IEEE P1641.1™/D1 Draft Guide for the Use of IEEE Std 1641, Standard for Signal and Test Definition

Sponsor

SCC20 Committee
of the
IEEE <Society Name> Society

Approved <XX MONTH 20XX>

IEEE-SA Standards Board

IEEE Standards Association Department
445 Hoes Lane
Piscataway, NJ 08854, USA

Abstract: Guidance in the use of the signal and test definition standard is provided, IEEE 1641 provides the means to define and describe signals used in testing. This guide describes how to form complex signals usable across all test platforms.
Keywords: ATE, ATLAS, ATML, ATS, automatic test equipment, signal definitions, test definitions, test requirements, test signals, unit under test, UUT

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 or the soundness of any judgments contained in its standards.

Use of an IEEE Standard is wholly voluntary. The IEEE disclaims 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 IEEE Standard document.

The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims 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. IEEE Standards documents are supplied "**AS IS**."

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation, or every ten years for stabilization. When a document is more than five years old and has not been reaffirmed, or more than ten years old and has not been stabilized, 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 IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon his or her independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

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

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Recommendations to change the status of a stabilized standard should include a rationale as to why a revision or withdrawal is required. Comments and recommendations on standards, and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854
USA

Introduction

This introduction is not part of IEEE P1641.1/D1, Draft Guide for the Use of IEEE Std 1641, Standard for Signal and Test Definition

IEEE Std 1641™, IEEE Standard for Signal and Test Definition, defines a method of accurately defining signals and their timing as used in test procedures and requirements. This guide supplements IEEE Std 1641 and is part of the document set.

This guide has been prepared to help all users of IEEE Std 1641. The STD standard may be used within any test discipline, and with any carrier language, and the examples provided herein should be seen as typical.

The guide explains how each of the layers in the STD standard are built up on the preceding (lower) layer and to inform the user how to use the layer (or layers) that are important to a specific application. It describes how a signal may be created from Basic Signal Components or other signals. It also shows the application and measurement of signals using a text-only format.

The need for the guide arose from the experience of users in the creation of signals and tests using IEEE Std 1641. This experience showed that further information in the implementation and application of signal definitions was required. The purpose of the guide is to provide guidance in the technique of implementation, application and usage of the basic signals defined in IEEE Std 1641 to create signal definitions and test requirements. This is seen as particularly important in promoting the use of a relatively new and unambiguous method of describing signals. This document is not intended to be used as an instruction manual for IEEE Std 1641 nor as a substitute for formal training, but by its nature it should find some application in the training environment.

The initial clauses are intended as a brief introduction to the application of STD. Subsequent clauses concentrate on the description of signals used in test. Stimulus signals, conditioning elements, and the acquisition of information from response signals are all covered.

The STD standard does not specify any specific carrier (test program sequencing) language, and this guide does not provide any advice about the selection of a suitable carrier language. It does show STD used within typical carrier languages and includes examples of the definition and use of signal models in different environments.

Notice to users

Laws and regulations

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

Copyrights

This document is copyrighted by the IEEE. It is made available for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making this document available for use and adoption by public authorities and private users, the IEEE does not waive any rights in copyright to this document.

Updating of IEEE documents

Users of IEEE standards should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect. In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE Standards Association web site at http://ieeexplore.ieee.org/xpl/standards.jsp, or contact the IEEE at the address listed previously.

For more information about the IEEE Standards Association or the IEEE standards development process, visit the IEEE-SA web site at http://standards.ieee.org.

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

[If the IEEE has not received letters of assurance prior to the time of publication, the following notice shall appear:]

Attention is called to the possibility that implementation of this guide may require use of subject matter covered by patent rights. By publication of this guide, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this guide are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

[The following notice shall appear when the IEEE receives assurance from a known patent holder or patent applicant prior to the time of publication that a license will be made available to all applicants either without compensation or under reasonable rates, terms, and conditions that are demonstrably free of any unfair discrimination.]

Attention is called to the possibility that implementation of this guide may require use of subject matter covered by patent rights. By publication of this guide, no position is taken with respect to the existence or validity of any patent rights in connection therewith. A patent holder or patent applicant has filed a statement of assurance that it will grant licenses under these rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses. Other Essential Patent Claims may exist for which a statement of assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this guide are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Participants

At the time this draft guide was submitted to the IEEE-SA Standards Board for approval, the Test and ATS Description Subcommittee had the following membership:

Ashley Hulme, *CoChair*
Ion Neag, *CoChair*

Participant1	Participant4	Participant7
Participant2	Participant5	Participant8
Participant3	Participant6	Participant9

The following members of the <individual/entity> balloting committee voted on this guide. Balloters may have voted for approval, disapproval, or abstention.

(to be supplied by IEEE)

Balloter1	Balloter4	Balloter7
Balloter2	Balloter5	Balloter8
Balloter3	Balloter6	Balloter9

When the IEEE-SA Standards Board approved this guide on <XX MONTH 20XX>, it had the following membership:

(to be supplied by IEEE)

<Name>, *Chair*
<Name>, *Vice Chair*
<Name>, *Past President*
<Name>, *Secretary*

SBMember1	SBMember4	SBMember7
SBMember2	SBMember5	SBMember8
SBMember3	SBMember6	SBMember9

*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

<Name>, *NRC Representative*
<Name>, *DOE Representative*
<Name>, *NIST Representative*

<Name>
IEEE Standards Program Manager, Document Development

<Name>
IEEE Standards Program Manager, Technical Program Development

Contents

1. Overview 1
1.1 Scope 1
1.2 Purpose 2

2. Normative references 2

3. Definitions, acronyms, and abbreviations 2
3.1 Definitions 2
3.2 Acronyms and abbreviations 5

4. Introduction to IEEE 1641 5
4.1 Requirements for signal and test definition (STD) standard 5
4.2 Features of the STD standard 6
4.3 Hierarchy of signal definitions 6
4.4 Improvements to the standad 8
4.5 Signals and streams 8
4.6 Inclusion of support for Automatic Test Markup Language (ATML) 11

5. Describing signals using IEEE 1641 11
5.1 Overview 11
5.2 Physical types 14
5.3 Signals 15
5.4 Using signal graphs to create a signal 20
5.5 Documenting signal descriptions 24
5.6 Using signal definitions 30

6. Signal models 30
6.1 General 30
6.2 TSF model as signal template 30
6.3 Typical signal model (TSF) 32
6.4 Using TSFs in a test requirement or program 34
6.5 Measurement models (signal measurement) 35

7. Defining measurements with STD 35
7.1 General 35
7.2 Sensors 36
7.3 Measurement maps 41
7.4 Intrinsic measurements 42
7.5 Generic measurement 51
7.6 Reference signals description 55
7.7 Different valid measurement methods 55

8. Describing Tests and Test Requirements 58
8.1 Structure—sequence and signals 58
8.2 Using tools such as graphical environments 59
8.3 Portable test requirements 61
8.4 ATML Test Description 69
8.5 Examples of test requirements using TPL, ATLAS, and native languages 70

9. Basic signal components (BSCs) 93
9.1 Introduction of new BSCs 93

9.2 BSC interfaces 93
9.3 Diagrammatic representation of BSCs 94
9.4 SignalFunctions and events 96

10. Test Signal Framework (TSF) 109
10.1 Introduction to TSF 109
10.2 Building TSF signal models using BSCs 110
10.3 Examples of source signal models 110
10.4 Dual or multiple use TSF models 118
10.5 Signal models with preset internal attributes 121
10.6 Examples of signal models that process input signal 125
10.7 Example of signal models that include connection BSCs 130
10.8 SignalDelay TSF model for SWEEP 133
10.9 TSF model for linear sweep using frequency modulation (FM) BSC 134
10.10 TSF model for logarithmic sweep using FM BSC 135
10.11 TSF attributes mappings and formulae 137
10.12 Synchronization of signal model 138
10.13 Gating a signal model 147
10.14 Use of XML to specify TSF signal model information 149
10.15 Use of IDL to specify TSF signal model information 159

11. Digital Signals 163
11.1 Possible states for digital stream 163
11.2 Generating a digital stream 163
11.3 Converting digital data into a physical digital signal. 165
11.4 Using the SelectCase BSC 169
11.5 Extracting digital data from a physical digital signal. 173

12. More about Events and their interaction 173
12.1 Interaction between streams 173
12.2 Recovering Event information from digital streams 178

13. Test Procedure Language (TPL) 179
13.1 Introduction to TPL 179
13.2 Simple test requirement in TPL 180
13.3 Further test requirement in TPL 188
13.4 Examples of test statements in TPL 196
13.5 Quantities, units, and unit symbols 206

14. Signal Modeling Language (SML) 207
14.1 Introduction to SML 207
14.2 Using SML to define a BSC 208

Annex A (informative) Glossary 214

Annex B (informative) Intrinsic measurement 216
B.1 Introduction 216
B.2 Essential aspects and parts of a measurement 216
B.3 Interpreting measurement information 220
B.4 Examples showing breakdown of derivation of attributes 221
B.5 Creating IEEE 1641 measurements for legacy ATLAS systems 228

Annex C (informative) Generic measurement 231
C.1 Introduction 231
C.2 Information required to complete a measurement 231
C.3 Interpreting measurement information 232

C.4 Examples showing breakdown of derivation of attributes 233
C.5 Creating IEEE 1641 measurements for legacy ATLAS systems 239
C.6 Dependence upon reference signal....241
C.7 Measurements on a complex signal (square wave) 245

Annex D (informative) The role of Resource Adapter Information (RAI) in IEEE 1641 247
D.1 Introduction 247
D.2 Maximizing the platform independence of test requirements....247
D.3 Interpreting the principles and rules 248
D.4 Test requirement presentation....249

Annex E (informative) Understanding IEEE 1641 capabilities....250
E.1 How to define and connect Loads 250
E.2 Implementing short circuits in IEEE 1641 262
E.3 Signals and triggering....263

Annex F (informative) Implementation of IEEE 1641—Application techniques 268
F.1 Implementing IEEE 1641—RF stimulus and measurement....268
F.2 Implementing IEEE 1641—Amplifier characterization using IEEE 1641 277
F.3 Implementing IEEE 1641—Envelope testing of waveforms....290
F.4 Implementing IEEE 1641—Resource drivers & COTS languages 301
F.5 Implementing IEEE 1641—Compilation techniques 311

Annex G (informative) Bibliography 319

Table of Figures (DELETE BEFORE PASSING TO IEEE)

Figure 1 —The four STD layers 7
Figure 2 —State diagram for a signal stream 9
Figure 3 —State diagram for an event stream 10
Figure 4 —State diagram for a digital stream....11
Figure 5 —Signal flow between two SignalFunctions 12
Figure 6 —SignalFunction diagram....13
Figure 7 —Simple signal summing two sinusoidal waveforms....13
Figure 8 —Constant (Voltage, Time) signal....16
Figure 9 —Sinusoid (Voltage, Time) signal....17
Figure 10 —Example showing phase relationship on a signal....18
Figure 11 —Effect of connection: both waveforms valid for signal....18
Figure 12 —Relationship between instantaneous voltage and power....19
Figure 13 —Sources and conditioners 21
Figure 14 —Events 22
Figure 15 —Measurements....23
Figure 16 —Connections 24
Figure 17 —Generic TSF model diagram....28
Figure 18 —Simple TSF model (AC_SIGNAL) 28
Figure 19 —Multiple languages using common signal component definitions....29
Figure 20 —Annotated SYNCHRO signal TSF model 33
Figure 21 —Output from SYNCHRO TSF model....34
Figure 22 —STD components and their typical use 36
Figure 23 —Measurement window....37

Figure 24 —Measurement windows with positive gateTime and gated event stream 38
Figure 25 —Measurement windows with negative gateTime and gated event stream 39
Figure 26 —Continuous monitoring over rolling measurement window..40
Figure 27 —Generating event for input signal above zero 40
Figure 28 —Measurement map of rms value of damped sinusoid signal 42
Figure 29 —Waveform from <Instantaneous type="Voltage" reftype="Time" /> 43
Figure 30 —Waveform from <Instantaneous type="Power" reftype="Time" /> 44
Figure 31 —Waveform from <Instantaneous type="Voltage" reftype="Frequency" />.......................44
Figure 32 —Average measurement 45
Figure 33 —Average measurement with longer gate time...46
Figure 34 —Average measurement where gate time is too short 46
Figure 35 —Average measurement with short gate time & 5 samples 47
Figure 36 —RMS Measurement 47
Figure 37 —MaxInstantaneous measurement...48
Figure 38 — MinInstantaneous measurement 48
Figure 39 —PeakPos measurement 49
Figure 40 — PeakNeg measurement...49
Figure 41 — Peak measurement 50
Figure 42 — PeaktoPeak measurement 50
Figure 43 —Count measurement with Samples=2 51
Figure 44 —Interval measurement...51
Figure 45 —Generic measure best match example..54
Figure 46 —Errors (rms) for best match example 54
Figure 47 —Measurement of complex square wave...56
Figure 48 —Measurements on complex square wave...57
Figure 49 —Distribution of values for amplitude levels..58
Figure 50 —Part of graphical test plan 59
Figure 51 —Section of graphical program showing detail of program flow 60
Figure 52 —Simple TSF for measuring voltage 60
Figure 53 —TPL used with any carrier language 61
Figure 54 —Simple block diagram design...62
Figure 55 —Noisy AC_Signal 65
Figure 56 —DC amplifier example...70
Figure 57 —AC amplifier test block diagram design 91
Figure 58 —Start of sweep signal 92
Figure 59 —End of sweep signal 93
Figure 60 —BSC diagram..95
Figure 61 —Sinusoid BSC...96
Figure 62 —Event stream only 98
Figure 63 — Source with Sync event 98
Figure 64 —Effect of Sync event on sine wave..99
Figure 65 —Source with Gate event..99
Figure 66 —Effect of Gate event on sine wave 99
Figure 67 —Source with Gate and Sync events..100
Figure 68 —Effect of Gate and Sync events on sine wave 100
Figure 69 —Source with separate Gate and Sync events...101
Figure 70 —Effect of different Gate and Sync events on sinusoid wave...101
Figure 71 —Conditioner with sync input..102
Figure 72 —Effect of Sync event on conditioner 102
Figure 73 —Conditioner with gate input 103
Figure 74 —Effect of Gate event on a conditioner 103
Figure 75 —Conditioner with sync and gate inputs..104
Figure 76 —Effect of Sync and Gate events on conditioner...104
Figure 77 —Source with sync input (followed by conditioner)...104
Figure 78 —Effect of Sync event on source (followed by conditioner) 105
Figure 79 —Source and conditioner with sync inputs 105

Figure 80 —Effect of Sync event on both source and conditioner 106
Figure 81 —Source with gate input and conditioner with sync input 106
Figure 82 —Effect of Gate event on source and Sync event on conditioner 107
Figure 83 —Source with gate and sync input plus conditioner with sync input 107
Figure 84 —Effect of Gate and Sync events on source plus Sync event on conditioner 107
Figure 85 —Source with sync input plus conditioner with gate and sync input 108
Figure 86 —Effect of Sync event on source plus Gate and Sync events on conditioner 108
Figure 87 —Source and conditioner with separate sync inputs 109
Figure 88 —Effect of different Sync events on source and conditioner 109
Figure 89 —AC_SIGNAL model 111
Figure 90 —AC_SIGNAL example....113
Figure 91 —NOISY_AC_SIGNAL model 114
Figure 92 —NOISY_DC_SIGNAL model 117
Figure 93 —AM_SIGNAL model 119
Figure 94 —PULSED_DC_TRAIN model....121
Figure 95 —DME_INTERROGATION signal model 123
Figure 96 —TSF RADAR_RX_SIGNAL 126
Figure 97 —Signal model with incoming SignalFunction....128
Figure 98 —SYNCHRO signal model....131
Figure 99 —SWEEP signal model....133
Figure 100 —Linear sweep using FM BSC 135
Figure 101 —Logarithmic sweep using FM BSC....136
Figure 102 —AM_SIGNAL model with all sync inputs connected 139
Figure 103 —Signal model with sync input 139
Figure 104 —Signal model showing implicit sync connections 140
Figure 105 —Signal model as collection of BSCs....141
Figure 106 —Signal model with explicit sync input to TimedEvent BSC 143
Figure 107 —Signal model with randomizing input to sinusoid BSC 145
Figure 108 —Signal model with single implicit internal Gate....147
Figure 109 —Signal model with additional gating BSC....148
Figure 110 —Signal model showing explicit internal Gate 149
Figure 111 —State diagram for a digital stream 163
Figure 112 —A digital stream ("HLXZHLXZ") 164
Figure 113 —Converting a digital streams into a physical signal....164
Figure 114 —Encode BSC....167
Figure 115 —Generating a BasicNRZ digital signal 168
Figure 116 —Simple DAC using SelectCase 170
Figure 117 —Distorted sinusoid generated from 3-bit code 172
Figure 118 —Sinusoid controlled by event stream....174
Figure 119 —Waveform from Sinusoid controlled by event stream 174
Figure 120 —Event generator with Clock followed by NotEvent 175
Figure 121 —Waveform showing Clock followed by NotEvent....175
Figure 122 —Waveform from Figure 120 showing effect of a gate on Clock BSC 175
Figure 123 —Constant gated by Sinusoid 176
Figure 124 —Waveform showing Constant gated on continuously by Sinusoid....177
Figure 125 —Recovering event data via a NotEvent....177
Figure 126 —Waveform showing Constant gated by recovered event data 178
Figure 127 —UUT for example TPL program 181
Figure 128 —Power supply UUT for TPL program 189
Figure 129 —Time-based events 201
Figure 130 —Plot of sinusoid BSC from SML information 210
Figure 131 —Plot of AM BSC from SML information....213

Figure B.1—Model of average voltage measurement TSF 229
Figure B.2—TSF used for measurement....229

Figure B.3—Enlarged view of average voltage measurement 230
Figure B.4—XML version of average voltage measurement TSF 230
Figure C.1—Generic measurement of an AM signal 240
Figure C.2—Model segment of TSF for measuring modulation amplitude 240
Figure C.3—Measurement showing difference between expected and actual signal....241
Figure C.4—Amplitude of sinusoid signal....242
Figure C.5—Amplitude of square wave signal 242
Figure C.6—Amplitude of trapezoid signal 243
Figure C.7—Measuring trapezoid signal as square wave....244
Figure C.8—Measuring exponential signal as trapezoid....244
Figure C.9—Generic measurement of instantaneous error aberration 245
Figure C.10—Measurement of complex square wave....246
Figure C.11—Waveform aberration for complex square wave....246
Figure E.1—Load impedance connected to a UUT....251
Figure E.2—Load effectively connected to output connections....252
Figure E.3—Load connected to two output pins 253
Figure E.4—Load as a transformation BSC with output port 254
Figure E.5—Load used with a serial port (Via BSC) 254
Figure E.6—Parallel Load on the input to a UUT....255
Figure E.7—Serial Load on the input to a UUT....256
Figure E.8—Load as a parallel network element 257
Figure E.9—Load as a serial network element....258
Figure E.10—Load BSC on a two-wire connection 259
Figure E.11—Load BSC on a three-wire connection 259
Figure E.12—Load BSC on a three phase wye connection....260
Figure E.13—Load BSC on a three phase wye connection (alternative view)....260
Figure E.14—Load BSC on a three phase delta connection....261
Figure E.15—Load BSC on a parallel digital connection 261
Figure E.16—dc voltage triggered by third cycle of sinusoid....265
Figure E.17—Sinusoid starting with 90 deg offset....266
Figure E.18—Simulation showing trigger on rising edge 267
Figure F.1—Synthetic instrumentation architecture of the ATS used....269
Figure F.2—Spectrum measurement setup....270
Figure F.3—Source signal TSF 270
Figure F.4—Spectrum measurement TSF 271
Figure F.5—Actual results from an implementation of a characterization measurement....273
Figure F.6—Maximum power TSF 274
Figure F.7—Spur search TSF....275
Figure F.8—UUT power supply TSF....278
Figure F.9—RF CW stimulus TSF....279
Figure F.10—Example of output from Sinusoid BSC....279
Figure F.11—Example of Sinusoidal Power....280
Figure F.12—1 dB compression point formula....281
Figure F.13—1 dB compression point TSF....282
Figure F.14—Example average power *vs* $V_{RMS} \times I_{RMS}$ 283
Figure F.15—Gain measurement TSF....283
Figure F.16—Gain transfer measurement TSF....284
Figure F.17—IEEE 1641 simulator interface....285
Figure F.18—Modeling the UUT using BSCs 285
Figure F.19—Gain transfer measurement TSF Simulation 286
Figure F.20—Simulation of Gain transfer measurement....287
Figure F.21—Signal to Driver Translation Process....288
Figure F.22—Typical code generated by a Resource Manager....289
Figure F.23—Signal graph of measurement....291
Figure F.24—Test envelope for waveform 292
Figure F.25—Test waveform 293

