22.5 Rules

The outputs of the resolver secondaries are given by the following two equations:

Sine output

$$e_{s1} = KE_r \sin\theta \sin(2\pi f_r t + \varphi)$$

Cosine output

$$e_{s2} = KE_r \cos\theta \sin(2\pi f_r t + \varphi)$$

or

$$e_{s2} = KE_r \sin(\theta + \pi/2) \sin(2\pi f_r t + \varphi)$$

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);
θ is angular displacement of the rotor (angle);
f_r is the reference frequency of the signal in the primary (freq);
φ 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

$$e_{s1} = (E_r \sin(2\pi f_r t + \varphi)) \times (K \sin\theta)$$

Cosine output

$$e_{s2} = (E_r \sin(2\pi f_r t + \varphi)) \times (K \sin(\theta + \pi/2))$$

E.22.6 Example

See Figure E.42 for an example of RESOLVER.

XML Static Signal Description:

```
<RESOLVER name="RESOLVER9" angle_rate="5 Hz" freq="100 Hz" />
```

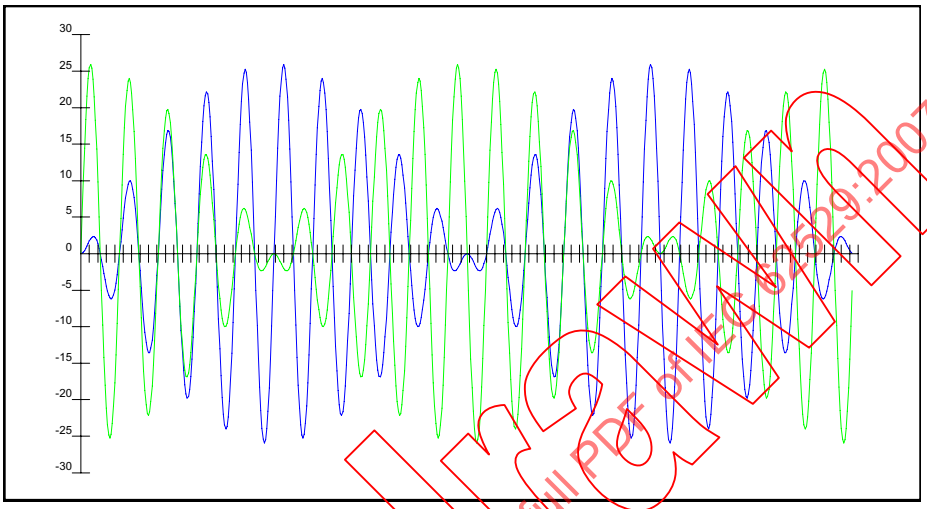


Figure E.42—RESOLVER example

E.23 RS_232

E.23.1 Definition

A serial databus signal that transmits and receives strings of characters and operates according to TIA-232 [B12].

E.23.2 Interface properties

See Table E.43 for details of the TSF RS_232 interface.

Table E.43—TSF RS_232 interface

Description	Name	Type	Default	Range
Data Word	data_word	Character String		
Baud Rate	baud_rate	Numeric–integer	9600	75| 110| 134| 150| 300| 600| 1200| 1800| 2400| 4800 7200| 9600| 14400| 19200| 38400| 57600| 115200
Data Bits	data_bits	Numeric–integer	8	4| 5| 6| 7| 8
Parity	parity	Enumeration	None	Even| Odd| None| Mark| Space

Table E.43—TSF RS_232 interface *(continued)*

Description	Name	Type	Default	Range
Stop Bits	stop_bits	Enumeration	1	1| 1.5| 2
Flow Control	flow_control	Enumeration	None	None| Hardware| Xon-Xoff

E.23.3 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 physically connected, but are not considered by this TSF.

E.23.4 Model description

See Table E.44 for details of the TSF RS_232 model.

Table E.44—TSF RS_232 model

Name	Type	Terminal	Inputs	Output	Formula
RS-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.23.5 Rules

For this signal, the data word supplied is transmitted via the serial bus connections according to the rules specified in TIA-232 [B12]. Data received via the serial bus connections will be available when the signal is used in a measurement.

E.24 SQUARE_WAVE<type: Current|| Voltage|| Power>

E.24.1 Definition

A periodic wave that alternately assumes one of two fixed values of amplitude for equal lengths of time. See Figure E.43.

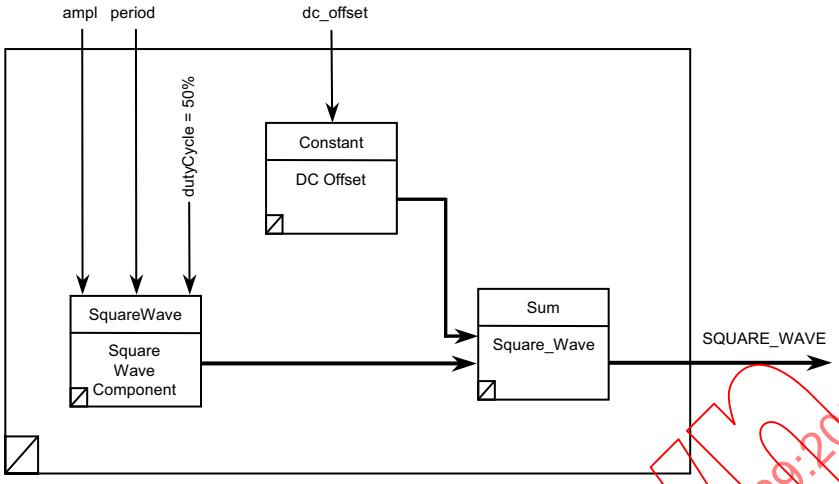


Figure E.43—TSF SQUARE_WAVE

E.24.2 Interface properties

See Table E.45 for details of the TSF SQUARE_WAVE interface.

Table E.45—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 V	

E.24.3 Notes

E.24.4 Model description

See Table E.46 for details of the TSF SQUARE_WAVE model.

Table E.46—TSF SQUARE_WAVE model

Name	Type	Terminal	Inputs	Output	Formula
Square Wave	Sum	Signal [Out]		SQUARE_WAVE	
		Signal [In]	Square Wave Component		
		Signal [In]	DC Offset		

Table E.46—TSF SQUARE_WAVE model *(continued)*

Name	Type	Terminal	Inputs	Output	Formula
Square Wave Component	SquareWave	Signal [Out]		Square Wave	
		amplitude	ampl		
		period	period		
		dutyCycle			50%
DC Offset	Constant	Signal [Out]		Square Wave	
		amplitude	dc_offset		

E.24.5 Rules

For this signal, the allowable types are voltage, current, and power. All types must be consistent. Thus, for example, if the square wave amplitude is specified in volts, then the dc offset **must** also be specified in volts.

E.24.6 Example

See Figure E.44 for an example of SQUARE_WAVE.

XML Static Signal Description:

```
<SQUARE_WAVE name="SQUARE_WAVE6" ampl="1" dc_offset="500 mV"
period="10 us" />
```

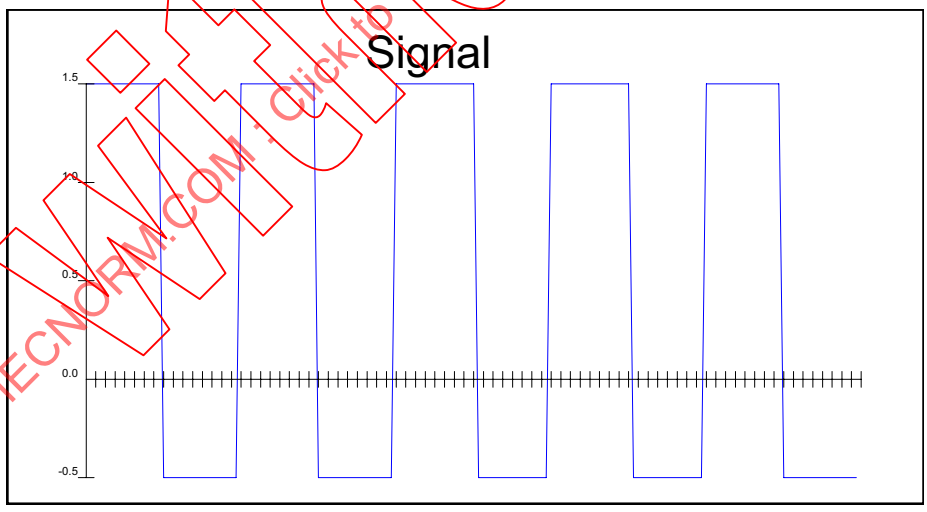


Figure E.44—SQUARE_WAVE example

E.25 SSR_INTERROGATION<type: Voltage|| Current|| Power>

E.25.1 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, system (IFF), STANAG 4193 [B11]. An aircraft on-board transponder will sense an interrogation from a ground (or airborne) station on a specific frequency (i.e., 1030 MHz) and will respond with coded signals on another frequency (i.e., 1090 MHz). See Figure E.45.

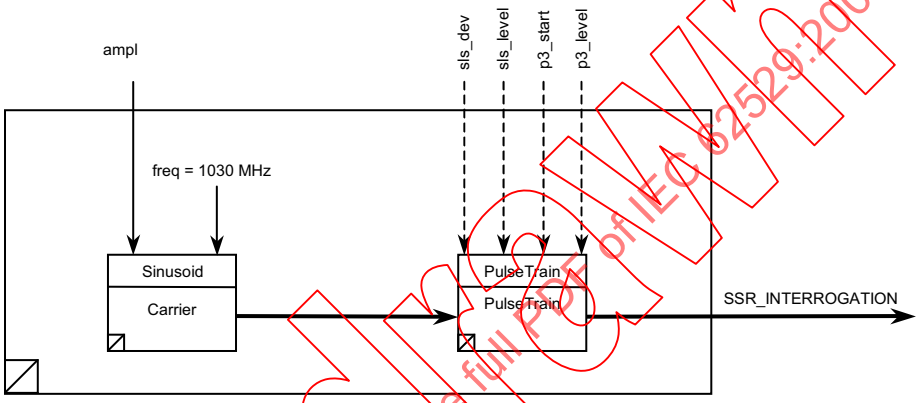


Figure E.45—TSF SSR_INTERROGATION

E.25.2 Interface properties

See Table E.47 for details of the TSF SSR_INTERROGATION interface.

Table E.47—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 | 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.25.3 Notes

The interrogation signal comprises three pulses, called P1, P2, P3, respectively. See Table E.48. The normal spacing between P1 and P2 is 2 µs. Normal spacing between P1 and P3 depends on the choice of mode.

While interrogators will 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 ensure correct response.

Table E.48—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

E.25.3.1 Equations

```
SSR_pulses = 0, 0.0000008, 1),
             (0.000002+sls_dev, 0.0000008, sls_level),
             (p3_start, 0.0000008, 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 determined by the value of interrogation mode from the table below;

p3_level is the P3 level from the interface properties.

Interrogation mode (mode)	P3 start time (p3_start) (µs)
1	3
2	5
3	8
A	8
B	17
C	21
D	25

E.25.4 Model description

See Table E.49 for details of the TSF SSR_INTERROGATION model.

Table E.49—TSF SSR_INTERROGATION model

Name	Type	Terminal	Inputs	Output	Formula
PulseTrain	Pulse-Train	Signal [Out]		SSR_INTERROGATION	
		pulses			SSR_pulses See Equation (9)
		repetition			1
		Signal [In]	Carrier		
Carrier	Sinusoid	Signal [Out]		PulseTrain	
		amplitude	ampl		
		frequency			1030 MHz
		phase			0 rad

E.25.5 Rules

For this signal, the allowable types are voltage, current, and power.

E.25.6 Example

See Figure E.46 for an example of SSR_INTERROGATION.

XML Static Signal Description:

```
<SSR_INTERROGATION name="SSR_INTERROGATION3" mode="3" p3_start="8 us"
p3_level="2" />
```

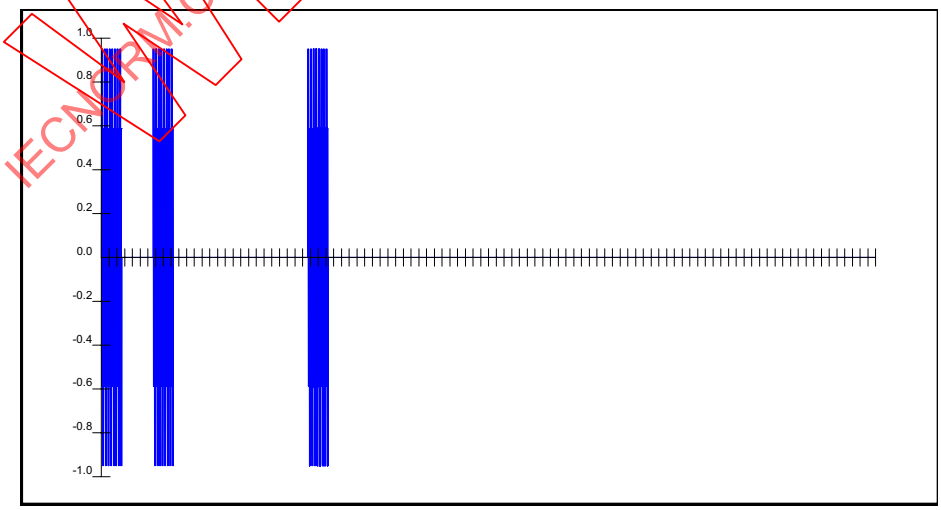


Figure E.46—SSR_INTERROGATION example

E.26 SSR_RESPONSE<type: Voltage|| Current|| Power>

E.26.1 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 will determine which pulses are present. See Figure E.47.

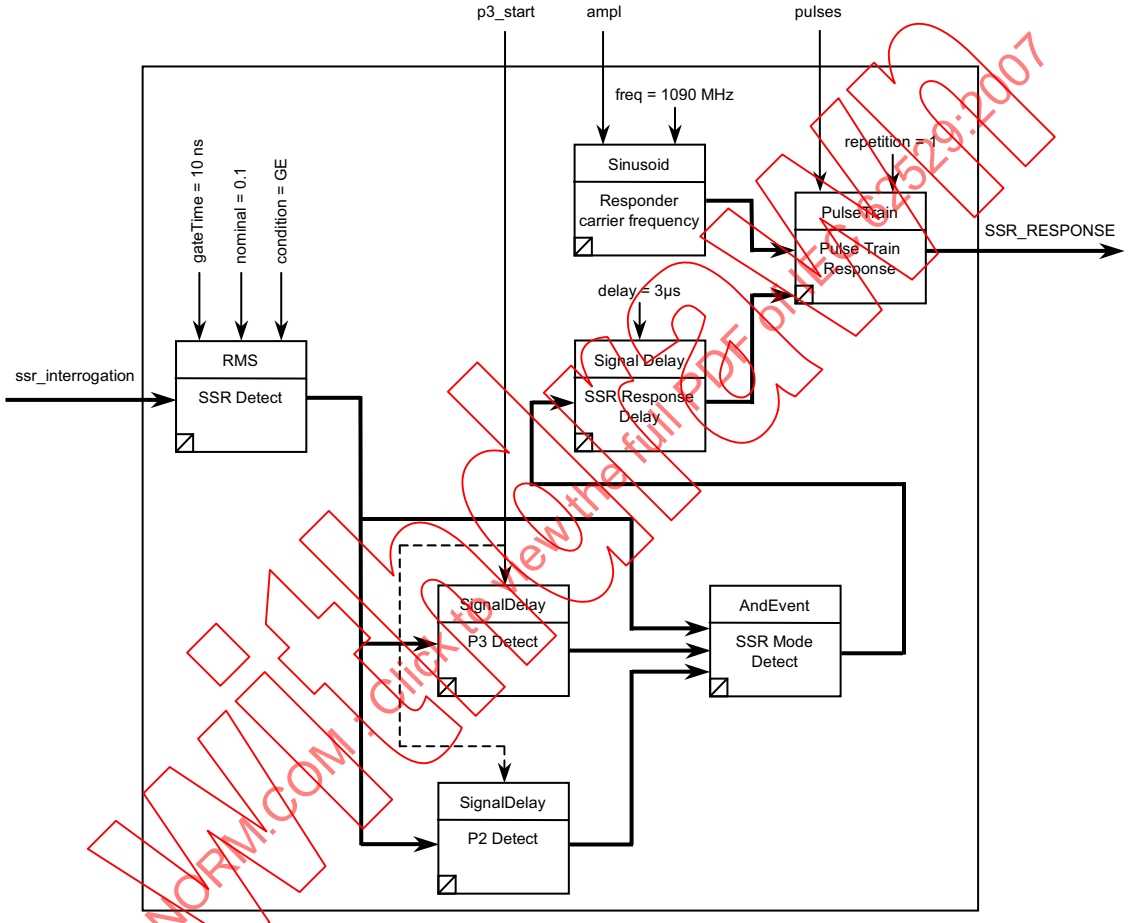


Figure E.47—TSF SSR_RESPONSE

E.26.2 Interface properties

See Table E.50 for details of the TSF SSR_RESPONSE interface.

Table E.50—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 | 8 µs | 17 µs | 21 µs | 25 µs
SSR Response Pulse Train	pulses	PulseDefns	[]	
Transmitted Interrogation signal	ssr_Interrogation	SignalFunction		

E.26.3 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.51. Pulse F1 and pulse F2 must be present. Pulse X is not currently used and should be omitted. Other pulses may be specified as required.

Table E.51—SSR_RESPONSE pulse descriptions

Pulse	Start_Time (µs)	Pulse_Width (µs)	Level_Factor
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.26.4 Model description

See Table E.52 for details of the TSF SSR_RESPONSE model.

Table E.52—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
SSR Response Delay	SignalDelay	Signal [Out]		Pulse Train Response	
		acceleration			0 Hz
		delay			3 µs
		rate			0%
		Signal [In]	SSR Mode Detect		
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 - $2.0e^{-6}$

Table E.52—TSF SSR_RESPONSE model *(continued)*

Name	Type	Terminal	Inputs	Output	Formula
		rate			0%
		Signal [In]	SSR Detect		
SSR Detect	RMS	[Out]		SSR Mode Detect, P2 Detect, P3 Detect	
		measurement			
		sample			
		count			
		gateTime			$10.0e^{-9}$ s
		nominal			0.1
		condition			GE
		NOGO			
		GO			
		UL			
		LL			
		Signal [In]	ssr_interrogation		

E.26.5 Rules

For this signal, the allowable types are voltage, current, and power. The type selected must agree with the type of the SSR_INTERROGATION signal that triggers the SSR_RESPONSE.

E.26.6 Example

See Figure E.48 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), (0.00000435,0.00000045,1),
(0.0000058,0.00000045,1), (0.00000725,0.00000045,1),
(0.0000087,0.00000045,1), (0.00001015,0.00000045,1),
(0.0000116,0.00000045,1), (0.00001305,0.00000045,1),
(0.0000145,0.00000045,1), (0.00001595,0.00000045,1),
(0.0000174,0.00000045,1), (0.00001885,0.00000045,1),
(0.0000203,0.00000045,1), (0.00002465,0.00000045,1)"
In="SSR_INTERROGATION3"/>

<SSR_INTERROGATION name="SSR_INTERROGATION3" mode="3" p3_start="8 us"
p3_level="2" />
```

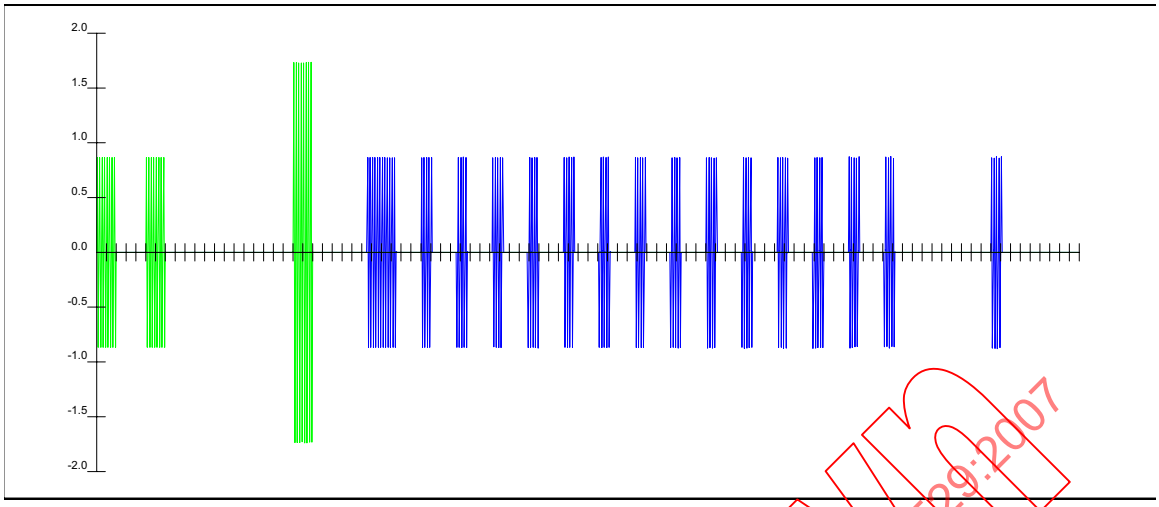


Figure E.48—SSR_RESPONSE example

E.27 STEP_SIGNAL

E.27.1 Definition

A change of dc electrical potential from one level to another, either positive or negative. See Figure E.49.

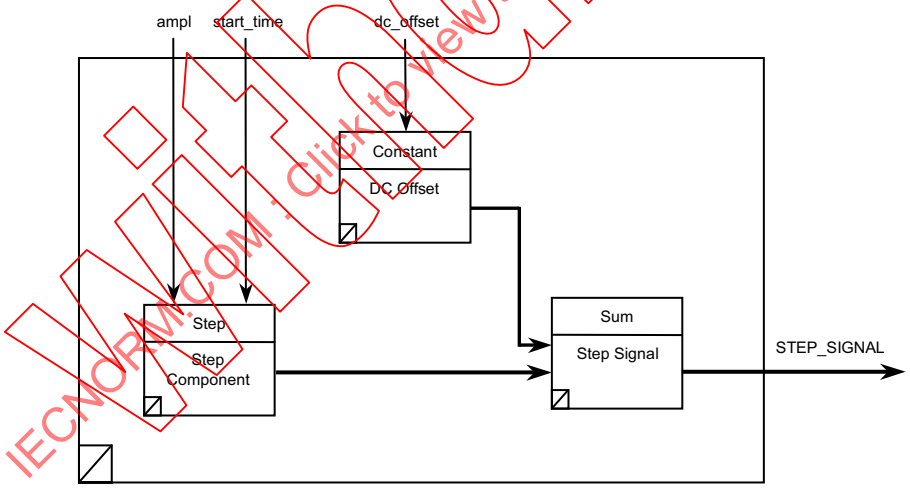


Figure E.49—TSF STEP_SIGNAL

E.27.2 Interface properties

See Table E.53 for details of the TSF STEP_SIGNAL interface.

Table E.53—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.27.3 Notes

E.27.4 Model description

See Table E.54 for details of the TSF STEP_SIGNAL model.

Table E.54—TSF STEP_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
Step Signal	Sum	Signal [Out]		STEP_SIGNAL	
		Signal [In]	Step		
		Signal [In]	DC Offset		
Step Component	Step	Signal [Out]		Step Signal	
		amplitude	Ampl		
		startTime	start_time		
DC Offset	Constant	Signal [Out]		Step Signal	
		amplitude	dc_offset		

E.27.5 Rules

E.27.6 Example

See Figure E.50 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" />
```

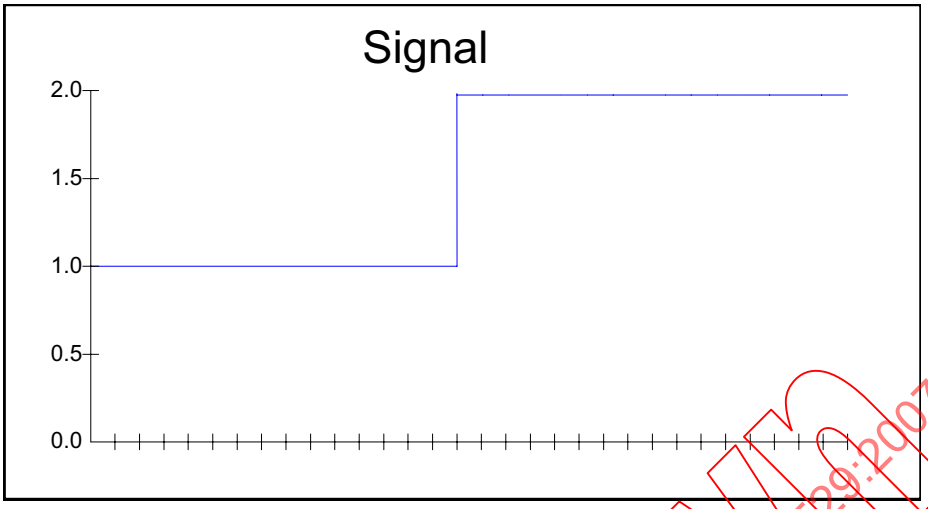


Figure E.50—STEP_SIGNAL example

E.28 SUP_CAR_SIGNAL

E.28.1 Definition

An amplitude-modulated signal in which the carrier is suppressed. See Figure E.51.

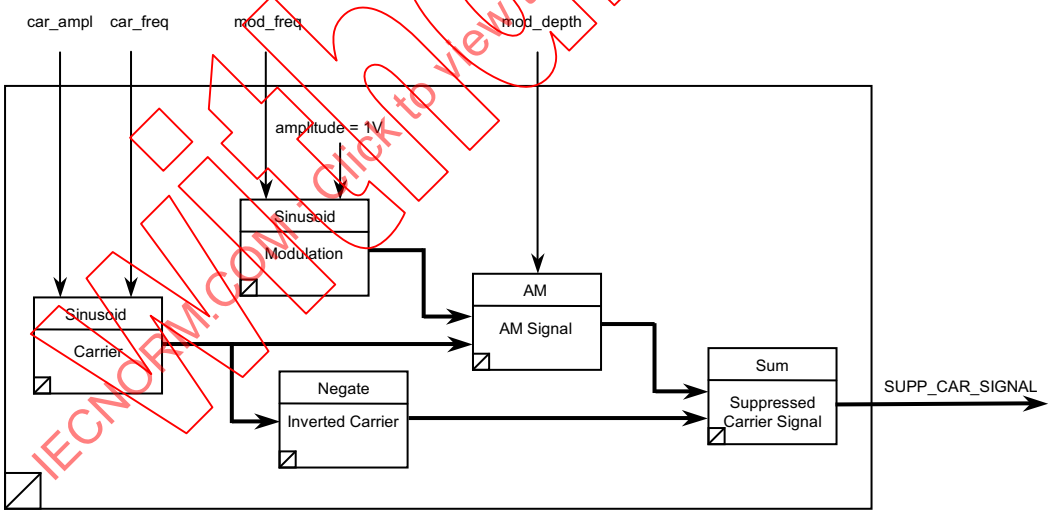


Figure E.51—TSF SUP_CAR_SIGNAL

E.28.2 Interface properties

See Table E.55 for details of the TSF SUP_CAR_SIGNAL interface.

Table E.55—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.28.3 Notes

E.28.4 Model description

See Table E.56 for details of the TSF SUP_CAR_SIGNAL model.

Table E.56—TSF SUP_CAR_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
Suppressed Carrier Signal	Sum	Signal [Out]		SUP_CAR_SIGNAL	Equation (7)
		Signal [In]	Inverted Carrier		
		Signal [In]	AM Signal		
Inverted Carrier	Negate	Signal [Out]		Suppressed Carrier signal	
		Signal [In]	Carrier		
AM Signal	AM	Signal [Out]		Suppressed Carrier signal	
		modIndex	mod_depth		
		Carrier [In]	Carrier		
		Signal [In]	Modulation		
Modulation	Sinusoid	Signal [Out]		AM Signal	
		amplitude			1 V (see note)
		frequency	mod_freq		
		phase			0 rad
Carrier	Sinusoid	Signal [Out]		Inverted Carrier, 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.28.5 Rules

The output is defined by the following equation:

$$e = (E_m E_c/2)\cos(\omega_c + \omega_m)t + (E_m E_c/2)\cos(\omega - \omega_m)t$$

where

E_m is the modulation signal amplitude;

E_c is the carrier amplitude (unmodulated);

ω_m is 2π × modulating frequency;

ω_c is 2π × carrier frequency.

E.28.6 Example

See Figure E.52 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" />
```

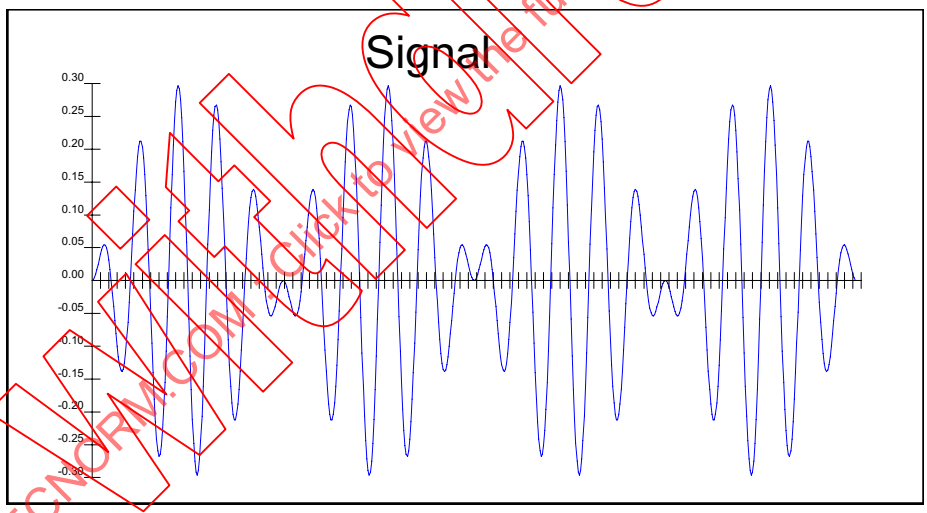


Figure E.52—SUP_CAR_SIGNAL example

E.29 SYNCHRO

E.29.1 Definition

Three ac sinusoid voltages whose relationships of amplitude represent the rotational shaft position of an electromechanical transducer. See Figure E.53.

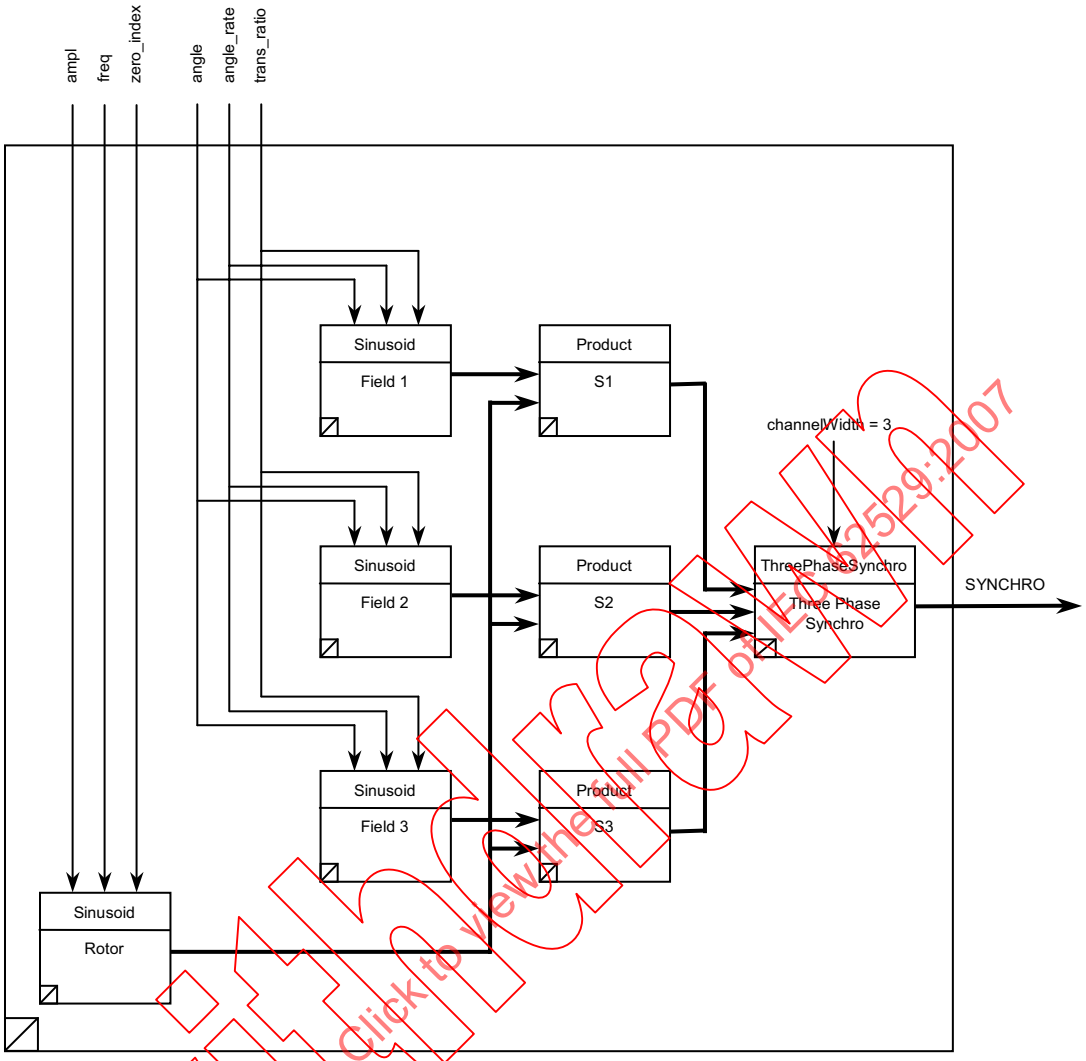


Figure E.53—TSF SYNCHRO

E.29.2 Interface properties

See Table E.57 for details of the TSF SYNCHRO interface.

Table E.57—TSF SYNCHRO interface

Description	Name	Type	Default	Range
Shaft angle	angle	PlaneAngle	0	
Reference amplitude	ampl	Voltage	26 V	26–119 V
Reference frequency	freq	Frequency	400 Hz	30 Hz – 54 kHz
Zero index	zero_index	PlaneAngle	0 rad	0–2 rad
Shaft angle rate	angle_rate	Frequency	0 Hz	
Transformer Ratio	trans_ratio	Ratio	1	

E.29.3 Notes

This model does not consider the effects of angular velocity of the rotor and the quadrature voltages generated in the stator windings.

E.29.4 Model description

See Table E.58 for details of the TSF SYNCHRO model.

Table E.58—TSF SYNCHRO model

Name	Type	Terminal	Inputs	Output	Formula
ThreePhaseSyncro	ThreePhaseSyncro	Signal [Out]		SYNCHRO	
		channelWidth			3
		Signal [In]	S1		
		Signal [In]	S2		
		Signal [In]	S3		
S1	Product	Signal [Out]		ThreePhaseSyncro	
		Signal [In]	Field 1		
		Signal [In]	Rotor		
S2	Product	Signal [Out]		ThreePhaseSyncro	
		Signal [In]	Field 2		
		Signal [In]	Rotor		
S3	Product	Signal [Out]		ThreePhaseSyncro	
		Signal [In]	Field 3		
		Signal [In]	Rotor		
Field 1	Sinusoid	Signal [Out]		S1	
		amplitude			trans_ratio
		frequency			(angle_rate)
		phase			angle - (2 /3)
Field 2	Sinusoid	Signal [Out]		S2	
		amplitude			trans_ratio
		frequency			(angle_rate)
		phase	Angle		
Field 3	Sinusoid	Signal [Out]		S3	
		amplitude			trans_ratio
		frequency			(angle_rate)
		phase			angle + (2/3)

Table E.58—TSF SYNCHRO model *(continued)*

Name	Type	Terminal	Inputs	Output	Formula
Rotor	Sinusoid	Signal [Out]		S2, S1, S3	
		amplitude	Ampl		
		frequency	Freq		
		phase	zero_index		

E.29.5 Rules

The outputs of the synchro stator windings are given by the following equations:

S1

$$E_{s1} = KE_r \sin(\theta - 2\pi/3)\sin(2\pi f_r t + \varphi)$$

S2

$$E_{s2} = KE_r \sin\theta \sin(2\pi f_r t + \varphi)$$

S3

$$E_{s2} = KE_r \sin(\theta + 2\pi/3)\sin(2\pi f_r t + \varphi)$$

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);
θ is angular displacement of the rotor (angle);
f_r is the reference frequency of the signal in the primary (freq);
φ 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

$$E_{s1} = (E_r \sin(2\pi f_r t + \varphi)) \times (K\sin(\theta - 2\pi/3))$$

S2

$$E_{s2} = (E_r \sin(2\pi f_r t + \varphi)) \times (K\sin(\theta))$$

S3

$$E_{s3} = (E_r \sin(2\pi f_r t + \varphi)) \times (K\sin(\theta + 2\pi/3))$$

E.29.6 Example

See Figure E.54 for an example of SYNCHRO.

XML Static Signal Description:

```
<SYNCHRO name="SYNCHRO5" angle_rate="5" freq="20 Hz" />
```

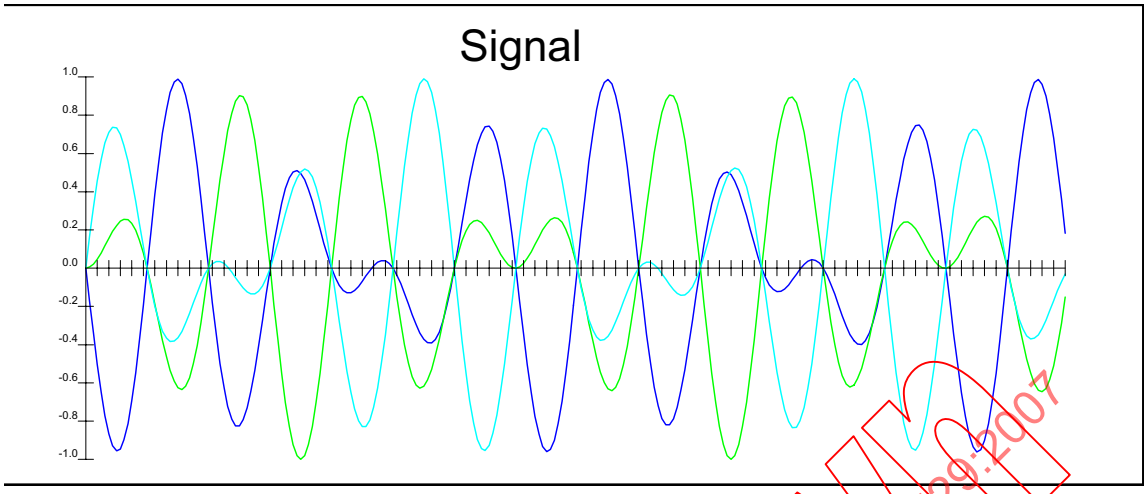


Figure E.54—SYNCHRO example

E.30 TACAN

E.30.1 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 [B10]. See Figure E.55.

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 0º phase of the 15 Hz component and, second, from the ARB to 0º of the 135 Hz component.

The TACAN beacon also generates a two- or three-letter Morse identification signal every 37.5 s.

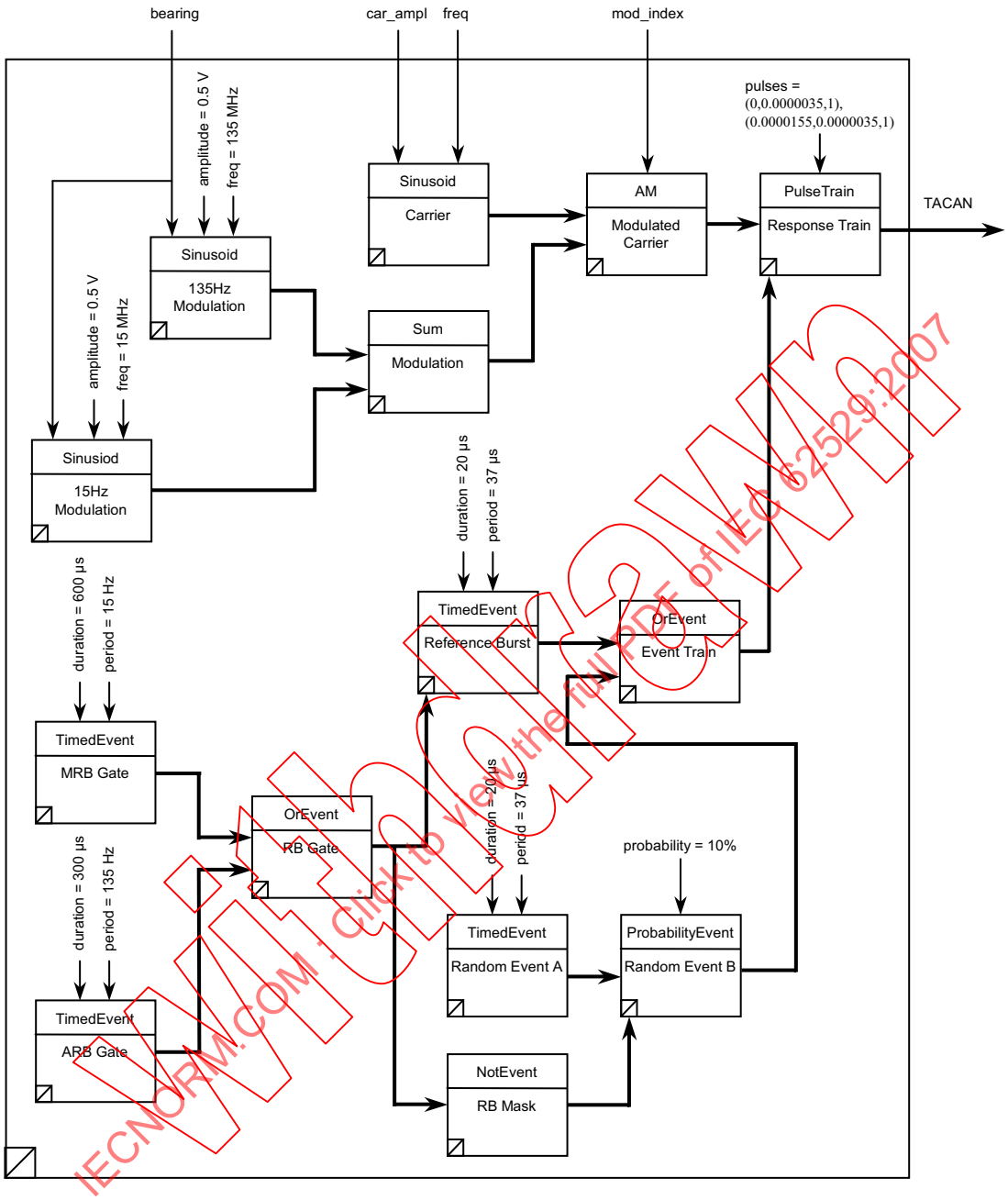


Figure E.55—TSF TACAN

E.30.2 Interface properties

See Table E.59 for details of the TSF TACAN interface.

Table E.59—TSF TACAN interface

Description	Name	Type	Default	Range
Transponder Frequency	freq	Frequency	962 MHz	962–1213 MHz
Modulation Index	mod_index	Ratio	0.3	0–1
Magnetic Bearing	bearing	PlaneAngle	0°	0–360°
Carrier Amplitude	car_ampl	Voltage		

E.30.3 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 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.30.4 Model description

See Table E.60 for details of the TSF TACAN model.

Table E.60—TSF TACAN model

Name	Type	Terminal	Inputs	Output	Formula
Response Train	Pulse Train	Signal [Out]		TACAN	
		pulses			(0,0.0000035,1), (0.0000155,0.0000035,1)
		repetition			0
		Signal [In]	Modulated Carrier		
		Gate [In]	Event Train		
Modulated Carrier	AM	Signal [Out]		Response Train	
		modIndex	mod_index		
		Carrier [In]	Carrier		
		Signal [In]	Modulation		

Table E.60—TSF TACAN model *(continued)*

Name	Type	Terminal	Inputs	Output	Formula
Event Train	OrEvent	Event [Out]		Response Train	
		Signal [In]	Random Event B		
		Signal [In]	Reference Burst		
Modulation	Sum	Signal [Out]		Modulated Carrier	
		Signal [In]	135 Hz Modulation		
		Signal [In]	15 Hz Modulation		
Reference Burst	TimedEvent	Event [Out]		Event Train	
		delay			10 µs
		duration			20 µs
		period			50 µs
		repetition			0
		Gate [In]	RB Gate		
Random Event B	ProbabilityEvent	Event [Out]		Event Train	
		seed			0
		probability			10% (reply efficiency)
		Signal [In]	Random Event A		
		Gate[In]	RB Mask		
Carrier	Sinusoid	Signal [Out]		Modulated Carrier	
		amplitude	car_ampl		
		frequency	Freq		
		phase			0 rad
Random Event A	TimedEvent	Event [Out]		Random Event B	
		delay			0 s
		duration			20 µs
		period			37 µs
		repetition			0
15 Hz Modulation	Sinusoid	Signal [Out]		Modulation	
		amplitude			0.5 V
		frequency			15 Hz

Table E.60—TSF TACAN model *(continued)*

Name	Type	Terminal	Inputs	Output	Formula
		phase	Bearing		
135 Hz Modulation	Sinusoid	Signal [Out]		Modulation	
		amplitude			0.5 V
		frequency			135 Hz
		phase	Bearing		
RB Mask	NotEvent	Event [Out]		Random Event B	
		Signal [In]	RB Gate		
RB Gate	OrEvent	Event [Out]		RB Mask Reference Burst	
		Signal [In]	ARB Gate		
		Signal [In]	MRB Gate		
MRB Gate	TimedEvent	Event [Out]		RB Gate	
		delay			0 s
		duration			600 µs
		period			15 Hz
		repetition			0
ARB Gate	TimedEvent	Event [Out]		RB Gate	
		delay			0 s
		duration			300 µs
		period			135 Hz
		repetition			0

E.30.5 Rules

E.30.6 Example

See Figure E.56 for an example of TACAN.

XML Static Signal Description:

```
<TACAN name="TACAN2" />
```

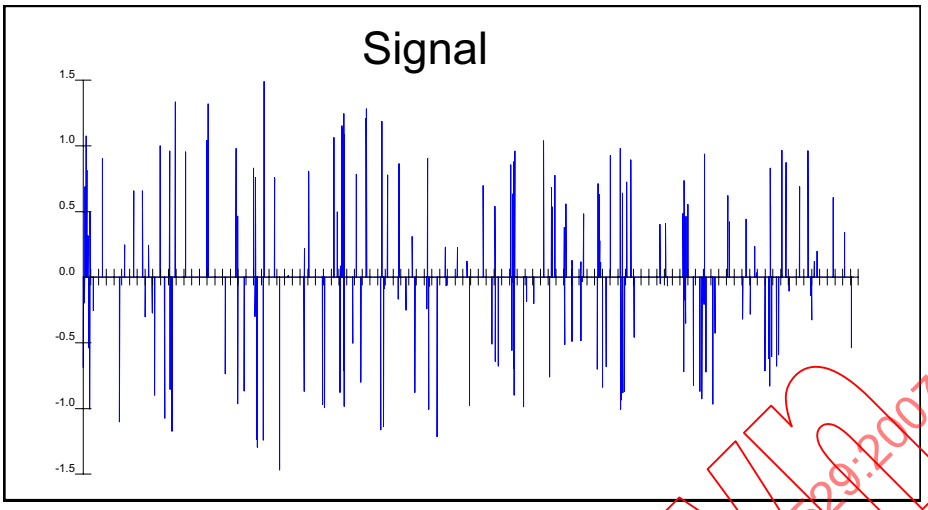


Figure E.56—TACAN example

E.31 TRIANGULAR_WAVE_SIGNAL<type: Voltage|| Current|| Power>

E.31.1 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.57.

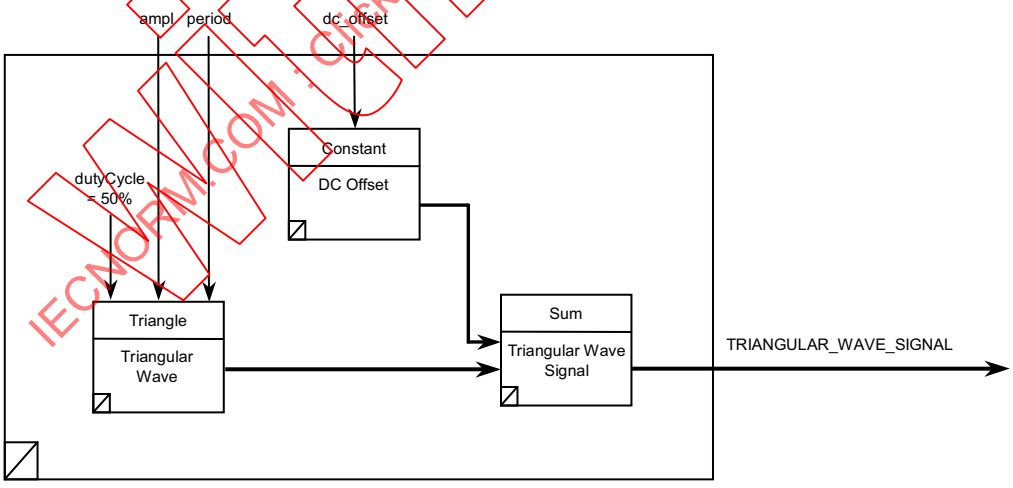


Figure E.57—TSF TRIANGULAR_WAVE_SIGNAL

E.31.2 Interface properties

See Table E.61 for details of the TSF TRIANGULAR_WAVE_SIGNAL interface.

Table E.61—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.31.3 Notes

E.31.4 Model description

See Table E.62 for details of the TSF TRIANGULAR_WAVE_SIGNAL model.

Table E.62—TSF TRIANGULAR_WAVE_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
Triangular Wave Signal	Sum	Signal [Out]		TRIANGULAR_WAVE_SIGNAL	
		Signal [In]	DC Offset		
		Signal [In]	Triangular Wave		
Triangular Wave	Triangle	Signal [Out]		Triangular Wave Signal	
		amplitude	ampl		
		period	period		
		dutyCycle			50%
DC Offset	Constant	Signal [Out]		Triangular Wave Signal	
		amplitude	dc_offset		

E.31.5 Rules

For this signal, the allowable types are voltage, current, and power. All types must be consistent. Thus, for example, if the triangular wave signal amplitude is specified in volts, then the dc offset **must** also be specified in volts.

E.31.6 Example

See Figure E.58 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" />
```

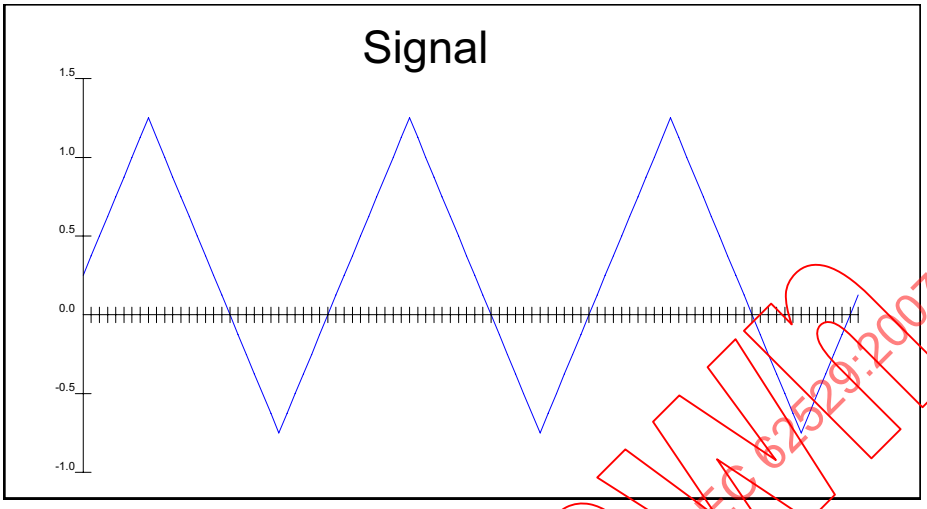


Figure E.58—TRIANGULAR_WAVE_SIGNAL example

E.32 VOR

E.32.1 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.59. 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*.

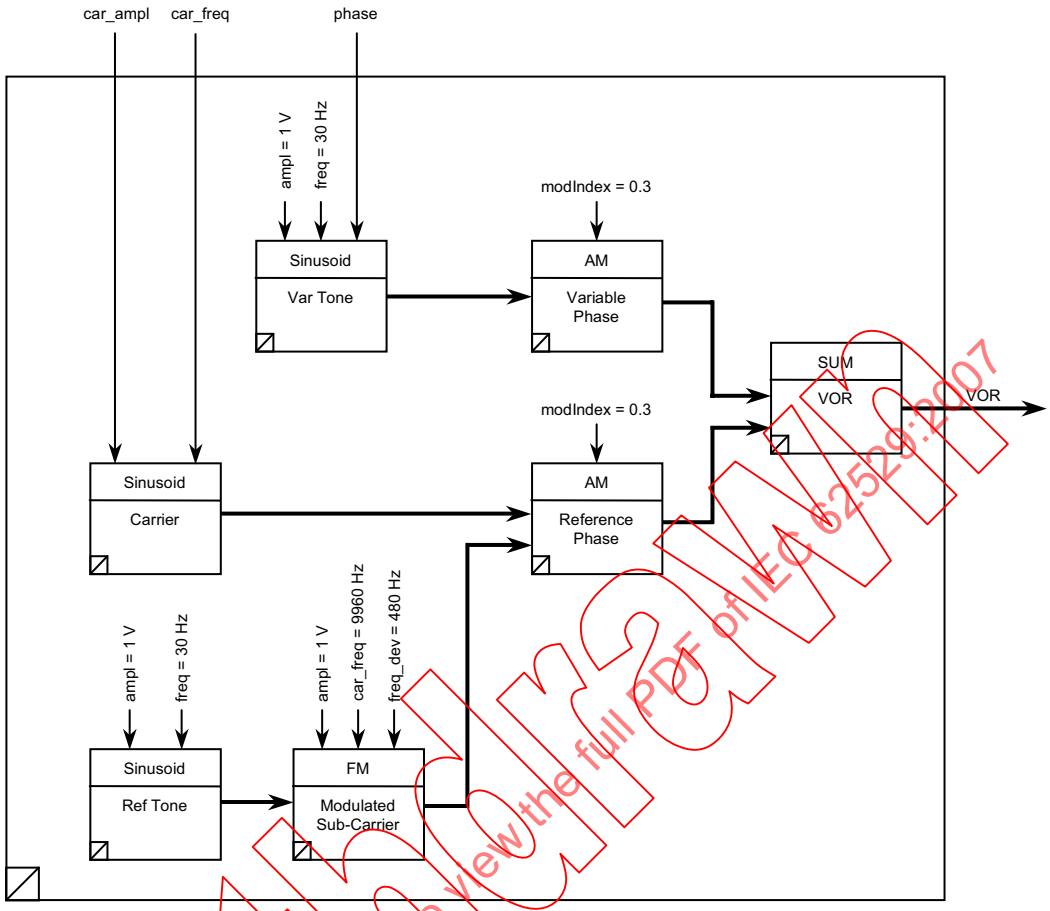


Figure E.59—TSF VOR

E.32.2 Interface properties

See Table E.63 for details of the TSF VOR interface.

Table E.63—TSF VOR interface

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Voltage	2 mV	
Carrier frequency	car_freq	Frequency	107.975 MHz	107.975–117.975 MHz
Radial bearing	phase	PlaneAngle	90°	0° – 360°

E.32.3 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.32.4 Model description

See Table E.64 for details of the TSF VOR model.

Table E.64—TSF VOR model

Name	Type	Terminal	Inputs	Output	Formula
VOR	Sum	Signal [Out]		VOR	
		Signal [In]	Reference Phase		
		Signal [In]	Variable Phase		
Reference Phase	AM	Signal [Out]		VOR	
		modIndex			0.3
		Carrier [In]	Carrier		
		Signal [In]	Modulated Subcarrier		
Variable Phase	AM	Signal [Out]		VOR	
		modIndex			0.3
		Carrier [In]	Carrier		
		Signal [In]	Var Tone		
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)

Table E.64—TSF VOR model *(continued)*

Name	Type	Terminal	Inputs	Output	Formula
		frequency			30 Hz
		phase			0
NOTE—The BSC requires a unity value for the amplitude of the modulating signal.					

E.32.5 Rules

E.32.6 Example

See Figure E.60 for an example of VOR.

XML Static Signal Description:

```
<VOR name="VOR7" />
```

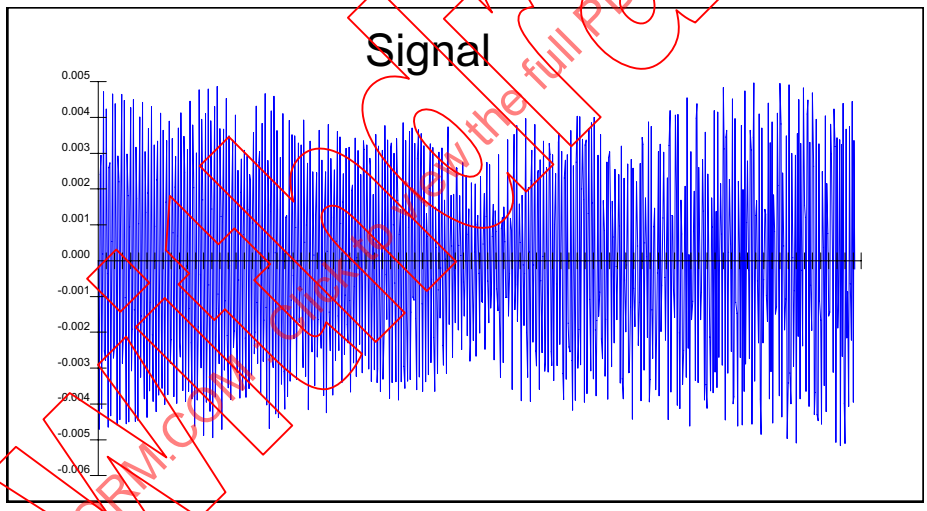


Figure E.60—VOR example

Annex F

(informative)

IDL for TSF for ATLAS

F.1 Introduction

The IDL in this annex provides the common interface description for all the TSFs for the ATLAS example 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 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 ATLAS components. All implementations that use these TSFs should use the same IDL to ensure 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.

NOTE—Annex F is an informative annex in that it provides the descriptions for the TSF components in IDL. Inclusion of the descriptions in this annex does not mean that the TSF components may not be described in other interface languages.

Where additional TSFs have been created for other test domains, they will require a new IDL library to ensure that they can be used by multiple programming environments. In such cases, the style and layout shown in this annex can be used.

F.2 IDL for TSF for ATLAS library

```
[
    uuid(AC957F30-5073-405F-9C1C-C59DB7F0CCD4),
    version(1.0),
    helpstring("STD TSF 1.0 Type Library")
]
library STDTSF
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    import "STDBSC.idl";

    //helper to forward declare interfaces and provide typedefs
    #define INTDECL( i )    interface I##i; typedef I##i* i

    INTDECL( TSF );

    //custom attribute to represent default value of this property
    #define GUID_DEFAULTVALUE    A6A997E7-94F8-4be2-8366-1814FF70EAFE


//TSF (Base Class)
    [
        object,
        uuid(AC957F31-5073-405F-9C1C-C59DB7F0CCD4),
        dual,
        helpstring("Test Signal Framework Interface"),
        pointer_default(unique)
    ]
```

```
    interface ITSF : ISignalFunction
    {
        enum {TSF_BASE=0};
    };


//AC_SIGNAL
    [
        object,
        uuid(CA4F7FD8-D05E-11D6-860D-00010214C4D2),
        dual,
        helpstring("IAC_SIGNAL Interface"),
        pointer_default(unique)
    ]
    interface IAC_SIGNAL : ITSF
    {
        enum {AC_SIGNAL_BASE=(TSF_BASE+256)};

        [propget, id(AC_SIGNAL_BASE+1), helpstring("AC Signal amplitude")]
            HRESULT ac_ampl([out, retval] Physical *pVal);
        [propputref, id(AC_SIGNAL_BASE+1), helpstring("AC Signal amplitude")]
            HRESULT ac_ampl([in] Physical newVal);
        [propget, id(AC_SIGNAL_BASE+2), helpstring("DC Offset")]
            HRESULT dc_offset([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(AC_SIGNAL_BASE+2), helpstring("DC Offset")]
            HRESULT dc_offset([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(AC_SIGNAL_BASE+3), helpstring("AC Signal frequency")]
            HRESULT freq([out, retval] Frequency *pVal);
        [propputref, id(AC_SIGNAL_BASE+3), helpstring("AC Signal frequency")]
            HRESULT freq([in] Frequency newVal);
        [propget, id(AC_SIGNAL_BASE+4), helpstring("AC Signal phase angle")]
            HRESULT phase([out, retval, custom(GUID_DEFAULTVALUE, "0 rad")] PlaneAngle
*pVal);
        [propputref, id(AC_SIGNAL_BASE+4), helpstring("AC Signal phase angle")]
            HRESULT phase([in, custom(GUID_DEFAULTVALUE, "0 rad")] PlaneAngle newVal);
    };
    [
        uuid(CA4F7FD8-D05E-11D6-860D-00010214C4D0),
        helpstring("AC_SIGNAL class"),
        noncreatable
    ]
    coclass AC_SIGNAL
    {
        [default] interface IAC_SIGNAL;
    };


//AM_SIGNAL
    [
        object,
        uuid(D7FAB36D-D075-11D6-860D-00010214C4D2),
        dual,
        helpstring("IAM_SIGNAL Interface"),
        pointer_default(unique)
    ]
    interface IAM_SIGNAL : ITSF
    {
        enum {AM_SIGNAL_BASE=(TSF_BASE+256)};