Figure F.26—Upper Boundary Test.....293
Figure F.27—Lower Boundary Test.....294
Figure F.28—Test showing point of failure.....294
Figure F.29—Signal graph for WaveformStep version.....295
Figure F.30—Waveform and permissible envelope showing point of failure.....296
Figure F.31—Signal graph of normalized measurement.....297
Figure F.32—Measurement task; check waveform remains between boundaries.....297
Figure F.33—Output from the two "Diff" BSCs.....298
Figure F.34—Normalized measurement.....298
Figure F.35—IEEE 1641 Layers showing information formats.....304
Figure F.36—An example of a TSF for measuring RF noise rejection.....305
Figure F.37—SML simulation of the TSF model.....306
Figure F.38—Defining the TSF interface.....307
Figure F.39—TSF signal modeling tool.....312
Figure F.40—IEEE 1641 core interface supplemented by an exported TSF interface.....313
Figure F.41—Actions required to implement capabilities.....314
Figure F.42—Driver command.....315
Figure F.43—Driver assignment.....316
Figure F.44—Signal to driver translation framework.....317
Figure F.45—Sample driver code.....318

Table of Tables (DELETE BEFORE PASSING TO IEEE)

Table 1 —Qualifiers and their meaning.....17
Table 2 —Interface properties.....26
Table 3 —Simple interface (AC_SIGNAL).....27
Table 4 —Model.....27
Table 5 —Simple model (AC_SIGNAL).....27
Table 6 —Measurement of two state levels on a square wave.....56
Table 7 — Attribute values obtained from two state-level values.....56
Table 8 —Intrinsic measurements on complex square wave.....57
Table 9 —AC_SIGNAL interface.....111
Table 10 —AC_SIGNAL model.....112
Table 11 —NOISY_AC_SIGNAL interface.....115
Table 12 —NOISY_AC_SIGNAL model.....115
Table 13 —NOISY_DC_SIGNAL interface.....117
Table 14 —NOISY_DC_SIGNAL model.....118
Table 15 —AM_SIGNAL interface.....119
Table 16 —AM_SIGNAL model.....120
Table 17 —PULSED_DC_TRAIN interface.....121
Table 18 — PULSED_DC_TRAIN model.....122
Table 19 —TSF DME_INTERROGATION interface.....123
Table 20 —TSF DME_INTERROGATION model.....124
Table 21 —TSF RADAR_RX_SIGNAL Interface.....126
Table 22 —TSF RADAR_RX_SIGNAL Model.....127
Table 23 —MODULATED_SIGNAL interface.....129
Table 24 —MODULATED_SIGNAL model.....129
Table 25 —SYNCHRO interface.....131
Table 26 —TSF SYNCHRO model.....132
Table 27 —SWEEP interface.....133

Table 28 —SWEEP model 134
Table 29 —LINEAR_SWEEP Interface 135
Table 30 —LOG_SWEEP Interface....136
Table 31 —Modified DME interrogation model 140
Table 32 —State of BSCs around first Sync event....142
Table 33 —State of BSCs around subsequent Sync events 142
Table 34 —State of BSCs around first Sync event....143
Table 35 —State of BSCs around subsequent Sync events 144
Table 36 —Model with explicit internal Sync....144
Table 37 —DME_INTERROGATION model with randomized input 145
Table 38 —State of BSCs around first Sync event....146
Table 39 —State of BSCs around subsequent Sync events 147
Table 40 —MODULATED_SIGNAL interface 154
Table 41 —MODULATED_SIGNAL model 155
Table 42 —MODULATED_SIGNAL interface 161
Table 43 —Effect of events on a digital stream 165
Table 44 —Digital streams for the same data with different attributes 166
Table 45 —Attribute values for BasicNRZ signal....168
Table 46 —Data flow through digital signal model 168
Table 47 —Data flow for 4 bit wide digital signal 169
Table 48 —Operation of SelectCase BSC 170
Table 49 —Signal values at SelectCase inputs....171
Table 50 —Generating the analogue outputs....172
Table 51 —Effect of events on a signal stream 174
Table 52 —Effect of events on an event stream 176
Table 53 —Recovery of event information from a signal stream....178
Table 54 —Recovery of event information from a digital stream 178
Table 55 —Effect of gating OrEvent & Sum BSCs 179
Table 56 —CONSTANT_I_LOAD interface....189
Table 57 —Constant current load model 189
Table 58 —Decimal codes for symbols....206
Table 59 —Keyboard entry codes for symbols 207

Table B.1—Default intrinsic measurement information 220
Table B.2—Derivation of attribute values for simple rms measurement 221
Table B.3—Derivation of attribute values for rms monitor 222
Table B.4—Derivation of attribute values for rms measurement with an expected value 223
Table B.5—Derivation of attribute values for rms measurement with type conversion....224
Table B.6—Derivation of attribute values for current measurement with capability information 225
Table B.7—Derivation of attribute values for voltage measurement with capability information....226
Table B.8—Derivation of attribute values for voltage measurement with capability information....227
Table B.9—Legacy ATLAS measurements and the IEEE 1641 equivalents 228
Table C.1—Default generic measurement information....232
Table C.2—Derivation of attribute values for simple 'lazy' measurement 233
Table C.3—Derivation of attribute values for measurement with expected value 234
Table C.4—Derivation of attribute values for simple 'lazy' monitor....235
Table C.5—Derivation of attribute values for average current measurement 235
Table C.6—Derivation of attribute values for C.4.5 error measurement using 'As' signal....236
Table C.7—Derivation of attribute values for measurement of current error....237
Table C.8—Derivation of attribute values for input signal measurement 238
Table C.9—Generic measurements on square wave 246
Table F.1—TSF Interface for normalized waveform measurement 300

Draft Guide for the Use of IEEE Std 1641, Standard for Signal and Test Definition

***IMPORTANT NOTICE:** This standard is not intended to ensure safety, security, health, or environmental protection. Implementers of the standard are responsible for determining appropriate safety, security, environmental, and health practices or regulatory requirements.*

This IEEE document is made available for use subject to important notices and legal disclaimers. These notices and disclaimers appear in all publications containing this document and may be found under the heading "Important Notice" or "Important Notices and Disclaimers Concerning IEEE Documents." They can also be obtained on request from IEEE or viewed at http://standards.ieee.org/IPR/disclaimers.html.

1. Overview

1.1 Scope

This guide provides application information and guidance for users who write, develop, implement, and support test requirements, signal definitions, and signal responses using IEEE Std 1641™, the signal and test definition (STD) standard. Examples of the definition and use of signal models in different environments are included.

1.2 Purpose

This guide explains how signal definitions and test requirements may be implemented in conformance with IEEE Std 1641. It also provides background information, tutorial support, and examples of signal definitions and test requirements for users of the standard.

2. Normative references

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

IEEE Std 1641-2010, IEEE Standard for Signal and Test Definition.[2]

3. Definitions, acronyms, and abbreviations

3.1 Definitions

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary: Glossary of Terms & Definitions* should be consulted for terms not defined in this clause.[3]

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.[4]

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

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

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

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

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

carrier language: A native computer language used to support signal and test definition (STD) signals (whether in the form of test statements or graphical representations) and to provide all other programming

[2] IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).
[3] *The IEEE Standards Dictionary: Glossary of Terms & Definitions* is available at http://shop.ieee.org/.
[4] In this standard, the term “ATLAS” refers to any version or subset whether it is a formal standardized version or a project specific modified subset.

requirements. STD signals will need to be converted into carrier language code before program execution can commence.

class: A generic set of predefined abstract test objects.

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

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

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

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

event stream: A signal comprising a stream of active/inactive pulses generated from signal models.

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.

gated event stream: An event stream used to gate a basic signal component (BSC) or test signal framework (TSF).

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.

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

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

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

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

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

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

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

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

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

signal graph: See: signal model.

signal model: A collection of test signal frameworks (TSFs) or basic signal components (BSCs) interconnected to define a signal. Used to define the model part of a TSF (TSF model) or a signal component used with the Require method. Syn: signal graph.

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

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

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

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

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

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

test: (A) An action or group of actions that are performed on a unit under test (UUT) to evaluate its parameter(s) or characteristic(s). (Derived from IEEE Std 771-1989). (B) An observed activity that may be caused to occur (e.g., stimulus-response) in order to obtain information about the behavior of a test subject. (Derived from IEEE Std 1671-2006). (C) A set of stimuli, either applied or known, combined with a set of observed responses and criteria for comparing these responses to a known standard. (Derived from IEEE Std 1232-2002).

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

Test Signal Framework (TSF): A combination of basic signal components (BSCs) to form more complex signals.

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

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

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

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

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

value: The quantitative magnitude of a signal attribute.

3.2 Acronyms and abbreviations

ac	alternating current
ADC	analog-to-digital converter
AM	amplitude modulation
API	application programming interface
ARB	arbitrary waveform generator
ASCII	American Standard Code for Information Interchange
ATE	automatic test equipment
ATLAS	Abbreviated Test Language for All Systems
ATML	Automatic Test Markup Language
ATS	automatic test system
BSC	basic signal component
C/ATLAS	Common/Abbreviated Test Language for All Systems – also see [B9]
COTS	commercial off-the-shelf
CW	continuous wave
dc	direct current
DAC	digital-to-analog converter
DCE	Distributed Computer Environment
DME	distance measuring equipment
DOD	Department of Defense
FG	function generator
FM	frequency modulation
IDL	Interface Definition Language
ILS	instrument landing system
LSB	least significant bit
MSB	most significant bit
PM	phase modulation
RAI	Resource Adapter Information
RF	radio frequency
rms	root-mean-square
SI	Système International d'Unités (International System of Units) – also see [B6]
SML	Signal Modeling Language
STD	signal and test definition
TPL	Test Procedure Language
TRD	test requirement document
trms	true root-mean-square
TSF	test signal framework
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
UTF-8	Unicode 8 bit encoding table
UUT	unit under test
XML	Extensible Markup Language
XSD	XML schema document

4. Introduction to IEEE 1641

4.1 Requirements for signal and test definition (STD) standard

At the time that IEEE Std 1641 was originally developed, the working group generated a series of requirements. The principal requirements for the STD standard were determined to be as follows

a) Provide the means to unambiguously define all signals.

b) Provide a sound mathematical foundation for the definition of signals so that signals could be accurately exported, re-created, and used directly by different test equipment.

c) Enable signals to be defined by providing an architecture that permitted the combination of existing signals.

d) Support the collection of signals with similar application domains into readily extensible libraries.

e) Provide a means whereby signals could be defined, used, and controlled in several programming languages.

f) Enable test programs to be written for any test system that supported the same STD signals.

These requirements resulted in a standard with a hierarchical structure that satisfied these requirements. The features and structure of the standard are outlined 4.2 and 4.3.

4.2 Features of the STD standard

The key features of the STD standard are as follows:

a) *Definition of signals.* STD has a rigorous mathematical and definitive foundation for all of its signal components. In other words, any signal defined using this standard will be the same regardless of what equipment is used.

b) *Modern language structures.* STD does not specify a programming language. Any carrier language is acceptable. Its suitability is determined by its support of the minimum requirements as described in the STD standard.

c) *Extensibility.* All signals are defined mathematically based on basic signal components (BSCs). New signals may be described by combining existing signals and BSCs. Thus, any desired signal can be described, and there is no limit on the extensibility of signals supported by the standard.

d) *Implementation of new technologies.* The STD standard supports the implementation of new technologies by providing users with the ability to describe their own signals.

e) *Flexibility of use.* The STD standard not only supports traditional textual test requirements and sequences, using the Test Procedure Language (TPL), but also supports modern graphical programming techniques, diagnostic driven programming techniques, and other environments where the test signals and timing need to be separated from the program sequencing and signal routing instructions and implementations.

4.3 Hierarchy of signal definitions

The STD standard uses a hierarchy of definitions to describe any signal. This hierarchy is illustrated in Figure 1, which shows the four STD layers. Most users will not need to familiarize themselves with every level in the hierarchy. The lowest level, the Signal Modeling Language (SML) level, will be used only by certain specialists within an organization. It may be used for the introduction of a new keyword to cover a new test application not currently embraced. The SML definitions are presented using Haskell 98 [B15].

The level above the SML is the BSC layer. This layer provides all the basic building blocks that enable more complex signals to be constructed. BSCs create SignalFunction base classes and BSC subclasses of up to four levels. A BSC may be used on its own as a signal to be applied to a unit under test (UUT). For example, a simple ac signal may require no more than is provided with the sinusoid BSC. If the ac signal requires further parameters, then the sinusoid BSC will have to be combined with one or more other BSCs to provide the overall signal desired.

Combining one or more BSCs into a more complex model results in a test signal framework (TSF) model. TSF models compose the next level in the hierarchy. These represent the majority of the signals required by a test requirement to be applied or measured at the interface with a UUT. The TSF models belonging to a particular test technology may be grouped together into a TSF library. The STD standard provides a typical TSF library in an informative annex (Annex E of IEEE Std 1641-2010). This annex presents examples of many of the signals described in C/ATLAS [B9]. It should be noted that TSF signals are not necessarily exclusive to a single TSF library. For example, a dc signal may be in many different libraries. They do not even have to be the same dc signal as the definition of the signal that is carried with the TSF class definition in the TSF library.

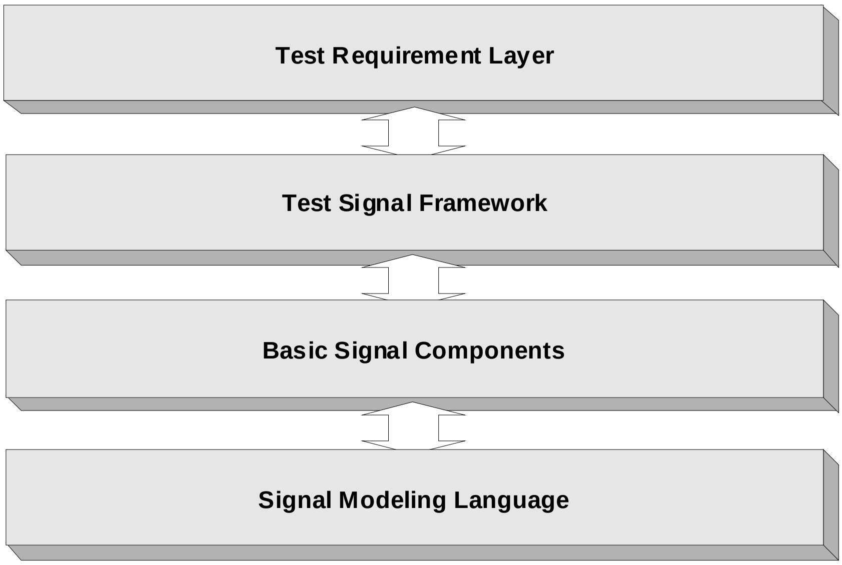


Figure 1—The four STD layers

It is the structured hierarchy of these three lower levels that ensures the portability of test requirements supported by the STD standard. As every signal applied or measured is founded upon the same lower level definitions, all implementations of the same test requirement should produce the same results.

The upper layer is the test requirement layer. This layer allows definitions of tests (e.g., test requirements, test procedures, test programs) to be formally described by combining signal definitions with procedural information. The signal information may be presented in different ways and the procedural information may be provided in a programming language of the user's choice. One option is to use the TPL defined in the standard for signals. TPL, together with a carrier language, provides the same programming facilities as any other test programming language, but it also includes signal-oriented descriptions that were a feature and an advantage of ATLAS. The same restricted programming features that ATLAS contained, however, do not constrict this layer.

The STD standard defines how signal statements should be written; in other words, all implementations of the STD standard will provide the same signal for an identical definition. The procedural part of a test requirement may be written in any suitable programming language. This language is referred to as the carrier language in the STD standard because it is the programming language that "carries" the signal statements. Annex H of IEEE Std 1641–2010 shows how TPL statements are mapped into a typical carrier language.

The test requirement layer is not necessarily restricted to a text-based program definition environment. Users are free to use a graphical programming environment should that be more relevant to their application.

The body of the STD standard concentrates upon the application of the STD standard to the electrical signal domain. The content of the STD standard can be applied to a much wider scope of signal definition, however, because the signal-defining terminology is not restricted to the electrical signal domain. For example, pneumatic, hydraulic, and mechanical signals can also be defined by the BSCs. The signal values and their dimensions have to be appropriate to the nonelectrical signal domain.4.5 Improvements to the standard

4.4 Improvements to the standad

A revision of IEEE Std 1641 was published in 2010. Most of the changes introduced in that revision do not represent new items but are clarifications and removal of ambiguities. Key to the revision process was the work to improve the standard whist maintaining backward compatibility. What is new is how various aspects of the standard are described. Some of the key areas for this improved definition include signal states, multi channel behavior, and measurement transforms.

There have been some additions to the revised standard. These include some new BSCs to simplify the definition of digital signals and signal streams. Although digital signals were supported by the original standard, describing complex digital structures or large amounts of digital data was comparatively unwieldy. During the revision process, considerable effort was put into methods of describing digital signals.

4.5 Signals and streams

4.5.1 Different types of stream

It should be emphasized that the behavior of signals has not been changed. Signals and events behave just as they did in the 2004 revision. What has happened is a new clarity in the relationship of the signals and events, plus the addition of digital streams.

The term “signal” is now used generically to refer to analog and digital signals as well as events. To distinguish between the different types, reference is made to signal streams, event streams and digital streams:

a) Signal stream—a physical signal stream, i.e. a signal of type Physical

b) Event stream—a stream of events

c) Digital stream—a stream of digital states.

A digital stream is a signal representing the digital states only, without any reference to a physical state that may be used to carry the digital information.

A physical signal, event or digital stream element may be active or inactive and also may be controlled by events, i.e., it may be gated on or off. Of course, it is quite common for a SignalFunction—a BSC or a TSF—not to have any gate connection at all. But this is the same as being permanently gated on, so does not require separate discussion. Thus, there are four possible states that may be adopted; active & gated on, active & gated off, inactive & gated on, and inactive & gated off. The states that can be adopted by each may be illustrated diagrammatically.

4.5.2 States available to a signal stream

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

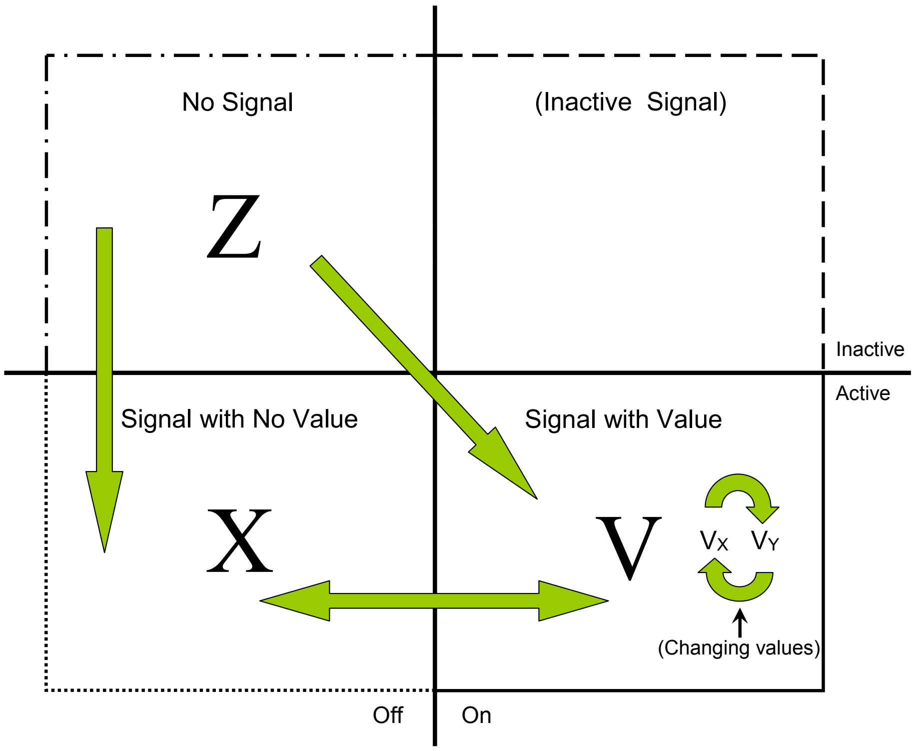


Figure 2—State diagram for a signal stream

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

4.5.3 States available to an event stream

Figure 3 shows the states available to an event stream. The states attained by an event conditioner will also depend upon the signal on its input. Before an event exists it may be considered to be in the "No Event" (Z) state. When it is invoked it passes into the gated on state but may be active or inactive. An event may change between active and inactive states while gated on. Events may be gated on and off by other events and switched between the "No Event" state and one of "Event Inactive" (I) or "Event Active" (A) as appropriate. The fourth state, "Event Off" (X) in the diagram, may be considered not to exist for an event source.

When used to gate a signal, there is very little difference between an event in the "No Event" state and the "Event Inactive" state; either of these will effectively gate the signal off. Only the "Event Active" state will gate a signal on.

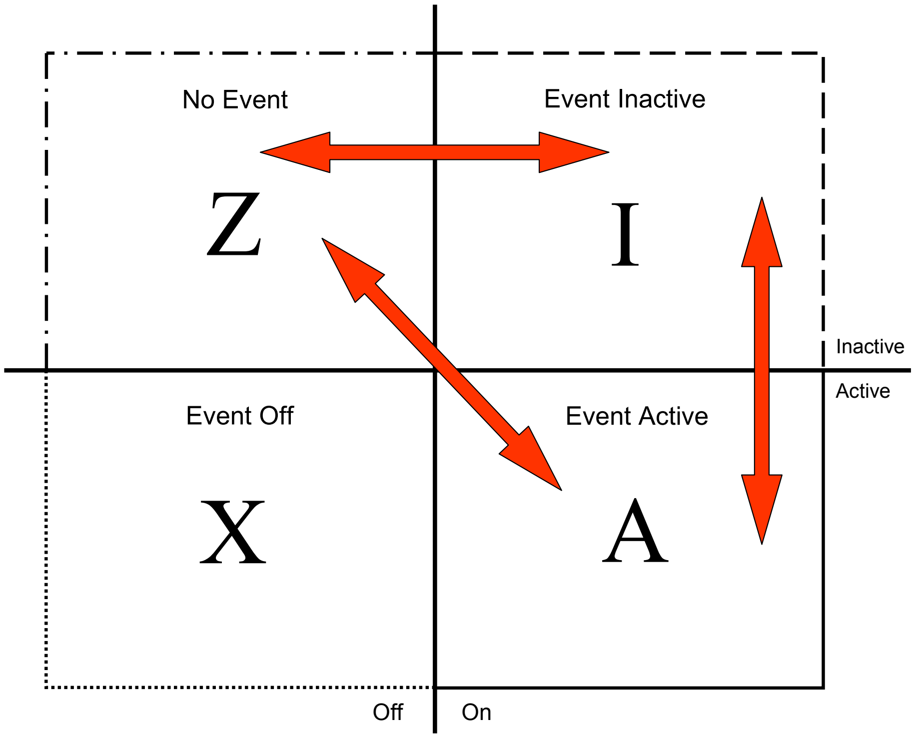


Figure 3—State diagram for an event stream

4.5.4 States available to a digital stream

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

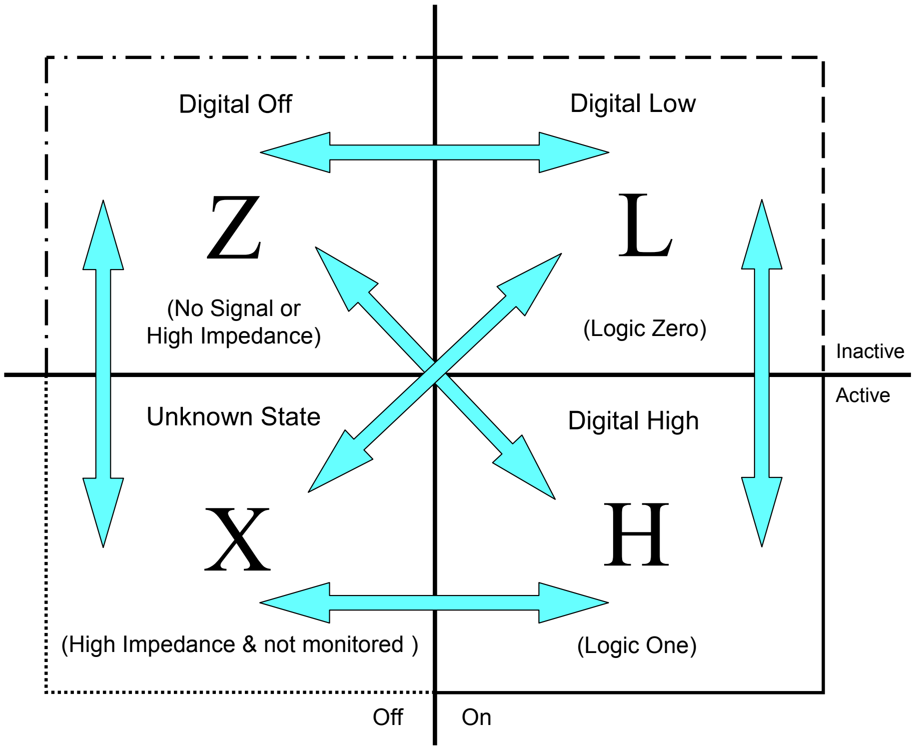


Figure 4—State diagram for a digital stream

A "Digital Off" (Z) state indicates that there is no signal. The "Digital High" (H) and "Digital Low" (L) states represent the normal two digital states (0 and 1). The "Unknown State" (X) represents the fact that a digital signal is present but it is not possible to know what it is, i.e. it is not monitored when used as response information (a "don't care").

4.6 Inclusion of support for Automatic Test Markup Language (ATML)

During the time that the STD standard was being developed, the ATML was initiated. The purpose of ATML is to define a collection of Extensible Markup Language (XML) schemas that allow automatic test equipment (ATE) and test information to be exchanged in a common format adhering to the XML standard (see IEEE 1671™ [B10]). This led to the inclusion of XML schemas for the BSCs andTSFs in the STD standard, providing a basis for test requirements written using ATML to be supported by the same signal definitions.

5. Describing signals using IEEE 1641

5.1 Overview

The STD standard provides a means to describe and model the test requirements needed to test a UUT. In describing these test requirements, the external behavior of a test subject is defined unambiguously, by modeling the signals, attributes, and values that pertain to the UUT.

The STD standard defines entities known as SignalFunctions. New SignalFunctions can be created by combining existing SignalFunctions. The behavior and specification of new SignalFunctions so created is defined (in part) by their hierarchical decomposition and their subcomponent SignalFunctions. At the lowest level, SignalFunctions are BSCs. At this level, their behavior and specification are defined by the STD standard.

Interaction between SignalFunctions occurs only via the information that passes between them (see Figure 5).

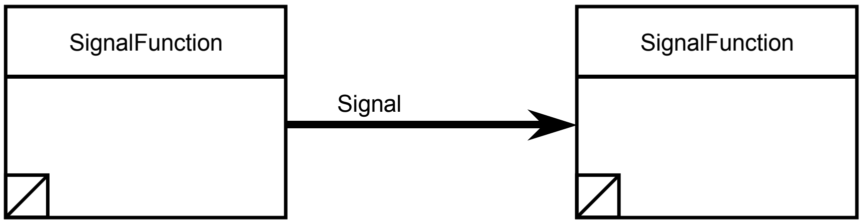


Figure 5—Signal flow between two SignalFunctions

The information transfer between SignalFunctions falls into two categories, continuous and discrete. The continuous transfer of information represents signals, and the discrete transfer of information represents events.

The behavior that a SignalFunction exhibits is controlled through attributes. The behavior is a combination of the class of the SignalFunction, such as Sinusoid (Voltage, Time), and the values of its attributes, such as frequency, amplitude, and phase. As well as describing a source of a signal, SignalFunctions can measure an observable behavior of the signal by producing values. Conceptually, control attributes help define the signal whereas value attributes provide measurable aspects of the signal. Put another way, control attributes represent inputs to the SignalFunctions whereas value attributes represent outputs from SignalFunctions. All attributes are heavily typed using the same basic types.

Figure 6 is used to illustrate a generic SignalFunction component. The concept of a SignalFunction is the key reusable component within the system. The SignalFunction provides the signal interface that transfers a signal, its attributes, and its values to other components. The SignalFunction has a reference to an input signal (In). The SignalFunction operates on the input signal and produces an output signal (Out). The behavior may also be through events (Sync and Gate). In addition, the SignalFunction has a collection of attributes that affect its behavior. An STD model depicts the passing of parameters to other components pictorially as lines.

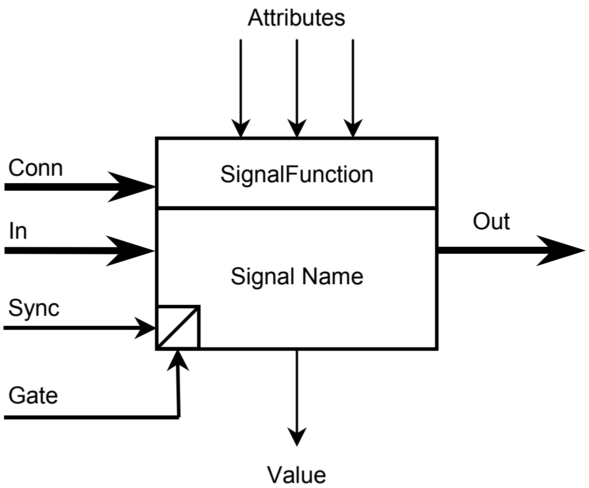


Figure 6—SignalFunction diagram

The fundamental SignalFunctions are BSCs and have formal definitions that mathematically define both the function and behavior. The concept of the STD standard is that a network of these SignalFunctions can be constructed based on a consistent set of rules of composition. These rules and the SignalFunction formal definitions allow the building of complex SignalFunctions, which are formally defined via their base components. Using these SignalFunctions and attributes, a model may be created that defines the complete signal (see Figure 7).

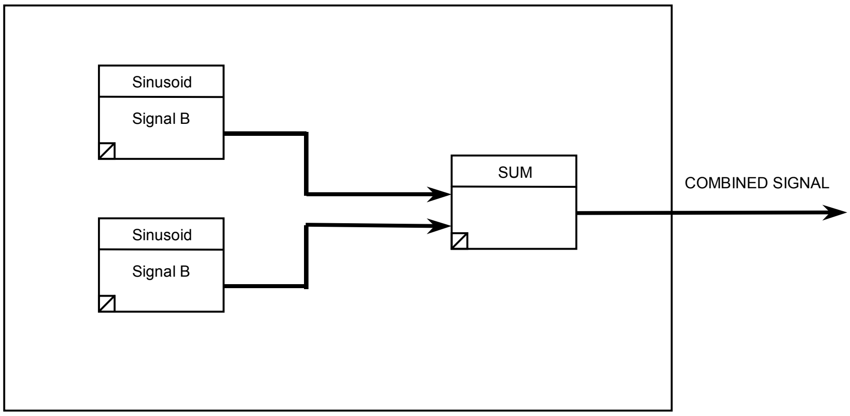


Figure 7—Simple signal summing two sinusoidal waveforms

By assigning the reference input signal to another SignalFunction output, it is possible to build up complex signal descriptions. These signal definitions become available to the application through the high-level signal model interfaces.

The STD standard deals directly with the concept of SignalFunctions and their interaction and defines the allowed set of attributes and values for each SignalFunction and their allowed types. It is the role of the carrier language program to deal with the values of every attribute and interpret the values correctly, including performing any analysis, computation, or evaluating expressions on the attributes.

In practice, some test requirements may not be fully defined without the need of some carrier language algorithms, in which case the test requirements may be nondeterministic across different language platforms. In these cases, it is important to realize that the signal descriptions will have fully deterministic behavior that is ensured by strict rules concerning event concurrency and state updates and that the nondeterministic nature lies with the different implementations of carrier languages, such as speed or variable precision.

SignalFunctions may be defined in several ways, including XML definitions, TSFs, or a carrier language utilizing the Interface Definition Language (IDL). When using a carrier language with IDL, each instance of a SignalFunction is created by a supplied application, but the SignalFunction's behavior and interface are defined through the STD standard. The purpose of the carrier language is to name and request the creation of all the necessary SignalFunctions, interconnect all of the SignalFunctions, set the attributes of the SignalFunctions, initiate the top-level sequencing (thereby, in effect, pressing the GO button), and interpreting the results.

Test specifications are built using static (functional specification) descriptions or dynamic (algorithmic) descriptions. This process necessitates that the carrier language program perform two tasks:

a) Modeling, i.e. providing a static description of the SignalFunctions and their interconnections.

b) Control, i.e. providing a dynamic use of the model by changing attributes and interpreting values.

5.2 Physical types

The STD standard defines a whole collection of quantity types, under a base type called Physical. These are the general engineering terms such as Time, Voltage, Resistance, and Power as well as some others that are less well used such as Amount of Information, Luminescence, Pressure, Heat, etc.

They are all equally valid, although the examples used in this guide are predominantly based on electrical signals with the default dependent (type) being Voltage and the independent (reftype) being Time. In general, any Physical type may be used.

The STD standard also defines the base physical types that can be used to define any quantity in terms of a combination of unit symbols, e.g., kB/s. The definition of these physical quantities is outside the scope of the STD standard. However, when used, their definition shall match the appropriate SI[5] or IEEE standard definition.

All physical types may take engineering unit prefixes as illustrated in the examples: pF, nm, µW, mA, cm, kg, and MHz.

When a Physical quantity is defined, e.g., 2 V, the following can also be used:

— A qualifier, such as inst, pk, trms, pk_pk, av

— A level of uncertainty, introduced by errlmt, ±, etc

— A range over which the quantity may change

If this additional information is not provided, then the interpretation is that these values are unimportant. In other words, if no uncertainty value is defined, any accuracy or stability will do.

Examples:

[5] Système International d'Unités (International System of Units).

— "25 °C" (or "25 degC"), meaning that on average the temperature should be 25 °C

— "trms 2 V ± 5% range MAX 10 V," meaning that the true root-mean-square (trms) value must be within 5% of 2 V

— "pk 2 W errlmt 100 mW range 1 W to 3 W," meaning that the peak power must be within 100 mW of 2 W

The range value in the last two examples indicates the extremes that the attribute value may take. Therefore, for a control attribute, it will never be programmed outside these values, and for a measurement, the value will be within the range.

It is strongly recommended that a level of uncertainty is provided, because the person specifying the value is in the best position to define a suitable value. If this is not done, it is left to the implementer to guess what would be a suitable value for the uncertainty, and it would be possible, if not sensible, to choose a significantly different value. For example, if a 12 V supply is specified without an errlmt, what would be an acceptable value; 12.000 V, 12.1 V, 12.5 V, 13 V, 18 V,...?

Within engineering, there are several terms where it is more convenient to express the value using a different set of units. The STD standard supports this practice by allowing mappings between different units onto the defined quantity. In all cases, the quantity, regardless of how it is defined, represents the same single value; therefore, defining "32 °F" or "0 °C" defines the same value.

Some quantities provide mappings between units, e.g., a frequency value of 1 MHz is identical to a value of 1 µs because frequency has a mapping of 1/Time. In some cases, there are relationships between mapped quantities that require additional information. An example of this situation is Power and Voltage where a value for the load resistance is required.

Only the mappings defined in the STD standard should be used because abuse of this feature makes signal verification difficult and masks user mistakes within signal definitions.

— Frequency = 1/Time

— Time = 1/Frequency

— Power = Voltage × Current

— Power = Voltage × Voltage / Resistance

— Power = Current × Current × Resistance

— Voltage = Current × Resistance

5.3 Signals

5.3.1 General

Electrical engineers often visualize signals as Voltage *vs*. Time; i.e., a signal is always some voltage changing in time. For most electrical signals, this view is reasonable. The question that should be asked is the following:

Are these types the best way to describe the required signal?

The STD standard allows a signal to be defined in any combination of Physical types. A signal identifies the relationship between the dependent variable (type) and independent variable (reftype). The concept is that any signal exhibiting the specified relationship will be adequate. The implication of this concept is that there may be many waveforms that will meet the STD definition of the signal. This implication falls in line with the intent of the standard:

a) Do not over-constrain a requirement.

b) Do not pin down a signal definition to a particular implementation, instrument, or resource.

A simple dc voltage, something that may be expressed as 5 V dc, would be described (using XML) as

```
<Constant amplitude="5 V" />
```

or more fully

```
<Constant type="Voltage" reftype="Time" amplitude="5 V" />
```

This example defines a signal whose relationship between Voltage and Time is such that the voltage amplitude is a constant 5 V. It is illustrated in Figure 8.

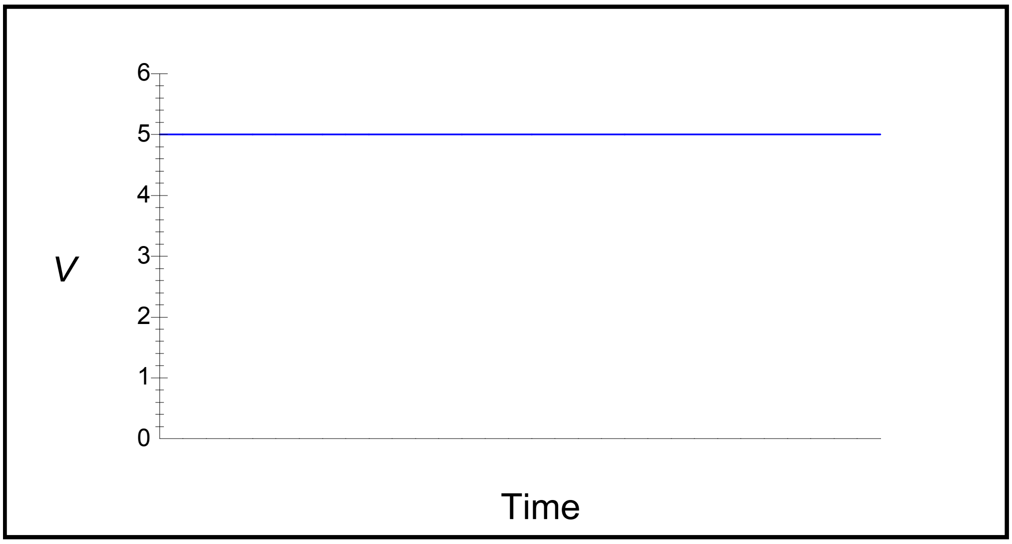


Figure 8—Constant (Voltage, Time) signal

5.3.2 Effect of qualifiers on signal definition

The definition could be expressed in a different way. The following definition could be used to describe the identical signal:

```
<Constant amplitude="trms 5 V" />
```

In both cases, the definitions describe a constant voltage with respect to time where the (constant) value of the amplitude of the voltage is expressed as trms 5 V, rather than a constant trms voltage with respect to time.

It is important to remember that using a qualifier does not change the type of the signal being defined. It just allows a quantity to be expressed in a different form. The meaning of qualifiers when used with a stimulus signal aligns with the definition of the corresponding measurement BSC. For example, if a signal is described thus:

```
<Sinusoid type="Voltage" amplitude="pk_pk 5 V"frequency="200 kHz" />
```

a measurement of the signal using the PeaktoPeak BSC would produce a value of 5 V.

The qualifiers available are shown in Table 1 and are listed with the corresponding Sensor BSC.

Table 1—Qualifiers and their meaning

Qualifier	Meaning	Corresponding Sensor BSC
trms	true root mean square	RMS
pk_pk	Peak to peak	PeakToPeak
pk	peak	Peak
pk_pos	positive peak	PeakPos
pk_neg	negative peak	PeakNeg
av	average	Average
inst	instantaneous	Instantaneous
inst_max	instantaneous maximum value	MaxInstantaneous
inst_min	instantaneous minimum value	MinInstantaneous

Consider the following signal:

```
<Sinusoid type="Voltage" reftype="Time" frequency="1 MHz ±0.1 Hz"
amplitude="trms 2 V" />
```

This example defines a signal whose relationship between Voltage and Time (the default types) is a sinusoid where the frequency is 1 MHz (with an accuracy of ± 0.1 Hz) and the trms value of its amplitude is 2 V. Therefore, the amplitude pk is 2.8284 V (2√2), the pk_pk value is 5.6569 V (4√2), etc. The phase of the signal starts at 0°, the default value of the phase attribute. This signal is illustrated in Figure 9.

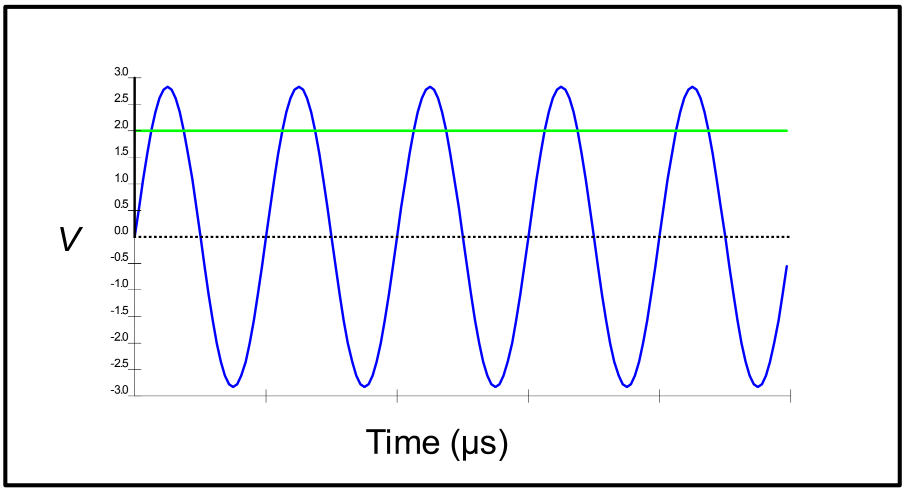


Figure 9—Sinusoid (Voltage, Time) signal

5.3.3 Phase information within signal

This example may be extended to illustrate how phase is handled by the STD standard. The signal was specified with a defined phase. If two such sinusoid signals with different phase attributes were to be summed, the resulting signal would be defined as follows:

```
<Sum In="s1 s2" />
<Sinusoid name="s1" frequency="1 MHz" amplitude="1 V" />
<Sinusoid name="s2" frequency="3 MHz" amplitude="1 V" phase="45 deg" />
```

Phase is relative between signals within the same time frame; therefore, the second sinusoid signal (s2) has an initial phase angle of 45° different from the first signal (s1). This case is illustrated in Figure 10, which shows the two signals, one combined in phase and the other with a 45° phase shift.