        [propget, id(AM_SIGNAL_BASE+1), helpstring("Carrier amplitude")]
            HRESULT car_ampl([out, retval] Voltage *pVal);
        [propputref, id(AM_SIGNAL_BASE+1), helpstring("Carrier amplitude")]
            HRESULT car_ampl([in] Voltage newVal);
        [propget, id(AM_SIGNAL_BASE+2), helpstring("Carrier frequency")]
            HRESULT car_freq([out, retval] Frequency *pVal);
        [propputref, id(AM_SIGNAL_BASE+2), helpstring("Carrier frequency")]
            HRESULT car_freq([in] Frequency newVal);
        [propget, id(AM_SIGNAL_BASE+3), helpstring("Modulation frequency")]
            HRESULT mod_freq([out, retval] Frequency *pVal);
        [propputref, id(AM_SIGNAL_BASE+3), helpstring("Modulation frequency")]
            HRESULT mod_freq([in] Frequency newVal);
```

```
        [propget, id(AM_SIGNAL_BASE+4), helpstring("Depth of modulation")]
            HRESULT mod_depth([out, retval] Ratio *pVal);
        [propputref, id(AM_SIGNAL_BASE+4), helpstring("Depth of modulation")]
            HRESULT mod_depth([in] Ratio newVal);
    };
    [
        uuid(D7FAB36D-D075-11D6-860D-00010214C4D0),
        helpstring("AM_SIGNAL class"),
        noncreatable
    ]
    coclass AM_SIGNAL
    {
        [default] interface IAM_SIGNAL;
    };


//DC_SIGNAL
    [
        object,
        uuid(B598AC87-86F1-44C4-A395-C0AD107FE85B),
        dual,
        helpstring("IDC_SIGNAL Interface"),
        pointer_default(unique)
    ]
    interface IDC_SIGNAL : ITSF
    {
        enum {DC_SIGNAL_BASE=(TSF_BASE+256)};

        [propget, id(DC_SIGNAL_BASE+1), helpstring("DC Level")]
            HRESULT dc_ampl([out, retval] Physical *pVal);
        [propputref, id(DC_SIGNAL_BASE+1), helpstring("DC Level")]
            HRESULT dc_ampl([in] Physical newVal);
        [propget, id(DC_SIGNAL_BASE+2), helpstring("AC Component amplitude")]
            HRESULT ac_ampl([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical *pVal);
        [propputref, id(DC_SIGNAL_BASE+2), helpstring("AC Component amplitude")]
            HRESULT ac_ampl([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(DC_SIGNAL_BASE+3), helpstring("AC Component frequency")]
           HRESULT freq([out, retval, custom(GUID_DEFAULTVALUE, "0 Hz")] Frequency *pVal);
        [propputref, id(DC_SIGNAL_BASE+3), helpstring("AC Component frequency")]
            HRESULT freq([in, custom(GUID_DEFAULTVALUE, "0 Hz")] Frequency newVal);
        [propget, id(DC_SIGNAL_BASE+4), helpstring("AC Component phase angle")]
            HRESULT phase([out, retval, custom(GUID_DEFAULTVALUE, "0 rad")] PlaneAngle
*pVal);
        [propputref, id(DC_SIGNAL_BASE+4), helpstring("AC Component phase angle")]
            HRESULT phase([in, custom(GUID_DEFAULTVALUE, "0 rad")] PlaneAngle newVal);
    };
    [
        uuid(B598AC87-86F1-44C4-A395-C0AD107FE850),
        helpstring("DC_SIGNAL class"),
        noncreatable
    ]
    coclass DC_SIGNAL
    {
        [default] interface IDC_SIGNAL;
    };


//DIGITAL_PARALLEL
    [
        object,
        uuid(A6B62348-B0BE-4900-95B1-FF513B8A6B17),
        dual,
        helpstring("IDIGITAL_PARALLEL Interface"),
        pointer_default(unique)
    ]
    interface IDIGITAL_PARALLEL : ITSF
    {
        enum {DIGITAL_PARALLEL_BASE=(TSF_BASE+256)};

        [propget, id(DIGITAL_PARALLEL_BASE+1), helpstring("Data Value")]
            HRESULT data_value([out, retval] SAFEARRAY(BSTR) *pVal);
        [propputref, id(DIGITAL_PARALLEL_BASE+1), helpstring("Data Value")]
```

```
            HRESULT data_value([in] SAFEARRAY(BSTR) newVal);
        [propget, id(DIGITAL_PARALLEL_BASE+2), helpstring("Clock period")]
            HRESULT clock_period([out, retval] Time *pVal);
        [propputref, id(DIGITAL_PARALLEL_BASE+2), helpstring("Clock period")]
            HRESULT clock_period([in] Time newVal);
        [propget, id(DIGITAL_PARALLEL_BASE+3), helpstring("Logic One level")]
            HRESULT logic_one_value([out, retval] Voltage *pVal);
        [propputref, id(DIGITAL_PARALLEL_BASE+3), helpstring("Logic One level")]
            HRESULT logic_one_value([in] Voltage newVal);
        [propget, id(DIGITAL_PARALLEL_BASE+4), helpstring("Logic Zero level")]
            HRESULT logic_zero_value([out, retval] Voltage *pVal);
        [propputref, id(DIGITAL_PARALLEL_BASE+4), helpstring("Logic Zero level")]
            HRESULT logic_zero_value([in] Voltage newVal);
    };
    [
        uuid(A6B62348-B0BE-4900-95B1-FF513B8A6B10),
        helpstring("DIGITAL_PARALLEL class"),
        noncreatable
    ]
    coclass DIGITAL_PARALLEL
    {
        [default] interface IDIGITAL_PARALLEL;
    };


//DIGITAL_SERIAL
    [
        object,
        uuid(3BEF21EF-C09C-4190-962B-24D90C301B17),
        dual,
        helpstring("IDIGITAL_SERIAL Interface"),
        pointer_default(unique)
    ]
    interface IDIGITAL_SERIAL : ITSF
    {
        enum {DIGITAL_SERIAL_BASE=(TSF_BASE+256)};

        [propget, id(DIGITAL_SERIAL_BASE+1), helpstring("Data Value")]
            HRESULT data_value([out, retval] BSTR *pVal);
        [propput, id(DIGITAL_SERIAL_BASE+1), helpstring("Data Value")]
            HRESULT data_value([in] BSTR newVal);
        [propget, id(DIGITAL_SERIAL_BASE+2), helpstring("Clock period. Zero denotes
infinite time for static digital data.")]
            HRESULT clock_period([out, retval] Time *pVal);
        [propputref, id(DIGITAL_SERIAL_BASE+2), helpstring("Clock period. Zero denotes
infinite time for static digital data.")]
            HRESULT clock_period([in] Time newVal);
        [propget, id(DIGITAL_SERIAL_BASE+3), helpstring("Logic One level")]
            HRESULT logic_one_value([out, retval] Voltage *pVal);
        [propputref, id(DIGITAL_SERIAL_BASE+3), helpstring("Logic One level")]
            HRESULT logic_one_value([in] Voltage newVal);
        [propget, id(DIGITAL_SERIAL_BASE+4), helpstring("Logic Zero level")]
            HRESULT logic_zero_value([out, retval] Voltage *pVal);
        [propputref, id(DIGITAL_SERIAL_BASE+4), helpstring("Logic Zero level")]
            HRESULT logic_zero_value([in] Voltage newVal);
    };
    [
        uuid(3BEF21EF-C09C-4190-962B-24D90C301B10),
        helpstring("DIGITAL_SERIAL class"),
        noncreatable
    ]
    coclass DIGITAL_SERIAL
    {
        [default] interface IDIGITAL_SERIAL;
    };


//DME_INTERROGATION
    [
        object,
        uuid(61911490-F7E1-4E43-A2DA-32BB8C29FAFC),
        dual,
```

```
        helpstring("IDME_INTERROGATION Interface"),
        pointer_default(unique)
    ]
    interface IDME_INTERROGATION : ITSF
    {
        enum {DME_INTERROGATION_BASE=(TSF_BASE+256)};

        [propget, id(DME_INTERROGATION_BASE+1), helpstring("Carrier Amplitude")]
            HRESULT car_ampl([out, retval] Voltage *pVal);
        [propputref, id(DME_INTERROGATION_BASE+1), helpstring("Carrier Amplitude")]
            HRESULT car_ampl([in] Voltage newVal);
        [propget, id(DME_INTERROGATION_BASE+2), helpstring("Interrogator Frequency")]
           HRESULT int_freq([out, retval, custom(GUID_DEFAULTVALUE, "1025 MHz")] Frequency
*pVal);
        [propputref, id(DME_INTERROGATION_BASE+2), helpstring("Interrogator Frequency")]
           HRESULT int_freq([in, custom(GUID_DEFAULTVALUE, "1025 MHz")] Frequency newVal);
        [propget, id(DME_INTERROGATION_BASE+3), helpstring("Interrogation Rate")]
            HRESULT int_rate([out, retval, custom(GUID_DEFAULTVALUE, "27Hz")] Frequency
*pVal);
        [propputref, id(DME_INTERROGATION_BASE+3), helpstring("Interrogation Rate")]
            HRESULT int_rate([in, custom(GUID_DEFAULTVALUE, "27Hz")] Frequency newVal);
    };
    [
        uuid(61911490-F7E1-4E43-A2DA-32BB8C29FAF0),
        helpstring("DME_INTERROGATION class"),
        noncreatable
    ]
    coclass DME_INTERROGATION
    {
        [default] interface IDME_INTERROGATION;
    };


//DME_RESPONSE
    [
        object,
        uuid(49DEA8E3-CD12-4A5E-9055-0DE92E731505),
        dual,
        helpstring("IDME_RESPONSE Interface"),
        pointer_default(unique)
    ]
    interface IDME_RESPONSE : ITSF
    {
        enum {DME_RESPONSE_BASE=(TSF_BASE+256)};

        [propget, id(DME_RESPONSE_BASE+1), helpstring("Transponder Frequency")]
           HRESULT resp_freq([out, retval, custom(GUID_DEFAULTVALUE, "962MHz")] Frequency
*pVal);
        [propputref, id(DME_RESPONSE_BASE+1), helpstring("Transponder Frequency")]
            HRESULT resp_freq([in, custom(GUID_DEFAULTVALUE, "962MHz")] Frequency newVal);
        [propget, id(DME_RESPONSE_BASE+2), helpstring("Carrier Amplitude")]
            HRESULT car_ampl([out, retval] Voltage *pVal);
        [propputref, id(DME_RESPONSE_BASE+2), helpstring("Carrier Amplitude")]
            HRESULT car_ampl([in] Voltage newVal);
        [propget, id(DME_RESPONSE_BASE+3), helpstring("Slant Range")]
            HRESULT range([out, retval, custom(GUID_DEFAULTVALUE, "0 m")] Distance *pVal);
        [propputref, id(DME_RESPONSE_BASE+3), helpstring("Slant Range")]
            HRESULT range([in, custom(GUID_DEFAULTVALUE, "0 m")] Distance newVal);
        [propget, id(DME_RESPONSE_BASE+4), helpstring("Range Rate")]
            HRESULT rate([out, retval, custom(GUID_DEFAULTVALUE, "0 m/s")] Speed *pVal);
        [propputref, id(DME_RESPONSE_BASE+4), helpstring("Range Rate")]
            HRESULT rate([in, custom(GUID_DEFAULTVALUE, "0 m/s")] Speed newVal);
        [propget, id(DME_RESPONSE_BASE+5), helpstring("Rate of Change of Range Rate")]
            HRESULT accn([out, retval, custom(GUID_DEFAULTVALUE, "0 m/s2")] Acceleration
*pVal);
        [propputref, id(DME_RESPONSE_BASE+5), helpstring("Rate of Change of Range Rate")]
            HRESULT accn([in, custom(GUID_DEFAULTVALUE, "0 m/s2")] Acceleration newVal);
    };
    [
        uuid(49DEA8E3-CD12-4A5E-9055-0DE92E731500),
        helpstring("DME_RESPONSE class"),
        noncreatable
```

```
    ]
    coclass DME_RESPONSE
    {
        [default] interface IDME_RESPONSE;
    };


//FM_SIGNAL
    [
        object,
        uuid(C4DE5309-7194-45C0-9A2A-BC2FB7EE832C),
        dual,
        helpstring("IFM_SIGNAL Interface"),
        pointer_default(unique)
    ]
    interface IFM_SIGNAL : ITSF
    {
        enum {FM_SIGNAL_BASE=(TSF_BASE+256)};

        [propget, id(FM_SIGNAL_BASE+1), helpstring("Carrier amplitude")]
            HRESULT car_ampl([out, retval] Physical *pVal);
        [propputref, id(FM_SIGNAL_BASE+1), helpstring("Carrier amplitude")]
            HRESULT car_ampl([in] Physical newVal);
        [propget, id(FM_SIGNAL_BASE+2), helpstring("Carrier frequency")]
            HRESULT car_freq([out, retval] Frequency *pVal);
        [propputref, id(FM_SIGNAL_BASE+2), helpstring("Carrier frequency")]
            HRESULT car_freq([in] Frequency newVal);
        [propget, id(FM_SIGNAL_BASE+3), helpstring("Frequency Deviation")]
            HRESULT freq_dev([out, retval] Frequency *pVal);
        [propputref, id(FM_SIGNAL_BASE+3), helpstring("Frequency Deviation")]
            HRESULT freq_dev([in] Frequency newVal);
        [propget, id(FM_SIGNAL_BASE+4), helpstring("Modulation frequency")]
            HRESULT mod_freq([out, retval] Frequency *pVal);
        [propputref, id(FM_SIGNAL_BASE+4), helpstring("Modulation frequency")]
            HRESULT mod_freq([in] Frequency newVal);
    };
    [
        uuid(C4DE5309-7194-45C0-9A2A-BC2FB7EE8320),
        helpstring("FM_SIGNAL class"),
        noncreatable
    ]
    coclass FM_SIGNAL
    {
        [default] interface IFM_SIGNAL;
    };


//ILS_GLIDE_SLOPE
    [
        object,
        uuid(9A319ED0-E8B1-4F27-998F-E670CB80EEDF),
        dual,
        helpstring("IILS_GLIDE_SLOPE Interface"),
        pointer_default(unique)
    ]
    interface IILS_GLIDE_SLOPE : ITSF
    {
        enum {ILS_GLIDE_SLOPE_BASE=(TSF_BASE+256)};

        [propget, id(ILS_GLIDE_SLOPE_BASE+1), helpstring("Carrier amplitude")]
            HRESULT car_ampl([out, retval, custom(GUID_DEFAULTVALUE, "2 mV")] Physical
*pVal);
        [propputref, id(ILS_GLIDE_SLOPE_BASE+1), helpstring("Carrier amplitude")]
            HRESULT car_ampl([in, custom(GUID_DEFAULTVALUE, "2 mV")] Physical newVal);
        [propget, id(ILS_GLIDE_SLOPE_BASE+2), helpstring("Frequency")]
            HRESULT car_freq([out, retval, custom(GUID_DEFAULTVALUE, "328.6 MHz")]
Frequency *pVal);
        [propputref, id(ILS_GLIDE_SLOPE_BASE+2), helpstring("Frequency")]
            HRESULT car_freq([in, custom(GUID_DEFAULTVALUE, "328.6 MHz")] Frequency
newVal);
        [propget, id(ILS_GLIDE_SLOPE_BASE+3), helpstring("150 Hz attenuation depth")]
```

```
            HRESULT onefifty_level([out, retval, custom(GUID_DEFAULTVALUE, "1")] Ratio
*pVal);
        [propputref, id(ILS_GLIDE_SLOPE_BASE+3), helpstring("150 Hz attenuation depth")]
            HRESULT onefifty_level([in, custom(GUID_DEFAULTVALUE, "1")] Ratio newVal);
        [propget, id(ILS_GLIDE_SLOPE_BASE+4), helpstring("90 Hz attenuation depth")]
            HRESULT ninety_level([out, retval, custom(GUID_DEFAULTVALUE, "1")] Ratio
*pVal);
        [propputref, id(ILS_GLIDE_SLOPE_BASE+4), helpstring("90 Hz attenuation depth")]
            HRESULT ninety_level([in, custom(GUID_DEFAULTVALUE, "1")] Ratio newVal);
    };
    [
        uuid(9A319ED0-E8B1-4F27-998F-E670CB80EED0),
        helpstring("ILS_GLIDE_SLOPE class"),
        noncreatable
    ]
    coclass ILS_GLIDE_SLOPE
    {
        [default] interface IILS_GLIDE_SLOPE;
    };


//ILS_LOCALIZER
    [
        object,
        uuid(19D4E568-E533-4232-AEC2-B319D7DB0E12),
        dual,
        helpstring("IILS_LOCALIZER Interface"),
        pointer_default(unique)
    ]
    interface IILS_LOCALIZER : ITSF
    {
        enum {ILS_LOCALIZER_BASE=(TSF_BASE+256)};

        [propget, id(ILS_LOCALIZER_BASE+1), helpstring("Carrier Amplitude")]
            HRESULT car_ampl([out, retval, custom(GUID_DEFAULTVALUE, "2 mW")] Physical
*pVal);
        [propputref, id(ILS_LOCALIZER_BASE+1), helpstring("Carrier Amplitude")]
            HRESULT car_ampl([in, custom(GUID_DEFAULTVALUE, "2 mW")] Physical newVal);
        [propget, id(ILS_LOCALIZER_BASE+2), helpstring("Carrier frequency")]
            HRESULT car_freq([out, retval, custom(GUID_DEFAULTVALUE, "108.1 MHz")]
Frequency *pVal);
        [propputref, id(ILS_LOCALIZER_BASE+2), helpstring("Carrier frequency")]
            HRESULT car_freq([in, custom(GUID_DEFAULTVALUE, "108.1 MHz")] Frequency
newVal);
        [propget, id(ILS_LOCALIZER_BASE+3), helpstring("150Hz attenuation depth")]
            HRESULT onefifty_level([out, retval, custom(GUID_DEFAULTVALUE, "1")] Ratio
*pVal);
        [propputref, id(ILS_LOCALIZER_BASE+3), helpstring("150Hz attenuation depth")]
            HRESULT onefifty_level([in, custom(GUID_DEFAULTVALUE, "1")] Ratio newVal);
        [propget, id(ILS_LOCALIZER_BASE+4), helpstring("90Hz attenuation depth")]
            HRESULT ninety_level([out, retval, custom(GUID_DEFAULTVALUE, "1")] Ratio
*pVal);
        [propputref, id(ILS_LOCALIZER_BASE+4), helpstring("90Hz attenuation depth")]
            HRESULT ninety_level([in, custom(GUID_DEFAULTVALUE, "1")] Ratio newVal);
    };
    [
        uuid(19D4E568-E533-4232-AEC2-B319D7DB0E10),
        helpstring("ILS_LOCALIZER class"),
        noncreatable
    ]
    coclass ILS_LOCALIZER
    {
        [default] interface IILS_LOCALIZER;
    };


//ILS_MARKER
    [
        object,
        uuid(627E9A00-4B31-477F-9E57-CAB38DB31E39),
        dual,
        helpstring("IILS_MARKER Interface"),
```

```
        pointer_default(unique)
    ]
    interface IILS_MARKER : ITSF
    {
        enum {ILS_MARKER_BASE=(TSF_BASE+256)};

        [propget, id(ILS_MARKER_BASE+1), helpstring("Marker Frequency")]
           HRESULT marker_freq([out, retval, custom(GUID_DEFAULTVALUE, "400Hz")] Frequency
*pVal);
        [propputref, id(ILS_MARKER_BASE+1), helpstring("Marker Frequency")]
           HRESULT marker_freq([in, custom(GUID_DEFAULTVALUE, "400Hz")] Frequency newVal);
        [propget, id(ILS_MARKER_BASE+2), helpstring("Carrier Frequency")]
            HRESULT car_ampl([out, retval, custom(GUID_DEFAULTVALUE, "2mW")] Power *pVal);
        [propputref, id(ILS_MARKER_BASE+2), helpstring("Carrier Frequency")]
            HRESULT car_ampl([in, custom(GUID_DEFAULTVALUE, "2mW")] Power newVal);
    };
    [
        uuid(627E9A00-4B31-477F-9E57-CAB38DB31E30),
        helpstring("ILS_MARKER class"),
        noncreatable
    ]
    coclass ILS_MARKER
    {
        [default] interface IILS_MARKER;
    };


//PM_SIGNAL
    [
        object,
        uuid(2636CC15-81A4-43B7-8A7F-0F7CD9C5B442),
        dual,
        helpstring("IPM_SIGNAL Interface"),
        pointer_default(unique)
    ]
    interface IPM_SIGNAL : ITSF
    {
        enum {PM_SIGNAL_BASE=(TSF_BASE+256)};

        [propget, id(PM_SIGNAL_BASE+1), helpstring("Carrier amplitude")]
            HRESULT car_ampl([out, retval] Voltage *pVal);
        [propputref, id(PM_SIGNAL_BASE+1), helpstring("Carrier amplitude")]
            HRESULT car_ampl([in] Voltage newVal);
        [propget, id(PM_SIGNAL_BASE+2), helpstring("Carrier frequency")]
            HRESULT car_freq([out, retval] Frequency *pVal);
        [propputref, id(PM_SIGNAL_BASE+2), helpstring("Carrier frequency")]
            HRESULT car_freq([in] Frequency newVal);
        [propget, id(PM_SIGNAL_BASE+3), helpstring("Phase Deviation")]
            HRESULT phase_dev([out, retval] PlaneAngle *pVal);
        [propputref, id(PM_SIGNAL_BASE+3), helpstring("Phase Deviation")]
            HRESULT phase_dev([in] PlaneAngle newVal);
        [propget, id(PM_SIGNAL_BASE+4), helpstring("Modulation frequency")]
            HRESULT mod_freq([out, retval] Frequency *pVal);
        [propputref, id(PM_SIGNAL_BASE+4), helpstring("Modulation frequency")]
            HRESULT mod_freq([in] Frequency newVal);
        [propget, id(PM_SIGNAL_BASE+5), helpstring("Modulation amplitude")]
            HRESULT mod_ampl([out, retval] Voltage *pVal);
        [propputref, id(PM_SIGNAL_BASE+5), helpstring("Modulation amplitude")]
            HRESULT mod_ampl([in] Voltage newVal);
    };
    [
        uuid(2636CC15-81A4-43B7-8A7F-0F7CD9C5B440),
        helpstring("PM_SIGNAL class"),
        noncreatable
    ]
    coclass PM_SIGNAL
    {
        [default] interface IPM_SIGNAL;
    };


//PULSED_AC_SIGNAL
```

```
    [
        object,
        uuid(1764E313-2284-41F7-872D-C34C9FF4B6FA),
        dual,
        helpstring("IPULSED_AC_SIGNAL Interface"),
        pointer_default(unique)
    ]
    interface IPULSED_AC_SIGNAL : ITSF
    {
        enum {PULSED_AC_SIGNAL_BASE=(TSF_BASE+256)};

        [propget, id(PULSED_AC_SIGNAL_BASE+1), helpstring("AC Signal amplitude")]
            HRESULT ac_ampl([out, retval] Physical *pVal);
        [propputref, id(PULSED_AC_SIGNAL_BASE+1), helpstring("AC Signal amplitude")]
            HRESULT ac_ampl([in] Physical newVal);
        [propget, id(PULSED_AC_SIGNAL_BASE+2), helpstring("AC Signal frequency")]
            HRESULT freq([out, retval] Frequency *pVal);
        [propputref, id(PULSED_AC_SIGNAL_BASE+2), helpstring("AC Signal frequency")]
            HRESULT freq([in] Frequency newVal);
        [propget, id(PULSED_AC_SIGNAL_BASE+3), helpstring("DC offset")]
            HRESULT dc_offset([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(PULSED_AC_SIGNAL_BASE+3), helpstring("DC offset")]
            HRESULT dc_offset([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(PULSED_AC_SIGNAL_BASE+4), helpstring("Initial delay")]
            HRESULT p_delay([out, retval, custom(GUID_DEFAULTVALUE, "0 s")] Time *pVal);
        [propputref, id(PULSED_AC_SIGNAL_BASE+4), helpstring("Initial delay")]
            HRESULT p_delay([in, custom(GUID_DEFAULTVALUE, "0 s")] Time newVal);
        [propget, id(PULSED_AC_SIGNAL_BASE+5), helpstring("Pulse width")]
            HRESULT p_duration([out, retval] Time *pVal);
        [propputref, id(PULSED_AC_SIGNAL_BASE+5), helpstring("Pulse width")]
            HRESULT p_duration([in] Time newVal);
        [propget, id(PULSED_AC_SIGNAL_BASE+6), helpstring("Pulse repetition frequency")]
            HRESULT prf([out, retval] Frequency *pVal);
       [propputref, id(PULSED_AC_SIGNAL_BASE+6), helpstring("Pulse repetition frequency")]

            HRESULT prf([in] Frequency newVal);
        [propget, id(PULSED_AC_SIGNAL_BASE+7), helpstring("Number of pulses")]
            HRESULT p_repetition([out, retval, custom(GUID_DEFAULTVALUE, "0")] int *pVal);
        [propput, id(PULSED_AC_SIGNAL_BASE+7), helpstring("Number of pulses")]
            HRESULT p_repetition([in, custom(GUID_DEFAULTVALUE, "0")] int newVal);
    };
    [
        uuid(1764E313-2284-41F7-872D-C34C9FF4B6F0),
        helpstring("PULSED_AC_SIGNAL class"),
        noncreatable
    ]
    coclass PULSED_AC_SIGNAL
    {
        [default] interface IPULSED_AC_SIGNAL;
    };


//PULSED_AC_TRAIN
    [
        object,
        uuid(7ED5ECB7-1B38-4E31-90E8-21DCDCA55F92),
        dual,
        helpstring("IPULSED_AC_TRAIN Interface"),
        pointer_default(unique)
    ]
    interface IPULSED_AC_TRAIN : ITSF
    {
        enum {PULSED_AC_TRAIN_BASE=(TSF_BASE+256)};

        [propget, id(PULSED_AC_TRAIN_BASE+1), helpstring("AC amplitude")]
            HRESULT ac_ampl([out, retval] Physical *pVal);
        [propputref, id(PULSED_AC_TRAIN_BASE+1), helpstring("AC amplitude")]
            HRESULT ac_ampl([in] Physical newVal);
        [propget, id(PULSED_AC_TRAIN_BASE+2), helpstring("AC frequency")]
            HRESULT freq([out, retval] Frequency *pVal);
        [propputref, id(PULSED_AC_TRAIN_BASE+2), helpstring("AC frequency")]
```

```
            HRESULT freq([in] Frequency newVal);
        [propget, id(PULSED_AC_TRAIN_BASE+3), helpstring("DC offset")]
            HRESULT dc_offset([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(PULSED_AC_TRAIN_BASE+3), helpstring("DC offset")]
            HRESULT dc_offset([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(PULSED_AC_TRAIN_BASE+4), helpstring("Pulse train")]
            HRESULT pulse_train([out, retval] PulseDefns *pVal);
        [propputref, id(PULSED_AC_TRAIN_BASE+4), helpstring("Pulse train")]
            HRESULT pulse_train([in] PulseDefns newVal);
    };
    [
        uuid(7ED5ECB7-1B38-4E31-90E8-21DCDCA55F90),
        helpstring("PULSED_AC_TRAIN class"),
        noncreatable
    ]
    coclass PULSED_AC_TRAIN
    {
        [default] interface IPULSED_AC_TRAIN;
    };


//PULSED_DC_SIGNAL
    [
        object,
        uuid(EB5BB2D2-24A2-4DCA-96F3-4090B5FFD68B),
        dual,
        helpstring("IPULSED_DC_SIGNAL Interface"),
        pointer_default(unique)
    ]
    interface IPULSED_DC_SIGNAL : ITSF
    {
        enum {PULSED_DC_SIGNAL_BASE=(TSF_BASE+256)};

        [propget, id(PULSED_DC_SIGNAL_BASE+1), helpstring("DC level")]
            HRESULT dc_ampl([out, retval] Physical *pVal);
        [propputref, id(PULSED_DC_SIGNAL_BASE+1), helpstring("DC level")]
            HRESULT dc_ampl([in] Physical newVal);
        [propget, id(PULSED_DC_SIGNAL_BASE+2), helpstring("AC component amplitude")]
            HRESULT ac_ampl([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical *pVal);
        [propputref, id(PULSED_DC_SIGNAL_BASE+2), helpstring("AC component amplitude")]
            HRESULT ac_ampl([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(PULSED_DC_SIGNAL_BASE+3), helpstring("AC component frequency")]
           HRESULT freq([out, retval, custom(GUID_DEFAULTVALUE, "0 Hz")] Frequency *pVal);
        [propputref, id(PULSED_DC_SIGNAL_BASE+3), helpstring("AC component frequency")]
            HRESULT freq([in, custom(GUID_DEFAULTVALUE, "0 Hz")] Frequency newVal);
        [propget, id(PULSED_DC_SIGNAL_BASE+4), helpstring("Delay before first pulse")]
            HRESULT p_delay([out, retval, custom(GUID_DEFAULTVALUE, "0 s")] Time *pVal);
        [propputref, id(PULSED_DC_SIGNAL_BASE+4), helpstring("Delay before first pulse")]
            HRESULT p_delay([in, custom(GUID_DEFAULTVALUE, "0 s")] Time newVal);
        [propget, id(PULSED_DC_SIGNAL_BASE+5), helpstring("Pulse width")]
            HRESULT p_duration([out, retval] Time *pVal);
        [propputref, id(PULSED_DC_SIGNAL_BASE+5), helpstring("Pulse width")]
            HRESULT p_duration([in] Time newVal);
        [propget, id(PULSED_DC_SIGNAL_BASE+6), helpstring("Pulse repetition frequency")]
            HRESULT prf([out, retval] Frequency *pVal);
       [propputref, id(PULSED_DC_SIGNAL_BASE+6), helpstring("Pulse repetition frequency")]

            HRESULT prf([in] Frequency newVal);
        [propget, id(PULSED_DC_SIGNAL_BASE+7), helpstring("Number of pulses")]
            HRESULT p_repetition([out, retval, custom(GUID_DEFAULTVALUE, "0")] int *pVal);
        [propput, id(PULSED_DC_SIGNAL_BASE+7), helpstring("Number of pulses")]
            HRESULT p_repetition([in, custom(GUID_DEFAULTVALUE, "0")] int newVal);
    };
    [
        uuid(EB5BB2D2-24A2-4DCA-96F3-4090B5FFD680),
        helpstring("PULSED_DC_SIGNAL class"),
        noncreatable
    ]
    coclass PULSED_DC_SIGNAL
    {
        [default] interface IPULSED_DC_SIGNAL;
```

```
    };


//PULSED_DC_TRAIN
    [
        object,
        uuid(5A00D58E-F7A7-4285-AB6D-55D533A63239),
        dual,
        helpstring("IPULSED_DC_TRAIN Interface"),
        pointer_default(unique)
    ]
    interface IPULSED_DC_TRAIN : ITSF
    {
        enum {PULSED_DC_TRAIN_BASE=(TSF_BASE+256)};

        [propget, id(PULSED_DC_TRAIN_BASE+1), helpstring("DC level")]
            HRESULT dc_ampl([out, retval] Physical *pVal);
        [propputref, id(PULSED_DC_TRAIN_BASE+1), helpstring("DC level")]
            HRESULT dc_ampl([in] Physical newVal);
        [propget, id(PULSED_DC_TRAIN_BASE+2), helpstring("Pulse train")]
            HRESULT pulse_train([out, retval] PulseDefns *pVal);
        [propputref, id(PULSED_DC_TRAIN_BASE+2), helpstring("Pulse train")]
            HRESULT pulse_train([in] PulseDefns newVal);
        [propget, id(PULSED_DC_TRAIN_BASE+3), helpstring("AC Component amplitude")]
            HRESULT ac_ampl([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical *pVal);
        [propputref, id(PULSED_DC_TRAIN_BASE+3), helpstring("AC Component amplitude")]
            HRESULT ac_ampl([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(PULSED_DC_TRAIN_BASE+4), helpstring("AC Component frequency")]
           HRESULT freq([out, retval, custom(GUID_DEFAULTVALUE, "0 Hz")] Frequency *pVal);
        [propputref, id(PULSED_DC_TRAIN_BASE+4), helpstring("AC Component frequency")]
            HRESULT freq([in, custom(GUID_DEFAULTVALUE, "0 Hz")] Frequency newVal);
    };
    [
        uuid(5A00D58E-F7A7-4285-AB6D-55D533A63230),
        helpstring("PULSED_DC_TRAIN class"),
        noncreatable
    ]
    coclass PULSED_DC_TRAIN
    {
        [default] interface IPULSED_DC_TRAIN;
    };


//RADAR_RX_SIGNAL
    [
        object,
        uuid(7D866851-EDEA-4A99-9801-658439350C9C),
        dual,
        helpstring("IRADAR_RX_SIGNAL Interface"),
        pointer_default(unique)
    ]
    interface IRADAR_RX_SIGNAL : ITSF
    {
        enum {RADAR_RX_SIGNAL_BASE=(TSF_BASE+256)};

        [propget, id(RADAR_RX_SIGNAL_BASE+1), helpstring("Atten")]
            HRESULT atten([out, retval, custom(GUID_DEFAULTVALUE, "1")] Ratio *pVal);
        [propputref, id(RADAR_RX_SIGNAL_BASE+1), helpstring("Atten")]
            HRESULT atten([in, custom(GUID_DEFAULTVALUE, "1")] Ratio newVal);
        [propget, id(RADAR_RX_SIGNAL_BASE+2), helpstring("Range of simulated target")]
            HRESULT range([out, retval] Distance *pVal);
        [propputref, id(RADAR_RX_SIGNAL_BASE+2), helpstring("Range of simulated target")]
            HRESULT range([in] Distance newVal);
       [propget, id(RADAR_RX_SIGNAL_BASE+3), helpstring("Rate of change of rate change")]
            HRESULT range_accn([out, retval, custom(GUID_DEFAULTVALUE, "0")] Acceleration
*pVal);
        [propputref, id(RADAR_RX_SIGNAL_BASE+3), helpstring("Rate of change of rate
change")]
            HRESULT range_accn([in, custom(GUID_DEFAULTVALUE, "0")] Acceleration newVal);
       [propget, id(RADAR_RX_SIGNAL_BASE+4), helpstring("Rate of change of target range")]

            HRESULT range_rate([out, retval, custom(GUID_DEFAULTVALUE, "0")] Speed *pVal);
```

```
        [propputref, id(RADAR_RX_SIGNAL_BASE+4), helpstring("Rate of change of target
range")]
            HRESULT range_rate([in, custom(GUID_DEFAULTVALUE, "0")] Speed newVal);
        [propget, id(RADAR_RX_SIGNAL_BASE+5), helpstring("Proportion of Tx pulses
returned")]
            HRESULT reply_eff([out, retval, custom(GUID_DEFAULTVALUE, "100%")] Ratio
*pVal);
        [propputref, id(RADAR_RX_SIGNAL_BASE+5), helpstring("Proportion of Tx pulses
returned")]
            HRESULT reply_eff([in, custom(GUID_DEFAULTVALUE, "100%")] Ratio newVal);
    };
    [
        uuid(7D866851-EDFA-4A99-9801-658439350C90),
        helpstring("RADAR_RX_SIGNAL class"),
        noncreatable
    ]
    coclass RADAR_RX_SIGNAL
    {
        [default] interface IRADAR_RX_SIGNAL;
    };


//RADAR_TX_SIGNAL
    [
        object,
        uuid(333E8A10-DB09-4E9E-A924-DEDA53F34DA2),
        dual,
        helpstring("IRADAR_TX_SIGNAL Interface"),
        pointer_default(unique)
    ]
    interface IRADAR_TX_SIGNAL : ITSF
    {
        enum {RADAR_TX_SIGNAL_BASE=(TSF_BASE+256)};

        [propget, id(RADAR_TX_SIGNAL_BASE+1), helpstring("Tx signal amplitude")]
            HRESULT ampl([out, retval] Physical *pVal);
        [propputref, id(RADAR_TX_SIGNAL_BASE+1), helpstring("Tx signal amplitude")]
            HRESULT ampl([in] Physical newVal);
        [propget, id(RADAR_TX_SIGNAL_BASE+2), helpstring("Tx signal frequency")]
            HRESULT freq([out, retval] Frequency *pVal);
        [propputref, id(RADAR_TX_SIGNAL_BASE+2), helpstring("Tx signal frequency")]
            HRESULT freq([in] Frequency newVal);
        [propget, id(RADAR_TX_SIGNAL_BASE+3), helpstring("Initial delay")]
            HRESULT delay([out, retval, custom(GUID_DEFAULTVALUE, "0 s")] Time *pVal);
        [propputref, id(RADAR_TX_SIGNAL_BASE+3), helpstring("Initial delay")]
            HRESULT delay([in, custom(GUID_DEFAULTVALUE, "0 s")] Time newVal);
        [propget, id(RADAR_TX_SIGNAL_BASE+4), helpstring("Pulse duration")]
            HRESULT duration([out, retval] Time *pVal);
        [propputref, id(RADAR_TX_SIGNAL_BASE+4), helpstring("Pulse duration")]
            HRESULT duration([in] Time newVal);
        [propget, id(RADAR_TX_SIGNAL_BASE+5), helpstring("Pulse repetition frequency")]
            HRESULT prf([out, retval] Frequency *pVal);
        [propputref, id(RADAR_TX_SIGNAL_BASE+5), helpstring("Pulse repetition frequency")]
            HRESULT prf([in] Frequency newVal);
        [propget, id(RADAR_TX_SIGNAL_BASE+6), helpstring("Number of pulses")]
            HRESULT repetition([out, retval, custom(GUID_DEFAULTVALUE, "0")] int *pVal);
        [propput, id(RADAR_TX_SIGNAL_BASE+6), helpstring("Number of pulses")]
            HRESULT repetition([in, custom(GUID_DEFAULTVALUE, "0")] int newVal);
    };
    [
        uuid(333E8A10-DB09-4E9E-A924-DEDA53F34DA0),
        helpstring("RADAR_TX_SIGNAL class"),
        noncreatable
    ]
    coclass RADAR_TX_SIGNAL
    {
        [default] interface IRADAR_TX_SIGNAL;
    };


//RAMP_SIGNAL
    [
```

```
        object,
        uuid(C4E0A145-A6DE-454C-A3EF-D9959F8D4959),
        dual,
        helpstring("IRAMP_SIGNAL Interface"),
        pointer_default(unique)
    ]
    interface IRAMP_SIGNAL : ITSF
    {
        enum {RAMP_SIGNAL_BASE=(TSF_BASE+256)};

        [propget, id(RAMP_SIGNAL_BASE+1), helpstring("Ramp signal amplitude")]
            HRESULT ampl([out, retval] Physical *pVal);
        [propputref, id(RAMP_SIGNAL_BASE+1), helpstring("Ramp signal amplitude")]
            HRESULT ampl([in] Physical newVal);
        [propget, id(RAMP_SIGNAL_BASE+2), helpstring("DC offset")]
            HRESULT dc_offset([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(RAMP_SIGNAL_BASE+2), helpstring("DC offset")]
            HRESULT dc_offset([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
        [propget, id(RAMP_SIGNAL_BASE+3), helpstring("Ramp signal period")]
            HRESULT period([out, retval] Time *pVal);
        [propputref, id(RAMP_SIGNAL_BASE+3), helpstring("Ramp signal period")]
            HRESULT period([in] Time newVal);
        [propget, id(RAMP_SIGNAL_BASE+4), helpstring("Ramp signal time to rise")]
            HRESULT rise_time([out, retval] Time *pVal);
        [propputref, id(RAMP_SIGNAL_BASE+4), helpstring("Ramp signal time to rise")]
            HRESULT rise_time([in] Time newVal);
    };
    [
        uuid(C4E0A145-A6DE-454C-A3EF-D9959F8D4950),
        helpstring("RAMP_SIGNAL class"),
        noncreatable
    ]
    coclass RAMP_SIGNAL
    {
        [default] interface IRAMP_SIGNAL;
    };


//RANDOM_NOISE
    [
        object,
        uuid(29C8F5CF-0539-4986-A104-52D65E89C5E3),
        dual,
        helpstring("IRANDOM_NOISE Interface"),
        pointer_default(unique)
    ]
    interface IRANDOM_NOISE : ITSF
    {
        enum {RANDOM_NOISE_BASE=(TSF_BASE+256)};

        [propget, id(RANDOM_NOISE_BASE+1), helpstring("Noise signal amplitude")]
            HRESULT ampl([out, retval] Physical *pVal);
        [propputref, id(RANDOM_NOISE_BASE+1), helpstring("Noise signal amplitude")]
            HRESULT ampl([in] Physical newVal);
        [propget, id(RANDOM_NOISE_BASE+2), helpstring("Pseudo random noise frequency")]
            HRESULT freq([out, retval, custom(GUID_DEFAULTVALUE, "0")] Frequency *pVal);
        [propputref, id(RANDOM_NOISE_BASE+2), helpstring("Pseudo random noise frequency")]
            HRESULT freq([in, custom(GUID_DEFAULTVALUE, "0")] Frequency newVal);
        [propget, id(RANDOM_NOISE_BASE+3), helpstring("Pseudo random noise seed")]
            HRESULT seed([out, retval, custom(GUID_DEFAULTVALUE, "0")] long *pVal);
        [propput, id(RANDOM_NOISE_BASE+3), helpstring("Pseudo random noise seed")]
            HRESULT seed([in, custom(GUID_DEFAULTVALUE, "0")] long newVal);
    };
    [
        uuid(29C8F5CF-0539-4986-A104-52D65E89C5E0),
        helpstring("RANDOM_NOISE class"),
        noncreatable
    ]
    coclass RANDOM_NOISE
    {
        [default] interface IRANDOM_NOISE;
```

```
    };


//RESOLVER
    [
        object,
        uuid(8899C7D8-F946-46AF-9698-3B3FDE0C026F),
        dual,
        helpstring("IRESOLVER Interface"),
        pointer_default(unique)
    ]
    interface IRESOLVER : ITSF
    {
        enum {RESOLVER_BASE=(TSF_BASE+256)};

        [propget, id(RESOLVER_BASE+1), helpstring("Shaft angle")]
            HRESULT angle([out, retval, custom(GUID_DEFAULTVALUE, "0")] PlaneAngle *pVal);
        [propputref, id(RESOLVER_BASE+1), helpstring("Shaft angle")]
            HRESULT angle([in, custom(GUID_DEFAULTVALUE, "0")] PlaneAngle newVal);
        [propget, id(RESOLVER_BASE+2), helpstring("Reference amplitude")]
            HRESULT ampl([out, retval, custom(GUID_DEFAULTVALUE, "26 V")] Voltage *pVal);
        [propputref, id(RESOLVER_BASE+2), helpstring("Reference amplitude")]
            HRESULT ampl([in, custom(GUID_DEFAULTVALUE, "26 V")] Voltage newVal);
        [propget, id(RESOLVER_BASE+3), helpstring("Reference frequency")]
            HRESULT freq([out, retval, custom(GUID_DEFAULTVALUE, "400 Hz")] Frequency
*pVal);
        [propputref, id(RESOLVER_BASE+3), helpstring("Reference frequency")]
            HRESULT freq([in, custom(GUID_DEFAULTVALUE, "400 Hz")] Frequency newVal);
        [propget, id(RESOLVER_BASE+4), helpstring("Zero index")]
           HRESULT zero_index([out, retval, custom(GUID_DEFAULTVALUE, "0 rad")] PlaneAngle
*pVal);
        [propputref, id(RESOLVER_BASE+4), helpstring("Zero index")]
           HRESULT zero_index([in, custom(GUID_DEFAULTVALUE, "0 rad")] PlaneAngle newVal);
        [propget, id(RESOLVER_BASE+5), helpstring("Shaft angle rate")]
            HRESULT angle_rate([out, retval, custom(GUID_DEFAULTVALUE, "0 Hz")] Frequency
*pVal);
        [propputref, id(RESOLVER_BASE+5), helpstring("Shaft angle rate")]
            HRESULT angle_rate([in, custom(GUID_DEFAULTVALUE, "0 Hz")] Frequency newVal);
        [propget, id(RESOLVER_BASE+6), helpstring("Transformer Ratio")]
           HRESULT trans_ratio([out, retval, custom(GUID_DEFAULTVALUE, "1")] Ratio *pVal);
        [propputref, id(RESOLVER_BASE+6), helpstring("Transformer Ratio")]
            HRESULT trans_ratio([in, custom(GUID_DEFAULTVALUE, "1")] Ratio newVal);
    };
    [
        uuid(8899C7D8-F946-46AF-9698-3B3FDE0C0260),
        helpstring("RESOLVER class"),
        noncreatable
    ]
    coclass RESOLVER
    {
        [default] interface IRESOLVER;
    };


//RS_232
    [
        object,
        uuid(444194DE-209D-487C-B37C-5D6FD1F14E53),
        dual,
        helpstring("IRS_232 Interface"),
        pointer_default(unique)
    ]
    interface IRS_232 : ITSF
    {
        enum {RS_232_BASE=(TSF_BASE+256)};

        [propget, id(RS_232_BASE+1), helpstring("Data Word")]
            HRESULT data_word([out, retval] BSTR *pVal);
        [propput, id(RS_232_BASE+1), helpstring("Data Word")]
            HRESULT data_word([in] BSTR newVal);
        [propget, id(RS_232_BASE+2), helpstring("Baud Rate")]
            HRESULT baud_rate([out, retval, custom(GUID_DEFAULTVALUE, "9600")] int *pVal);
```

```
        [propput, id(RS_232_BASE+2), helpstring("Baud Rate")]
            HRESULT baud_rate([in, custom(GUID_DEFAULTVALUE, "9600")] int newVal);
        [propget, id(RS_232_BASE+3), helpstring("Data Bits")]
            HRESULT data_bits([out, retval, custom(GUID_DEFAULTVALUE, "8")] int *pVal);
        [propput, id(RS_232_BASE+3), helpstring("Data Bits")]
            HRESULT data_bits([in, custom(GUID_DEFAULTVALUE, "8")] int newVal);
        [propget, id(RS_232_BASE+4), helpstring("Parity")]
            HRESULT parity([out, retval, custom(GUID_DEFAULTVALUE, "None")] BSTR *pVal);
        [propput, id(RS_232_BASE+4), helpstring("Parity")]
            HRESULT parity([in, custom(GUID_DEFAULTVALUE, "None")] BSTR newVal);
        [propget, id(RS_232_BASE+5), helpstring("Stop Bits")]
            HRESULT stop_bits([out, retval, custom(GUID_DEFAULTVALUE, "1")] BSTR *pVal);
        [propput, id(RS_232_BASE+5), helpstring("Stop Bits")]
            HRESULT stop_bits([in, custom(GUID_DEFAULTVALUE, "1")] BSTR newVal);
        [propget, id(RS_232_BASE+6), helpstring("Flow Control")]
            HRESULT flow_control([out, retval, custom(GUID_DEFAULTVALUE, "None")] BSTR
*pVal);
        [propput, id(RS_232_BASE+6), helpstring("Flow Control")]
            HRESULT flow_control([in, custom(GUID_DEFAULTVALUE, "None")] BSTR newVal);
    };
    [
        uuid(444194DE-209D-487C-B37C-5D6FD1F14E50),
        helpstring("RS_232 class"),
        noncreatable
    ]
    coclass RS_232
    {
        [default] interface IRS_232;
    };


//SQUARE_WAVE
    [
        object,
        uuid(A978E1A2-3B51-4122-A0AB-2FD4F8A1AF22),
        dual,
        helpstring("ISQUARE_WAVE Interface"),
        pointer_default(unique)
    ]
    interface ISQUARE_WAVE : ITSF
    {
        enum {SQUARE_WAVE_BASE=(ITSF_BASE+256)};

        [propget, id(SQUARE_WAVE_BASE+1), helpstring("Square wave amplitude")]
            HRESULT ampl([out, retval] Physical *pVal);
        [propputref, id(SQUARE_WAVE_BASE+1), helpstring("Square wave amplitude")]
            HRESULT ampl([in] Physical newVal);
        [propget, id(SQUARE_WAVE_BASE+2), helpstring("Square wave period")]
            HRESULT period([out, retval] Time *pVal);
        [propputref, id(SQUARE_WAVE_BASE+2), helpstring("Square wave period")]
            HRESULT period([in] Time newVal);
        [propget, id(SQUARE_WAVE_BASE+3), helpstring("DC offset")]
            HRESULT dc_offset([out, retval, custom(GUID_DEFAULTVALUE, "0 V")] Physical
*pVal);
        [propputref, id(SQUARE_WAVE_BASE+3), helpstring("DC offset")]
            HRESULT dc_offset([in, custom(GUID_DEFAULTVALUE, "0 V")] Physical newVal);
    };
    [
        uuid(A978E1A2-3B51-4122-A0AB-2FD4F8A1AF20),
        helpstring("SQUARE_WAVE class"),
        noncreatable
    ]
    coclass SQUARE_WAVE
    {
        [default] interface ISQUARE_WAVE;
    };


//SSR_INTERROGATION
    [
        object,
        uuid(F947BEDD-0BC7-4419-9846-E90374E36AC9),
```

```
        dual,
        helpstring("ISSR_INTERROGATION Interface"),
        pointer_default(unique)
    ]
    interface ISSR_INTERROGATION : ITSF
    {
        enum {SSR_INTERROGATION_BASE=(TSF_BASE+256)};

        [propget, id(SSR_INTERROGATION_BASE+1), helpstring("P1 Amplitude")]
            HRESULT ampl([out, retval] Physical *pVal);
        [propputref, id(SSR_INTERROGATION_BASE+1), helpstring("P1 Amplitude")]
            HRESULT ampl([in] Physical newVal);
        [propget, id(SSR_INTERROGATION_BASE+2), helpstring("Interrogation mode")]
            HRESULT mode([out, retval, custom(GUID_DEFAULTVALUE, "1")] BSTR *pVal);
        [propput, id(SSR_INTERROGATION_BASE+2), helpstring("Interrogation mode")]
            HRESULT mode([in, custom(GUID_DEFAULTVALUE, "1")] BSTR newVal);
        [propget, id(SSR_INTERROGATION_BASE+3), helpstring("P3 Start Time ")]
            HRESULT p3_start([out, retval, custom(GUID_DEFAULTVALUE, "3us")] Time *pVal);
        [propputref, id(SSR_INTERROGATION_BASE+3), helpstring("P3 Start Time ")]
            HRESULT p3_start([in, custom(GUID_DEFAULTVALUE, "3us")] Time newVal);
        [propget, id(SSR_INTERROGATION_BASE+4), helpstring("P3 level")]
            HRESULT p3_level([out, retval, custom(GUID_DEFAULTVALUE, "1")] Ratio *pVal);
        [propputref, id(SSR_INTERROGATION_BASE+4), helpstring("P3 level")]
            HRESULT p3_level([in, custom(GUID_DEFAULTVALUE, "1")] Ratio newVal);
        [propget, id(SSR_INTERROGATION_BASE+5), helpstring("SLS Deviation")]
            HRESULT sls_dev([out, retval, custom(GUID_DEFAULTVALUE, "0 us")] Time *pVal);
        [propputref, id(SSR_INTERROGATION_BASE+5), helpstring("SLS Deviation")]
            HRESULT sls_dev([in, custom(GUID_DEFAULTVALUE, "0 us")] Time newVal);
        [propget, id(SSR_INTERROGATION_BASE+6), helpstring("SLS Level")]
            HRESULT sls_level([out, retval, custom(GUID_DEFAULTVALUE, "1")] Ratio *pVal);
        [propputref, id(SSR_INTERROGATION_BASE+6), helpstring("SLS Level")]
            HRESULT sls_level([in, custom(GUID_DEFAULTVALUE, "1")] Ratio newVal);
    };
    [
        uuid(F947BEDD-0BC7-4419-9846-E90374E36AC0),
        helpstring("SSR_INTERROGATION class"),
        noncreatable
    ]
    coclass SSR_INTERROGATION
    {
        [default] interface ISSR_INTERROGATION;
    };


//SSR_RESPONSE
    [
        object,
        uuid(C2A1609E-55BB-4098-AD3B-08AB6C21F92A),
        dual,
        helpstring("ISSR_RESPONSE Interface"),
        pointer_default(unique)
    ]
    interface ISSR_RESPONSE : ITSF
    {
        enum {SSR_RESPONSE_BASE=(TSF_BASE+256)};

        [propget, id(SSR_RESPONSE_BASE+1), helpstring("Carrier Amplitude")]
            HRESULT ampl([out, retval, custom(GUID_DEFAULTVALUE, "1")] Physical *pVal);
        [propputref, id(SSR_RESPONSE_BASE+1), helpstring("Carrier Amplitude")]
            HRESULT ampl([in, custom(GUID_DEFAULTVALUE, "1")] Physical newVal);
        [propget, id(SSR_RESPONSE_BASE+2), helpstring("P3 pulse start time")]
            HRESULT p3_start([out, retval, custom(GUID_DEFAULTVALUE, "3us")] Time *pVal);
        [propputref, id(SSR_RESPONSE_BASE+2), helpstring("P3 pulse start time")]
            HRESULT p3_start([in, custom(GUID_DEFAULTVALUE, "3us")] Time newVal);
        [propget, id(SSR_RESPONSE_BASE+3), helpstring("SSR Response Pulse Train")]
            HRESULT pulses([out, retval, custom(GUID_DEFAULTVALUE, "[]")] PulseDefns
*pVal);
        [propputref, id(SSR_RESPONSE_BASE+3), helpstring("SSR Response Pulse Train")]
            HRESULT pulses([in, custom(GUID_DEFAULTVALUE, "[]")] PulseDefns newVal);
    };
    [
        uuid(C2A1609E-55BB-4098-AD3B-08AB6C21F920),
```

```
        helpstring("SSR_RESPONSE class"),
        noncreatable
    ]
    coclass SSR_RESPONSE
    {
        [default] interface ISSR_RESPONSE;
    };


//STEP_SIGNAL
    [
        object,
        uuid(DA473EFE-2F0F-4A4F-96FE-A39BA96DB523),
        dual,
        helpstring("ISTEP_SIGNAL Interface"),
        pointer_default(unique)
    ]
    interface ISTEP_SIGNAL : ITSF
    {
        enum {STEP_SIGNAL_BASE=(TSF_BASE+256)};

        [propget, id(STEP_SIGNAL_BASE+1), helpstring("Step size")]
            HRESULT ampl([out, retval] Voltage *pVal);
        [propputref, id(STEP_SIGNAL_BASE+1), helpstring("Step size")]
            HRESULT ampl([in] Voltage newVal);
        [propget, id(STEP_SIGNAL_BASE+2), helpstring("DC offset")]
           HRESULT dc_offset([out, retval, custom(GUID_DEFAULTVALUE, "0")] Voltage *pVal);
        [propputref, id(STEP_SIGNAL_BASE+2), helpstring("DC offset")]
            HRESULT dc_offset([in, custom(GUID_DEFAULTVALUE, "0")] Voltage newVal);
        [propget, id(STEP_SIGNAL_BASE+3), helpstring("Step time")]
            HRESULT start_time([out, retval] Time *pVal);
        [propputref, id(STEP_SIGNAL_BASE+3), helpstring("Step time")]
            HRESULT start_time([in] Time newVal);
    };
    [
        uuid(DA473EFE-2F0F-4A4F-96FE-A39BA96DB520),
        helpstring("STEP_SIGNAL class"),
        noncreatable
    ]
    coclass STEP_SIGNAL
    {
        [default] interface ISTEP_SIGNAL;
    };


//SUP_CAR_SIGNAL
    [
        object,
        uuid(DCCE2F12-CFC2-11D6-860C-00010214C4D2),
        dual,
        helpstring("ISUP_CAR_SIGNAL Interface"),
        pointer_default(unique)
    ]
    interface ISUP_CAR_SIGNAL : ITSF
    {
        enum {SUP_CAR_SIGNAL_BASE=(TSF_BASE+256)};

        [propget, id(SUP_CAR_SIGNAL_BASE+1), helpstring("Carrier amplitude")]
            HRESULT car_ampl([out, retval] Voltage *pVal);
        [propputref, id(SUP_CAR_SIGNAL_BASE+1), helpstring("Carrier amplitude")]
            HRESULT car_ampl([in] Voltage newVal);
        [propget, id(SUP_CAR_SIGNAL_BASE+2), helpstring("Carrier frequency")]
            HRESULT car_freq([out, retval] Frequency *pVal);
        [propputref, id(SUP_CAR_SIGNAL_BASE+2), helpstring("Carrier frequency")]
            HRESULT car_freq([in] Frequency newVal);
        [propget, id(SUP_CAR_SIGNAL_BASE+3), helpstring("Modulation frequency")]
            HRESULT mod_freq([out, retval] Frequency *pVal);
        [propputref, id(SUP_CAR_SIGNAL_BASE+3), helpstring("Modulation frequency")]
            HRESULT mod_freq([in] Frequency newVal);
        [propget, id(SUP_CAR_SIGNAL_BASE+4), helpstring("Depth of modulation")]
            HRESULT mod_depth([out, retval] Ratio *pVal);
        [propputref, id(SUP_CAR_SIGNAL_BASE+4), helpstring("Depth of modulation")]
```

```
            HRESULT mod_depth([in] Ratio newVal);
    };
    [
        uuid(DCCE2F12-CFC2-11D6-860C-00010214C4D0),
        helpstring("SUP_CAR_SIGNAL class"),
        noncreatable
    ]
    coclass SUP_CAR_SIGNAL
    {
        [default] interface ISUP_CAR_SIGNAL;
    };


//SYNCHRO
    [
        object,
        uuid(30034740-EA13-4F52-A4EF-47C4CDA96FEE),
        dual,
        helpstring("ISYNCHRO Interface"),
        pointer_default(unique)
    ]
    interface ISYNCHRO : ITSF
    {
        enum {SYNCHRO_BASE=(TSF_BASE+256)};

        [propget, id(SYNCHRO_BASE+1), helpstring("Shaft angle")]
            HRESULT angle([out, retval, custom(GUID_DEFAULTVALUE, "0")] PlaneAngle *pVal);
        [propputref, id(SYNCHRO_BASE+1), helpstring("Shaft angle")]
            HRESULT angle([in, custom(GUID_DEFAULTVALUE, "0")] PlaneAngle newVal);
        [propget, id(SYNCHRO_BASE+2), helpstring("Reference amplitude")]
            HRESULT ampl([out, retval, custom(GUID_DEFAULTVALUE, "26 V")] Voltage *pVal);
        [propputref, id(SYNCHRO_BASE+2), helpstring("Reference amplitude")]
            HRESULT ampl([in, custom(GUID_DEFAULTVALUE, "26 V")] Voltage newVal);
        [propget, id(SYNCHRO_BASE+3), helpstring("Reference frequency")]
            HRESULT freq([out, retval, custom(GUID_DEFAULTVALUE, "400 Hz")] Frequency
*pVal);
        [propputref, id(SYNCHRO_BASE+3), helpstring("Reference frequency")]
            HRESULT freq([in, custom(GUID_DEFAULTVALUE, "400 Hz")] Frequency newVal);
        [propget, id(SYNCHRO_BASE+4), helpstring("Zero index")]
            HRESULT zero_index([out, retval, custom(GUID_DEFAULTVALUE, "0")] PlaneAngle
*pVal);
        [propputref, id(SYNCHRO_BASE+4), helpstring("Zero index")]
            HRESULT zero_index([in, custom(GUID_DEFAULTVALUE, "0")] PlaneAngle newVal);
        [propget, id(SYNCHRO_BASE+5), helpstring("Shaft angle rate")]
            HRESULT angle_rate([out, retval, custom(GUID_DEFAULTVALUE, "0")] Frequency
*pVal);
        [propputref, id(SYNCHRO_BASE+5), helpstring("Shaft angle rate")]
            HRESULT angle_rate([in, custom(GUID_DEFAULTVALUE, "0")] Frequency newVal);
        [propget, id(SYNCHRO_BASE+6), helpstring("Transformer Ratio")]
            HRESULT trans_ratio([out, retval, custom(GUID_DEFAULTVALUE, "1")] Ratio *pVal);
        [propputref, id(SYNCHRO_BASE+6), helpstring("Transformer Ratio")]
            HRESULT trans_ratio([in, custom(GUID_DEFAULTVALUE, "1")] Ratio newVal);
    };
    [
        uuid(30034740-EA13-4F52-A4EF-47C4CDA96FE0),
        helpstring("SYNCHRO class"),
        noncreatable
    ]
    coclass SYNCHRO
    {
        [default] interface ISYNCHRO;
    };


//TACAN
    [
        object,
        uuid(A9829AB1-9EBA-4440-9AD7-D5C8B8E95C42),
        dual,
        helpstring("ITACAN Interface"),
        pointer_default(unique)
    ]
```

```
    interface ITACAN : ITSF
    {
        enum {TACAN_BASE=(TSF_BASE+256)};

        [propget, id(TACAN_BASE+1), helpstring("Transponder Frequency")]
            HRESULT freq([out, retval, custom(GUID_DEFAULTVALUE, "962 MHz")] Frequency
*pVal);
        [propputref, id(TACAN_BASE+1), helpstring("Transponder Frequency")]
            HRESULT freq([in, custom(GUID_DEFAULTVALUE, "962 MHz")] Frequency newVal);
        [propget, id(TACAN_BASE+2), helpstring("Modulation Index")]
           HRESULT mod_index([out, retval, custom(GUID_DEFAULTVALUE, "0.3")] Ratio *pVal);
        [propputref, id(TACAN_BASE+2), helpstring("Modulation Index")]
            HRESULT mod_index([in, custom(GUID_DEFAULTVALUE, "0.3")] Ratio newVal);
        [propget, id(TACAN_BASE+3), helpstring("Magnetic Bearing")]
            HRESULT bearing([out, retval, custom(GUID_DEFAULTVALUE, "0")] PlaneAngle
*pVal);
        [propputref, id(TACAN_BASE+3), helpstring("Magnetic Bearing")]
            HRESULT bearing([in, custom(GUID_DEFAULTVALUE, "0")] PlaneAngle newVal);
        [propget, id(TACAN_BASE+4), helpstring("er Amplitude")]
            HRESULT car_ampl([out, retval] Voltage *pVal);
        [propputref, id(TACAN_BASE+4), helpstring("er Amplitude")]
            HRESULT car_ampl([in] Voltage newVal);
    };
    [
        uuid(A9829AB1-9EBA-4440-9AD7-D5C8B8E95C40),
        helpstring("TACAN class"),
        noncreatable
    ]
    coclass TACAN
    {
        [default] interface ITACAN;
    };