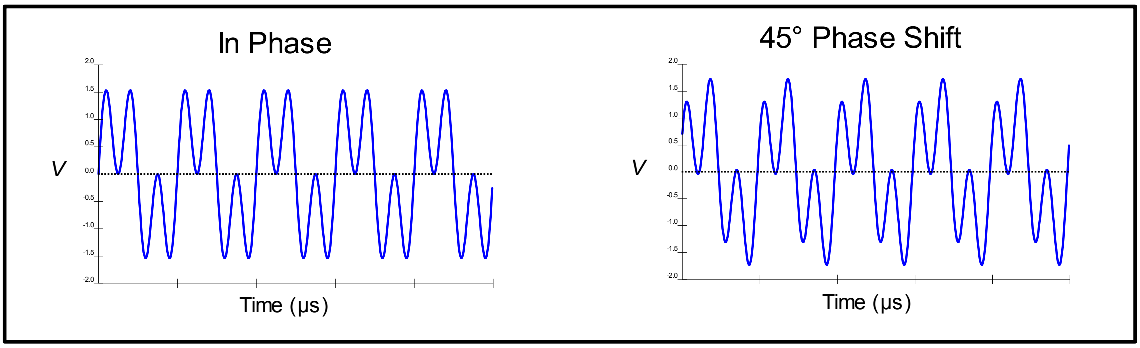


Figure 10—Example showing phase relationship on a signal

A signal may pass through different time frames. For example, when a signal goes through any derived STD connection object, the phase information of the time frame is lost. In other words, the relative phase of the signal is maintained, but the specific phase of the time frame is not guaranteed. A consequence of this situation is that if a simple single sinusoid is applied to the UUT, defining the signal as sinusoid (0° initial phase angle) or cosine (90° initial phase angle) has little effect because at the connection the waveform that emerges is not guaranteed to start at the time t0. Using the signal example above, both waveforms in Figure 11 are valid at the UUT.

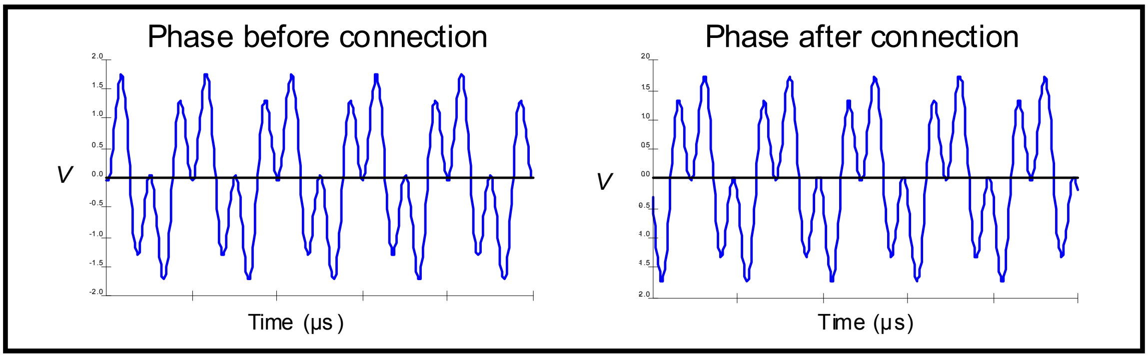


Figure 11—Effect of connection: both waveforms valid for signal

5.3.4 Signals with insufficient information

Consider signals that are expressed in terms of Power.

```
<Sinusoid type="Voltage" frequency="1 MHz" amplitude="trms 2 mW" />
```

This example defines a signal whose relationship between Voltage and Time is a sinusoid where the voltage amplitude of the sinusoid is such that the trms power (voltage × current) is 2 mW. Because it is unlikely that the signal can affect the impedance of the system, it will need to adjust the voltage, or know in advance what the resistance load is, in order to achieve this goal. The signal definition, however, does not change.

```
<Sinusoid type="Voltage" frequency="1 MHz" amplitude="inst_max 2 μW" />
```

This example defines a signal whose relationship between Voltage and Time is a sinusoid where the voltage amplitude of the sinusoid is such that the inst_max power (voltage × current) is 2 μW. If an impedance of 50 Ω is assumed, the maximum instantaneous power of 2 μW would indicate that a voltage amplitude of the signal at that instant would be √(2 μW × 50 Ω), or 10 mV. Figure 12 shows the waveforms for instantaneous voltage and power.

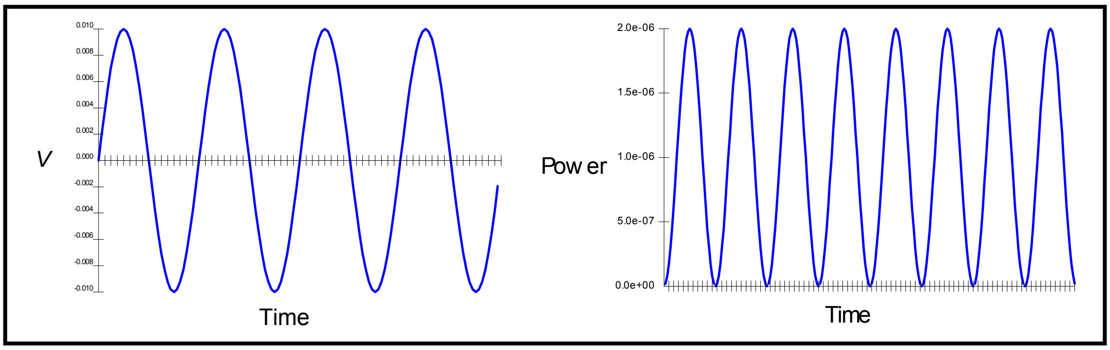


Figure 12—Relationship between instantaneous voltage and power

The above examples show that either the signal voltage has to be dynamically adjusted to arrive at the required power or the impedance must be known in advance of creating the signal. It should be noted that it may be difficult or even impossible to achieve the specified signal, e.g., if the impedance is not a linear or open circuit. Specification of stimulus signals that rely on the characteristics of the UUT for their implementation is not recommended.

All of the following describe the same signal (assuming an impedance of 50 Ω).

```
<Sinusoid frequency="1 MHz" amplitude="inst_max 2 μW" />
<Sinusoid frequency="1 MHz" amplitude="pk_pk 2 μW" />
<Sinusoid frequency="1 MHz" amplitude="pk 1 μW" />
<Sinusoid frequency="1 MHz" amplitude="trms 1.2247 μW" />
<Sinusoid frequency="1 MHz" amplitude="inst_max 10 mV" />
<Sinusoid frequency="1 MHz" amplitude="inst_min -10 mV" />
<Sinusoid frequency="1 MHz" amplitude="pk_pk 20 mV" />
<Sinusoid frequency="1 MHz" amplitude="pk 10 mV" />
<Sinusoid frequency="1 MHz" amplitude="pk_neg -10 mV" />
<Sinusoid frequency="1 MHz" amplitude="trms 7.0711 mV" />
<Sinusoid frequency="1 MHz" amplitude="10 mV" />
```

Consider this example in which the signal is defined in terms of Power.

```
<Constant type="Power" reftype="Frequency" amplitude="0.1 uW" />
```

This example defines a signal whose relationship between Power and Frequency is constant, i.e., at all frequencies, a power of 0.1 μW exists. This example is a description of white noise. In this example, the key properties that are required have been defined, but not the method of turning it into a traditional voltage-time waveform. This ability represents a major strength when defining signals because in reality there may be many types of waveforms that will do the job and it must be possible to include them all.

5.4 Using signal graphs to create a signal

5.4.1 Introduction

BSCs are specified to be the basic building blocks required to create any signal. A TSF model is a higher level grouping of building blocks designed to create reusable blocks of signal functionality or measurement definition, as an arrangement of BSCs and/or TSFs, connected together to form a signal. A TSF model is also known as a signal graph or signal model.

There are a number of different ways in which to express signal graphs. They may be shown graphically, which is useful for visualization; XML provides the benefits of a mark-up language, such as ready parsing; and IDL provides an interface to software objects implementing signal control functionality. In all cases, the functional SML underpins all BSCs to provide consistent definition across all implementations.

5.4.2 Sources and conditioners

Sources are BSCs that create a signal according to the values of the attributes associated with it. Conditioners are BSCs that take one or more input signals and output a signal depending upon the relationship of the input signals and/or the value of the associated attributes.

Most BSCs have attributes, which are used to configure the behavior of that component. Examples of attributes are amplitude, frequency, phase, etc.

Attributes have default values that simplify their use should the designer not need to manipulate that attribute. For example, the design in Figure 13 shows sinusoid components where the phase attribute has been ignored. In this case, the result is no phase offset. Figure 13 demonstrates an additional format to present the same information. This format is also used in Figure 14, Figure 15, and Figure 16 in 5.4.3, 5.4.4, and 5.4.5, respectively.

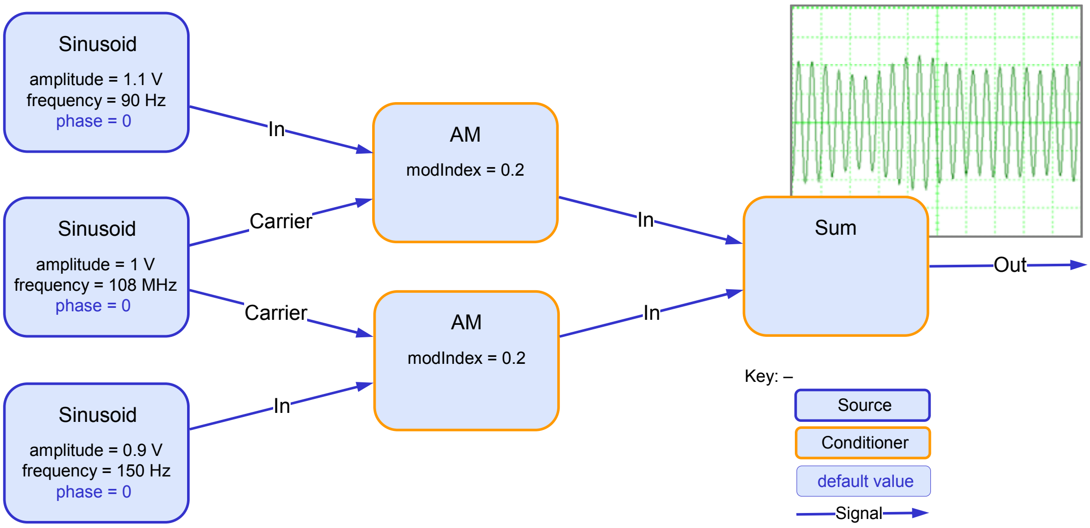


Figure 13—Sources and conditioners

The In connection is used to indicate that the signal generated by one BSC Out connection should be passed to another. This statement is true for almost all BSCs. An exception to this rule is made for the case of the carrier input to the amplitude modulation (AM) BSC, where a distinction between two different In connections must be made. Usually, as for the case of the sum conditioner, no such distinction is required.

The signal graph shown in Figure 13 represents an instrument landing system (ILS) signal. One sinusoid BSC provides a carrier signal into the two AM BSCs. The other two sinusoids provide the modulation signals for the two AM BSCs. This setup results in two separate AM signals with the same carrier frequency but with different modulation signals. These two signals are then added in the sum BSC to provide the composite signal.

5.4.3 Events and timing

Much like signal sources and conditioners, event BSCs provide and condition event signals. Events can be used to synchronize or gate signals.

— Synchronizing a signal restarts the BSC or TSF, e.g., time becomes zero.

— Gating a signal turns on the operation, i.e., for a source or conditioner, the output is on, and for a measurement, the measurement is performed over the gate.

In addition to default values, the Clock event source provides an example of a default unit. In this case, if no unit is specified, hertz is presumed. In the case of the clock BSC, it is possible to specify clockRate in seconds or hertz. The component model will automatically convert one unit to the other.

The signal graph shown in Figure 14 represents random bursts of a decaying sinusoid. The sinusoid Out signal is modified by the exponential conditioner, which will produce an exponentially decaying sinusoid. The effect of the Sync signal from the clock will be to restart the exponential each time the Sync event occurs, resulting in a series of decaying sinusoids. Notice also that the clock provides a signal into the ProbabilityEvent (labeled ProbabilityBasedEvent in Figure 14); this results in an output from the ProbabilityEvent for 50% of the inputs. The Gate signal from the ProbabilityEvent is used to gate the exponential BSC on and off, resulting in the random bursts of a decaying sinusoid.

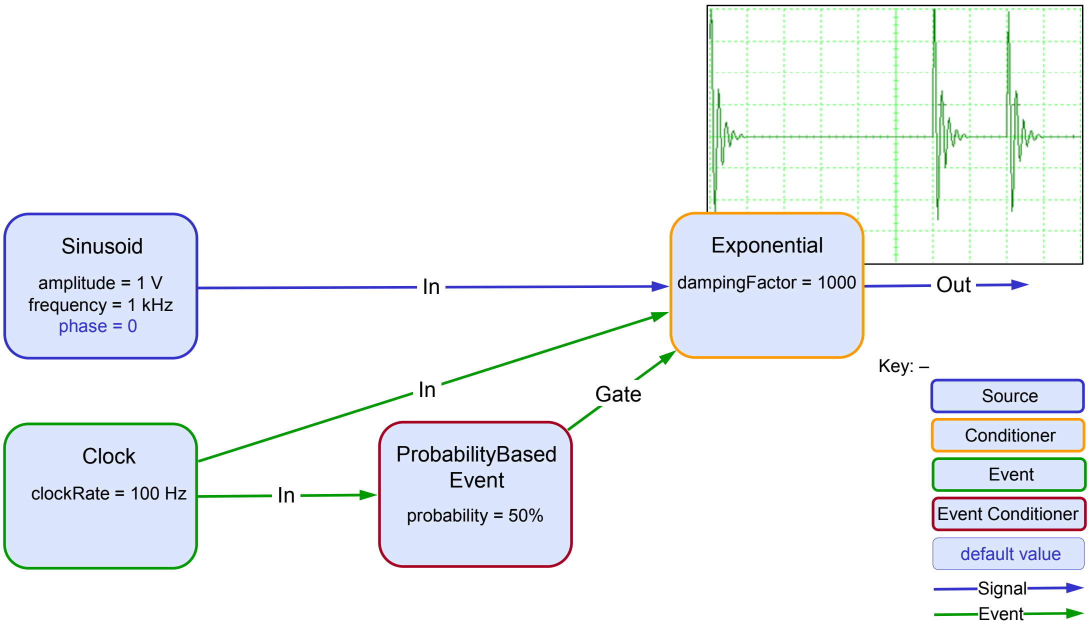


Figure 14—Events

5.4.4 Sensors and measurements

Sensors perform combinations of three main functions:

a) A monitor function for raising an event when a particular condition is met

b) Measurements of a required value or values of a signal

c) Comparison of the measurement with upper and lower limits to yield a pass/fail result

Figure 15 shows a sinusoid being measured by a sensor BSC that provides the root-mean-square (rms) value of the input signal. The signal from the SingleRamp BSC is monitored by a sensor BSC that will register the maximum instantaneous value of the ramp signal. In addition, as the instantaneous value reaches 0.9 V, the monitor function will raise an event that is used to gate the sinusoid BSC on.

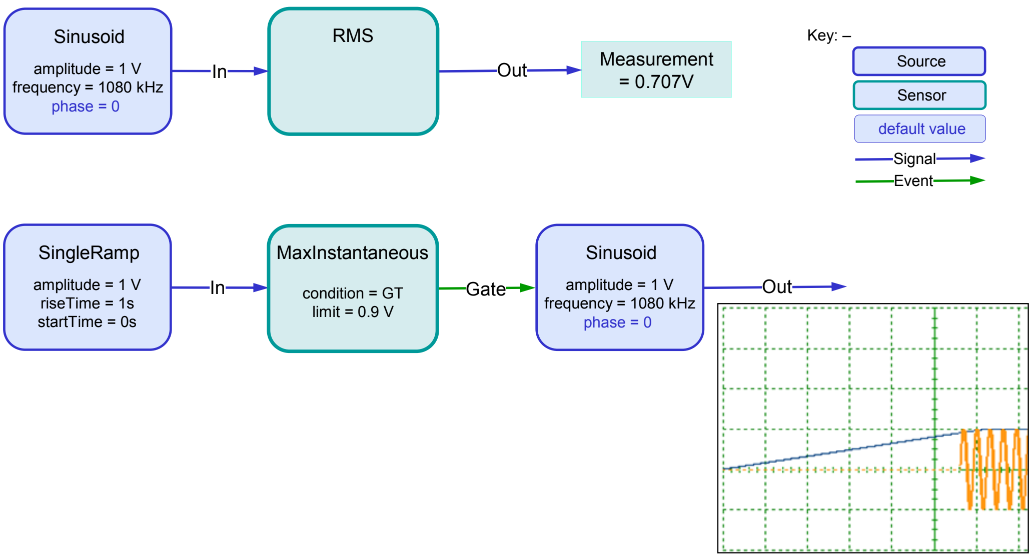


Figure 15—Measurements

When taking a measurement, the attribute value of gateTime has the following effect:

a) +ve value: the gateTime is the specified value in time (seconds).

b) zero: the measurement resource determines the required time.

c) -ve value: the gateTime is determined by the gate input.

Prior to any measurement, the signal can be conditioned, e.g., by filtering the signal or demodulating the signal. Conditioning allows measurements to be made on processed signals.

Parallel measurements can be defined by using sensors in parallel. As an example to measure both rms and instantaneous frequency, it is needed only to add sensors in parallel, for both rms and instantaneous (frequency).

As an alternative measurement method, the STD standard provides the generic measure sensor. This method allows multiple TSF or BSC attributes to be measured simultaneously. The measurement may represent a multidimensional series of results that can be further analyzed.

Example:

```
<Measure As="Sinusoid" attribute="amplitude phase" />
```

This example provides the simultaneous amplitude and phase measurement of the input signal. The use of user-defined TSFs and the generic measure sensor allows a wide range of measurements to be defined by the standard.

5.4.5 Connections and locations

Connections provide the method to map signals from signal graphs to real pins on a real UUT.

The STD standard provides a further group of BSCs called connections. Connections are used to define a point in the signal graph where the signal is required to appear in the real world, typically, as an electrical signal. The attributes of a connection are primarily pin names used to describe a physical pin or location, i.e., what would commonly be the UUT (see Figure 16).

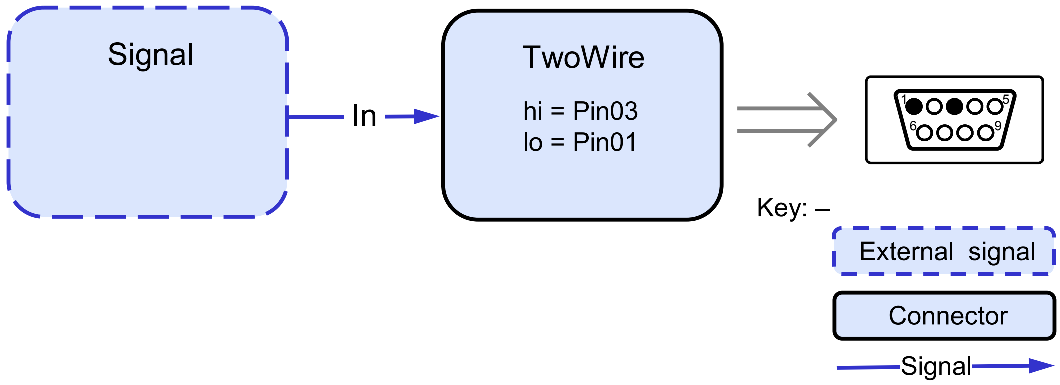


Figure 16—Connections

An additional aspect of using a connector is that any implied phase or synchronous information with respect to t = 0 is lost across the connector. Phase and synchronism between signal components is maintained. This aspect allows TSFs to define signals that have ambiguous phase or synchronous characteristics without over-constraining a solution. As an example, consider a pulsed ac signal where the phase of the ac signal to that of the pulse may be unknown. By separating the components by a connector (with no pin information), this ambiguity is resolved.

5.5 Documenting signal descriptions

The STD standard is an interface standard that identifies how to model signals. It is not a computer language. Remember that the TPL describes only signals and requires the support of a carrier language.

In order to allow consistent access to signal definitions across multiple languages, it is necessary to describe the STD classes using a language-neutral method. This approach allows each independent language to define how it accesses STD classes. In effect, each language reads the language-neutral description and coverts it into something it can use (e.g., in the case of C++, a header file is used containing client classes). The medium chosen for the definition of the signal definitions is the IDL, which is defined in the Distributed Computer Environment (DCE) Specifications [B2] controlled by The Open Group.

The STD standard provides interface definitions detailing the names of all the properties that the STD classes possess. In addition, the standard provides the semantics of what each property does. The STD standard also defines the rules of use and composition between these elements and effectively defines interfaces to non-signal types that are used by signals.

5.5.1 BSC definitions

The STD standard defines a set of fundamental SignalFunctions (i.e., the BSCs). These fundamental classes represent the elementary components available in STD and are used to define all higher level signals. For each BSC, there exists a description in English and an SML definition to form the basis of a formal description. The SML definitions underwrite the BSCs in a rigorous mathematical way.

As well as defining the BSCs, the STD standard contains higher level signal models defined in a TSF library. The TSF library provided in Annex E of IEEE Std 1641-2010 defines examples of signals derived from the existing C/ATLAS standard [B9].

Clause 10 of this guide provides a more detailed description of the BSCs.

5.5.2 Defining TSFs

TSF models are built up using the STD BSCs and by reusing existing TSF models. The TSF models represent static model descriptions with an interface that represents its attributes. A TSF model has an interface description and a static "network list" description. A TSF class shall be defined with a valid XML TSF element utilizing the attribute model and description elements. These may also be shown in a diagram and as tables. A TSF class model shall be described only as a static signal model and may contain elements from the following:

— BSCs

— Other TSF classes

— Reference to the TSF class being defined

A TSF IDL library shall be defined in terms of an IDL library module, will contain an entry for each TSF class within the library domain, and shall be derived from the XML TSF library. No SML definition is required because the TSF classes are constructed only from STD BSCs or other TSF models. Because the fundamental classes (BSCs) are formalized and traceable to SML definitions, the higher level signals are also defined.

The use of TSFs allows users to define their own signals and then use them in the same way as any of the BSCs. The STD standard does not mandate the use of TSFs; in fact, applications concentrate on the underlying capability of BSCs. TSFs provide a very powerful tool to define and support legacy signals as well as provide a vehicle for capturing domain specific signals.

It should be noted that the signal model attributed to a TSF only identifies that the TSF model can be defined; it does not mandate any particular implementation. This feature allows TSF implementations to be optimized and enables different TSF definitions to be compared.

The key use of TSF models is that, once defined, they provide an ideal vehicle through which to describe resource capability. The strengths of this approach are that resources can be configured for specific signal domains and users can match test requirement to capability. The downside is that each resource's capability needs to be defined in terms of the TSF models it can support. In traditional instruments, this offers a practical solution; however, when considering synthetic instruments or looking at combining instruments into a complex resource, this approach requires that all the TSF models are known and mapped into resource capability.

5.5.2.1 TSF structure

A TSF comprises two parts; an interface description and a model description. Basically, the interface description provides the information that allows the TSF to connect to other TSFs and a UUT, and the model description defines its internal operation. Although the formal method of defining TSFs requires the use of XML, it is simpler to explain how an interface is described and a model is defined using tables and a graphical representation. Commercial tools that assist in the creation of TSFs may use graphical methods and then automatically create the XML definition.

5.5.2.2 Interface properties

The different types of property for a TSF model include the following:

— Control

— Measurement

— Input

— Output

— Expression or formula

The interface must include all inputs for the model as shown in Table 2.

Table 2—Interface properties

Description	Name	Type	Default	Range
Attribute 1	attr_name1	Physical		
Attribute 2	attr_name2	Frequency		
Attribute 3	attr_name3	Ratio		
Input Signal	input_Signal	SignalFunction		

This interface properties table provides the following information:

— Description is a short, helpful description describing the attribute.

— Name is the attribute's name with initial lowercase letter and no spaces. This information becomes the name of the attribute in the interface.

— Type is the primary type, such as a signal type, a Physical type, or a basic data type. An attribute can be of only one type, which may be of type Variant. Associated with the type may be a direction relative to the SignalFunction. Types have initial uppercase letter and no spaces.

— Default is the value given to the attribute if no value is provided by the user.

— Range is a constraint that limits the range of the values for an attribute.

The output signal and the sync and gate inputs are not shown in the interface properties table. This omission does not mean that they are not present.

Therefore, for a simple signal like an ac signal, the interface properties table could be as shown in Table 3.

Table 3—Simple interface (AC_SIGNAL)

Description	Name	Type	Default	Range
AC Signal amplitude	ac_ampl	Physical		
AC Signal frequency	freq	Frequency		

In a TSF interface properties table, the lack of a default entry indicates that the user is expected to provide a value. Normal engineering considerations require that, for an ac signal, the user will (as a minimum) state the amplitude and frequency of the signal. If the values are not provided, the default attributes of the supporting BSC will be assumed. Omitting these parameters from a signal is bad practice, as it is then not clear at a glance what the values are intended to be. Users are not expected to remember (or care about) the default attributes of a BSC.

5.5.2.3 Model description

Each TSF model has a signal model description in the form of a network list describing the interconnections. The model description table has the form shown in Table 4.

Table 4—Model

Name	Type	Terminal	Inputs	Output	Formula
BSC Name	BSC Class	Signal [Out]		OUTPUT_SIGNAL	
		bsc_attr_1	attr_name1		
		bsc_attr_2	attr_name1		
		bsc_attr_3	attr_name1		
		bsc_attr_4			0

where

Name — is a short, helpful description identifying the specific instance of the BSC.

Type — is the class or subclass of the BSC in the model.

Terminal — is the name of the attribute or signal of the BSC.

Input — is the name of the input from the interface properties table.

Output — is the name of the BSC output signal.

Formula — provides information about the attribute, such as how the value is fixed or a calculated value if the attribute does not have an entry in the interface properties table.

For the same simple ac signal the model description table could be as shown in Table 5.

Table 5—Simple model (AC_SIGNAL)

Name	Type	Terminal	Inputs	Output	Formula
AC Signal	Sinusoid	Signal [Out]		AC_SIGNAL	
		amplitude	ac_ampl		
		frequency	freq		
		phase			0 rad

The name of the TSF model provides the class of the SignalFunction it describes. For multi-component models, the Sync event input is generally connected to the last SignalFunction Sync event.

5.5.2.4 Graphical presentation

Each TSF model diagram shows the network list in a graphical presentation where the TSF is contained within an interface box. Each component is placed within the interface box, and its class and name identified. The interconnections between signals and between attributes are shown as lines with arrows.

The inputs and outputs to the TSF interface box are the same as for an individual BSC as shown in Figure 17.

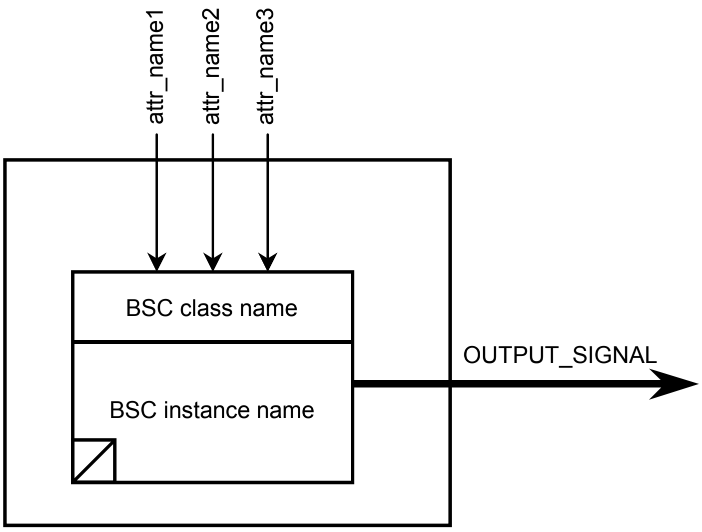


Figure 17 —Generic TSF model diagram

The TSF model diagram for the simple example of an ac signal would be as shown in Figure 18.

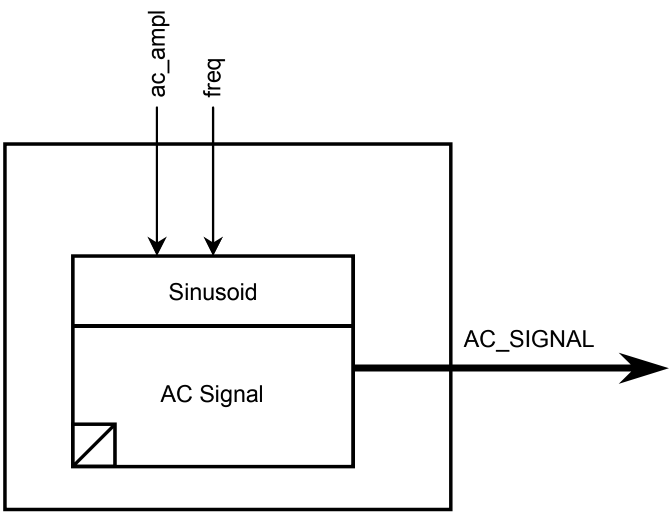


Figure 18 —Simple TSF model (AC_SIGNAL)

Clause 10 of this guide provides a more detailed description of the creation and use of TSF models together with examples.

5.5.3 Interfacing STD signal descriptions to carrier language

The STD standard allows almost any language to be used as the carrier language for a signal definition environment. To achieve this feature, the signal definitions have to be made available to the carrier language in a form that is compatible with the carrier language. As previously discussed, the STD standard provides a description of the SignalFunctions, together with their controllable properties, in a language-neutral form, which can be mapped into carrier[6] language environments. This language-neutral form (IDL) allows computer language environments (such as C/C++ or Basic) to generate their own header and input files and allows access to these signal building blocks. Some language development environments are powerful enough that they can use the IDL description directly, extend their programming features, and eliminate the need for additional interfacing files.

The STD standard allows the use of any of the commercial off-the-shelf (COTS) languages to define common signal descriptions for UUT testing. These signal descriptions, together with the programmatic information, become the UUT test requirements (see Figure 19).

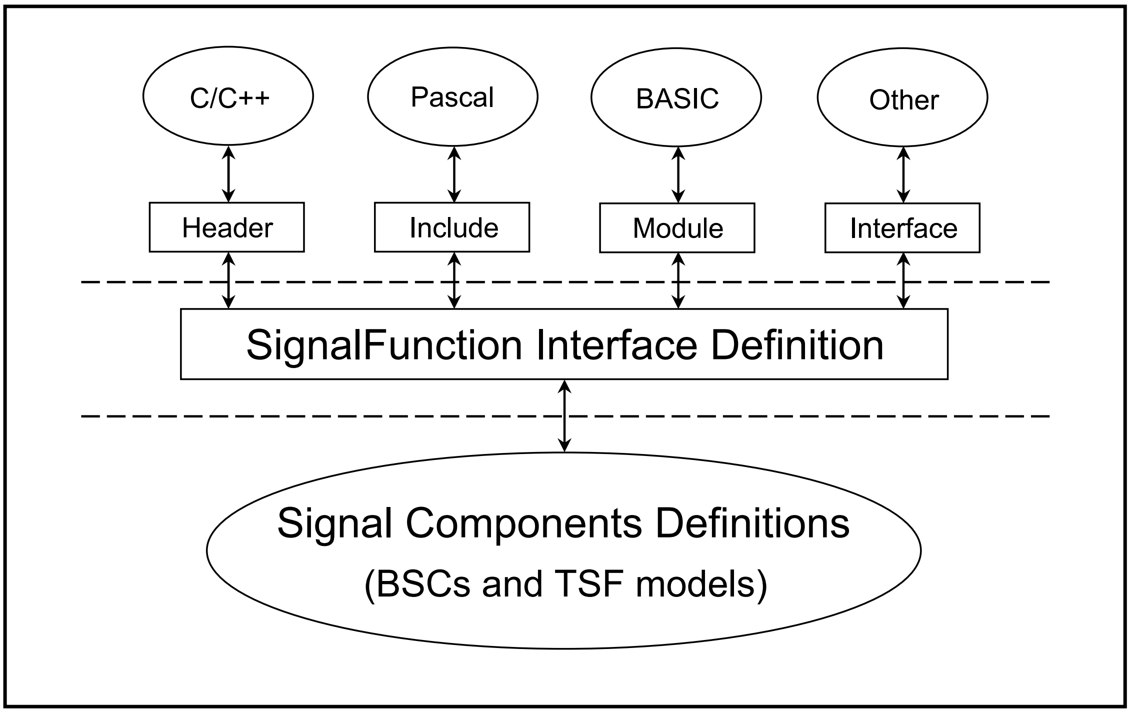


Figure 19—Multiple languages using common signal component definitions

[6] There are certain criteria a carrier language must meet to support this feature.

5.6 Using signal definitions

When defining signals, it is important to consider that they will ultimately have to be implemented practically. That does not mean that the detailed implementation has to be described or even implied. It is important that the signal or test remains implementation independent. But, whatever is described will have to be run on a test system, and maybe several different types of test system at different maintenance levels. The simplest way of describing a signal is usually the best. Designing a very clever and complicated signal is of no use when it cannot be implemented using available test instrumentation.

In many cases instruments may be described using the same elements as used for signal definitions, which means that aligning or allocating real instruments with signal definitions can be an automatic process. But even if manual intervention is required to ensure that the correct instrumentation is used, the signal definitions will leave no doubt in what is required, and the implementer's task is therefore straightforward.

6. Signal models

6.1 General

One of the primary use cases for the STD standard is to capture test requirements and create test programs. The STD standard has a variety of methods that could be used for this task, and initially it can be daunting to identify a process using the STD standard. This clause introduces some best practices that can be followed for this activity, and users can refine these practices to fulfill their individual requirements.

A key feature of the STD standard is the ability to create user signal components and maintain them within a library environment. Within the STD standard, these signal components are called TSF models, and a collection of TSF models used for a particular application, technology, or project is a TSF library. TSF models can be used to support stimulus signals, events, measurements, and the processing of existing signals.

It is also possible to define a complete test, including all the timing and location information, within a single template signal or TSF model.

6.2 TSF model as signal template

A TSF model may be considered as a reusable signal template with parameters, signal inputs, and signal outputs. The parameters are known as attributes and will be one of the types described in 6.2.1. The signal model (template) may be reused by providing different values for the attributes.

6.2.1 Types of TSF attributes

The following attributes can be found in a TSF model:

a) Control attributes. A control attribute is mapped onto a model property. It provides information about the characteristics of a signal.

   Example: voltage 10 V. The resource provides a signal with a voltage of 10 V.

b) Capability attributes. An attribute that is not mapped onto any signal model is regarded as a capability attribute and may be thought of as a requirement attribute. It holds information about the capability of the resource required to supply the signal

Example: current 2 A. The resource must be capable of supplying a signal that may draw up to 2 A.

c) Value attributes. A value attribute represents a value obtained from the signal model. These attributes are normally read¬ only.

Example: when an interface attribute is mapped onto a model measurement attribute. The resource must be capable of providing measurement information about a signal, such as the amplitude of the signal.

d) Expression or formula attributes. An expression or formula attribute is mapped to or from a model property via an expression.

Examples:

—Type conversion, such as “period = 1/freq”

—Measurement analysis, such as “mode = HistMode (Measurements)”

—Text manipulation

6.2.2 Creating a TSF

These TSF models may be defined in XML as shown in the following examples:

a) Provide the name of the library component.

```
<TSF name="AC_SIGNAL" uuid=...>
```

b) Specify TSF base type.

```
<xs:extension base=“SignalFunctionType”/>
```

c) Define attributes in the <interface> section.

```
<xs:attribute name="ac_ampl" type="Physical">
```

d) Specify any default attribute values.

```
default="0 rad"
```

e) Specify any restrictions to attribute values.

```
<xs:maxInclusive value="2*pi rad"/>
```

f) Define the behavior model in the <model> section.

```
<Signal Out="AC_Signal" …>
   <Constant name="DC_Offset" amplitude="dc_offset"/>
   <Sinusoid name="AC_Component" amplitude="ac_ampl"
                                 frequency="freq"
                                 phase="phase"/>
   <Sum name="AC_Signal" In="DC_Offset AC_Component"/>
</Signal>
```

g) Specify any attribute formula.

```
(period="1/freq")
```

h) Assign value attributes formula.
`(default="rms.measurement/carrier.amplitude")`

Many of the above examples are shown in the following XML description of an AC_SIGNAL.

```
<TSF name="AC_SIGNAL" uuid="{CA4F7FD8-D05E-11D6-860D-00010214C4D2}">
   <interface>
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 elementFormDefault="qualified">
         <xs:element name="AC_SIGNAL">
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType">
                     <xs:attribute name="ac_ampl" type="Physical">
                     </xs:attribute>
                     <xs:attribute name="dc_offset" type="Physical" default="0">
                     </xs:attribute>
                     <xs:attribute name="freq" type="Frequency">
                     </xs:attribute>
                     <xs:attribute name="phase" default="0 rad">
                        <xs:simpleType>
                           <xs:restriction base="PlaneAngle">
                              <xs:minInclusive value="0"/>
                              <xs:maxInclusive value="2*pi rad"/>
                           </xs:restriction>
                        </xs:simpleType>
                     </xs:attribute>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
      </xs:schema>
   </interface>
   <model>
      <Signal Out="AC_Signal" xmlns="STDBSC" >
         <Constant name="DC_Offset" amplitude="dc_offset"/>
         <Sinusoid name="AC_Component" amplitude="ac_ampl"
                                       frequency="freq"
                                       phase="phase"/>
         <Sum name="AC_Signal" In="DC_Offset AC_Component"/>
      </Signal>
   </model>
</TSF>
```

Additional tools may be required to deal with formulae within a signal model:

— Formula translations allow TSF models to manipulate attribute values.

— Formula translations may be used to specify measurement analysis.

— The formulae formats are normally in Haskell 98 [B15], which may not be the most suitable form. This situation leads to a requirement for formula support for other unspecified formulae and for other analysis tools, and this requirement should be documented in the TSF library documentation.

Further details about the creation and use of TSF models are provided in Clause 10.

6.3 Typical signal model (TSF)

Figure 20 represents a model for a typical TSF model. The example used is for a SYNCHRO, which is described in more detail in 10.7. This model shows the input parameters (attributes) at the top and the signal output on the right. The attributes shown are control attributes. Figure 21 shows the output waveforms derived from the TSF model.

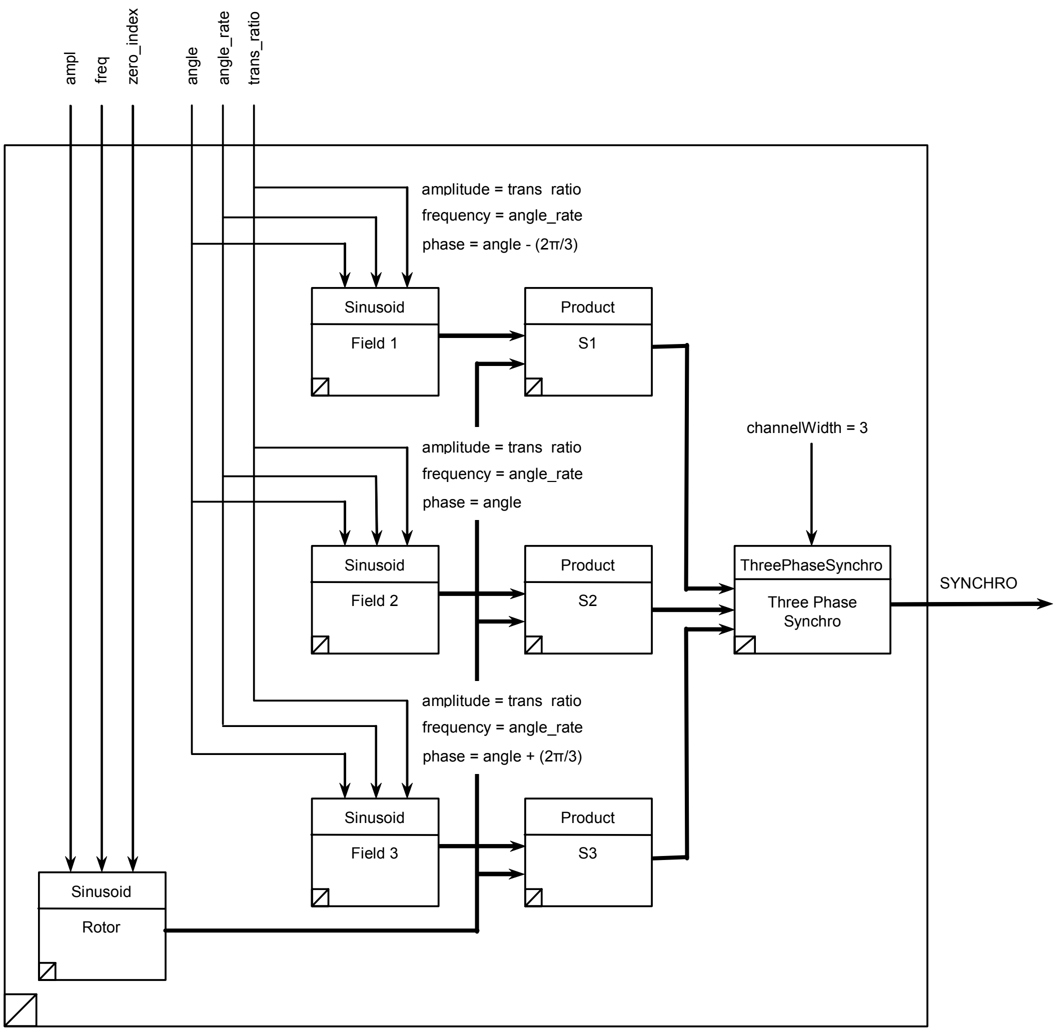


Figure 20—Annotated SYNCHRO signal TSF model

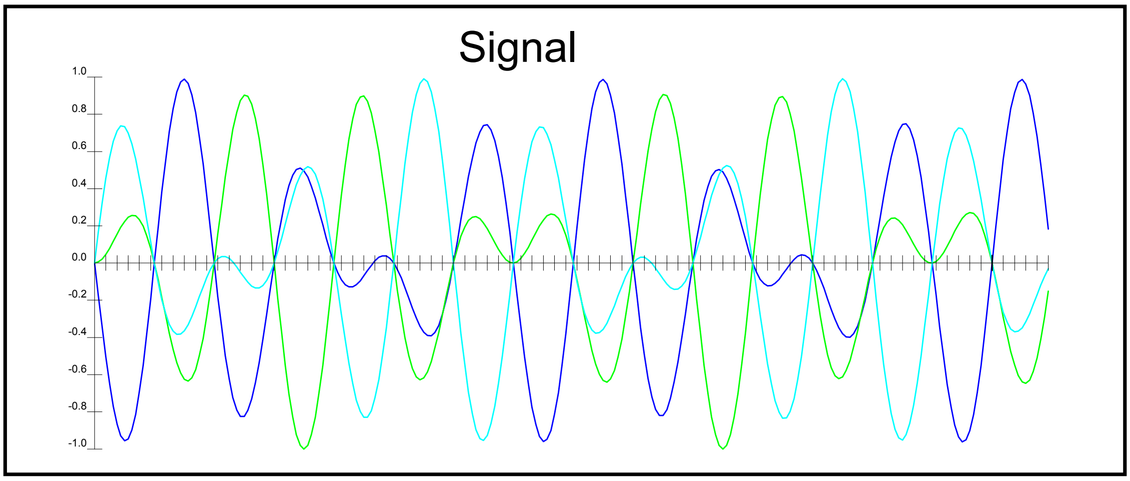


Figure 21—Output from SYNCHRO TSF model

6.4 Using TSFs in a test requirement or program

Signal templates may be used in a test requirement or program by adding the test parameters to the template to create an instance of a signal. The reason to use a template is to enable the signal definition to be reused.