//TRIANGULAR_WAVE_SIGNAL
    [
        object,
        uuid(7C5304C2-A118-46D6-83F2-8AD2176B6161),
        dual,
        helpstring("ITRIANGULAR_WAVE_SIGNAL Interface"),
        pointer_default(unique)
    ]
    interface ITRIANGULAR_WAVE_SIGNAL : ITSF
    {
        enum {TRIANGULAR_WAVE_SIGNAL_BASE=(TSF_BASE+256)};

        [propget, id(TRIANGULAR_WAVE_SIGNAL_BASE+1), helpstring("Triangular wave signal
amplitude")]
            HRESULT ampl([out, retval] Physical *pVal);
        [propputref, id(TRIANGULAR_WAVE_SIGNAL_BASE+1), helpstring("Triangular wave signal
amplitude")]
            HRESULT ampl([in] Physical newVal);
        [propget, id(TRIANGULAR_WAVE_SIGNAL_BASE+2), helpstring("Trianglar wave signal
period")]
            HRESULT period([out, retval] Time *pVal);
        [propputref, id(TRIANGULAR_WAVE_SIGNAL_BASE+2), helpstring("Trianglar wave signal
period")]
            HRESULT period([in] Time newVal);
        [propget, id(TRIANGULAR_WAVE_SIGNAL_BASE+3), helpstring("DC offset")]
            HRESULT dc_offset([out, retval, custom(GUID_DEFAULTVALUE, "0")] Physical
*pVal);
        [propputref, id(TRIANGULAR_WAVE_SIGNAL_BASE+3), helpstring("DC offset")]
            HRESULT dc_offset([in, custom(GUID_DEFAULTVALUE, "0")] Physical newVal);
    };
    [
        uuid(7C5304C2-A118-46D6-83F2-8AD2176B6160),
        helpstring("TRIANGULAR_WAVE_SIGNAL class"),
        noncreatable
    ]
    coclass TRIANGULAR_WAVE_SIGNAL
    {
        [default] interface ITRIANGULAR_WAVE_SIGNAL;
```

```
    };


//VOR
    [
        object,
        uuid(F3F9EB3B-3BF2-4460-8BB9-192D7C71FC65),
        dual,
        helpstring("IVOR Interface"),
        pointer_default(unique)
    ]
    interface IVOR : ITSF
    {
        enum {VOR_BASE=(TSF_BASE+256)};

        [propget, id(VOR_BASE+1), helpstring("Carrier amplitude")]
            HRESULT car_ampl([out, retval, custom(GUID_DEFAULTVALUE, "2 mV")] Voltage
*pVal);
        [propputref, id(VOR_BASE+1), helpstring("Carrier amplitude")]
            HRESULT car_ampl([in, custom(GUID_DEFAULTVALUE, "2 mV")] Voltage newVal);
        [propget, id(VOR_BASE+2), helpstring("Carrier frequency")]
            HRESULT car_freq([out, retval, custom(GUID_DEFAULTVALUE, "107.975 MHz")]
Frequency *pVal);
        [propputref, id(VOR_BASE+2), helpstring("Carrier frequency")]
            HRESULT car_freq([in, custom(GUID_DEFAULTVALUE, "107.975 MHz")] Frequency
newVal);
        [propget, id(VOR_BASE+3), helpstring("Radial bearing")]
            HRESULT phase([out, retval, custom(GUID_DEFAULTVALUE, "90deg")] PlaneAngle
*pVal);
        [propputref, id(VOR_BASE+3), helpstring("Radial bearing")]
            HRESULT phase([in, custom(GUID_DEFAULTVALUE, "90deg")] PlaneAngle newVal);
    };
    [
        uuid(F3F9EB3B-3BF2-4460-8BB9-192D7C71FC60),
        helpstring("VOR class"),
        noncreatable
    ]
    coclass VOR
    {
        [default] interface IVOR;
    };


};
```

Annex G

(normative)

Carrier language requirements

G.1 Carrier language requirements

This annex describes the minimum requirements for a suitable carrier language for the STD methodology. The requirements address data definition, data processing, and control structures.

G.1.1 General requirements

The carrier language will 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 will control the test instrumentation, which may be part of or connected to the host system, directly or indirectly.

G.1.2 Human interface and communication

The carrier language shall be able to communicate with the operator via the host system in order that a test requirement may pass instructions for manual interventions. It shall provide support so that the operator shall be able to provide the input required by the test requirement (e.g., serial numbers, identifiers). The human interface may be provided via an operating system.

G.2 IDL

The carrier language shall support the IDL as defined in the DCE Specifications [B5].

G.3 Data types

The carrier language shall provide either a set of data types or a set of language constructs to establish, label, and identify data types. Any data type shall be accessible at the outer structural level where it is established and at any inner nested structural level.

The data types defined in this clause shall support the data types defined in the IDL. Table G.1 shows the relationship between the data types required, their IDL names, and the carrier language data type that supports them.

Table G.1—Data types used in STD

Name used in STD	IDL name	XML	Supported by (in carrier language)
Array of …..	SAFEARRAY(…..)	List …	Array data type
Character String	BSTR	xs:string	16-bit character data type
Enumeration	enum	xs:string (restricted)	Enumeration

Table G.1—Data types used in STD *(continued)*

Name used in STD	IDL name	XML	Supported by (in carrier language)
Measurement Flags	VARIANT_BOOL	xs:boolean	Boolean data type
Numeric			
integer (Numeric_integer)	long	xs:long	Long integer (32 bit) data type
real (Numeric_real)	double	xs:double	Double-precision real data type
Any type	VARIANT	#any	Implementation dependent
Array of Any Type	VARIANT	List	Implementation dependent
NOTE—The numeric value of any physical type shall be supported by the real data type (double).			

G.3.1 Enumeration data type

To establish, label, and identify sets of enumerated data.

G.3.2 Integer data type

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 integer type data shall be in the range of at least –2 147 483 647 to +2 147 483 647.

G.3.3 Real data type

G.3.3.1 Fixed-point number

To establish, label, and identify decimal fixed-point numbers. 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), followed by a decimal point, followed by one or more digits (0 through 9).

For single-precision real numbers, the minimum precision of the coded representation of fixed-point data shall be at least 15 significant digits over a magnitude range of at least $+10^{38}$ to -10^{38}.

For double-precision real numbers, the precision of the coded representation of fixed-point data shall be at least 15 significant digits over a magnitude range of at least $+10^{307}$ to -10^{307}.

G.3.3.2 Decimal floating-point number

To establish, label, and identify decimal floating-point numbers. A decimal floating-point number is written in the form '± n.mE ± p', where *n*, *m*, and *p* are numerical strings consisting of one or more digits (0 through 9).

For single-precision real numbers, the minimum precision of the coded representation of floating-point data shall be at least 15 significant digits over a magnitude range of at least $+10^{38}$ to -10^{38}.

For double-precision real numbers, the precision of the coded representation of floating-point data shall be at least 15 significant digits over a magnitude range of at least $+10^{307}$ to -10^{307}.

G.3.4 Character data type

To establish, label, and identify one ASCII 8-bit character or a string of ASCII 8-bit characters.

To establish, label, and identify one ASCII 16-bit character or a string of ASCII 16-bit characters.

G.3.5 Boolean data type

To establish, label, and identify the data values TRUE or FALSE.

G.3.6 File data type

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.3.7 Array data type

To establish, label, and identify either a one-dimensional or a multi-dimensional 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.3.8 Record data type

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.3.9 Variables and constants

To establish, label, and identify variables and constants. A unique data type shall be assigned to an established variable or constant. A facility for initializing variables shall be provided.

Any created variable or constant shall be accessible at the structural level where it is established and at any inner nested structural level.

G.4 Data-processing requirements

To provide data-processing statements having the capability to assign a value to a variable, to perform calculations on values, and to make comparisons of values. The data-processing requirements are further defined in G.4.1 through G.4.10.

G.4.1 Data manipulation

To do one of the following:

— To assign a value to a variable, or

— To evaluate an expression on the right side of an assignment statement and then assign the value of an expression to the variable on the left side.

One or more evaluations and assignments may be made.

G.4.2 Arithmetic operators

To perform arithmetic operations on arguments in an expression.

The following arithmetic operators shall be provided:

a) Addition
b) Subtraction
c) Multiplication
d) Floating-point division
e) Exponentiation
f) Modulo (result remainder)
g) Integer division
h) Unary addition
i) Unary subtraction

G.4.3 Relational operators

To perform relational operations on arguments in an expression. The result of a relational operation shall be of Boolean data type.

The following relational operators shall be provided:

a) Equal to
b) Not equal to
c) Greater than
d) Less than
e) Greater than or equal to
f) Less than or equal to

G.4.4 Logical operators

To perform logical operations on one or more arguments of either a bit or Boolean data type in an expression. The result of a logical operation on a Boolean data type shall be a Boolean data type.

The following logical operators shall be provided:

a) Logical NOT
b) Logical exclusive OR
c) Logical AND
d) Logical OR
e) Bitwise exclusive OR
f) Bitwise AND
g) Bitwise OR

h) Ones complement (bitwise NOT).

G.4.5 Other operators

The following additional operators shall be provided:

a) Concatenation of (two or more) character strings
b) Concatenation of (two or more) bit strings

G.4.6 Mathematical functions

To provide mathematical functions that operate on integer and real data types.

Functions shall be provided to perform the following:

a) Compute the integer part of a real data type number
b) Round an argument to the nearest integer
c) Truncate an argument to an integer
d) Compute an absolute value
e) Compute a sine
f) Compute a cosine
g) Compute a tangent
h) Compute an arctangent (in degrees and radians)
i) Compute an arcsine (in degrees and radians)
j) Compute an arccosine (in degrees and radians)
k) Compute a natural logarithm
l) Compute a common logarithm
m) Compute an antilogarithm
n) Compute an exponential function (*e* to a power of *x*)
o) Compute a square root
p) Compute a hypotenuse of a right triangle
q) Compute a Bessel function
r) Return the larger (maximum) of two numbers
s) Return the smaller (minimum) of two numbers
t) Add two binary numbers
u) Subtract two binary numbers
v) Multiply two binary numbers
w) Divide two binary numbers
x) Generate cyclic redundancy check (CRC) characters
y) Check CRC characters
z) Compute Fourier transforms

G.4.7 File-handling functions

To provide functions that operate with files on the host system.

Functions shall be provided to perform the following:

a) Create a file
b) Delete a file
c) Read from a file
d) Write to a file
e) Test to determine whether a file exists (returns a result of true or false)
f) Test for the end of a file (returns a result of true or false)
g) Obtain the size of a file, i.e., the number of records within a file (returns an integer value)

G.4.8 Type conversion functions

To provide functions that convert one data type to another

Functions shall be provided to perform the following:

a) Convert an integer data type into a character string
b) Convert a real data type into a character string
c) Convert a character string (containing numeric characters) to integer
d) Convert a character string (containing numeric characters) to real
e) Convert a character into a bit string
f) Convert a bit string into a character
g) Convert a character to the integer value of an ASCII character code
h) Convert the integer value of an ASCII character code to a character

G.4.9 String related functions

To provide functions that operate on either bit strings or character strings.

Functions shall be provided to perform the following string manipulations and tests:

a) Determine the length of a string, i.e., returns the length of a character string or a bit string (returns an integer value)
b) Determine the location of a string within another string, i.e., determines the position of the first occurrence of a string within another string (returns an integer value)
c) Determine the number of occurrences of a string within another string (returns an integer value)
d) Copy a substring from a string, i.e., copy a substring determined by its start position and length from another string
e) Delete a substring from a string, i.e., delete a substring determined by its start position and length from another string
f) Insert a substring into another string, i.e., insert a substring into another string at a defined start position
g) Rotate a bit string, i.e., rotate the contents of a bit string to the left or right
h) Shift a bit string, i.e., shift the contents of a bit string to the left or right
i) Test for an alphanumeric character, i.e., determine whether a character within a character string is alphanumeric
j) Test for an alpha character, i.e., determine whether a character within a character string is alpha (returns a result of true or false)
k) Test for a control character, i.e., determine whether a character within a character string is numeric (returns a result of true or false)

l) Check for even parity, i.e., determine whether the parity of a bit string is even (returns a result of true or false)

m) Check for odd parity, i.e., determine whether the parity of a bit string is odd (returns a result of true or false)

G.4.10 Other functions

To provide the following additional functions:

a) Date (returns the current date from the host system)

b) Time (returns the current time from the host system)

G.5 Control structures

The carrier language shall provide the control structures defined in G.5.1 through G.5.5.

G.5.1 If

To branch between two segments of clearly delimited code dependent upon the condition of an expression.

It shall be possible to nest the IF control structure.

G.5.2 Else

To provide optional branching to segments of clearly delimited code. It is used in conjunction with an IF control structure.

G.5.3 Case

To branch to one or more segments of clearly delimited code dependent upon the evaluation of an expression. Each code segment shall be executed sequentially unless a break is encountered. A break directs the program control to the end of the case structure. A default section is mandatory.

G.5.4 For

To allow the repetitive execution of a segment of procedural statements, to establish the bounds of the segment, and to identify a control variable that will be assigned a value prior to each iteration of the segment.

There shall be two forms of the FOR control structure as follows:

a) List-Form, for which the control variable is a list of values.

b) Sequence-Form, for which the control variable is a range of values.

G.5.5 While

To identify the logical condition under which a segment of procedural code is to be interactively performed while a specified condition is valid within the boundaries of the segment of code.

There shall be two forms of the WHILE control structure as follows:

a) When the condition is evaluated at the start of a segment of code and there are zero or more iterations.

b) When the condition is evaluated at the end of the segment of code and there is at least one iteration.

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.

H.2 Elements of the TPL

The TPL comprises two 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.

H.3 Structure of test requirements

A test requirement written using the TPL will include the following elements:

a) Pragmas, such as native language *includes*, *defines*, and 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 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 UUT.

H.5.1 Definition of signal statements

Signal statement definitions have the following components:

a) Description, i.e., a formal textual description of the signal statement

b) Language neutral representation, i.e., shows the syntax and semantics of test statements, including test statement name and formal parameter list

c) Mapping information, i.e., specifies the functionality of the test statement. A simple pseudo code notation has been adopted as the language for describing the functionality of the test statement and how it relates to the BSCs and TSF signals.

The mapping component will be used by test requirement authors to describe the processing carried out by the signal statement. For system implementers, it specifies the mapping of STD test statements onto the methods and attributes of BSCs and TSF signals.

H.5.2 Structure of signal statements

Test program language statements are focused on single actions. Single action test statements describe a critical testing action that cannot be further subdivided with respect to the UUT. These statements are used to describe sources, sensors, events, test actions, and test comparisons.

Each test statement follows a similar format, although each has its own specific syntax and includes differences due to the particular requirements of the test statement. In general terms, the structure of each statement is as follows:

a) Each TPL statement starts with a keyword defining the function of the statement, such as setup, connect, or enable.

b) Normally, signal information is then given to describe the signal that is to be applied, measured, or otherwise referenced. The signal information normally comprises the <TSFClass>, followed by one or more Attribute-Value groups. The Attribute-Value group comprises a <TSFClass attribute>, an optional <Qualifier>, and a <Value>. The <TSFClass attribute> shall be valid for the <TSFClass>. The <Value> contains the numeric value of the attribute (which may be in the form of a variable or a literal) and may also include the dimension, tolerance, and range information. The optional <Qualifier> indicates how the attribute is observed. If no <Qualifier> is specified, then true root mean square (trms) is assumed.

c) If required, optional synchronization and gating information can be added. Keywords sync and gate are used to reference objects defined in other TPL statements.

d) A user-defined object name is then given to the signal or action.

H.5.3 Syntax of signal statements

The syntax employed in the formal description definitions is as follows:

— **Bold** indicates a TPL keyword or symbol (see note after this list).
— < > (angle brackets) denote a user-supplied name or literal, e.g., <name>.
— { } (braces) indicate a group.
— [ ] (brackets) indicate an optional field or element.
— | (vertical bar) indicates that the elements on each side of the bar are alternatives.
— * (asterisk) indicates that the previous element is repeated 0 or more times.
— + (plus symbol) indicates that the previous element is repeated 1 or more times.
— , (comma) is used as a parameter separator.
— ; (semicolon) is used as a statement terminator.

NOTE—Where a TPL symbol could be confused with a syntax symbol, the TPL symbol is in bold and underscored. Hence, if a brace is used as a TPL symbol, it is defined as **{**. The symbol is written in the TPL without the underscore.

H.6 Mapping of test statements to carrier language

Each test statement definition includes a mapping to the carrier language. As the carrier language may vary with different implementations, the mapping uses a pseudo-language.

The following program words are used in the language mapping:

— *Declare … as* indicates a class declaration.

— *Assign* indicates an assignment statement in which an object is assigned to a variable.

— *Comment* indicates a (nonexecutable) comment statement.

— No language word indicates an assignment statement in which a value is assigned to a variable.

For example, in VisualBasic the words *Declare*, *as*, *Assign*, and *Comment* would be replaced by "Dim", "As", "Set" and "'", respectively.

The syntax employed in the language mapping is similar to that used in the formal description definitions:

— **Bold** indicates a keyword.

— < > (angle brackets) denote a user-supplied name or string, e.g., <name>.

— { } (braces) indicate a group.

— [ ] (brackets) indicate an optional field or element.

— | (vertical bar) indicates that the elements on each side of the bar are alternatives.

— * (asterisk) indicates that the previous element is repeated 0 or more times.

— + (plus symbol) indicates that the previous element is repeated 1 or more times.

Throughout this annex, the use of a **ResourceManager** object named ***STD*** is used. It is assumed that this object was previously declared and instantiated as a **ResourceManager** object in the carrier language.

H.7 Test statement definitions

TPL statements are focused on single actions. Single action test statements describe a critical testing action that cannot be further subdivided with respect to the UUT. In the following statement definitions, the keywords are in bold for the sake of clarity. It is not necessary to use bold in a TPL requirement.

H.7.1 Setup statements

Setup is used to describe (but not create or invoke) events, sources, and sensors.

H.7.1.1 Setup source statement

The setup statement for a source describes a signal to be applied to a UUT.

H.7.1.1.1 Formal description

```
Setup <TSFClass> {<TSFClass attribute>[<Qualifier>]<Value>}
                 {,<TSFClass attribute>[<Qualifier>]<Value>}*
[sync to <EventSyncName>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}]
[as [<identifier>]]  source <SourceSignalName>;
```

where

<EventSyncName>, <EventGateName>, <EventFromName>, and <EventToName> are previously defined event names.

[**as** [<identifier>]] **source** associates an arbitrary user-supplied object name <SourceSignalName> with the signal definition and a user-defined identifier used by the **Require** method.

NOTE—An appropriate {<TSFClass attribute>[<Qualifier>]<Value>} group shall be supplied for every attribute that does not have a valid default value.

H.7.1.1.2 Language mapping

```
Declare <SourceSignalName> as <TSFClass>
Assign <SourceSignalName> = STD.Require("<TSFClass>"[,<identifier>])
{<SourceSignalName>.<attribute>="<Qualifier> <Value>"}+
[Assign <SourceSignalName>.Sync = <EventSyncName>]
[{Assign <SourceSignalName>.Gate = <EventGateName>}
|{Assign <SourceSignalName>.Gate = {STD.Require("EventedEvent")
 Assign <SourceSignalName>.Gate.Enable = <EventFromName>
       Assign <SourceSignalName>.Gate.Disable = <EventToName>}]
```

H.7.1.2 Setup sensor statement

The setup statement for a sensor describes a signal to be monitored or measured.

H.7.1.2.1 Formal description

```
Setup <TSFClass><TSFClass measure_attribute> [<mQualifier>] [mValue]
{,<TSFClass attribute>[<Qualifier>]<Value>}*
[sync to <EventSyncName>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}]
[as [<identifier>]]   sensor  <SensorSignalName>;
```

where

<EventSyncName>, <EventGateName>, <EventFromName>, and <EventToName> are previously defined event names.

[**as** [<identifier>]] **sensor** associates an arbitrary user-supplied object name <SensorSignalName> with the signal definition and a user-defined identifier used by the **Require** method.

NOTES

1—The <TSFClass measure_attribute> to be measured may be any valid controllable TSFClass attribute for the specified <TSFClass>.

2—The <TSFClass measure_attribute> is the attribute to be measured (i.e., the measured attribute) and has no <Value> specified. Subsequent <TSFClass Attribute>s (Attribute-Value groups) provide additional signal description information.

H.7.1.2.2 Language mapping

```
Declare <SensorSignalName> as Measure
Assign  <SensorSignalName> = STD.Require("Measure" [,<identifier>])
        <SensorSignalName>.attribute="<measure_attribute>"
Assign  <SensorSignalName>.As = STD.Require("<TSFClass>")
```

```
      [<SensorSignalName>.As.<measure-attribute>="<mQualifier>
<mValue>"]
      {<SensorSignalName>.As.<attribute>="<Qualifier> <Value>"}+
[Assign <SensorSignalName>.Sync = <EvntSyncName>]
[{Assign <SensorSignalName>.Gate = <EventGateName>}
|{Assign <SensorSignalName>.Gate = {STD.Require("EventedEvent")
      Assign <SensorSignalName>.Gate.Enable = <EventFromName>
      Assign <SensorSignalName>.Gate.Disable = <EventToName>}]
```

H.7.1.3 Setup sensor statement (for undefined signal)

The setup statement for a sensor for an undefined signal describes a qualifier to be monitored or measured.

H.7.1.3.1 Formal description

```
Setup [Undefined_Signal] <Attribute>[<Qualifier>][<Errlmt>][<Range>]
[sync to <EventSyncName>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}]
[as [<identifier>]]  sensor  <SensorSignalName>;
```

where

<Attribute> is the physical type being observed by the monitor.

<EventSyncName>, <EventGateName>, <EventFromName>, and <EventToName> are previously defined event names.

[**as** [<identifier>]] **sensor** associates an arbitrary user-supplied object name <SensorSignalName> with the signal definition and a user-defined identifier used by the **Require** method.

Table H.1 shows the qualifiers that may be used with each attribute. If the attribute cannot be measured as a trms value, then a valid qualifier shall be supplied.

Table H.1—Attributes and qualifiers for use with an undefined signal

Attribute	Valid qualifiers
Voltage	inst, av, pk, trms, pk_pk
Current	inst, av, pk, trms, pk_pk
Power	inst, av, pk, trms
Frequency	inst, av, pk,
Resistance	Inst
Capacitance	Inst
Conductance	Inst
Inductance	Inst
Reactance	Inst
Susceptance	Inst

H.7.1.3.2 Language mapping

```
Declare <SensorSignalName> as <SensorFunction>
Assign <SensorSignalName> = STD.Require("<SensorFunction>"
[,<identifier>])
[Assign <SensorSignalName>.Sync = <EvntSyncName>]
[{Assign <SensorSignalName>.Gate = <EventGateName>}
|{Assign <SensorSignalName>.Gate = {STD.Require("EventedEvent")
       Assign <SensorSignalName>.Gate.Enable = <EventFromName>
       Assign <SensorSignalName>.Gate.Disable = <EventToName>}]
```

where

<SensorFunction> is described as <SensorType>(<Attribute>) where <Attribute> is an valid physical type and <SensorType> is mapped as

a) inst Instantaneous(<Type>)
b) trms RMS(<Type>)
c) pk Peak(<Type>)
d) pk_pk PeakToPeak(<Type>)
e) pk_pos Peak(<Type>)
f) pk_neg PeakNeg (<Type>)
g) av Average(<Type>)
h) inst_max MaxInstantaneous(<Type>)
i) inst_min MinInstantaneous (<Type>)

H.7.1.4 Setup signal-based event statement

The setup statement for a signal-based event describes a signal to be monitored to generate an event when the specified conditions are satisfied.

H.7.1.4.1 Formal description

```
Setup <Attribute>[<Qualifier>] <condition> <Value>
[sync to <EventSyncName>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}]
[as [<identifier>]]  event <EventName>;
```

where

[**as** [<identifier>]] **event** associates an arbitrary user-supplied name <EventName> with the event definition and a user-defined identifier used by the **Require** method.
<Attribute> is the physical type being observed by the monitor.
<Qualifier>s are defined in Table H.1.
<condition> is one of {**GT**|**LT**}

where

GT indicates that the monitored signal must be greater than the specified value in order to satisfy the condition and generate the event.
LT indicates that the monitored signal must be less than the specified value in order to satisfy the condition and generate the event.

<EventSyncName>, <EventGateName>, <EventFromName>, and <EventToName> are previously defined event names.

During the period that the signal-based-event is enabled, an event will occur at the instant that the specified signal condition is satisfied. An event will occur each time the signal condition is satisfied (following the condition becoming unsatisfied) until the event is disabled.

An associated event interval will occur between the condition becoming satisfied and the condition becoming unsatisfied.

The event <EventName> may, therefore, be used for synchronization and for gating.

H.7.1.4.2 Language mapping

```
Declare <EventName> as <SensorFunction>
Assign  <EventName> = STD.Require("<SensorFunction>")
        <EventName>.condition=<condition>
        <EventName>.Nominal=<Value>
[Assign <EventName>.Sync = <EvntSyncName>]
[{Assign <EventName>.Gate = <EventGateName>}
|{Assign <EventName>.Gate = {STD.Require("EventedEvent")
       Assign <EventName>.Gate.Enable = <EventFromName>
       Assign <EventName>.Gate.Disable = <EventToName>}]
```

where

<SensorFunction> is described as <SensorType>(<Attribute>) where <Attribute> is an valid physical type and <SensorType> is mapped as

a) inst Instantaneous(<Type>)
b) trms RMS(<Type>)
c) pk Peak(<Type>)
d) pk_pk PeakToPeak(<Type>)
e) pk_pos Peak(<Type>)
f) pk_neg PeakNeg (<Type>)
g) av Average(<Type>)
h) inst_max MaxInstantaneous(<Type>)
i) inst_min MinInstantaneous (<Type>)

H.7.1.5 Setup event-based event statement

The setup statement for an event-based event describes an event when the monitored events are present.

H.7.1.5.1 Formal description

```
Setup from <EventFrom> to <EventTo>
[sync to <EventSyncName>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}]
[as [<identifier>]]  event <EventName>;
```

where

[as [<identifier>]] **event** associates an arbitrary user-supplied name <EventName> with the event definition and a user-defined identifier used by the **Require** method.
<EventFrom> and <EventTo> are previously defined event names.
<EventSyncName>, <EventGateName>, <EventFromName>, and <EventToName> are previously defined event names.

During the period that the event-based-event is enabled, an event will occur at the instant that the <EventFrom> occurs. Subsequently, an event will occur each time the <EventFrom> occurs following an <EventTo> until the event is disabled.

An associated event interval will occur between the <EventFrom> event and the <EventTo> event.

The event <EventName> may, therefore, be used for synchronization and for gating.

H.7.1.5.2 Language mapping

```
Declare <EventName> as EventedEvent
Assign  <EventName> = STD.Require("EventedEvent")
Assign  <EventName>.Enable=<EventFrom>
Assign  <EventName>.Disable=<EventTo>
[Assign <EventName>.Sync = <EvntSyncName>]
[{Assign <EventName>.Gate = <EventGateName>}
|{Assign <EventName>.Gate = {STD.Require("EventedEvent")
       Assign <EventName>.Gate.Enable = <EventFromName>
       Assign <EventName>.Gate.Disable = <EventToName>}]
```

H.7.1.6 Setup time-based event statement

The setup statement for a time-based event describes an event when the specified time conditions are satisfied.

H.7.1.6.1 Formal description

```
Setup [after <TimeValue Delay>] [for <TimeValue Duration>]
      every <TimeValue Period> [<Integer Repetition> times]
[sync to <EventSyncName>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}]
[as [<identifier>]]  event <EventName>;
```

where

[as [<identifier>]] **event** associates an arbitrary user-supplied name <EventName> with the event definition and a user-defined identifier used by the **Require** method.
<TimeValue Delay>, <TimeValue Duration>, and <TimeValue Period> are <Value>s in which the <UnitSymbol> shall be a valid time interval symbol.
<Integer> is an expression that evaluates to a positive integer value.
<EventSyncName>, <EventGateName>, <EventFromName>, and <EventToName> are previously defined event names.

During the period that the time-based-event is enabled, an event will occur at the instant that the initial **after** time value expires. Subsequent events will occur with a period equal to the time defined following the **every** keyword until the event is disabled. The optional <Integer Repetition> **times** field defines the number of events to be generated. Omitting this field causes events to be generated continuously until the event is disabled.

The optional **for** <TimeValue_Duration> field defines the duration of the associated event interval. If this field is omitted, the event interval period will be undefined and should not be used for gating.

The event <EventName> may be used for synchronization and for gating if the event interval period is defined.

H.7.1.6.2 Language mapping

```
Declare <EventName> as TimedEvent
Assign  <EventName> = STD.Require("TimedEvent")
        [<EventName>.delay="<TimeValue Delay>"]
        [<EventName>.duration ="<TimeValue Duration >"]
        <EventName>.period="<TimeValue Period>"
        [<EventName>.repetition="<TimeValue Repetition>"]
[Assign <EventName>.Sync = <EvntSyncName>]
[{Assign <EventName>.Gate = <EventGateName>}
|{Assign <EventName>.Gate = {STD.Require("EventedEvent")
       Assign <EventName>.Gate.Enable = <EventFromName>
       Assign <EventName>.Gate.Disable = <EventToName>}]
```

H.7.1.7 Setup clock statement

The setup statement for a clock describes a stream of events at the specified frequency.

H.7.1.7.1 Formal description

```
Setup <frequency>|<period>
[sync to <EventSyncName>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}]
[as [<identifier>]]  clock <ClockName>;
```

where

[as [<identifier>]] **clock** associates an arbitrary user-supplied name <ClockName> with the clock definition and a user-defined identifier used by the **Require** method.
<frequency> is a <Value> in which the <UnitSymbol> shall be a valid frequency symbol.
<period> is a <Value> in which the <UnitSymbol> shall be a valid time interval symbol.
<EventSyncName>, <EventGateName>, <EventFromName>, and <EventToName> are previously defined event names.

While the clock is enabled, an event will occur with the frequency specified or a frequency derived from the period specified until the event is disabled.

An associated event interval will occur starting at the event with a duration equal to half the period.

H.7.1.7.2 Language mapping

```
Declare <ClockName> as Clock
Assign  <EventName> = STD.Require("Clock")
        <EventName>.clockRate="<period>"
[Assign <EventName>.Sync = <EvntSyncName>]
[{Assign <EventName>.Gate = <EventGateName>}
|{Assign <EventName>.Gate = {STD.Require("EventedEvent")
       Assign <EventName>.Gate.Enable = <EventFromName>
```

```
        Assign <EventName>.Gate.Disable = <EventToName>}]
```

H.7.1.8 Setup time interval measurement statement

The setup statement for a time interval measurement describes a requirement to measure the time between two events.

H.7.1.8.1 Formal description