The STD standard does not mandate how the requirement is to be captured. It is possible to use the TPL as defined in Annex H of the STD standard. This method allows a more traditional textual definition of a program in a simple format; however it does requires a process to convert the TPL program into the test program native language (see examples in 8.5).

The test program may also be written directly in the native language using the signals instantiated from BSCs or the TSF models (see 8.5.1.3).

Both these approaches provide the same test requirement in a procedural (dynamic) form, i.e., “do this; then do that; etc.”

Test requirements may also be captured in a nonprocedural form for use in an information or declarative (static) system, e.g., using XML descriptions such as ATML or other nonprocedural environment (see 8.5.2.6). The STD standard itself uses the nonprocedural method to define test requirements, as seen in the behavioral model of a TSF (Figure 3).

The STD standard allows the creation of “anonymous TSFs,” which represent a signal’s behavior without any interface definitions, using the Require method and XML signal definition. Anonymous TSFs support a default interface that provides a “Document Object Model” interface utilizing late binding (IDespatch).

Example:

```
mySig.DC_Offset.dc_offset = "1V"
mySig.AC_Component.freq = "10MHz"
```

These anonymous TSFs can be used to create and define nonprocedural test requirements and can be used interchangeably within a procedural test requirement definition.

The test requirement may then be built up in a procedural or declarative manner.

Clause 8 provides many examples of tests and test requirements that use TSFs.

6.5 Measurement models (signal measurement)

Thus far, TSF models have been discussed as signals. Signals and, therefore, TSFs can be equally considered sources, events, or measurements. This flexibility allows TSF models to encapsulate measurements or even complete tests.

Measurements are performed using the Sensor BSCs. Many of these perform the measurement of specific characteristics of a signal, i.e. they perform intrinsic measurements. The Measurement BSC requires additional information about the measurement required, but can measure characteristics of a signal as described by a signal attribute, i.e. a generic measurement.

Clause 7 deals with measurement in 1641 in some detail, and Annex B and Annex C include further discussion of Intrinsic and Generic measurement, respectively.

The Decode BSC in one of the Sensor BSCs, which will be described separately.

7. Defining measurements with STD

7.1 General

Measurement within the STD standard should be considered within the following context:

— Precursor signals are supplied (to the UUT).

— Output signal is returned (from the UUT).

— The signal is conditioned (optional).

— Characteristics of the signals are measured at specific points.

— The measurement characteristics may then be further analyzed.

Figure 22 shows a typical signal into a UUT and a measurement being performed on an output of the UUT.

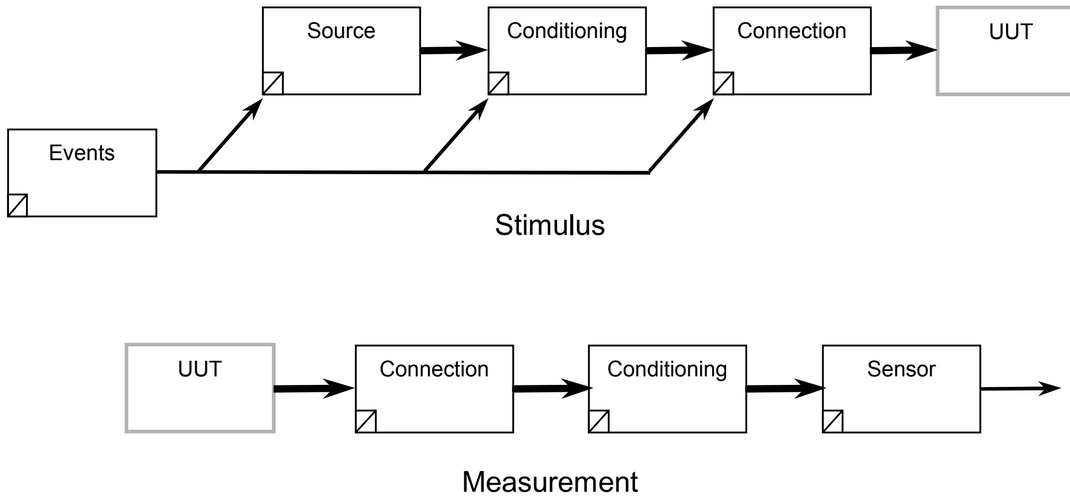


Figure 22—STD components and their typical use

Within the STD standard, each of these areas is dealt with by a separate item:

- Sources
- Connection
- Conditioners
- Sensors
- Carrier language analysis libraries (outside the scope of the STD standard)

Within this list, the sensors are the STD components that provide the raw measurement feature.

There are two distinct measurement methods defined by sensors within the STD standard:

- Generic measurement
- Intrinsic measurement techniques

Before describing the different types of intrinsic measurements supported by STD, it is worth examining some common measurement principles and those provided by all STD sensors.

7.2 Sensors

7.2.1 General

All sensors provide the same basic capability:

a) Number of measurements (samples)

b) The interval over which the measurement should be taken (gateTime)

c) Event when the measurement should be made (Gate)

d) Expected value and range of values (nominal).

7.2.2 Measurements

A sensor can take a series of measurements. The number of consecutive measurements that will be taken is defined by the samples attribute.

In general, each individual measurement is made over the measurement window specified by the gateTime attribute. The use of the word Time in the attribute name is a misnomer; gateTime refers to the width of the measurement window in terms of the independent variable (reftype). Time just happens to be a very common independent variable.

Figure 23 shows a series of measurements gated by a positive gateTime. The measurement window is the same as the gateTime. Measurements are made when the input signal is available (active) and when the sensor is gated by an event.

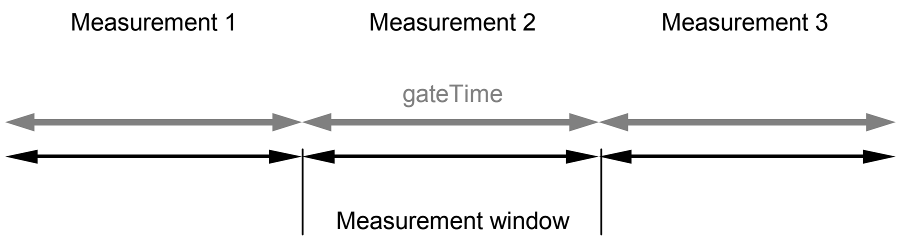


Figure 23—Measurement window

The value of gateTime provides more information than just the size of the measurement window. The following added complexities should be noted:

a) gateTime is zero, and

   1) there is a gated event stream. The measurement window is defined by the duration of the Gate event, and each successive measurement is made when the next Gate event occurs.

   2) there is no gated event stream. The measurement window is implementation dependent, and each successive measurement is made sequentially.

   NOTE— This mode relies on the measuring system determining the appropriate measurement window to suit the signal being measured. This may be satisfactory for very simple applications or for initial use during program development, but could give rise to different results on different test equipment. It is strongly recommended that a non-zero gateTime is provided so that the measurement requirement is explicit and will provide the same result across different test equipment.

b) gateTime is positive, and the sensor is defined with respect to time, and

   1) there is a gated event stream. The measurement window is defined by the value of gateTime, and the time when each successive measurement is made is determined by the next Gate event after the gateTime has elapsed, following the previous measurement (see Figure 24).

2) there is no gated event stream. The measurement window is defined by the value of gateTime, and the time when each successive measurement is made is also determined by the value of gateTime.

c) gateTime is negative, and the sensor is defined with respect to time, and

1) there is a gated event stream. The measurement window is defined by the duration of the Gate event, and the time when each successive measurement is made is determined by the next Gate event after the (absolute) gateTime has elapsed, following the previous measurement (see Figure 25, in which the gateTime arrows represent the absolute gateTime).

2) there is no gated event stream. The measurement window is implementation dependent.

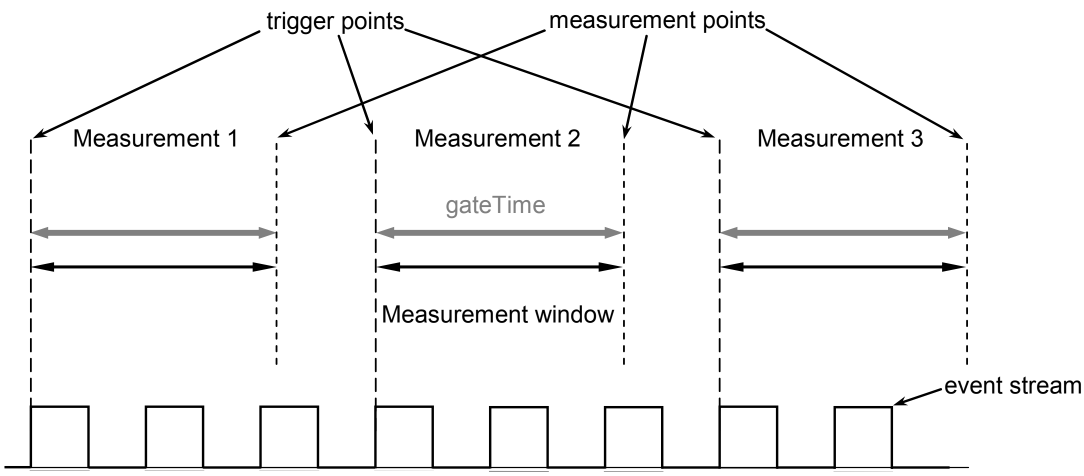


Figure 24 —Measurement windows with positive gateTime and gated event stream

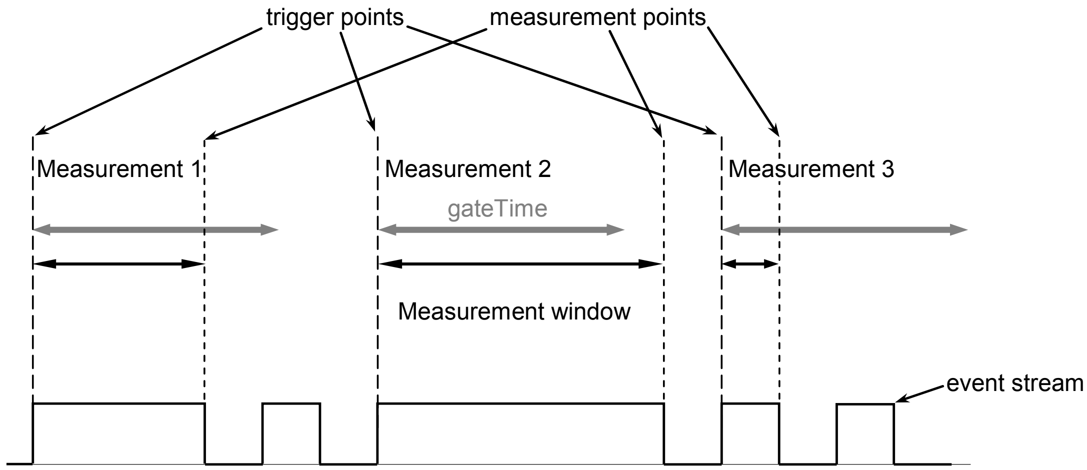


Figure 25 —Measurement windows with negative gateTime and gated event stream

If the measurement is not defined with respect to time (e.g., with respect to frequency instead) and the sensor uses a gated event stream, all the measurements are taken when the gate event arrives.

When multiple measurement attributes are defined, each measurement has the same gateTime value.

The nominal attribute indicates the expected value and units, the range of the expected value, the uncertainty by which it should be measured, and the qualifier by which it should be represented.

The output from a sensor can be both an event stream and a signal. When a sensor is performing a measurement, it will transition to the Active state once the measurement(s) have been taken. If multiple measurements are defined, this transition occurs when the last measurement has been taken.

7.2.3 Monitoring and conditioning events

As well as the ability to perform measurements, sensors also have the capability to monitor the signal in order to generate events. The monitoring criteria control when events are generated (these events are known as conditioning events).

A sensor is in monitoring mode when its samples attribute is set to zero. If its condition attribute is anything other than none, then it will also generate conditioning events, by comparing against the current value that would be measured against the value in the nominal attribute, and is active while the condition is true. When the condition attribute is none, the signal is active while monitoring and has a signal value of the measured value.

When a sensor is in the monitoring mode, it does not record any measurements. The value that would be measured at any given point is the value used for the conditioning comparison. Conversely, a measured value is simply a snapshot of the monitored signal (see Figure 26). The arrows in the diagram represent the measurement window for any particular snapshot. A measurement window cannot start before t = 0; therefore, a monitored value obtained immediately after t = 0 will be for less than a full measurement-window period. Generally, the monitored value and measurement value are the value at the end of the measurement window. When a measure is initiated (triggered), the measurement window is started, and the value of the measurement is taken from that point over the measurement window.

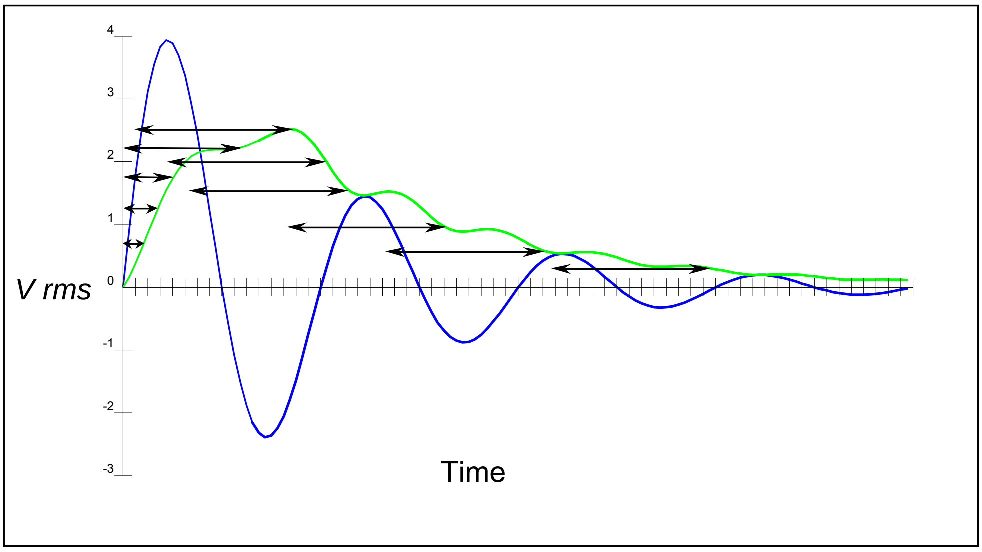


Figure 26—Continuous monitoring over rolling measurement window

As an example of this principle, the XML description of a sensor that generates a stream of events when the input signal is greater than 0 V would include

```
<Instantaneous condition="GT" nominal="0 V" samples="0" In=.../>
```

The effect is shown in Figure 27.

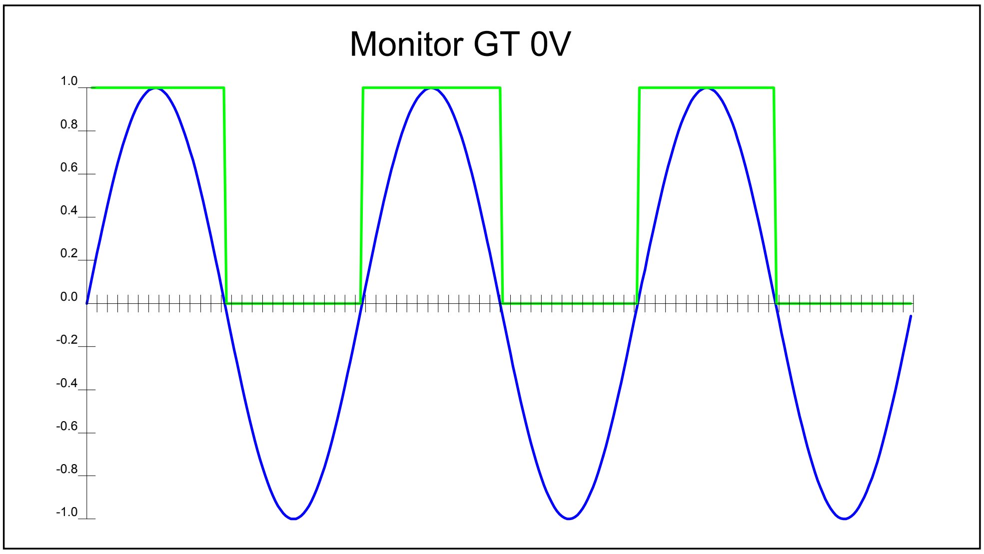


Figure 27—Generating event for input signal above zero

7.2.4 Relative triggering

In addition to specifying an absolute trigger value, e.g., 1.2 mV, it is also possible to specify a relative trigger value by using a ratio quantity as the nominal value, e.g., 10%, 50%, or 90%.

As an example of this approach, the XML to trigger when the instantaneous value gets above 50% of its value would include the following:

```
<Instantaneous condition="GT" nominal="50%" samples="0" In=.../>
```

In all cases, the relative value is with respect to the reference level or nominal amplitude, which is the value of the amplitude without any ringing, droop, or other effects. For a pulse, 0% is the value of the low state or state 0, and 100% is the value of the high state or state 1. These definitions are defined in IEEE Std 181™-2011 [B7].

Therefore, by definition, to measure the amplitude of a square or pulse waveform without transitory effects, measure the instantaneous value when the signal reaches 100%.

```
<Instantaneous name="State1" condition="EQ" nominal="100%" In=.../>
<Instantaneous gate="State1" samples="0" In=.../>
```

This step can be reduced down into a signal sensor statement as follows:

```
<Instantaneous name="level1" condition="EQ" nominal="100%" samples=”1”
In=.../>
```

This reduced form has the advantage of not using separate events and, therefore, can be used when the independent variable (reftype) is not with respect to Time. If the above example were to declare reftype='frequency', then the amplitude when the frequency reaches the value that gives the largest (100%) value over the complete range would be measured. In other words, the amplitude of the dominant frequency would be measured.

7.3 Measurement maps

A signal is defined as a dependent variable varying over the independent variable, e.g., as a voltage signal varying over time. The definition of such signals uses attributes, e.g., a sinusoidal signal is defined with the attributes amplitude, frequency, and phase.

When a measurement is taken, the values of these attributes are returned at particular positions in terms of the independent variable, e.g., an amplitude may be returned at particular points in time. Consider the value of the measurand attribute(s) over all values of the independent variable, and the measurement returns those values at specific locations. As an example, the measurement map of an rms voltage level of a damped sinusoid signal is illustrated in Figure 28.

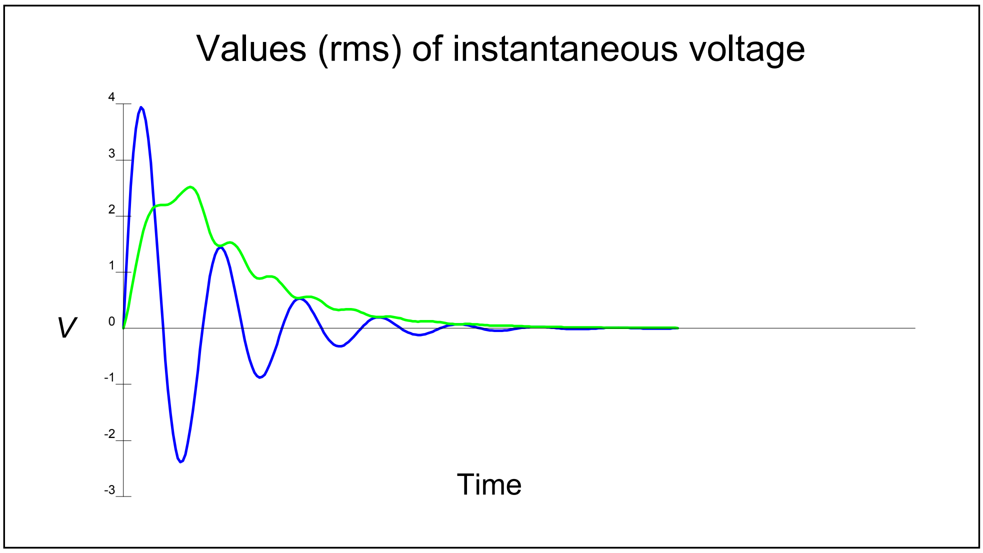


Figure 28—Measurement map of rms value of damped sinusoid signal

7.4 Intrinsic measurements

7.4.1 Components supporting intrinsic measurement

The STD standard supports eleven specific measurement components:

a) Counter
b) Interval
c) Instantaneous (inst)
d) RMS (trms)
e) Average (av)
f) PeakToPeak (pk_pk)
g) Peak (pk)
h) PeakPos (pk_pos)
i) PeakNeg (pk_neg)
j) MaxInstantaneous (inst_max)
k) MinInstantaneous (inst_min)

Of these, the Counter and Interval are special cases.

— Counter provides a dimensionless quantity indicating the number of measurements made.
— Interval is applicable to the independent variable because it provides the difference between events in terms of the independent variable.

— The measurement BSC name TimeInterval is available in the 2010 revision of the STD standard to support any legacy program that may use it. The BSC name TimeInterval is only applicable when the independent variable is Time. The use of the name TimeInterval is deprecated.

By definition, the intrinsic measurements are equivalent to the generic measurement without any reference signal or attribute value and using the appropriate nominal qualifier.

Example:

```
<Instantaneous/> is equivalent to <Measure nominal="inst"/>
<RMS/> is equivalent to <Measure nominal="trms"/>
<Average/> is equivalent to <Measure nominal="av"/>
```

These measurements can apply to any combination of dependent variable (type) and independent variable (reftype). In other words, these measurement components convert or consider the input signal as the required type. By selecting different types, the signal is converted into a signal represented by the dependent variable and independent variable. If a nominal value is specified, it must be an allowed type or physical mapping.

In all cases, the measurement describes the measurement map between the dependent and independent variables. Hence, for an input signal of sinusoidal voltage, the expected results of the measurement over a range of values would appear as shown in Figure 29.

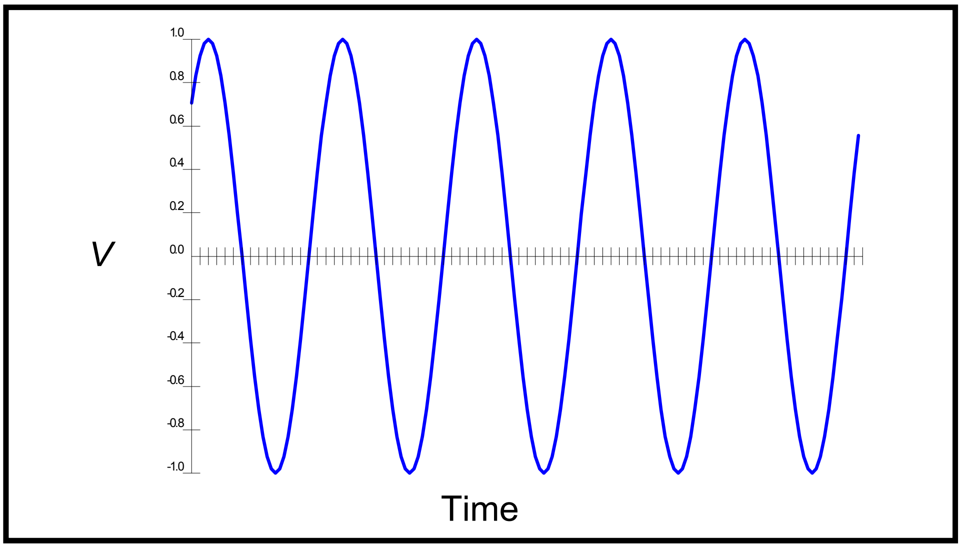


Figure 29—Waveform from <Instantaneous type="Voltage" reftype="Time" />

Figure 30 and Figure 31 show the same sinusoidal signal but with different dependent and independent variables.

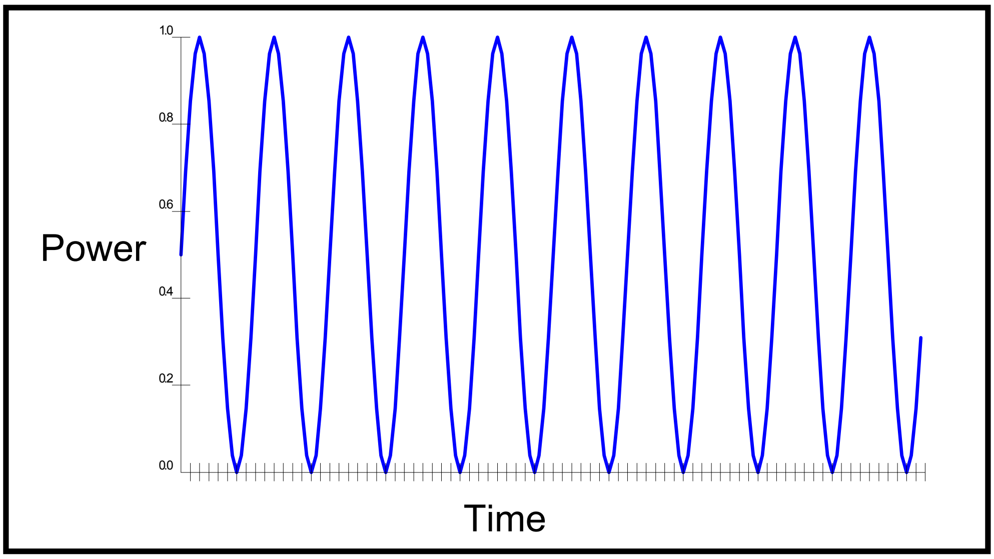


Figure 30—Waveform from <Instantaneous type="Power" reftype="Time" />

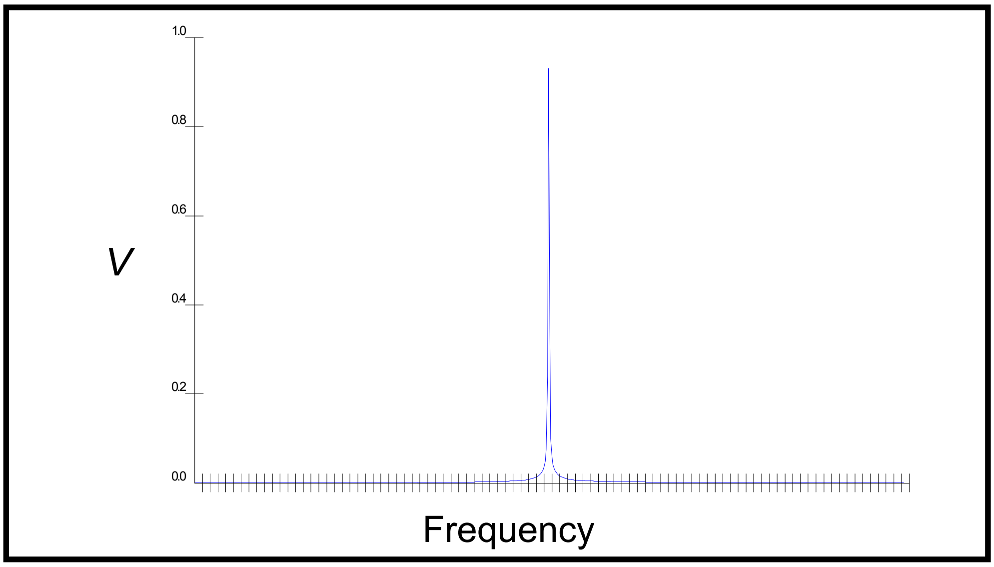


Figure 31—Waveform from <Instantaneous type="Voltage" reftype="Frequency" />

Note that some measurement maps may be invalid or meaningless. Consider the case of a two-tone signal. A measurement map of instantaneous frequency against time would not be valid. However, a measurement map of amplitude against frequency would be perfectly reasonable.

It is also possible to define measurements on quantities that do not have a unit name. It may be necessary to define network traffic in terms of kilobytes per second. Although there is no quantity defined in these physical units, the type Physical may be used with the units specified in the nominal attribute.

```
<Average type=”Physical” nominal="128 kB/s" In=.../>
```

Further examples of the use of intrinsic measurement are provided in Annex B

7.4.2 Illustrations of intrinsic measurement

7.4.2.1 Explanation of example signal used

Each of the examples shown use the same signal, which is a 5 kHz sinusoid with an amplitude of 1 V added to a 1.75 V dc offset. The signal is initially inactive, as is shown by the "dot-dash" trace, and then after a short period of time (0.75 ms) is triggred by an event (not shown). Each figure includes two sets of traces; the upper set represent the measurement of the signal and the lower set represent the monitoring of the signal. The gate time for the measurement is set at 0.2 ms (GateTime="0.0002") or one complete cycle of the sinusoid. The measurement trace is superimposed upon the upper signal trace and it may be seen that the trace changes to a "dashed" line (inactive) until the measurement is complete, at which point it jumps to the measured value as a solid line.

The monitor trace is superimposed on the lower signal trace and is also inactive until the event occurs, at which point it becomes active and outputs the appropriate monitored value. The value output by the monitor varies until it has enough information to acertain the overall correct value. In these examples it requires at least one complete cycle of the signal.

7.4.2.2 Average measurement

Figure 32 shows the average measurement of the signal. The measured value may be seen to be equivalent to the dc offset value as the average of the sinusoidal component is zero. The monitor trace shows the instantaneously calculated average value of the signal until the first complete cycle is complete.

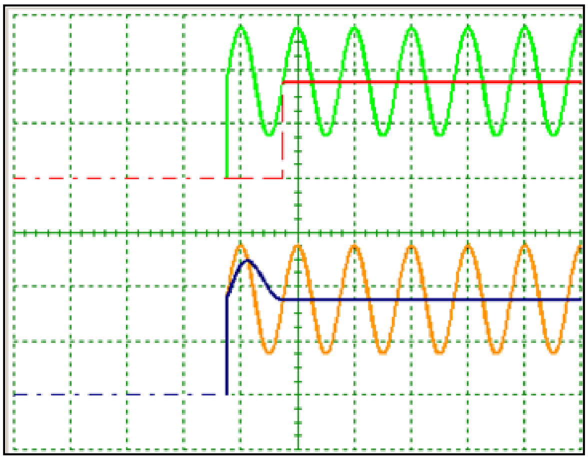


Figure 32—Average measurement

If a longer gate time had been used, the monitored value would have taken longer to settle, but the final result would have been correct. Figure 33 shows an example of this where the gate time has been increased to 0.6 ms.

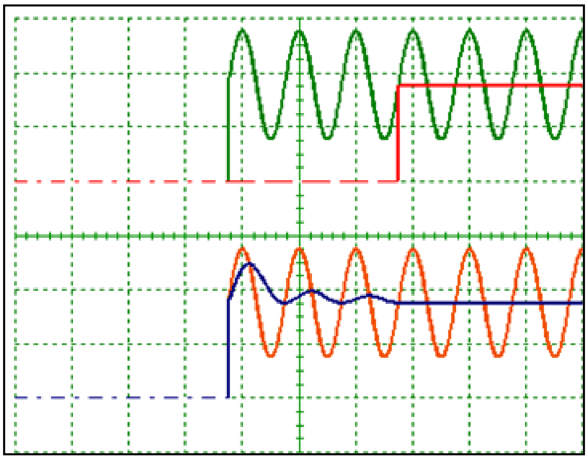


Figure 33—Average measurement with longer gate time

If a shorter gate time had been chosen, the monitor trace will never settle to the correct overall value but will continuously calculate the value over the previous running gate time. The measured value will also be incorrect because the signal is not measured over a full cycle. This is illustrayed in Figure 34 where the gate time is only 0.14 ms.

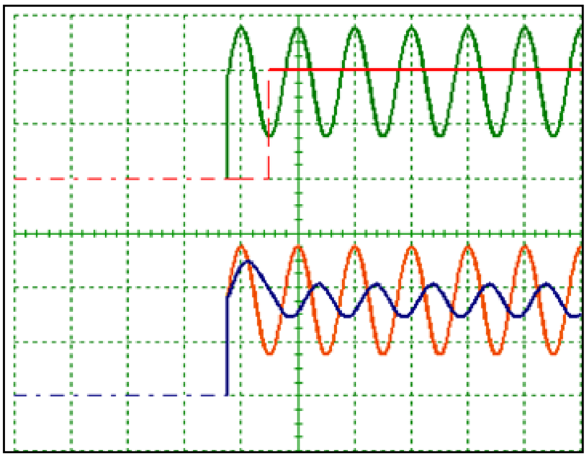


Figure 34—Average measurement where gate time is too short

If more that one measurement is taken with a gate time that is too short, a different value may be obtained for each measurement. Figure 35 shows five measurements being taken (Samples="5", GateTime="0.14 ms") and shows five different measurement values.

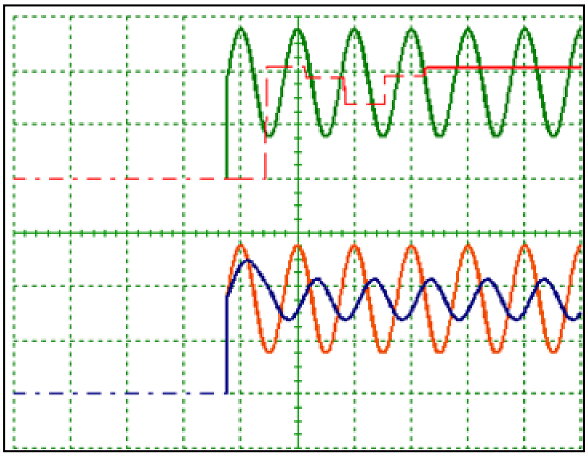


Figure 35—Average measurement with short gate time & 5 samples

7.4.2.3 RMS measurement

Figure 36 shows the RMS measurement and monitoring of the same signal.

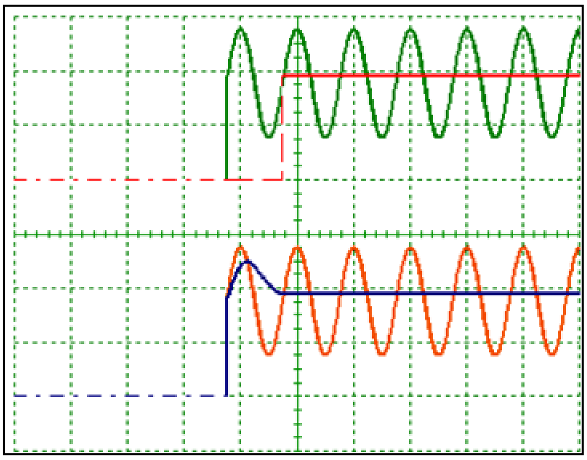


Figure 36—RMS Measurement

7.4.2.4 MaxInstantaneous measurement

Figure 37 shows the MaxInstantaneous measurement and monitoring of the signal. The monitor trace clearly shows how it follows the increasing instantaneous value of the signal until it reaches its maximum value.

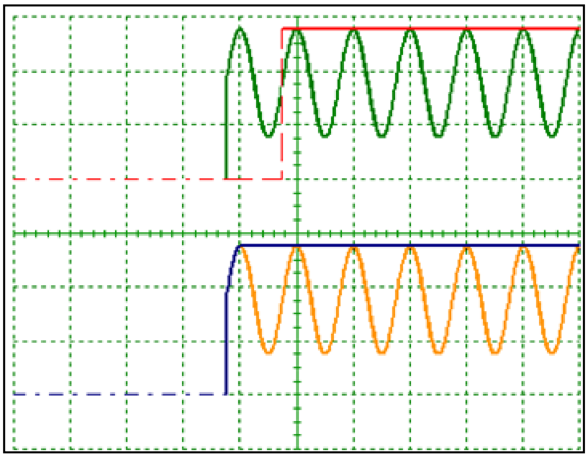


Figure 37 —MaxInstantaneous measurement

7.4.2.5 MinInstantaneous measurement

Figure 38 shows the MinInstantaneous measurement and monitoring of the signal. The monitor trace shows the value initially being the starting value, i.e. the dc offset value, until the trace goes below that level.

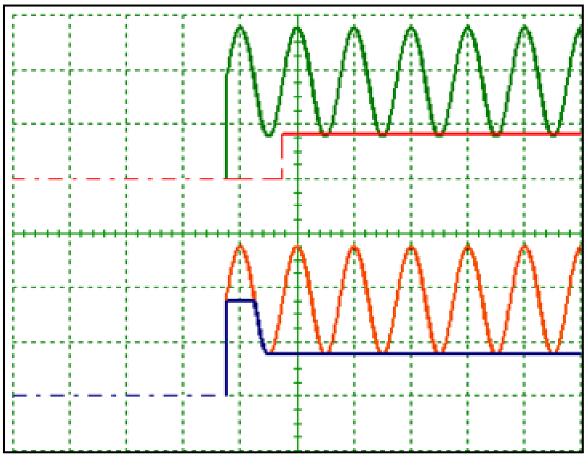


Figure 38— MinInstantaneous measurement

7.4.2.6 PeakPos measurement

Figure 39 shows the PeakPos measurement. In this case the monitor trace shows the current difference between the average value and the positive peak value until there is enough information to calculate the final result.

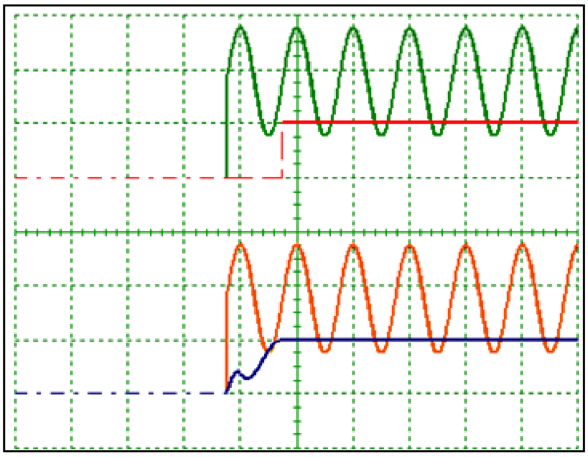


Figure 39—PeakPos measurement

7.4.2.7 PeakNeg measurement

Figure 40 shows the PeakNeg measurement. In this case the monitor trace shows the difference between the average value and the negative peak value. This gives a negative value as shown in both the measurement and monitor traces.

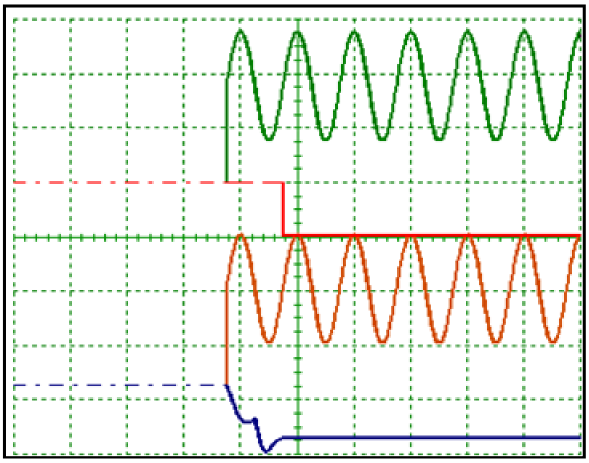


Figure 40— PeakNeg measurement

7.4.2.8 Peak measurement

Figure 41 shows the Peak measurement. This is the greater of the PeakPos and PeakNeg magnitudes. In this example the values are the same so the Peak will be the same as the PeakPos value.