```
Setup TimeInterval [<Errlmt>] [<Range>]
[sync to <EventSyncName>]
[gate]  from <EventFromName> [to <EventToName>]
[as [<identifier>]]  timer  <TimerName>;
```

where

[**as** [<identifier>]] **timer** associates an arbitrary user-supplied name <TimerName> with the time interval measurement sensor definition and a user-defined identifier used by the **Require** method.

<EventSyncName>, <EventFromName>, and <EventToName> are previously defined event names.

This statement facilitates the measurement of time between two different events or, if the **to** <EventTo> field is omitted, between subsequent occurrences of the <EventFrom> event.

H.7.1.8.2 Language mapping

```
Declare <TimerName> as TimeInterval
Assign  <TimerName> = STD.Require("TimeInterval")
Assign  <TimerName>.In = <EventFromName>.Out
[Assign <TimerName>.Gate = <EventToName>.Out]
[Assign <TimerName>.Sync = <EventSyncName>]
```

H.7.1.9 Setup event counter statement

The setup statement for an event counter describes a requirement to monitor an event stream and to count the number of events.

H.7.1.9.1 Formal description

```
Setup Events [<Errlmt>] [<Range>]
             of <Event>
[sync to <EventSyncName>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}]
[as [<identifier>]]  counter <CounterName>;
```

where

[**as** [<identifier>]] **counter** associates an arbitrary user-supplied name <CounterName> with the event counter definition and a user-defined identifier used by the **Require** method.

The < UnitSymbol> in <Errlmt> and <Range> (if used) shall be a null symbol.

<Event> is a previously defined <EventName>.

<EventSyncName>, <EventGateName>, <EventFromName>, and <EventToName> are previously defined event names.

This statement facilitates the count of the number of events (in an event stream) while the counter <Counter-Name> is enabled.

H.7.1.9.2 Language mapping

```
Declare <CounterName> as Counter
Assign  <CounterName> = STD.Require("Counter")
Assign  <CounterName>.In = <Event>.Out
[Assign <CounterName>.Sync = <EvntSyncName>]
[{Assign <CounterName>.Gate = <EventGateName>}
|{Assign <CounterName>.Gate = {STD.Require("EventedEvent")
       Assign <CounterName>.Gate.Enable = <EventFromName>
 Assign <CounterName>.Gate.Disable = <EventToName>}]
```

H.7.1.10 Setup signal statement

The setup statement for a signal describes a requirement to provide a user-defined signal.

H.7.1.10.1 Formal description

```
Setup <XMLSignalDescription>
[sync to <EventSyncName>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}]
 [as [<identifier>]] <SignalName>;
```

where

[**as** [<identifier>]] associates an arbitrary user-supplied object name <SignalName> with the signal definition and a user-defined identifier used by the **Require** method.

<XMLSignalDescription> can be either a XML string or a string variable containing the XML signal description.

<EventSyncName>, <EventGateName>, <EventFromName>, and <EventToName> are previously defined event names.

H.7.1.10.2 Language mapping

```
Declare <SignalName> as SignalFunction
Assign  <SignalName> = STD.Require(<XMLSignalDescription>
[,<identifier>])
[Assign <SignalName>.Sync = <EvntSyncName>]
[{Assign <SignalName>.Gate = <EventGateName>}
|{Assign <SignalName>.Gate = {STD.Require("EventedEvent")
       Assign <SignalName.Gate.Enable = <EventFromName>
       Assign <SignalName.Gate.Disable = <EventToName>}]
```

H.7.2 Reset statement

The reset statement resets and releases a previously setup signal requirement.

H.7.2.1 Formal description

```
Reset {  <SignalName>|<EventName>|ConnectionName>
        |<ClockName>|<TimerName>|<CounterName>
```

```
        {,<SignalName>|<EventName>|ConnectionName>
         |<ClockName>|<TimerName>|<CounterName>}* }|all
[timeout <TimeOutValue>];
```

where

<TimeOutValue> is an integer value (in milliseconds) representing the maximum time the system should allow for the signal to reset.

NOTES

1—The optional **all** keyword indicates that all setup signal requirements are to be released.

2—If the optional timeout field is not used, the implementation-specific standard timeout value will be assumed.

H.7.2.2 Language mapping

```
<Name>.Out.Stop [<TimeOutValue>]
Assign <Name> = Nothing
```

If the keyword **all** is used**,** the implementation shall enumerate through all signal names performing the operations in H.7.2.1.

H.7.3 Connect statement

Connect is used to invoke sources and sensors and to connect them to the UUT.

H.7.3.1 Connect source signal statement

The connect statement for a source signal invokes the signal and connects it to the specified UUT pins.

H.7.3.1.1 Formal description

```
Connect <SourceSignalName>[<ConnectionClass>] to
        {<ConnectionClassPinName> <Pin>[{,<Pin>}*]}+
[timeout <TimeOutValue>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}]
[[as [<identifier>]]  connection <ConnectionName>];
```

where

[as [<identifier>]] **connection** associates an arbitrary user-supplied name <ConnectionName> with the connection definition and a user-defined identifier used by the **Require** method.
<ConnectionClass> is a valid connection class name, e.g., **TwoWire**, **ThreePhaseDelta**.
<ConnectionClassPinName> is a valid connection class pin name, e.g., **hi**, **lo**.
<Pin> is the UUT pin identifier or special pin name, e.g., Common, Earth.
<EventGateName>, <EventFromName>, and <EventToName> are previously defined event names.

NOTES

1—The optional <ConnectionName> allows the same signal to be connected to different pins at different times.

2—The optional <ConnectionClass> is required only if there may be ambiguity about the meaning of <Connection-ClassPinName>s, e.g., between three-phase wye or delta where A, B, and C are both used. The underlying translation mechanism shall determine the <ConnectionClass> from <Pin> names, if a <ConnectionClass> is not supplied in the statement.

3—If the optional Gate field is included, the event or clock is enabled at the instant the event occurs; and the signal will be removed at the end of the event interval period

4—If a connection is gated, the signal is being hot-switched. If the connection is not gated, the signal path is made prior to the signal being initiated and is, therefore, cold-switched. The default mode is cold-switching if the connection is not gated.

H.7.3.1.2 Language mapping

```
Declare <ConnectionName> as <ConnectionClass>
Assign  <ConnectionName> = STD.Require("<ConnectionClass>")
       {<ConnectionName>.<ConnectionClassPinName> = <Pin>}+
Assign  <ConnectionName>.In = <SourceSignalName>.Out
[{Assign <ConnectionName>.Gate = <EventGateName>}
|{Assign <ConnectionName>.Gate = STD.Require("EventedEvent")
       Assign <ConnectionName>.Gate.Enable = <EventFromName>
 Assign <ConnectionName>.Gate.Disable = <EventToName>}]
<ConnectionName>.Out.Run [<TimeOutValue>]
```

H.7.3.2 Connect sensor signal statement

The connect statement for a sensor signal connects the signal monitor to the UUT pins on which the signal is to be measured.

H.7.3.2.1 Formal description

```
Connect {<ConnectionClassPinName> <Pin>[{,<Pin>}*]}+ to
<SensorSignalName>[<ConnectionClass>]
[timeout <TimeOutValue>]
 [gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}]
 [[as [<identifier>]]  connection <ConnectionName>];
```

where

[as [<identifier>]] **connection** associates an arbitrary user supplied name, <ConnectionName>, with the connection definition and a user-defined identifier used by the **Require** method.
<ConnectionClass> is a valid connection class name, e.g., **TwoWire**, **ThreePhaseDelta**.
<ConnectionClassPinName> is a valid connection class pin name, e.g., **hi**, **lo**.
<Pin> is the UUT pin identifier or special pin names, e.g., Common, Earth.
<EventGateName>, <EventFromName> and <EventToName> are previously defined event names.
<SensorSignalName> is a previously defined object from a setup sensor statement, following the **sensor** keyword.

NOTES

1—The optional <ConnectionName> allows the same signal to be connected to different pins at different times.

2—The optional <ConnectionClass> is required only if there may be ambiguity about the meaning of <Connection-ClassPinName>s, for example, between three-phase wye or delta where A, B, and C are used. The underlying translation mechanism shall determine the <ConnectionClass> from <Pin> names, if a <ConnectionClass> is not supplied in the statement.

H.7.3.2.2 Language mapping

```
Declare <connectionName> as <ConnectionClass>
Assign <ConnectionName> = STD.Require("<ConnectionClass>")
      {<ConnectionName>.<ConnectionClassPinName> = <Pin>}+
```

```
Assign <SensorSignalName>.In = <ConnectionName>.Out
[{Assign <ConnectionName>.Gate = <EventGateName>}
|{Assign <ConnectionName>.Gate = STD.Require("EventedEvent"
       Assign <ConnectionName>.Gate.Enable = <EventFromName>
       Assign <ConnectionName>.Gate.Disable = <EventToName>}]
<ConnectionName>.Out.Run [<TimeOutValue>]
```

H.7.3.3 Connect pin to pin statement

The connect pin to pin statement connects UUT pins together or to named pins.

H.7.3.3.1 Formal description

```
Connect  <Pin> and  < Pin >[{,<Pin>}*]
[timeout <TimeOutValue>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}]
[as [<identifier>]]  connection <ConnectionName>;
```

where

[**as** [<identifier>]] **connection** associates an arbitrary user-supplied name, <ConnectionName>, with the connection definition and a user-defined identifier used by the **Require** method.
<Pin> is the UUT pin identifier or special pin names, e.g., Common, Earth.
<EventGateName>, <EventFromName> and <EventToName> are previously defined event names.

H.7.3.3.2 Language mapping

```
Declare <connectionName> as <ConnectionClass>*
Assign  <ConnectionName> = STD.Require("<ConnectionClass>")*
       {<ConnectionName>.<ConnectionClassPinName> = <Pin>}+
Assign  <ConnectionName#1>.In = <ConnectionName#2>.Out
[{Assign <ConnectionName>.Gate = <EventGateName>}
|{Assign <ConnectionName>.Gate = STD.Require("EventedEvent"
       Assign <ConnectionName>.Gate.Enable = <EventFromName>
       Assign <ConnectionName>.Gate.Disable = <EventToName>}]
<ConnectionName>.Out.Run [<TimeOutValue>]
```

H.7.4 Disconnect statement

The disconnect statement removes the signal at the specified connections. It does not release the <Signal-Name>; therefore, the same signal does not have to be described many times with identical setup statements. A <ConnectionName> will be released.

H.7.4.1 Formal description

```
Disconnect {<SignalName>|<ConnectionName>|all
[timeout <TimeOutValue>];
```

NOTE—The optional **all** keyword indicates that all connected resources are disconnected.

H.7.4.2 Language mapping

```
<ConnectionName>.Out.Stop [<TimeOutValue>]
```

```
Assign <ConnectionName>.In = Nothing
Assign <ConnectionName> = Nothing
```

Or

```
<SignalName>.Out.Stop [<TimeOutValue>]
For Each conn in <SignalName>.Out
      Assign conn.In = Nothing
Next
```

If the keyword **all** is used, the implementation shall enumerate through all connections as described in H.7.4.1.

H.7.5 Enable statement

The enable statement causes the specified requirement to be enabled so that the events may be monitored or generated as appropriate.

H.7.5.1 Enable statement (general case)

The general case enable statement applies to all enabled requirements except the signal-based event.

H.7.5.1.1 Formal description

```
Enable <EventName>|<ClockName>|<TimerName>|<CounterName>
     {,<EventName>|<ClockName>|<TimerName>|<CounterName>}*
[timeout <TimeOutValue>]
[sync to <EventSyncName>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}];
```

where

<EventName> is a previously defined event except a signal-based event.
<ClockName> is any previously defined clock.
<TimerName> is any previously defined timer.
<CounterName> is any previously defined counter.
<EventSyncName>, <EventGateName>, <EventFromName> and <EventToName> are previously defined event names.
<TimeOutValue> is an integer value (in milliseconds) representing the maximum time the system should allow for the signal to be enabled.

NOTES

1—If the optional Sync or Gate fields are included, the event or clock is enabled at the instant the event occurs.

2—In the case of a Gate field, the signal will be removed at the end of the event interval period.

3—If the optional Timeout field is not used, the implementation-specific standard timeout value will be assumed.

H.7.5.1.2 Language mapping

```
[Assign <Name>.Sync = <EventSyncName>]
[{Assign <Name>.Gate = <EventGateName>}
|{Assign <Name>.Gate = STD.Require("EventedEvent")
      Assign <Name>.Gate.Enable = <EventFromName>
```

```
      Assign <Name>.Gate.Disable = <EventToName>}]
<Name>.Out.Run [<TimeOutValue>]
```

where

<Name> is one of <EventName>|<ClockName>|<TimerName>|<CounterName>.

H.7.5.2 Enable signal-based event statement

The enable statement for the signal-based event connects the signal monitor to the specified pins and enables the event generator.

H.7.5.2.1 Formal description

```
Enable  <EventName> on
            {<ConnectionClassPinName> <Pin>[{,<Pin>}*]}+
[timeout <TimeOutValue>]
[sync to <EventSyncName>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}];
```

where

<EventName> is a previously defined signal-based event.
<ConnectionClassPinName> is a valid connection class pin name, e.g., **hi**, **lo**.
<Pin> is the UUT pin identifier or special pin names, e.g., Common, Earth.
<EventSyncName>, <EventGateName>, <EventFromName> and <EventToName> are previously defined event names.
<TimeOutValue> is an integer value (in milliseconds) representing the maximum time the system should allow for the signal to be enabled.

NOTE—If the optional Timeout field is not used, the implementation-specific standard timeout value will be assumed.

H.7.5.2.2 Language mapping

```
[Assign <EventName>.Sync = <EventSyncName>]
[{Assign <EventName>.Gate = <EventGateName>]}
|{Assign <EventName>.Gate = STD.Require("EventedEvent")
      Assign <EventName>.Gate.Enable = <EventFromName>
      Assign <EventName>.Gate.Disable = <EventToName>}]
 Assign cnx = STD.Require("<ConnectionClass>")
 Assign <EventName>.In = cnx.Out
      {cnx.<ConnectionClassPinName> = "<Pin>"}+
 Assign cnx=Nothing
<EventName>.Out.Run [<TimeOutValue>]
```

H.7.6 Disable statement

H.7.6.1 Formal description

```
Disable {<EventName>|<ClockName>|<TimerName>|<CounterName>}|all
[timeout <TimeOutValue>];
```

where

<EventName> is any previously enabled event.
<ClockName> is any previously enabled clock.
<TimerName> is any previously enabled timer.
<CounterName> is any previously enabled counter.

NOTE—The optional **all** keyword indicates that all enabled resources are disabled.

H.7.6.2 Language mapping

```
<Name>.Out.Stop [<TimeOutValue>]
```

where

<Name> is one of <EventName>|<ClockName>|<TimerName>|<CounterName>.

If the keyword **all** is used, the implementation shall enumerate through all events performing the operations in H.7.6.1.

H.7.7 Read statement

Read returns into <Variable> the measured <TSFClassAttribute> of a setup and connected <SensorSignal-Name>.

H.7.7.1 Formal description

```
Read [(<samples>)]<SensorSignalName>|<TimerName>|<CounterName>
into <Variable>
[timeout <TimeOutValue>]
[sync to <EventSyncName>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}];
```

where

<Variable> is a previously declared valid carrier language variable used to store an array of the measured value.
<EventSyncName>, <EventGateName>, <EventFromName> and <EventToName> are previously defined event names.
<TimeOutValue> is an integer value (in milliseconds) representing the maximum time the system should allow for the signal to be read.

NOTES

1—If the optional Sync or Gate fields are included, the event or clock is enabled at the instant the event occurs.

2—In the case of a Gate field, the signal will be removed at the end of the event interval period.

3—If the measured attribute has multiple properties or multiple sets of properties, the <Variable> into which the values are to be saved shall be a previously declared valid carrier language array variable.

4—If the optional Timeout field is not used, the implementation-specific standard timeout value will be assumed.

H.7.7.2 Language mapping

```
[<SensorSignalName>.samples = <samples>]
[Assign <SensorSignalName>.Sync = <EventSyncName>]
[{Assign <SensorSignalName>.Gate = <EventGateName>}
```

```
|{Assign <SensorSignalName>.Gate = STD.Require("EventedEvent")
      Assign <SensorSignalName>.Gate.Enable = <EventFromName>
      Assign <SensorSignalName>.Gate.Disable = <EventToName>}]
<SensorSignalName>.Out.Run [<TimeOutValue>]
… wait for measurement
Set <Variable> = <SensorSignalName>.measurements
```

H.7.8 Change statement

Change adjusts the <NumericValue> of one or more <TSFClassAttribute>s of <SourceSignalName> identified by previous setup signal statements.

H.7.8.1 Formal description

```
Change <SourceSignalName>
{<TCFClassAttribute>[<Qualifier>]<Value>|{<Variable>[<UnitSymbol>]}}+
[timeout <TimeOutValue>]
[sync to <EventSyncName>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}];
```

where

<Qualifier> must remain the same as defined in the setup statement for the <SourceSignalName>.
<UnitSymbol> must remain the same as defined in the setup statement for the <SourceSignalName>.
<Value> must remain the same as defined in the setup statement for the <SourceSignalName> except that the <NumericValue> may change
<EventSyncName>, <EventGateName>, <EventFromName> and <EventToName> are previously defined event names
<TimeOutValue> is an integer value (in milliseconds) representing the maximum time the system should allow for the signal to change.

NOTES

1—If the optional Sync or Gate fields are included, the event or clock is enabled at the instant the event occurs.

2—In the case of a Gate field, the signal will be removed at the end of the event interval period.

3—If the optional Timeout field is not used, the implementation-specific standard timeout value will be assumed.

H.7.8.2 Language mapping

```
(<SourceSignalName>.<TCFClassAttribute>="<Modifer> <Value>"|Variable)+
[Assign < SourceSignalName>.Sync = <EventSyncName>]
[{Assign <SourceSignalName>.Gate = <EventGateName>}
|{Assign <SourceSignalName>.Gate = STD.Require("EventedEvent")
      Assign <SourceSignalName>.Gate.Enable = <EventFromName>
Assign <SourceSignalName>.Gate.Disable = <EventToName>}]
<SourceSignalName>.Out.Change [<TimeOutValue>]
```

H.7.9 Compare statement

The compare statement compares the value of <SensorSignalName> with the contents of the <Evaluation Field> and sets the sensor signal name's flags GO, NOGO, HI, and LO.

H.7.9.1 Formal description

```
Compare <SensorSignalName> <Evaluation Field>;
```

where

<SensorSignalName> is a previously declared sensor for which a read statement has been executed.
<Evaluation Field> is defined in H.7.9.1.1.

H.7.9.1.1 <Evaluation Field>

The syntax of the <Evaluation Field> is as follows:

```
{UL <NumericValue>[<UnitSymbol>]   LL <NumericValue>[<UnitSymbol>]}|
{LL <NumericValue>[<UnitSymbol>]   UL <NumericValue>[<UnitSymbol>]}|
{>  <NumericValue>[<UnitSymbol>]}|{>= <NumericValue>[<UnitSymbol>]}|
{<  <NumericValue>[<UnitSymbol>]}|{<= <NumericValue>[<UnitSymbol>]}|
{=  <NumericValue>[<UnitSymbol>]}|{<> <NumericValue>[<UnitSymbol>]}
```

RULE—<NumericValue> shall be of the same physical type and units as the measured attribute in the <SensorSignalName>.

NOTES

1—The optional <UnitSymbol> is not required to achieve a valid evaluation (see the above rule), but may be used to create clearer test requirements.

2—The measured attribute is saved in the variable associated with the <SensorSignalName> in the last read statement executed.

Given the possible values of the measured attribute attr and <NumericValue>(s) x and y, the measurement flags are set as follows:

Compare attr UL x LL y	attr > x	HI and NOGO
	y≥ attr ≥ x	GO
	attr < y	LO and NOGO
Compare attr > x	attr > x	GO
	attr ≤ x	LO and NOGO
Compare attr ≥ x	attr ≥ x	GO
	attr < x	LO and NOGO
Compare attr < x	attr < x	GO
	attr ≥ x	HI and NOGO
Compare attr ≤ x	attr ≤ x	GO
	attr > x	HI and NOGO
Compare attr = x	attr = x	GO
	attr <> x	NOGO
Compare attr <> x	attr = x	NOGO
	attr <> x	GO

H.7.9.2 Language mapping

```
[<SensorSignalName>.UL=<ULValue>]
[<SensorSignalName>.LL=<LLValue>]
[<SensorSignalname>.Nominal=<ConditionValue>]
[<SensorSignalName>.Condition=<Condition>]
Comment <SensorSignalName> GO,NOGO, HI, LO Flags are now set
```

H.7.10 Wait_For statement

The wait_for statement pauses execution for the specified <TimeValue>.

H.7.10.1 Formal description

Wait_For <TimeValue>

where

<TimeValue> is a <Value> in which the <UnitSymbol> shall be a valid time interval symbol.

H.7.10.2 Language mapping

The language mapping for a wait_for statement is implementation dependent; it waits for a time greater than the <TimeValue> shown in the following example:

Sleep <TimeValue>

H.8 Elements used in test statement definitions

H.8.1 <TSFClass>

The <TSFClass> is selected from either a BSC or TSF class name. Associated with the <TSFClass> is a <type>, which comprises the dependent and independent variables. To provide a full definition of the signal class, this information would be given in the following form:

```
<TSFClass> := <BSCClassName>|<TSFClassName> [(<dependent variable>
[,<independent variable>)]]
```

To simplify the preparation of test requirements, providing this information explicitly is not necessary if it can be determined implicitly from the TPL statement or if there is a default.

H.8.1.1 Dependent variable

Every signal has a default for the dependent variable (usually voltage), and several show other optional types (usually current and power). These dependent variables have been determined to be the most common valid types for the technology under consideration. This statement does not exclude the use of other dependent variables with the <TSFClass>.

The dependent variable may be determined from the <UnitSymbol> within the associated <Value>.

If no <UnitSymbol> is provided, then the default dependent variable is assumed unless explicitly defined.

H.8.1.2 Independent variable

The default independent variable is time. Therefore, unless the independent variable is to be some other variable (such as frequency), it need not be stated.

If the independent variable is to be provided explicitly, the dependent variable shall also be provided even if it is the default.

H.8.2 Attribute-Value groups

Attribute-Value groups appear in most TPL statements and may include a qualifier as shown in the following example:

```
<TSFClass attribute> [<Qualifier>] <Value>
```

H.8.2.1 <TSFClass attribute>

The <TSFClass attribute> shall be a valid property name for the <TSFClass>.

H.8.2.2 <Qualifier>

The optional <Qualifier> refers to the different ways of observing the <TSFClass attribute> and may be one of the following:

- trms – true root mean square value
- pk_pk – peak-peak value
- pk – peak value (see Note 1)
- pk_pos – positive peak value (see Notes 1 and 2)
- pk_neg – negative peak value (see Note 1)
- av – average value
- inst – instantaneous value
- inst_max – maximum instantaneous value
- inst_min – minimum instantaneous value

NOTES

1—When qualifiers pk, pk_pos, and pk_neg are used in sensor statements, they normally give a value equal to 1/2 the pk_pk value. This result is due to the limitation of measurement techniques.

2—The qualifier pk_pos is an alias for pk. It is provided for use in test requirements in which the writer wishes to ensure that there is no confusion between pk and pk_neg.

Figure H.1 shows the amplitude of a signal varying with time and illustrates the <Qualifier>s as applying to that signal.

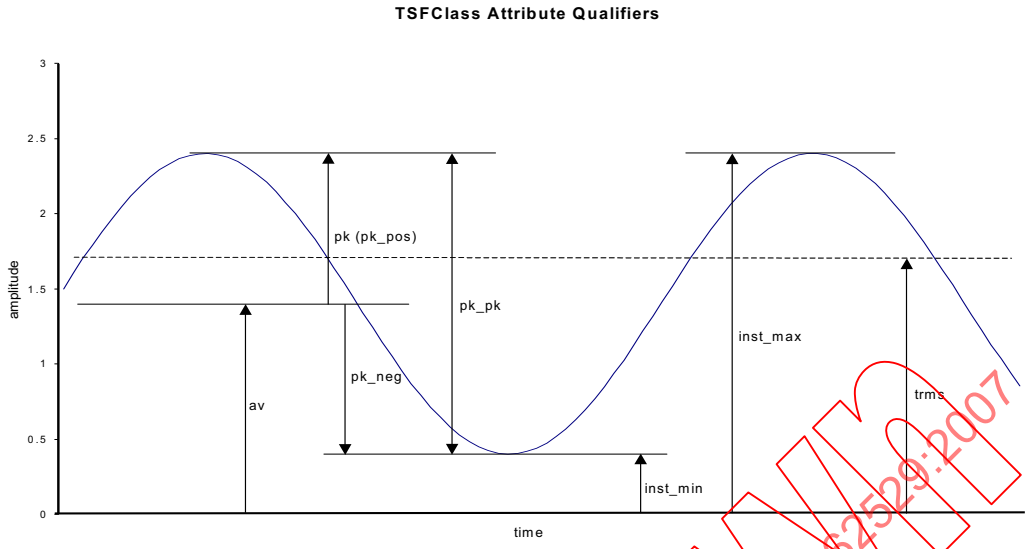


Figure H.1—Attribute qualifiers

H.8.2.3 <Value>

<Value> indicates the numeric value of the attribute together with the units of measure and any associated error limit and range information.

The syntax for <Value> is as follows:

```
<NumericExpression> [<UnitSymbol>] [<Errlmt>] [<Range>]
```

where

<NumericExpression> is an expression that evaluates to a valid <NumericValue>.
<NumericValue> is the numeric value of the physical type either in a variable (e.g., a previously declared carrier language variable) or as a literal (e.g., 3.0, 0.263, 293).
<UnitSymbol> is the symbol (from Table H.2) for the units of the numeric value (e.g., mV MHz, μs).
<Errlmt> is the required accuracy of the stimulus or measurement.
<Range> is the range of values that the stimulus is expected to provide or the sensor is expected to measure.

NOTES

1—If the optional<UnitSymbol> is omitted, then the value is assumed to be in the units without any metric prefix. If the value may take more than one unit, then the unit is assumed to be the first listed in Table H.2, i.e., the preferred SI unit.

2—If the optional <UnitSymbol> is used, it is recommended that the <UnitSymbol> should be the same throughout the <Value> string.

The syntax for <Errlmt> is as follows:

```
errlmt [±|+-|+|-]<NumericValue>[<UnitSymbol>]
[{-|:|to} <NumericValue>[<UnitSymbol>]]
```

where

<NumericValue> is always provided as an absolute/(positive) quantity.

The syntax for <Range> is as follows:

```
{range [+|-]<NumericValue>[<UnitSymbol>]
    to [+|-]<NumericValue>[<UnitSymbol>]}
|{range MAX|MIN <NumericValue> [<UnitSymbol>]}
```

H.8.2.4 Permissible quantities, units, and unit symbols

Table H.2 lists the quantities that may be used in the TPL together with units and their unit symbols (see Note 1 after the table). The table also shows the mapping to their physical types and units.

Table H.2—Quantities, units, and symbols for use with TPL

Quantity	Unit	SI unit	Unit symbol	Physical type mapping	Comment
Acceleration	meter per second squared	derived	m/s^2	Acceleration	
Admittance	—	—	—	—	See Note 14
Amount of information	byte	—	B	AmountOfInformation	
Amount of substance	mole	Base	mol	AmountOfSubstance	
Angular acceleration	radian per second squared	derived	rad/s^2	AngularAcceleration	
Angular velocity	radian per second	Derived	rad/s	AngularSpeed	
Area	square meter	derived	m^2	Area	
Capacitance	farad	Derived	F	Capacitance	
Concentration	mole per cubic meter	derived	mol/m^3	Concentration	
Current density	ampere per square meter	derived	A/m^2	CurrentDensity	
Dynamic viscosity	pascal second	derived	Pa·s	DynamicViscosity	
Electric charge	coulomb	Derived	C	Charge	
Electric charge density	coulomb per cubic meter	derived	C/m^3	ElectricChargeDensity	
Electric conductance	siemens	Derived	S	Conductance	See Note 14
Electric current	ampere	Base	A	Current	
Electric field strength	volt per meter	derived	V/m	ElectricFieldStrength	See Note 11
	newton per coulomb	derived	N/C	ElectricFieldStrength	See Note 11
Electric flux density	coulomb per square meter	derived	C/m^2	ElectricFluxDensity	
Electric potential difference	volt	derived	V	Voltage	

Table H.2—Quantities, units, and symbols for use with TPL *(continued)*

Quantity	Unit	SI unit	Unit symbol	Physical type mapping	Comment
Electric resistance	ohm	Derived	Ohm	Resistance	See Notes 5, 14
Electromotive force	volt	Derived	V	Voltage	
Energy	joule	Derived	J	Energy	
Energy density	joule per cubic meter	derived	J/m^3	EnergyDensity	
Entropy	joule per kelvin	derived	J/K	Entropy	
Exposure	coulomb per kilogram	derived	C/kg	Exposure	
Force	newton	Derived	N	Force	
Frequency	hertz	Derived	Hz	Frequency	
Heat	joule		J	Heat	
Heat capacity	joule per kelvin	derived	J/K	HeatCapacity	
Heat flux density	watt per square meter	derived	W/m^2	HeatFluxDensity	
Illuminance	lux	Derived	lx	Illuminance	
Impedance	—	—	—	—	See Notes 5, 14
Inductance	henry	Derived	H	Inductance	
Irradiance	watt per square meter	derived	W/m^2	Irradiance	
Kinematic viscosity	square meter per second	derived	m^2/s	KinematicViscosity	
Length	meter	Base	m	Distance	
	inch	—	in	Distance	See Note 13
	foot	—	ft	Distance	See Note 13
	mile (statute)	—	mi	Distance	See Note 13
	nautical mile	—	nmi	Distance	See Note 13
Luminance	candela per square meter	Derived	cd/m^2	Luminance	The use of nit is deprecated
Luminous Flux	lumen	Derived	lm	LuminFlux	
Luminous Intensity	candela	Base	cd	LuminInten	
Magnetic field strength	ampere per meter	Derived	A/m	MagneticFieldStrength	
Magnetic Flux	weber	Derived	Wb	MagneticFlux	
Magnetic flux density	tesla	Derived	T	MagneticFluxDensity	

Table H.2—Quantities, units, and symbols for use with TPL *(continued)*

Quantity	Unit	SI unit	Unit symbol	Physical type mapping	Comment
Mass	gram	kg (base)	g	Mass	See Note 2
Mass density	kilogram per square	Derived	kg/m^2	MassDensity	
MassFlow	kilogram per second	Derived	kg/s	MassFlow	
Molar energy	joule per mole	Derived	J/mol	MolarEnergy	
Molar entropy	joule per mole kelvin	Derived	J/(mol·K)	MolarEntropy	
Molar heat capacity	joule per mole kelvin	Derived	J/(mol·K)	MolarHeatCapacity	
Moment of force	newton meter	Derived	N·m	MomentOfForce	
Moment of inertia	kilogram meter squared	Derived	$kg·m^2$	MomentOfInertia	
Momentum	kilogram meter per sec-ond	Derived	kg·m/s	Momentum	
Permeability	henry per meter	Derived	H/m	Permeability	
Permittivity	farad per meter	Derived	F/m	Permittivity	
Plane Angle	radian	Derived	rad	PlaneAngle	
	degree	in use	°, deg	PlaneAngle	See Notes 8, 12
Power	watt	Derived	W	Power	
	decibel watt	—	dBW, dB(1 W)	Power	See Notes 3, 4
	decibel milliwatt	—	dBm, dB(1 mW)	Power	See Notes 3, 4
Power density	watt per square meter	Derived	W/m^2	PowerDensity	
Pressure	pascal	Derived	Pa	Pressure	
	millibar	—	mbar	Pressure	See Note 9
Radiance	watt per square meter steradian	Derived	$W/(m^2·sr)$	Radiance	
Radiant intensity	watt per steradian	Derived	W/sr	RadiantIntensity	
Ratio	—	—	dB, %, pc	Ratio	See Notes 3, 7
Reactance	Reactance	Derived	ohm	Ohm	See Notes 5, 14

Table H.2—Quantities, units, and symbols for use with TPL *(continued)*

Quantity	Unit	SI unit	Unit symbol	Physical type mapping	Comment
Solid Angle	steradian	Derived	sr	SolidAngle	
Specific energy	joule per kilogram	Derived	J/kg	SpecificEnergy	
Specific entropy	joule per kilogram kelvin	Derived	J/(kg·K)	SpecificEntropy	
Specific heat capacity	joule per kilogram kelvin	Derived	J/(kg·K)	SpecificHeatCapacity	
Specific volume	cubic meter per kilogram	Derived	m^3/kg	SpecificVolume	
Speed	meter per second	Derived	m/s	Speed	
Surface tension	newton per meter	Derived	N/m	SurfaceTension	
Susceptance	siemens	Derived	S	Susceptance	See Note 14
Thermal conductivity	watt per meter kelvin	Derived	W/(m·K)	ThermalConductivity	
Thermodynamic Temperature	kelvin	Base	K	Temperature	See Note 3
	degree Celsius	Derived	°C, degC	Temperature	See Note 6
	degree Fahrenheit	—	°F, degF	Temperature	See Notes 6, 13
Time	second	Base	s	Time	
	minute	in use	min	Time	See Notes 3, 12
	hour	in use	h	Time	See Notes 3, 12
Volume	cubic meter	Derived	m^3	Volume	
	liter	in use	L	Volume	See Note 12
Volume Flow	liter per second	Derived	L/s	VolumeFlow	

NOTES

1—It is preferred practice to leave one space between the numeric value and the unit symbol when defining a value. (See Note 8).

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

3—These units are not used with the SI prefixes.

4—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 abbreviations 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).

Table H.2—Quantities, units, and symbols for use with TPL *(continued)*

5—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.

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

7—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 (%).

8—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 abbreviation deg be used, e.g., Bearing deg. Using the degree symbol (°) with a variable may give rise to confusion if it was placed adjacent to the variable name.

9—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 (e.g., for barometric pressure). The value 1 mbar is equal to 1 hPa.

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

11—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.

12—For convenience, certain non-SI units are acceptable for use with SI. These units are marked as “in use” in the SI unit column of the table.

13—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.

14—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 *conductance* are reserved for future use as physical type names.

H.8.2.5 Unit prefixes

A unit string can be created by optionally adding one of the metric prefixes to the unit symbol from Table H.3.

Table H.3—Metric prefixes for use with TPL

Prefix	Name	Value	Comments
y	yocto	10^{-24}	
z	zepto	10^{-21}	
a	atto	10^{-18}	
f	femto	10^{-15}	
p	pico	10^{-12}	
n	nano	10^{-9}	
μ, u	micro	10^{-6}	See Note 1

Table H.3—Metric prefixes for use with TPL *(continued)*

Prefix	Name	Value	Comments
m	milli	10^{-3}	
c	centi	10^{-2}	See Note 2
d	deci	10^{-1}	See Note 2
h	hecto	10^{+2}	See Note 2
k	kilo	10^{+3}	
M	mega	10^{+6}	
G	giga	10^{+9}	
T	tera	10^{+12}	
P	peta	10^{+15}	
E	exa	10^{+18}	
Z	zetta	10^{+21}	
Y	yotta	10^{+24}	

NOTES

1—The preferred symbol for the prefix micro (i.e., μ) 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.

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).

H.9 Attributes with multiple properties

H.9.1 Entering literal data

Several TSF attributes require the entry of a set of multiple properties. Moreover, it may be necessary to enter more than one set of properties. For example, the PULSED_AC_TRAIN signal requires the attribute pulse_train to be entered as a series of pulses, and each pulse requires the properties start_time, pulse_width, and level-factor to be specified.

H.9.1.1 General case – multiple properties

The Attribute-Value group (used in source-type statements) becomes an Attribute-Properties-Values group where the attribute may have a series of properties, each with its own <Qualifier> and <Value>. The information for an attribute with a set of multiple properties would be entered as shown in H.9.1.1.1.

H.9.1.1.1 Attribute-Properties-Values group syntax (single set)

```
<TSFClass attribute> {<Property>[<Qualifier>]<Value>
                     {,<Property>[<Qualifier>]<Value>}*}
```

This information may be shown in an alternative format as follows to illustrate the language mapping:

```
<TSFClass attribute> {<Property1><Value1>
                     [,<Property2><Value2>
```

```
                    [,<Property3><Value3>
                                    ...]]}
```

H.9.1.1.2 Language mapping

```
 <SourceSignalName>.attribute.property1 = "<Value1>"
[<SourceSignalName>.attribute.property2 = "<Value2>"
[<SourceSignalName>.attribute.property3 = "<Value3>"
                                                    ...]]
```

H.9.1.2 General case – multiple sets

The Attribute-Properties-Values group information for a series of more than one set of multiple properties is an array of groups and would be entered as shown in H.9.1.2.1.

H.9.1.2.1 Attribute-Properties-Values group syntax (multiple sets)

```
<TSFClass attribute> [ {<Property>[<Qualifier>]<Value>
                        {,<Property>[<Qualifier>]<Value>}*}
                       {,{<Property>[<Qualifier>]<Value>
                        {,<Property>[<Qualifier>]<Value>}*}}*]
```

This information may be shown in an alternative format as follows to illustrate the language mapping:

```
<TSFClass attribute>
    [{<Property1><Value1>[,<Property2><Value2>[,<Property3><Value3>]]},
     {<Property1><Value1>[,<Property2><Value2>[,<Property3><Value3>]]},
     {<Property1><Value1>[,<Property2><Value2>[,<Property3><Value3>]]}]
     ...
```

H.9.1.2.2 Language mapping

```
 <SourceSignalName>.attribute.Add(1).property1 = "<Value1>(1)"
[<SourceSignalName>.attribute.Item(1).property2 = "<Value2>(1)"
[<SourceSignalName>.attribute.Item(1).property3 = "<Value3>(1)"]]
 <SourceSignalName>.attribute.Add(2).property1 = "<Value1>(2)"
[<SourceSignalName>.attribute.Item(2).property2 = "<Value2>(2)"
[<SourceSignalName>.attribute.Item(2).property3 = "<Value3>(2)"]]
 <SourceSignalName>.attribute.Add(3).property1 = "<Value1>(3)"
[<SourceSignalName>.attribute.Item(3).property2 = "<Value2>(3)"
[<SourceSignalName>.attribute.Item(3).property3 = "<Value3>(3)"]]
```

H.9.1.3 Examples (using pulses)

In the case of the PULSED_AC_TRAIN signal, the attribute pulse_train may include a series of pulses, each with up to three properties. The information would be presented in the following format:

```
pulse_train
      [{start_time  0 µs, pulse_width  8 µs, level_factor 1},
       {start_time 20 µs, pulse_width  5 µs, level_factor 1.1},
       {start_time 30 µs, pulse_width 10 µs, level_factor 1.2},
       {start_time 45 µs, pulse_width  4 µs, level_factor 0.95}]
```

This information maps to the following:

```
Sig.pulse_train.Add(1).start_time = "0us"
Sig.pulse_train.Item(1).pulse_width = "8us"
Sig.pulse_train.Item(1).level_factor = 1
Sig.pulse_train.Add(2).start_time = "20us"
Sig.pulse_train.Item(2).pulse_width = "5us"
Sig.pulse_train.Item(2).level_factor = 1.1
Sig.pulse_train.Add(3).start_time = "30us"
Sig.pulse_train.Item(3).pulse_width = "10us"
Sig.pulse_train.Item(3).level_factor = 1.2
Sig.pulse_train.Add(4).start_time = "45us"
Sig.pulse_train.Item(4).pulse_width = "4us"
Sig.pulse_train.Item(4).level_factor = 0.95
```

The <Value> associated with each property may include the optional <UnitSymbol>, <Errlmt>, and <Range> information. Also, the <NumericExpression> may be provided by a variable. This situation is illustrated in the following example:

```
pulse_train
      [{start_time Start1 µs errlmt ± Lmt % range 0 µs to 50 µs,
            pulse_width Pw1 µs errlmt ±1 µs, level_factor Factr1},
       {start_time Start2 µs errlmt ± Lmt % range 0 µs to 50 µs,
            pulse_width Pw2 µs errlmt ±1 µs, level_factor Factr2},
       {start_time Start3 µs errlmt ± Lmt % range 0 µs to 50 µs,
            pulse_width Pw3 µs errlmt ±1 µs, level_factor Factr3},
       {start_time Start4 µs errlmt ± Lmt % range 0 µs to 50 µs,
            pulse_width Pw4 µs errlmt ±1 µs, level_factor Factr4}]
```

where

Start1, Start2, Start3, and Start4 are real variables containing the value of the start_time;
Lmt is a real variable containing the error limit value for the start_time;
Pw1, Pw2, Pw3, and Pw4 are real variables containing the values of pulse_width;
Factr1, Factr2, Factr3, and Factr4 are real variables containing the values of the level_factor.

NOTE—Expressions may be used instead of variables, but must be enclosed in parentheses ().

H.9.2 Using arrays of data

Data may be provided in an array for convenience. The array will be defined in the carrier language. The elements of the array may then be used to provide numeric values for a property.

For example, it may be convenient to provide data for a series of pulses in an array. A shorthand method is required to extract the data from the array. This method requires the use of the keywords **for each** ... **in** together with an array index variable.

This method may be illustrated as follows:

```
pulse_train
      [for each x in puls
       {start_time puls[x].startTime µs
            errlmt ± Lmt % range 0 µs to 50 µs,
```

```
        pulse_width puls[x].pulseWidth µs errlmt ±1 µs,
        level_factor 1}]
```

where

puls is an array containing the start time and pulse width for a number of pulses;
Lmt is a real variable containing the error limit value for the start_time;
level_factor has a constant value (1).

This information maps to the following:

```
for each x in puls
      pulse_train.Add(x).start_time = puls[x].startTime
      pulse_train.Item(x).pulse_width = puls[x].pulseWidth
      pulse_train.Item(x).level_factor = 1
next
```

H.9.3 Acquiring sensor data

When a measured attribute has multiple properties or more than one set of properties, it is necessary to provide a suitable array variable in the read statement to store all the acquired data. The variable may need to be a one- or two-dimensional array depending on the number of attribute properties and the number of sets of attributes to be saved. The array may be storing values for multiple properties with different units. Hence, the <UnitSymbol> is not used with any read statement where the variable "into" is an array, and each result will then be saved in the base unit of the property.

The <Qualifier>, <Errlmt>, and <Range> of each property to be acquired may be defined with the measured attribute in the associated define sensor statement.

H.9.3.1 Measured attribute syntax

```
<TSFClass attribute>(<Property>[<Qualifier>][<Errlmt>][<Range>]
                    {,<Property>[<Qualifier>][<Errlmt>][<Range>]}*
```

H.9.3.2 Example of multiple sets of a single property

```
Setup PULSED_AC_TRAIN pulse_train (pulse_width errlmt ±2 µs)
as sensor MeasuredPulses;
.
.
Read (8) MeasuredPulses into PulseArray;
```

where

PulseArray is a one-dimensional (i.e., real) array that will be populated with a series of values equivalent to the pulse-width of each successive pulse in the measured signal.

NOTE—If the signal has more pulses than there are elements in the array, then the results for the remaining pulses will be discarded.

H.9.3.3 Example of a single set of multiple properties

```
Setup PULSED_AC_TRAIN pulse_train
```

```
            (start_time range 0 µs to 300 µs errlmt ±0.5 µs,
             pulse_width errlmt ±1 µs,
             level_factor)
as sensor PulseSensor;
.
.
Read PulseSensor into PulseData;
```

where

PulseData is a one-dimensional (i.e., real) array that will be populated with a series of values equivalent to the start_time, pulse-width, and the level_factor of the (first) pulse in the measured signal.

NOTE—If the signal has more than one pulse, then the results for the second and subsequent pulses will be discarded.

H.9.3.4 Example of multiple sets of multiple properties

```
Setup PULSED_AC_TRAIN pulse_train
            (start_time errlmt ±0.5 µs,
             pulse_width range 0 µs to 15 µs errlmt ±0.2 µs,
             level_factor)
as sensor PulseStream;
.
.
Read PulseStream into StreamArray;
```

where

StreamArray is a two-dimensional (i.e, real) array that will be populated with a series of values equivalent to the start_time, pulse-width, and level_factor of each successive pulse in the measured signal.

NOTES

1—If the signal has more pulses than there are rows in the array, then the results for the remaining pulses will be discarded.

2—If the signal has more properties than there are columns in the array, then the results for the remaining properties will be discarded.

H.10 Transferring data in digital signals

Digital signals include a data_value attribute that holds the data representing the digital pattern being passed. It may be required to pass more than one set of data, i.e., a series of patterns may be transmitted (or received) to (or from) the UUT. The size of the pattern of data depends on the width of the parallel word (for parallel digital signals) or the word_length (for serial digital signals).

H.10.1 Representation of digital data

Digital data are represented by the characters H, L, Z, and X where

— X represents an unknown state or undefined level;
— Z represents a high impedance state (no signal);
— L represents a logic 0;
— H represents a logic 1.

NOTE—The meaning of the X state varies according to whether the X is being transmitted, received, or used as a comparison. When being transmitted (or sourced), it indicates that the output is undefined and the test equipment may transmit an H, L, or Z state. When being received (or sensed), it indicates that the received signal is at an indeterminate level (i.e., it is between an L and an H state and not a Z, L, or H). When being used as a comparison, the X indicates that the value being compared is irrelevant (i.e., a "don't care").

A digital pattern may be of any number of digital characters. A single pattern is represented by a single literal character-string, a single character-string variable, or a single element of a character-string array. This pattern applies at the point at which the literal or variable is used in a TPL digital statement. Partial strings may be manipulated within the carrier language prior to being transmitted by a digital source or after being acquired by a digital sensor.

H.10.2 Transmitting digital data using digital sources

Digital source signals are defined using the TPL setup source statement in the same way that any analogue signal is defined. Many of the digital source attributes are analogue in nature and are, therefore, treated as analogue signal attributes. Only one attribute requires special consideration: the digital data attribute (data_value) that carries the digital information.

H.10.2.1 Attribute–Value group for digital data

<Value> for digital data takes a special form. In place of the <NumericExpression>, there is a <PatternExpression>, which represents digital data in a character string format. <UnitSymbol>, <Errlmt>, and <Range> are not required or included for digital data.

Hence, the Attribute-Value group for digital source data (i.e., for the data_value attribute) takes the following special form:

```
data_value <PatternExpression>
```

where

data_value is defined in the TSF entries for digital sources;
<PatternExpression> is a set of digital data provided as literal data or in a predefined carrier language variable or array variable.

NOTE—Attribute-Value groups for analogue (i.e., nondigital data) attributes in the digital TSF entries follow the normal syntax.

H.10.2.2 Literal representation of digital patterns

Literal data may be provided for a single digital pattern or a series of digital patterns. Formal descriptions are provided for both a single pattern and a series of patterns.

H.10.2.2.1 Attribute-Value group syntax for single literal pattern

```
data_value ["{X|Z|L|H}+"]
```

where
X represents an unknown state or undefined level;
Z represents a high impedance state;
L represents a logic 0;
H represents a logic 1.

H.10.2.2.2 Attribute-Value group syntax for series of literal patterns

```
data_value ["{X|Z|L|H}+"{,"{X|Z|L|H}+"}*]
```

where

X represents an unknown state or undefined level;

Z represents a high impedance state;

L represents a logic 0;

H represents a logic 1.

H.10.2.2.3 Example of literal representation of digital patterns

This example shows a literal data_value with four patterns of eight bits.

```
data_value
        ["HHLLHLHL",
         "HHLLHLHH",
         "HHLLHHLL",
         "HHLLZZZZ" ]
```

H.10.2.3 Representation of digital patterns in arrays

Digital testing usually requires the provision of multiple sets of patterns. These data are most easily provided via an array (of text) in which each element contains a single digital pattern. It is only necessary to provide the name of the array and the number of elements (within that array) of relevant data. If only a single pattern is to be provided, it may be provided in a string variable.

H.10.2.3.1 Attribute-Value group syntax for arrays and variables

```
data_value {<ArrayName>}|<VariableName>
```

where

<ArrayName> is a one-dimensional character-string array with an element for each pattern to be passed. Each of the *n* elements contains the digital pattern for a single data_value word.

<VariableName> is a user-defined character-string variable containing the digital pattern for a single data_value word.

H.10.2.3.2 Example of digital patterns presented in an array

This example shows a data_value with the digital data provided in an array DigiData. The number of bits is determined by other information provided with the setup statement.

```
data_value DigiData
```

where

DigiData is a one-dimensional character-string array that will be populated with a series of character strings representing the digital data to be transmitted. Each element in the array represents one digital pattern.

A pattern will be transmitted for each element in the array.

H.10.3 Acquiring digital sensor data

Digital sensor signals are defined using the TPL setup sensor statement in the same way that any analogue signal is defined. Many of the digital sensor attributes are analogue in nature and are, therefore, treated as analogue signal attributes. Only one attribute requires special consideration: the digital data attribute (data_value) that receives the digital information.

It is necessary to provide a suitable variable in the read statement to store all the acquired data. The variable may need to be a simple character-string variable or a one-dimensional array depending on the number of digital patterns to be saved. In the case of the digital data attribute (data_value), it would be inappropriate to include a <UnitSymbol> in the read statement.

H.10.3.1 Measured attribute for digital data

The measured attribute for digital data does not require <Qualifier>, <Errlmt>, or <Range>, and these elements shall not be included.

Hence, the measured attribute for digital data (i.e., the attribute data_value) stands alone in the setup sensor statement.

H.10.3.2 Example of acquisition of digital data

```
Setup DIGITAL_PARALLEL data_value
as sensor DigiSensor;
.
.
Read DigiSensor into DataArray;
```

where

DataArray is a one-dimensional character-string array that will be populated with the series of patterns received by the digital sensor DigiSensor.

NOTE—If the digital signal has more patterns than there are elements in the array, then the remaining patterns will be discarded.

H.10.4 Bidirectional digital signals

Bidirectional digital signals may be considered to be two separate actions, i.e., the transmission of digital data and the acquisition of digital sensor data on the same bus or set of connections.

H.10.4.1 Transmitting digital data on a bidirectional bus

In order to prevent any bus/data clashes, the programmer shall ensure that digital source data are sent at the correct time. This requirement may be achieved by appropriate timing control.

H.10.4.2 Sensing digital data on a bidirectional bus

Digital data may be sensed at any time. It is, therefore, both possible and valid to sense data being transmitted by the test equipment in addition to data being received from the UUT.

H.11 Creating test requirements

The user creates a test requirement by describing the test signals, measurements, and comparisons required to test a UUT using signal statements. The program flow is described in the carrier language of the user's choice. This flexibility enables the user to create a test requirement with sequential tests, tests in loops, result dependent tests, optional tests, diagnostic tests, etc.

Any variables used in the signal statements will be declared and/or defined according to the rules of the chosen carrier language. The variable names used in the signal statements will have to comply with the naming rules of that carrier language. It is good practice to select variable names that are meaningful and will also be suitable for use with any or most carrier languages. This approach will facilitate the conversion of a test requirement from one carrier language to another, should this be required in the future.

H.11.1 Creating test statements

A test statement is created from the formal description by substituting the appropriate TSF information and user-defined names. The keywords and symbols are copied unchanged. In the statement definitions, the keywords and symbols are in bold for the sake of clarity. It is not necessary to use bold in a TPL requirement.

H.11.1.1 Sample test statement from formal description

To illustrate how a signal statement is created from the formal description, the following example shows a setup source statement

```
'<TPL>
Setup DC_SIGNAL ampl DC-Power-Value V errlmt ±0.08 V range 0 V to 12 V
      gate with PowerSupplyEnable
      as source DC_POWER;
'</TPL>
```

This statement is derived from the formal description of setup source and is repeated here:

```
Setup <TSFClass> {<TSFClass attribute>[<Qualifier>]<Value>}
                 {,<TSFClass attribute>[<Qualifier>]<Value>}*
[sync to <EventSyncName>]
[gate {with <EventGateName>}|{from <EventFromName> to <EventToName>}]
[as [<identifier>]] source <SourceSignalName>;
```

The derivation of the setup source signal statement from the formal description is shown in Table H.4.

Table H.4—Relationship between formal description and typical signal statement

Formal description element	Signal statement	Comment
Setup	Setup	keyword
<TSFClass>	DC_SIGNAL	from TSF
[<Qualifier>]		optional — not used in this example
<TSFClass attribute>	Ampl	from TSF

Table H.4—Relationship between formal description and typical signal statement *(continued)*

Formal description element	Signal statement	Comment
<Value> (expands into:)		see separate definition
<NumericExpression>	DC-Power-Value	user-defined variable
[<UnitSymbol>]	V	from Table K.2
[<Errlmt>]	errlmt ±0.08 V	see separate definition
[<Range>]	range 0 V to 12 V	see separate definition
gate with	gate with	keywords
<EventGateName>	PowerSupplyEnable	user-defined object (in separate TPL statement)
[as [<identifier>]] source	as source	keywords
<SourceSignalName>	DC_POWER	user-defined object name
;	;	key symbol

The user-defined variable DC-Power-Value shall be a valid carrier language variable and will have been defined in the program before being referenced in the signal statement. The user=supplied object PowerSupplyEnable will have been previously defined in an appropriate setup statement.

H.11.1.2 Language mapping

The language mapping section shows the mapping to the carrier language of that particular TPL statement. It is assumed that a preprocessor will be used to convert all the signal statements into the correct, equivalent carrier language statements.

Mapping the sample signal statement in H.11.1.1 to the carrier language gives the following:

```
Declare DC_POWER as DC_SIGNAL
Assign DC_POWER = STD.Require("DC_SIGNAL")
       DC_POWER.ampl = "trms" & DC-Power-Value & "V errlmt ±0.08 V" &
                       "range 0 V to 12 V "
Assign DC_POWER.Gate = PowerSupplyEnable
```

H.11.2 Use of gate in signal statements

In order to give the maximum flexibility to test requirement developers, Gate fields are available in many of the signal statement types. In the case of a sensor, for example, it is possible to link a measurement to a gate event in the setup statement, the connect statement, and the read statement. A gate event linked to any of the statements will gate the signal (or, in the example, the read). It is not necessary to reference the same gate event in each of the statements, although doing so will not necessarily cause an incorrect result.

It is possible to refer to different gate events in more than one of the statements, and this reference may well be justified in the context of the test requirement. Care should be exercised when making such a reference, as a signal (or, in the sensor example, a successful read) will occur only if all the referenced events overlap. There must be a period when all the gate events are active at the same time for the signal to occur.

H.12 Delimiting TPL statements

In most cases, TPL statements will be unique within the carrier language. Where the choice of carrier language results in TPL statements not being unique, the TPL statements can be delimited from the carrier language using the optional delimiter statements described in this clause.

In order to clearly identify TPL statements when they are embedded within a carrier language test requirement, it is necessary to introduce a block of one or more TPL statements with an introductory character group. This step also minimizes the parsing that has to be done to identify a TPL statement should the TPL have elements identical with carrier language keywords.

The TPL statements may or may not be comments within the carrier language; therefore, the identifier used must be suitable for incorporation with different methods of delimiting comments.

H.12.1 Introducing a group of one or more TPL statements

The standard carrier language comment identifier <comment symbol>, followed by the characters "<TPL>", is used to introduce TPL statements.

`<comment symbol>`**`<TPL>`**

For example, for typical carrier languages the TPL introductory character group would be as follows:

— '<TPL> for Visual Basic
— //<TPL> for C/C++ and Java
— /*<TPL> for C/C++ and Java

Provided that at least one space follows the introductory character group, additional commentary may be included in the same comment. If additional commentary is added, then the comment shall be terminated prior to the start of the first TPL statement.

H.12.2 Indicating end of group of TPL statements

The two methods of indicating the end of a group of TPL statements depend upon whether the TPL statements appear as comments within the carrier language. Both these methods use the characters "<TPL/>".

H.12.2.1 TPL statements appearing within comment

Where the TPL statements appear within a multi-line comment, the characters "<TPL/>" shall appear after the TPL statements and before the standard carrier language end-of-comment identifier.

For example, the characters <TPL/> occurring before the symbol */ would indicate the end of the TPL statement group in C.

H.12.2.2 TPL statements not appearing as comments

Where the TPL statements do not appear as comments, the standard carrier language comment identifier <comment symbol>, followed by the characters "<TPL/>", is used to indicate the end of the TPL group.

`<comment symbol>`**`<TPL/>`**

For example, for typical carrier languages, the character group indicating the end of a TPL statement group would be as follows:

— '<TPL/> for Visual Basic
— //<TPL/> for C/C++ and Java
— /*<TPL/> for C/C++ and Java, where the TPL does not appear as a comment.

Annex I

(normative)

Extensible markup language (XML) signal descriptions (XSDs)

I.1 Introduction

Signals can be defined through an extensible markup language (XML) document adhering to the XML signal definitions (XSDs). This allows static signals to be defined in XML by writing XML signal definition adhering to the XML signal schema definition.

An example of such a XML signal describing a suppressed carrier signal is as follows:

```
<?xml version="1.0"?>
<!--encoding="UTF-8"-->
<Signal name="suppressedCarrier" Out="Diff3" xmlns="STDBSC" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance">
   <Sinusoid name="Sinusoidal9" amplitude="5V" frequency="10kHz"/>
   <Sinusoid name="Sinusoidal10" amplitude="1V" frequency="1kHz"/>
   <AM name="AM5" Carrier="Sinusoidal9" In="Sinusoidal10"/>
   <Diff name="Diff3" In="AM5 Sinusoidal9"/>
</Signal>
```

This annex describes the XML schema (W3) to which any XSD shall adhere, where each element name is any of the BSC names or TSF elements adhering to this standard.

I.2 XML signal schema definition

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="STDBSC" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="STDBSC" elementFormDefault="qualified">
   <xs:simpleType name="qualifier">
      <xs:restriction base="xs:string">
         <xs:enumeration value="trms"/>
         <xs:enumeration value="pk_pk"/>
         <xs:enumeration value="pk"/>
         <xs:enumeration value="pk_pos"/>
         <xs:enumeration value="pk_neg"/>
         <xs:enumeration value="inst"/>
         <xs:enumeration value="av"/>
         <xs:enumeration value="inst_max"/>
         <xs:enumeration value="inst_min"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="quantity">
      <xs:restriction base="xs:string">
         <xs:pattern value="(\+|\-)?\d+(\.\d*)?((E|e)(\+|\-)?\d+)?
*((y|z|a|f|p|n|µ|u|m|c|d|h|k|M|G|T|P|E|Z|Y)?(F|S|C|A|V|J|eV|T|N|Hz|lx|H|m|in|ft|mi|nmi|lm|
cd|Wb|g|rad|deg|°|W|BW|Bm|Pa|bar|B(\(\d
*m?W\))?|%|pc|Ohm|sr|K|degC|°C|degF|°F|s|min|h|L|mol)\d*((·|\.|/
)(y|z|a|f|p|n|µ|u|m|c|d|h|k|M|G|T|P|E|Z|Y)?(F|S|C|A|V|J|eV|T|N|Hz|lx|H|m|in|ft|mi|nmi|lm|c
d|Wb|g|rad|deg|°|W|BW|Bm|Pa|bar|B|%|pc|Ohm|sr|K|degC|°C|degF|°F|s|min|h|L|mol)\d*)*)?"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="errlmt">
      <xs:restriction base="xs:string">
         <xs:pattern value="errlmt|±|\+-"/>
      </xs:restriction>
```

```
   </xs:simpleType>
   <xs:simpleType name="range">
      <xs:restriction base="xs:string">
         <xs:pattern value="range|MAX|MIN"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="to">
      <xs:restriction base="xs:string">
         <xs:pattern value=":|\-|to"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="_physical">
      <xs:list>
         <xs:simpleType>
            <xs:union memberTypes="qualifier errlmt range to quantity"/>
         </xs:simpleType>
      </xs:list>
   </xs:simpleType>
   <xs:simpleType name="Physical">
      <xs:union>
         <xs:simpleType>
            <xs:restriction base="xs:string">
               <xs:pattern value="\(.+\)"/>
            </xs:restriction>
         </xs:simpleType>
         <xs:simpleType>
            <xs:restriction base="xs:string">
               <xs:pattern value="(
*(trms|pk_pk|pk|pk_pos|pk_neg|inst|av|inst_max|inst_min|errlmt (±|\+-)?|±|\+-|range
(MAX|MIN)?|MAX|MIN|to|:|\-)? *((\+|\-)?\d+(\.\d*)?((E|e)(\+|\-)?\d+)?
*((y|z|a|f|p|n|µ|u|m|c|d|h|k|M|G|T|P|E|Z|Y)?(F|S|C|A|V|J|eV|T|N|Hz|lx|H|m|in|ft|mi|nmi|lm|
cd|Wb|g|rad|deg|°|W|BW|Bm|Pa|bar|B(\(\d
*m?W\))?|%|pc|Ohm|sr|K|degC|°C|degF|°F|s|min|h|L|mol)\d*((·|\.|/
)\(*(y|z|a|f|p|n|µ|u|m|c|d|h|k|M|G|T|P|E|Z|Y)?(F|S|C|A|V|J|eV|T|N|Hz|lx|H|m|in|ft|mi|nmi|l
m|cd|Wb|g|rad|deg|°|W|BW|Bm|Pa|bar|B(\(\d
*m?W\))?|%|pc|Ohm|sr|K|degC|°C|degF|°F|s|min|h|L|mol)\d*)*)?\)*) *)+"/>
            </xs:restriction>
         </xs:simpleType>
      </xs:union>
   </xs:simpleType>
   <xs:simpleType name="Admittance">
      <xs:restriction base="Physical"/>
   </xs:simpleType>
   <xs:simpleType name="AmountOfInformation">
      <xs:restriction base="Physical"/>
   </xs:simpleType>
   <xs:simpleType name="AmountOfSubstance">
      <xs:restriction base="Physical"/>
   </xs:simpleType>
   <xs:simpleType name="Capacitance">
      <xs:restriction base="Physical"/>
   </xs:simpleType>
   <xs:simpleType name="Conductance">
      <xs:restriction base="Physical"/>
   </xs:simpleType>
   <xs:simpleType name="Charge">
      <xs:restriction base="Physical"/>
   </xs:simpleType>
   <xs:simpleType name="Current">
      <xs:restriction base="Physical"/>
   </xs:simpleType>
   <xs:simpleType name="Voltage">
      <xs:restriction base="Physical"/>
   </xs:simpleType>
   <xs:simpleType name="Resistance">
      <xs:restriction base="Physical"/>
   </xs:simpleType>
   <xs:simpleType name="Energy">
      <xs:restriction base="Physical"/>
   </xs:simpleType>
   <xs:simpleType name="Force">
      <xs:restriction base="Physical"/>
```

```
</xs:simpleType>
<xs:simpleType name="Frequency">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Heat">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Illuminance">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Impedance">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Inductance">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Distance">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="LuminousFlux">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="LuminousIntensity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="MagneticFlux">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="MagneticFluxDensity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Mass">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="PlaneAngle">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Power">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Pressure">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Ratio">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Reactance">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="SolidAngle">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Susceptance">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Temperature">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Time">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Acceleration">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="AngularAcceleration">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="AngularSpeed">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Area">
   <xs:restriction base="Physical"/>
```

```
</xs:simpleType>
<xs:simpleType name="Concentration">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="CurrentDensity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="DynamicViscosity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="ElectricChargeDensity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="ElectricFieldStrength">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="ElectricFluxDensity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="EnergyDensity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Enthropy">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Exposure">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="HeatCapacity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="HeatFluxDensity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Irradiance">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="KinematicViscosity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Luminance">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="MagneticFieldStrength">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="MassDensity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="MassFlow">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="MolarEnergy">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="MolarEntropy">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="MolarHeatCapacity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="MomentOfForce">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="MomentOfInertia">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Momentum">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Permeability">
   <xs:restriction base="Physical"/>
```

```
</xs:simpleType>
<xs:simpleType name="Permittivity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="PowerDensity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Radiance">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="RadiantIntensity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="SpecificHeatCapacity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="SpecificEnergy">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="SpecificEntropy">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="SpecificVolume">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="SurfaceTension">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="ThermalConductivity">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Speed">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="Volume">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="VolumeFlow">
   <xs:restriction base="Physical"/>
</xs:simpleType>
<xs:simpleType name="PulseDefn">
   <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="PulseDefns">
   <xs:list itemType="PulseDefn"/>
</xs:simpleType>
<xs:simpleType name="enumCondition">
   <xs:restriction base="xs:string">
      <xs:enumeration value="NONE"/>
      <xs:enumeration value="GT"/>
      <xs:enumeration value="GE"/>
      <xs:enumeration value="LE"/>
      <xs:enumeration value="LT"/>
      <xs:enumeration value="EQ"/>
      <xs:enumeration value="NE"/>
   </xs:restriction>
</xs:simpleType>
<xs:simpleType name="enumMeasuredVariable">
   <xs:restriction base="xs:string">
      <xs:enumeration value="DEPENDENT"/>
      <xs:enumeration value="INDEPENDENT"/>
   </xs:restriction>
</xs:simpleType>
<xs:simpleType name="SAFEARRAY_BSTR">
   <xs:list itemType="xs:string"/>
</xs:simpleType>
<xs:simpleType name="SAFEARRAY_Physical">
   <xs:list itemType="Physical"/>
</xs:simpleType>
<xs:simpleType name="SAFEARRAY_double">
   <xs:list itemType="xs:double"/>
</xs:simpleType>
```

```
   <xs:simpleType name="SAFEARRAY_VARIANT">
      <xs:list itemType="xs:string"/>
   </xs:simpleType>
   <xs:complexType name="SignalFunctionType">
      <xs:attribute name="type" type="xs:string" use="optional"/>
      <xs:attribute name="reftype" type="xs:string" use="optional"/>
      <xs:attribute name="name" type="xs:ID" use="required"/>
      <xs:attribute name="In" type="xs:IDREFS" use="optional"/>
      <xs:attribute name="Gate" type="xs:IDREF" use="optional"/>
      <xs:attribute name="Sync" type="xs:IDREF" use="optional"/>
      <xs:attribute name="Conn" type="xs:IDREFS" use="optional"/>
   </xs:complexType>
   <xs:element name="Signal">
      <xs:complexType>
         <xs:choice maxOccurs="unbounded">
            <xs:element name="Constant">
               <xs:annotation>
                  <xs:documentation>A constant signal retains its given level.</
xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="amplitude" type="Physical" default="0">
                           <xs:annotation>
                              <xs:documentation>the level of the signal.</
xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="Step">
               <xs:annotation>
                  <xs:documentation>A Step signal makes a transition from zero to a given
level.</xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="amplitude" type="Physical" default="0">
                           <xs:annotation>
                              <xs:documentation>final value of step signal</
xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="startTime" type="Time" default="0.5s">
                           <xs:annotation>
                              <xs:documentation>defines when the step transition starts</
xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="SingleTrapezoid">
               <xs:annotation>
                  <xs:documentation>A Single Trapezoid is a non-periodic signal defined by
the geometric trapezoid shape.</xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="amplitude" type="Physical" default="0">
                           <xs:annotation>
                              <xs:documentation>value of pulse amplitude</
xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="startTime" type="Time" default="0s">
```

```
                        <xs:annotation>
                           <xs:documentation>time at which trapezoid starts relative to
it initialization</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="riseTime" type="Time" default="0.25s">
                        <xs:annotation>
                           <xs:documentation>time taken to reach amplitude</
xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="pulseWidth" type="Time" default="0.5s">
                        <xs:annotation>
                           <xs:documentation>time that trapezoid is stable at
amplitude</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="fallTime" type="Time" default="0.25s">
                        <xs:annotation>
                           <xs:documentation>Time taken to fall back to transient
state</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
         <xs:element name="Noise">
            <xs:annotation>
               <xs:documentation>Noise may be considered as unwanted disturbances
superimposed upon a useful signal, which tend to obscure its information content. Noise may
be genuinely random (as in white noise) or may be pseudo-random. Noise occurs across a
range of frequencies and can be characterized by amplitude; it may take the form of a sense
or source signal. Pseudo-random 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</
xs:documentation>
            </xs:annotation>
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType">
                     <xs:attribute name="amplitude" type="Physical" default="0"/>
                     <xs:attribute name="seed" type="xs:long" default="0">
                        <xs:annotation>
                           <xs:documentation>used for pseudo-random noise</
xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="frequency" type="Frequency" default="50 Hz">
                        <xs:annotation>
                           <xs:documentation>upper bound frequency bandwidth for
transient disturbances</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
         <xs:element name="SingleRamp">
            <xs:annotation>
               <xs:documentation>A Single Ramp represents a linear transition from zero
to the defined amplitude level during a defined time period.</xs:documentation>
            </xs:annotation>
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType">
                     <xs:attribute name="amplitude" type="Physical" default="0">
                        <xs:annotation>
                           <xs:documentation>final value of ramp signal</
xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="riseTime" type="Time" default="1s">
```

```
                        <xs:annotation>
                           <xs:documentation>time for signal to reach final value</
xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="startTime" type="Time" default="0s">
                        <xs:annotation>
                           <xs:documentation>defines when the step transition starts</
xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
         <xs:element name="Sinusoid">
            <xs:annotation>
               <xs:documentation>A Sinusoid is a signal whereby the amplitude of the
dependent variable is given by the formula:</xs:documentation>
            </xs:annotation>
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType">
                     <xs:attribute name="amplitude" type="Physical" default="0">
                        <xs:annotation>
                           <xs:documentation>Amplitude</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="frequency" type="Frequency" default="1Hz">
                        <xs:annotation>
                           <xs:documentation>Frequency</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="phase" type="PlaneAngle" default="0 rad">
                        <xs:annotation>
                           <xs:documentation>Initial phase angle</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
         <xs:element name="Trapezoid">
            <xs:annotation>
               <xs:documentation>A Trapezoidal Signal is a periodic signal that
sequentially repeats the Single Trapezoid. The period is defined by the duration of the
Single Trapezoid.  All event times are referenced to local time, which is reset at the
start of each pulse.</xs:documentation>
            </xs:annotation>
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType">
                     <xs:attribute name="amplitude" type="Physical" default="0">
                        <xs:annotation>
                           <xs:documentation>value of pulse amplitude</
xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="period" type="Time" default="1 s">
                        <xs:annotation>
                           <xs:documentation>time in which the signal repeats itself</
xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="riseTime" type="Time" default="0.25 s">
                        <xs:annotation>
                           <xs:documentation>time taken to reach amplitude</
xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="pulseWidth" type="Time" default="0.5 s">
                        <xs:annotation>
```

```
                        <xs:documentation>time that trapezoid is stable at
amplitude</xs:documentation>
                     </xs:annotation>
                  </xs:attribute>
                  <xs:attribute name="fallTime" type="Time" default="0.25 s">
                     <xs:annotation>
                        <xs:documentation>Time taken to fall back to transiant
state</xs:documentation>
                     </xs:annotation>
                  </xs:attribute>
               </xs:extension>
            </xs:complexContent>
         </xs:complexType>
      </xs:element>
      <xs:element name="Ramp">
         <xs:annotation>
            <xs:documentation>A Ramp Signal is a periodic signal whose instantaneous
value varies alternately and linearly between two specified values.  Its parameters are
defined by its amplitude, the time to rise from zero and the period.</xs:documentation>
         </xs:annotation>
         <xs:complexType>
            <xs:complexContent>
               <xs:extension base="SignalFunctionType">
                  <xs:attribute name="amplitude" type="Physical" default="0">
                     <xs:annotation>
                        <xs:documentation>final level of the signal</
xs:documentation>
                     </xs:annotation>
                  </xs:attribute>
                  <xs:attribute name="period" type="Time" default="1s">
                     <xs:annotation>
                        <xs:documentation>time is which signal repeats itself</
xs:documentation>
                     </xs:annotation>
                  </xs:attribute>
                  <xs:attribute name="riseTime" type="Time" default="1s">
                     <xs:annotation>
                        <xs:documentation>Rise Time of ramp signal</
xs:documentation>
                     </xs:annotation>
                  </xs:attribute>
               </xs:extension>
            </xs:complexContent>
         </xs:complexType>
      </xs:element>
      <xs:element name="Triangle">
         <xs:annotation>
            <xs:documentation>A Triangular Signal is a periodic signal whose
instantaneous value varies linearly and equally about zero. 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 dutyCycle.</
xs:documentation>
         </xs:annotation>
         <xs:complexType>
            <xs:complexContent>
               <xs:extension base="SignalFunctionType">
                  <xs:attribute name="amplitude" type="Physical" default="0">
                     <xs:annotation>
                        <xs:documentation>maximum amplitude level of the signal</
xs:documentation>
                     </xs:annotation>
                  </xs:attribute>
                  <xs:attribute name="period" type="Time" default="1s">
                     <xs:annotation>
                        <xs:documentation>time in which signal repeats itself</
xs:documentation>
                     </xs:annotation>
                  </xs:attribute>
                  <xs:attribute name="dutyCycle" type="Ratio" default="50%">
                     <xs:annotation>
                        <xs:documentation>ratio between time taken to increase from
its minimum to its maximum value and the time for one period</xs:documentation>
```

```
                        </xs:annotation>
                     </xs:attribute>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
         <xs:element name="SquareWave">
            <xs:annotation>
               <xs:documentation>A Square Wave is a periodic signal whose amplitude
(dependent variable) alternately assumes one of two fixed values of amplitude, the
amplitudes are equal about zero which is the reference base line.  Its parameters are
defined by its amplitude and period.</xs:documentation>
            </xs:annotation>
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType">
                     <xs:attribute name="amplitude" type="Physical" default="0">
                        <xs:annotation>
                           <xs:documentation>Amplitude of signal</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="period" type="Time" default="1s">
                        <xs:annotation>
                           <xs:documentation>period of signal</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="dutyCycle" type="Ratio" default="50%">
                        <xs:annotation>
                           <xs:documentation>duty cycle of the wave</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
         <xs:element name="WaveformRamp">
            <xs:annotation>
               <xs:documentation>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 Ramp
Signal.</xs:documentation>
            </xs:annotation>
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType">
                     <xs:attribute name="amplitude" type="Physical" default="1">
                        <xs:annotation>
                           <xs:documentation>amplitude of the output signal where the
level factor (in points) </xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="period" type="Time" default="1s">
                        <xs:annotation>
                           <xs:documentation>the time between each sequence</
xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="samplingInterval" type="Time" default="0">
                        <xs:annotation>
                           <xs:documentation>the time between successive (points)
outputs</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="points" type="SAFEARRAY_double">
                        <xs:annotation>
                           <xs:documentation>level factor of each waveform sample</
xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
```

```
            </xs:element>
            <xs:element name="WaveformStep">
               <xs:annotation>
                  <xs:documentation>WaveformStep takes the form of a sequence of constant
signals, the level of the constant signal is defined by the points, a transition in level
occurring at each increment of the sampling interval. WaveformStep cycles through the
points sequentially and continuously.

If the samplingInterval is zero, the complete waveform described by the points is repeated
per period, otherwise the period is calculated as sampleInterval * number of points.
Assigning a non-zero period value sets the samplingInterval to zero.  </xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="amplitude" type="Physical" default="1">
                           <xs:annotation>
                              <xs:documentation>amplitude of the output signal where the
level factor (in points) </xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="period" type="Time" default="1s">
                           <xs:annotation>
                              <xs:documentation>the time between each sequence</
xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="samplingInterval" type="Time" default="0">
                           <xs:annotation>
                              <xs:documentation>the time between successive (points)
outputs</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="points" type="SAFEARRAY_double">
                           <xs:annotation>
                              <xs:documentation>level factor of each waveform sample</
xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="BandPass">
               <xs:annotation>
                  <xs:documentation>A BandPass filter passes a set of frequencies from an
input signal (with equal gain across all its frequencies) and filters out all frequencies
outside of the band. </xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="centerFrequency" type="Frequency" default="0">
                           <xs:annotation>
                              <xs:documentation>Center frequency of the filter's band</
xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="frequencyBand" type="Frequency" default="0">
                           <xs:annotation>
                              <xs:documentation>Bandwidth of filter. Zero implies narrowest
band </xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="LowPass">
               <xs:annotation>
```

```
                <xs:documentation>LowPass Filter suppresses all frequencies above the
Bandpass frequency. The LowPass Filter represents a perfect step filter.</
xs:documentation>
             </xs:annotation>
             <xs:complexType>
                <xs:complexContent>
                   <xs:extension base="SignalFunctionType">
                      <xs:attribute name="cutoff" type="Frequency" default="0Hz">
                         <xs:annotation>
                            <xs:documentation>Cut off frequency filter. Zero implies DC
offset</xs:documentation>
                         </xs:annotation>
                      </xs:attribute>
                   </xs:extension>
                </xs:complexContent>
             </xs:complexType>
          </xs:element>
          <xs:element name="HighPass">
             <xs:annotation>
                <xs:documentation>HighPass Filter suppresses all frequencies below the
Bandpass frequency. The HighPass Filter represents a perfect step filter.</
xs:documentation>
             </xs:annotation>
             <xs:complexType>
                <xs:complexContent>
                   <xs:extension base="SignalFunctionType">
                      <xs:attribute name="cutoff" type="Frequency" default="0Hz">
                         <xs:annotation>
                            <xs:documentation>Start frequency of filter. Zero implies AC
Coupled</xs:documentation>
                         </xs:annotation>
                      </xs:attribute>
                   </xs:extension>
                </xs:complexContent>
             </xs:complexType>
          </xs:element>
          <xs:element name="Notch">
             <xs:annotation>
                <xs:documentation>Notch filter suppresses with equal gain across its
frequencies, opposite to BandPass filter</xs:documentation>
             </xs:annotation>
             <xs:complexType>
                <xs:complexContent>
                   <xs:extension base="SignalFunctionType">
                      <xs:attribute name="centerFrequency" type="Frequency"
default="0Hz">
                         <xs:annotation>
                            <xs:documentation>Center frrequency of the filter's notch</
xs:documentation>
                         </xs:annotation>
                      </xs:attribute>
                      <xs:attribute name="frequencyBand" type="Frequency" default="0Hz">
                         <xs:annotation>
                            <xs:documentation>Frequency band of filter. Zero implies
minimum band</xs:documentation>
                         </xs:annotation>
                      </xs:attribute>
                   </xs:extension>
                </xs:complexContent>
             </xs:complexType>
          </xs:element>
          <xs:element name="Sum">
             <xs:annotation>
                <xs:documentation>Sum makes signals from other Signals by summing them
together.</xs:documentation>
             </xs:annotation>
             <xs:complexType>
                <xs:complexContent>
                   <xs:extension base="SignalFunctionType"/>
                </xs:complexContent>
             </xs:complexType>
          </xs:element>
```

```
            <xs:element name="Product">
               <xs:annotation>
                  <xs:documentation>Product makes signals from other signals by multiplying
them together.</xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType"/>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="Diff">
               <xs:annotation>
                  <xs:documentation>Diff makes signals from other signals by subtracting
them from the first signal.</xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType"/>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="FM">
               <xs:annotation>
                  <xs:documentation>The FM is a modulator where the instantaneous frequency
of the sinusoidal carrier varies with the amplitude of the modulating input signal.</
xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="amplitude" type="Physical" default="1">
                           <xs:annotation>
                              <xs:documentation>Amplitude of  Sinusoidal Carrier wave</
xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="carrierFrequency" type="Frequency"
default="1kHz">
                           <xs:annotation>
                              <xs:documentation>Frequency of Sinusoidal Carrier Wave</
xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="frequencyDeviation" type="Frequency"
default="100Hz">
                           <xs:annotation>
                              <xs:documentation>Frequency deviation</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="AM">
               <xs:annotation>
                  <xs:documentation>Amplitude modulation where the Amplitude of the
sinusoidal carrier varies with as a ratio to the  amplitude of the modulated input
signal.</xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="modIndex" type="Ratio" default="0.3">
                           <xs:annotation>
                              <xs:documentation>Modulation Index</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="Carrier" type="xs:IDREFS"/>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
```

```
            </xs:element>
            <xs:element name="PM">
               <xs:annotation>
                  <xs:documentation>Phase modulation where the phase of the sinusoidal
carrier varies with the amplitude of the modulated input signal.</xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="amplitude" type="Physical" default="1">
                           <xs:annotation>
                              <xs:documentation>Amplitude of  Sinusoidal Carrier wave</
xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="carrierFrequency" type="Frequency"
default="1kHz">
                           <xs:annotation>
                              <xs:documentation>Frequency of Sinusoidal Carrier Wave</
xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="phaseDeviation" type="PlaneAngle"
default="(pi/4)">
                           <xs:annotation>
                              <xs:documentation>Phase deviation</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="SignalDelay">
               <xs:annotation>
                  <xs:documentation>The InSignal is delayed to become the OutSignal, where
the delay is defined by an initial fixed delay and where the delay may change over time.</
xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="acceleration" type="Frequency" default="0">
                           <xs:annotation>
                              <xs:documentation>the rate at which the Rate will alter over
time (Acceleration)</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="delay" type="Time" default="0">
                           <xs:annotation>
                              <xs:documentation>Fixed delay that signal will be delayed
(Distance)</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="rate" type="Ratio" default="0">
                           <xs:annotation>
                              <xs:documentation>the rate at which the Delay will alter over
time (Speed)</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="Exponential">
               <xs:annotation>
                  <xs:documentation>A transformation that multiplies the input signal with
a coefficient that decays exponentially over time..</xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="dampingFactor" type="xs:double" default="1.0">
```

```
                        <xs:annotation>
                           <xs:documentation>value of damping factor</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
         <xs:element name="E">
            <xs:annotation>
               <xs:documentation>Similarly, we have an exponential operation on signals.
This signal takes expy, where y is the result of the signal used as input. </
xs:documentation>
            </xs:annotation>
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType"/>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
         <xs:element name="Negate">
            <xs:annotation>
               <xs:documentation>Modifies a signal such that its amplitude is the
negative of the In signal amplidude</xs:documentation>
            </xs:annotation>
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType"/>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
         <xs:element name="Inverse">
            <xs:annotation>
               <xs:documentation>Inverse is the mathematical reciprocal of a Signal.</
xs:documentation>
            </xs:annotation>
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType"/>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
         <xs:element name="PulseTrain">
            <xs:annotation>
               <xs:documentation>Creates a Train of Pulses of the In signal by
multiplying the input signal with the amplitude of the pulses.</xs:documentation>
            </xs:annotation>
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType">
                     <xs:attribute name="pulses" type="PulseDefns">
                        <xs:annotation>
                           <xs:documentation>list of pulses</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="repetition" type="xs:long" default="0"/>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
         <xs:element name="Attenuator">
            <xs:annotation>
               <xs:documentation>Scales the amplitude (dependent variable) of the In
signal. Allows both the increase and decrease of the signal.</xs:documentation>
            </xs:annotation>
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType">
                     <xs:attribute name="gain" type="Ratio" default="1.0"/>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
```

```
            </xs:element>
            <xs:element name="Load">
               <xs:annotation>
                  <xs:documentation>Provides an impedance to load a signal.</
xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="resistance" type="Resistance" default="0 Ohm"/
>
                        <xs:attribute name="reactance" type="Resistance" default="0 Ohm"/>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="Limit">
               <xs:annotation>
                  <xs:documentation>Limit restricts the values of the signal to +/- the limit
value.</xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="limit" type="Physical" default="1">
                           <xs:annotation>
                              <xs:documentation>limits the absolute value of the signal to
+/- limit value</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="FFT">
               <xs:annotation>
                  <xs:documentation>The Fourier Transform converts time domain signals to
frequency domain signals. 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 two),
the time over which the signal will be sampled, and the signal to be converted.</
xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="samples" type="xs:long" default="1023"/>
                        <xs:attribute name="interval" type="Time" default="1s"/>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="Clock">
               <xs:annotation>
                  <xs:documentation>Clock generates an event at regular intervals. Each
event is active for the first half of the Clock period.</xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="clockRate" type="Frequency" default="1Hz">
                           <xs:annotation>
                              <xs:documentation>Frequency of the clock.</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="TimedEvent">
               <xs:annotation>
                  <xs:documentation>TimedEvent generates an OutEvent at regular intervals.
Each event is active for a specific duration, if the duration is longer that than the event
```

```
interval (Every) the OutEvent is signaled Active at each interval but never becomes
Paused.</xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="delay" type="Time" default="0s">
                           <xs:annotation>
                              <xs:documentation>the delay time before the first event will
be start</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="duration" type="Time" default="1s">
                           <xs:annotation>
                              <xs:documentation>the duration that each event is active</
xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="period" type="Time" default="1s">
                           <xs:annotation>
                              <xs:documentation>the time interval for each event</
xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="repetition" type="xs:long" default="0"/>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="PulsedEvent">
               <xs:annotation>
                  <xs:documentation>PulsedEvent generates an Out event in the form of a
sequence of timing pulses primarily intended for use in generating source signals. The
sequence comprises a train of N pulses (where N may be any integer greater than zero).
Where N is greater than one, 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.</xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="pulses" type="PulseDefns">
                           <xs:annotation>
                              <xs:documentation>list of pulses</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="repetition" type="xs:long" default="0"/>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="EventedEvent">
               <xs:annotation>
                  <xs:documentation>An EventedEvent allows events to be combined to produce
complex event streams.</xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType"/>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="EventCount">
               <xs:annotation>
                  <xs:documentation>An EventCount filters out input events only allowing
every 'count' input event.</xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="count" type="xs:long" default="0">
                           <xs:annotation>
```

```
                        <xs:documentation>Identifies the number of events that must
happen before a event is generated</xs:documentation>
                     </xs:annotation>
                  </xs:attribute>
               </xs:extension>
            </xs:complexContent>
         </xs:complexType>
      </xs:element>
      <xs:element name="ProbabilityEvent">
         <xs:annotation>
            <xs:documentation>A ProbabilityEvent generates an event stream based upon
the event stream present at the In. It will replicate the same timing information but will
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.</xs:documentation>
         </xs:annotation>
         <xs:complexType>
            <xs:complexContent>
               <xs:extension base="SignalFunctionType">
                  <xs:attribute name="seed" type="xs:long" default="0"/>
                  <xs:attribute name="probability" type="Ratio" default="50"/>
               </xs:extension>
            </xs:complexContent>
         </xs:complexType>
      </xs:element>
      <xs:element name="NotEvent">
         <xs:annotation>
            <xs:documentation>The NotEvent is Active when the In Signal is not
Active</xs:documentation>
         </xs:annotation>
         <xs:complexType>
            <xs:complexContent>
               <xs:extension base="SignalFunctionType"/>
            </xs:complexContent>
         </xs:complexType>
      </xs:element>
      <xs:element name="OrEvent">
         <xs:annotation>
            <xs:documentation>The OrEvent is Active when any in events is Active</
xs:documentation>
         </xs:annotation>
         <xs:complexType>
            <xs:complexContent>
               <xs:extension base="SignalFunctionType"/>
            </xs:complexContent>
         </xs:complexType>
      </xs:element>
      <xs:element name="XOrEvent">
         <xs:annotation>
            <xs:documentation>The XOrEvent is Active when an odd number of in events
is Active</xs:documentation>
         </xs:annotation>
         <xs:complexType>
            <xs:complexContent>
               <xs:extension base="SignalFunctionType"/>
            </xs:complexContent>
         </xs:complexType>
      </xs:element>
      <xs:element name="AndEvent">
         <xs:annotation>
            <xs:documentation>The AndEvent is Active when all in events is Active</
xs:documentation>
         </xs:annotation>
         <xs:complexType>
            <xs:complexContent>
               <xs:extension base="SignalFunctionType"/>
            </xs:complexContent>
         </xs:complexType>
      </xs:element>
      <xs:element name="Counter">
```

```
            <xs:annotation>
               <xs:documentation>For all Sensors every time a measurement is taken, the
'count' property is incremented. A Counter is a sensor that counts when a measurement would
be taken, but does not take any specific measurement.</xs:documentation>
            </xs:annotation>
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType">
                     <xs:attribute name="measuredVariable" type="enumMeasuredVariable"
default="DEPENDENT">
                        <xs:annotation>
                           <xs:documentation>Whether the measurement made is of the
dependent or independent variable. </xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="measurement" type="xs:string" default="0">
                        <xs:annotation>
                           <xs:documentation>Current value measured</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="measurements" type="SAFEARRAY_VARIANT">
                        <xs:annotation>
                           <xs:documentation>Array of measurements made</
xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="samples" type="xs:long" default="0">
                        <xs:annotation>
                           <xs:documentation>Number of consecutive measurement to be
made. Zero indicates no measurement to be taken and indicates the Sensor is acting as a
monitor only</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="count" type="xs:long" default="0">
                        <xs:annotation>
                           <xs:documentation>Readonly number of measurements currently
made</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="gateTime" type="xs:double" default="1">
                        <xs:annotation>
                           <xs:documentation>Continuous range of independent variable
(Time) over which measurement is made.</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="nominal" type="Physical" default="0">
                        <xs:annotation>
                           <xs:documentation>Value against which any condition is
checked. This can be either an absolute value (5V) or a ratio value (50%) representing the
percentage value between the low-peak and high-peak values.</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="condition" type="enumCondition"
default="NONE">
                        <xs:annotation>
                           <xs:documentation>Test made between measurement and nominal
value</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="GO" type="xs:boolean" default="false">
                        <xs:annotation>
                           <xs:documentation>Read Only flag indicating last measurement
Pass/Fail Status. If no measurement is taken GO is False</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="NOGO" type="xs:boolean" default="false">
                        <xs:annotation>
                           <xs:documentation>Read Only flag indicating last measurement
Pass/Fail Status If no measurement is taken GO is False.</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="HI" type="xs:boolean" default="false">
```

```
                        <xs:annotation>
                           <xs:documentation>Read Only flag indicating the last
measurement High/Low Status.  If no measurement is taken HI is False</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="LO" type="xs:boolean" default="false">
                        <xs:annotation>
                           <xs:documentation>Read Only flag indicating the last
measurement High/Low Status.  If no measurement is taken LO is False</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="UL" type="Physical" default="0">
                        <xs:annotation>
                           <xs:documentation>Upper Limit value against which condition
is checked</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="LL" type="Physical" default="0">
                        <xs:annotation>
                           <xs:documentation>Lower Limit value against which condition
is checked</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
         <xs:element name="TimeInterval">
            <xs:annotation>
               <xs:documentation>Measures the time interval between the In/Sync event
going Active and the Gate event going Active.</xs:documentation>
            </xs:annotation>
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType">
                     <xs:attribute name="measuredVariable" type="enumMeasuredVariable"
default="DEPENDENT">
                        <xs:annotation>
                           <xs:documentation>Whether the measurement made is of the
dependent or independent variable.</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="measurement" type="xs:string" default="0">
                        <xs:annotation>
                           <xs:documentation>Current value measured</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="measurements" type="SAFEARRAY_VARIANT">
                        <xs:annotation>
                           <xs:documentation>Array of measurements made</
xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="samples" type="xs:long" default="0">
                        <xs:annotation>
                           <xs:documentation>Number of consecutive measurement to be
made. Zero indicates no measurement to be taken and indicates the Sensor is acting as a
monitor only</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="count" type="xs:long" default="0">
                        <xs:annotation>
                           <xs:documentation>Readonly number of measurements currently
made</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="gateTime" type="xs:double" default="1">
                        <xs:annotation>
                           <xs:documentation>Continuous range of independent variable
(Time) over which measurement is made.</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
```

```
                        <xs:attribute name="nominal" type="Physical" default="0">
                           <xs:annotation>
                              <xs:documentation>Value against which any condition is
checked. This can be either an absolute value (5V) or a ratio value (50%) representing the
percentage value between the low-peak and high-peak values.</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="condition" type="enumCondition"
default="NONE">
                           <xs:annotation>
                              <xs:documentation>Test made between measurement and nominal
value</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="GO" type="xs:boolean" default="false">
                           <xs:annotation>
                              <xs:documentation>Read Only flag indicating last measurement
Pass/Fail Status. If no measurement is taken GO is False</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="NOGO" type="xs:boolean" default="false">
                           <xs:annotation>
                              <xs:documentation>Read Only flag indicating last measurement
Pass/Fail Status If no measurement is taken GO is False.</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="HI" type="xs:boolean" default="false">
                           <xs:annotation>
                              <xs:documentation>Read Only flag indicating the last
measurement High/Low Status.  If no measurement is taken HI is False</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="LO" type="xs:boolean" default="false">
                           <xs:annotation>
                              <xs:documentation>Read Only flag indicating the last
measurement High/Low Status.  If no measurement is taken LO is False</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="UL" type="Physical" default="0">
                           <xs:annotation>
                              <xs:documentation>Upper Limit value against which condition
is checked</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="LL" type="Physical" default="0">
                           <xs:annotation>
                              <xs:documentation>Lower Limit value against which condition
is checked</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
            <xs:element name="Instantaneous">
               <xs:annotation>
                  <xs:documentation>Measures the amplitude (value) of the signal in the
dimension 'type' at specified instances in time.</xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="measuredVariable" type="enumMeasuredVariable"
default="DEPENDENT">
                           <xs:annotation>
                              <xs:documentation>Whether the measurement made is of the
dependent or independent variable. </xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="measurement" type="xs:string" default="0">
                           <xs:annotation>
                              <xs:documentation>Current value measured</xs:documentation>
```