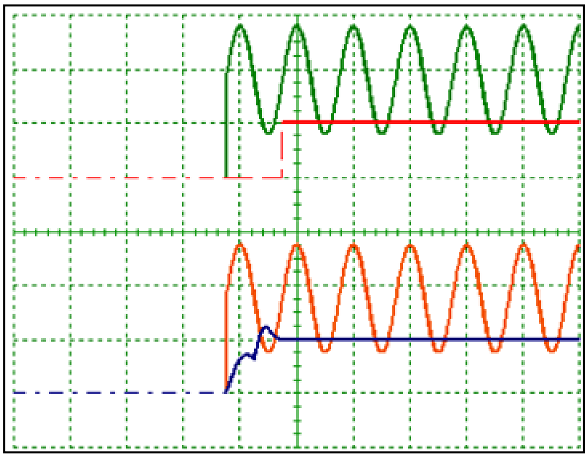


Figure 41 — Peak measurement

7.4.2.9 PeaktoPeak measurement

Figure 42 shows the PeaktoPeak measurement and monitor trsces.

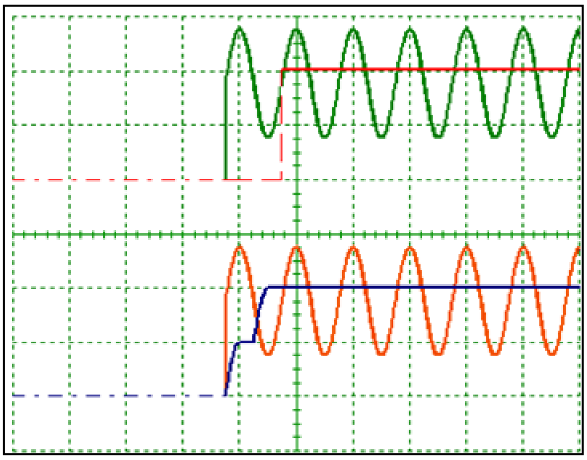


Figure 42 — PeaktoPeak measurement

7.4.2.10 Counter measurement

Figure 43 shows the count measurement and monitors traces. In the measurement set of traces at the top the number of measurements is set to two (Samples=2) and the measurement trace shows the count of the two measurements. The amplitude of the measurement trace represents counts not voltage. The monitor trace shows continuous counts for each gate time period.

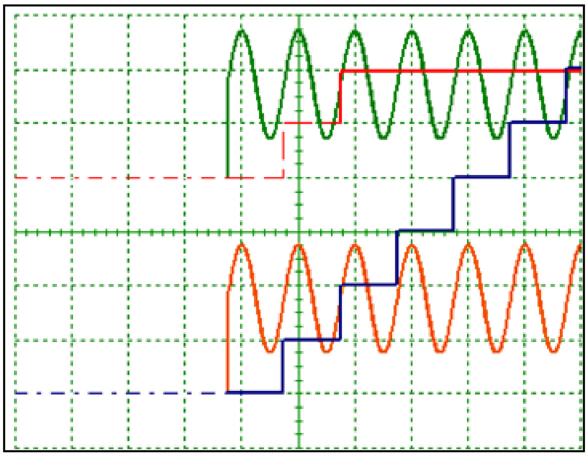


Figure 43—Count measurement with Samples=2

7.4.2.11 Interval measurement

Figure 44 shows the measurement of time interval. The amplitude of the measurement trace indicates time, in this case 20 ms, not voltage. The monitor trace just shows a continuous trace of increasing time.

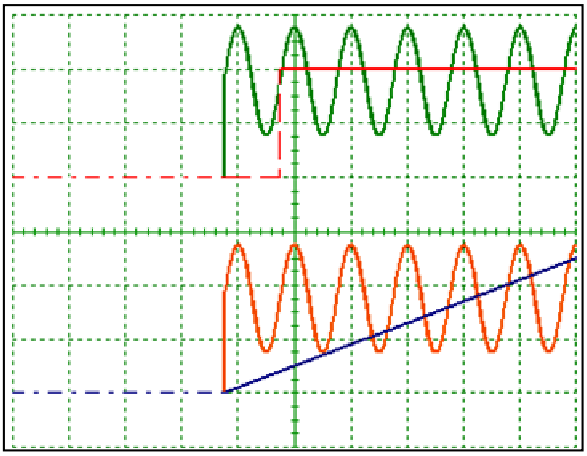


Figure 44—Interval measurement

7.5 Generic measurement

7.5.1 General

One major feature of STD measurements is known as generic measurement. This principle combines with TSF models so that any TSF model naturally defines its measurement requirements. It does not define the measurement method or the approach to achieving it.

Key features of generic measurements with a TSF model are the following:

a) Generic measurement provides an inverse function for any signal library component to measure any of its control attributes.

b) Because users can define library components, the generic measurement is naturally extended and can be used to measure any control attribute of such signal.

The concept of generic measurement provides an inverse function for a TSF or BSC to measure any of their control attributes. Because users can define TSFs, the generic measurement is naturally extended and can be used to measure any control attribute of such a TSF signal.

As an example of this principle, reading back an RS232 message may be simply achieved.

```
<Measure As="rs232" attribute="data_word" In=... />
```

To read back a list of pulses from a PULSED_AC_TRAIN is also simply expressed.

```
<Measure As="PULSED_AC_TRAIN" attribute="pulse_train" In=... />
```

To measure the rise time of a square wave is also simply expressed.

```
<Measure As="Trapezoid" attribute="riseTime" In=... />
```

The use of the generic measure is not restricted to measuring single attributes, as shown by the following example of performing multiple measurements:

```
<Measure As="Sinusoid" attribute="ac_ampl phase" In=... />
```

The XML for the STD standard identifies the As attribute as being an xs:IDREFS. Therefore, it has it has to contain the name of a signal defined separately and should not contain the signal definition directly.

Example:

```
<Sinusoid name=”sinusoid” />
            <Measure As=”sinusoid” attribute=”ac_ampl” In=... />
```

Using the XML definitions in this way is more verbose; therefore, for brevity, examples may use the abbreviated description that includes the reference descriptions within the As attribute. A style sheet could be used to convert them to valid STD XML.

7.5.2 Generic measurement definitions

To illustrate in more detail what is happening when generic measurement definitions are used, the examples in this subclause use the TSF AC_SIGNAL from Annex E of IEEE Std 1641-2010.

Consider the following measurement in XML:

```
<Measure As="tsf:AC_SIGNAL" attribute="ac_ampl" In=... />
```

This example is asking for the best-match value for the ac_ampl attribute assuming the input signal is an AC_SIGNAL. The AC_SIGNAL is defined as a signal whose voltage varies with time and has both a sinusoidal and dc component. Therefore, the default value of the ac_ampl is going to be the amplitude of the sinusoidal component in volts.

If the requirement was to measure the ac_ampl in trms volts, this value can also be specified. The definition for defining an AC_SIGNAL with a voltage output to provide a trms 2 V value is defined as follows:

```
<tsf:AC_SIGNAL ac_ampl="trms 2V"... />
```

Hence, the measurement could be specified as follows:

```
<Measure As="tsf:AC_SIGNAL ac_ampl='trms'" attribute="ac_ampl" In=.../>
```

Alternatively, the nominal attribute could be used to define the expected value:

```
<Measure As="tsf:AC_SIGNAL" attribute="ac_ampl" nominal="trms 2 V"
In=... />
```

Although there are two (similar) ways of defining the same measurement, they are not completely interchangeable.

a) It is not possible to use the nominal method when measuring multiple properties; strictly speaking, the nominal value refers to the dependent variable to be measured for the purposes of monitoring the signal.

b) The nominal definition overrides the reference signal and takes precedence; therefore, it can be used in creating measurement TSFs.

As an example of point b), a signal can be described as having a certain expected power (Measure As= ... 2 mW'"), but can indicate that the measurement is to return another value by providing a nominal value, e.g., the peak voltage (nominal="pk 10 mV"):

```
<Measure As="tsf:AC_SIGNAL ac_ampl='2 mW'" attribute="ac_ampl"
nominal="pk 10 mV"... />
```

It is not essential to provide the expected value, but in some cases, these additional values can aid a measurement and help in determining the best way to perform the measurement. When specifying units, the XML validation is that a numeric value must be entered. It is possible to specify a qualifier such as trms or pk on its own, but it is not possible to specify a unit such as mV or mW on its own. Units have to be written with an associated value, e.g., 10 mV, 1 mW.

Finally, if the power related to the ac_ampl was to be measured, it could be expressed as follows:

```
<Measure As="tsf:AC_SIGNAL ac_ampl='trms 10 mW'" attribute="ac_ampl"...
/>
```

The above definition is interpreted as “measure the ac component power of an AC_SIGNAL voltage signal whose expected voltage will provide a trms 10 mW ac component.” In order to measure the whole power of the signal, the As constraint must be removed. Alternatively, use one of the intrinsic measurement techniques.

Examples:

```
<Measure nominal="trms 10 mW" In=... />
<Measure type="Power" nominal="trms" In=... />
<RMS nominal="trms 10 mW" In=... />
<RMS type="Power" nominal="trms" In=... />
```

7.5.3 Principle of generic measurement

The principle of generic measurement is that the input waveform to be measured is compared against all possible instantiations of the reference signal definition, the best match is selected, and the attribute values of that match are returned. In this definition, the best match is defined as the solution with the least error, where the error is defined as the rms value of the difference between the input signal and the reference signal.

Figure 45 shows a simple example for an AC_SIGNAL, with various alternative signals and a best match highlighted. Figure 46 shows the corresponding rms error values. The best-match signal is the one with the smallest error, also shown highlighted close to the 0 V axis.

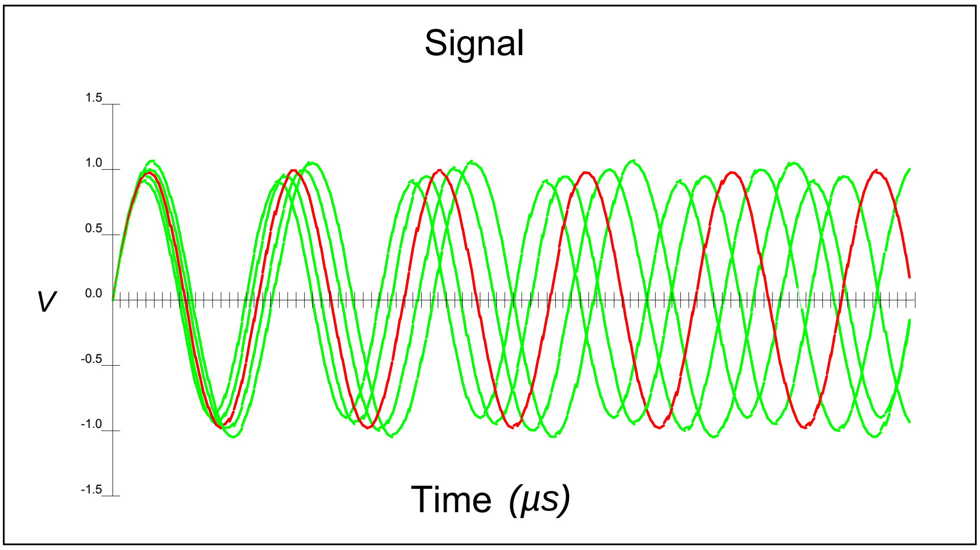


Figure 45—Generic measure best match example

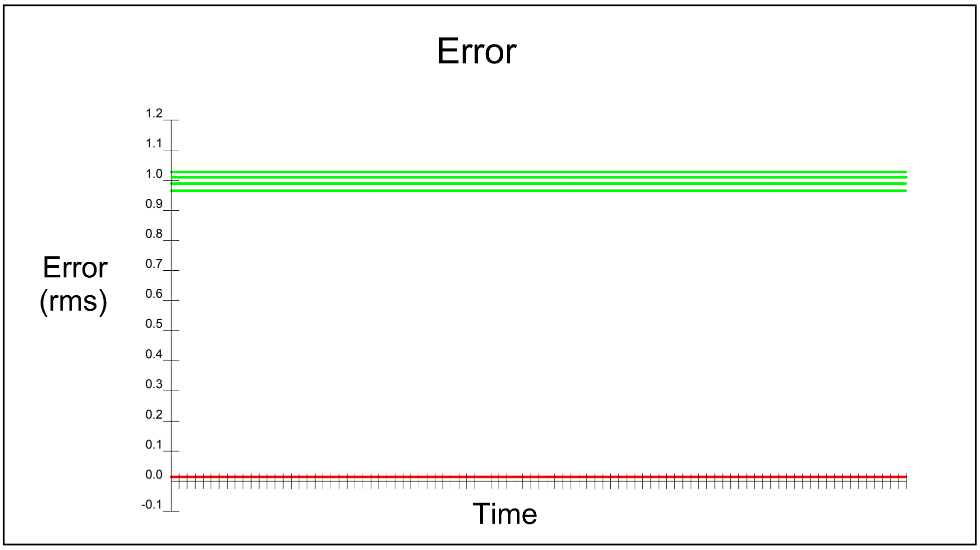


Figure 46—Errors (rms) for best match example

This definition leads to some practical difficulties with the result that different implementations will rarely actually be exact, although they will be close. The reason for this phenomenon is that implementations will optimize a solution based on their standard use cases. For example, AC_SIGNAL frequency measurement will probably result in some resource performing a frequency measurement assuming the input signal is an AC_SIGNAL. If, in practice, the input signal is not a pure AC_SIGNAL, different implementations may return different results, all equally defendable, but different from the ideal solution. The reason for this situation is that the definition is applicable to any input signal, whereas the implementation is optimized to the input signal being close to the reference definition. Of course, if the signal is a pure AC_SIGNAL, all implementations should return the same, correct result with their defined error limit (errlmt).

It is important to understand that the best-match approach is not necessarily simple to implement with some instrumentation. It is not how traditional instruments perform measurements. Assuming a solution can be implemented for synthetic instrumentation, it may be quite difficult to implement the best-match solution in some cases. The examples here show only one attribute (amplitude) being changed in order to achieve the match, while the other attributes (frequency and phase) are known. In a practical application, those attributes may also be required to be measured. If the best-match approach is used, optimization for more than one variable may run into convergence problems.

Further examples of the use of generic measurement are provided in Annex C

7.6 Reference signals description

The use of a reference signal, e.g., tsf:SQUARE_WAVE, identifies which waveforms are expected to be measured. The attributes of the reference signal may be used to constrain the range of values over which a solution is expected or to provide a nominal value where a solution is expected to lie.

```
<tsf:SQUARE_WAVE period='range min 1 ms' ampl='1 V max 5 V'
dc_offset='0 V'.../>
```

The period of the square wave will not be less than 1 ms; therefore, any measured value may not be less than this value.

The expected amplitude (ampl) is 1 V, but may be no greater than 5 V; therefore, any measured value may not exceed 5 V.

The expected dc offset (dc_offset) is 0 V, but is not constrained.

7.7 Different valid measurement methods

It is quite possible for different measurement techniques to provide different results for the same measurement. This possibility can be demonstrated using a square wave that has overshoot, ringing, and droop. In such a case, users must select the technique most suited to their requirement. Figure 47 shows such a complex square wave signal.

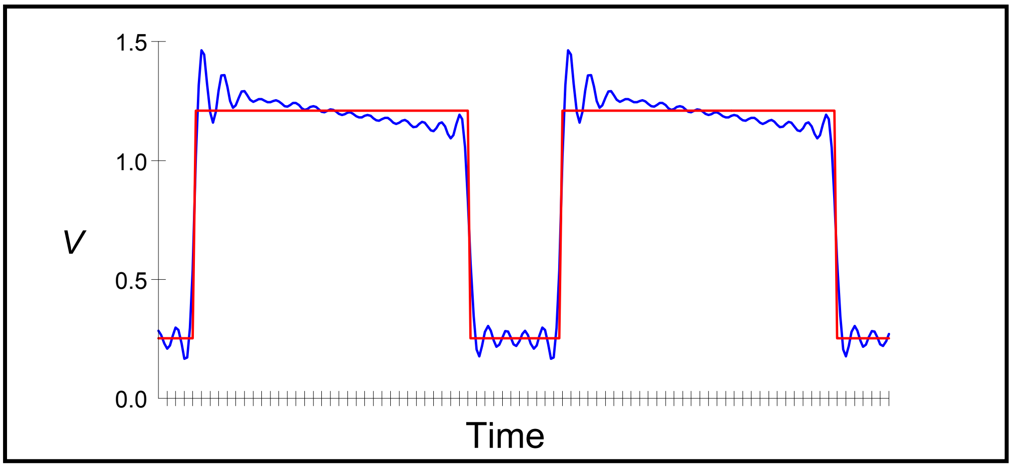


Figure 47 —Measurement of complex square wave

IEEE Std 181–2011 [B7] provides several methods for making measurements on pulse and square waveforms. Table 6 shows the results obtain when two of the methods from IEEE 181 are applied to the signal shown in Figure 47.

Table 6—Measurement of two state levels on a square wave

Level	Value (see Note 1) (V)	Value (see Note 2) (V)	XML measurement
Level 0	0.23	0.249 31	<Instantaneous name="Level0" Condition="LE" nominal="0%" In= .../>
Level 1	1.21	1.190 97	<Instantaneous name="Level1" Condition="GE" nominal="100%" In= …/>

NOTE 1—Values obtained using the histogram mode method and reports mode

NOTE 2—Values obtained using the 50% histogram average method.

Once the nominal levels are defined, formula methods may be used for the measurement TSF. The results shown in Table 6 and Table 7 may be compared with the values obtained using the generic measurement method in Annex C (see Table C.9).

Table 7— Attribute values obtained from two state-level values

Attribute	Value (see Note 1) (V)	Value (see Note 2) (V)	Calculation
nominal pk_pk amplitude	0.98	0.941 77	Level 1 – Level 0
nominal amplitude	0.49	0.470 83	(Level 1 – Level 0)/2
dc offset	0.72	0.720 14	(Level 1 + Level 0)/2

NOTE 1—Values obtained using the histogram mode method and reports mode.

NOTE 2—Values obtained using the 50% histogram average method.

Figure 48 shows the complex square wave with the attributes from Table 7 indicated on the waveform. The intrinsic measurements given in Table 8 are also shown on the waveform.

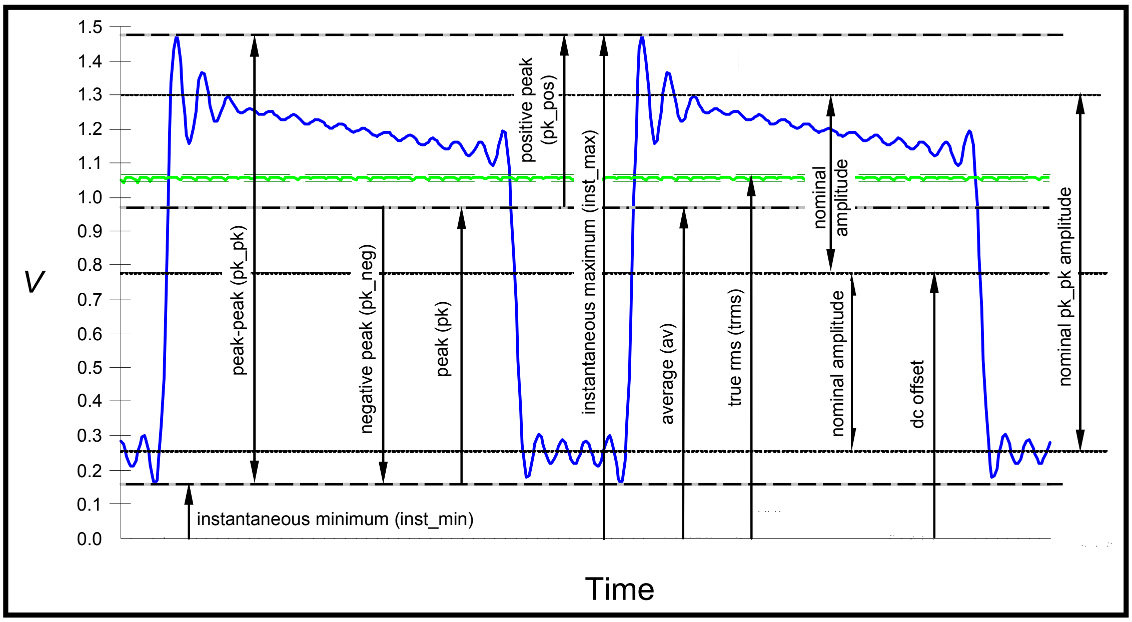


Figure 48—Measurements on complex square wave

Table 8—Intrinsic measurements on complex square wave

Measurement	Qualifier	Sensor subclass	Value obtained (V)	XML expression
instantaneous maximum	inst_max	MaxInstantaneous	1.464 46	<MaxInstantaneous In…/>
instantaneous minimum	inst_min	MinInstantaneous	1.296 81	<MinInstantaneous In…/>
peak-to-peak	pk_pk	PeakToPeak	1.296 81	<PeakToPeak In… />
average	av	Average	0.970 27	<Average In… />
rms	trms	RMS	1.054 44	<RMS In…/>
peak	pk	Peak	0.802 62	<Peak In… />
positive peak	pk_pos	PeakPos	0.493 19	<PeakPos In… />
negative peak	pk_neg	PeakNeg	0.802 62	<PeakNeg In… />

The fact that different techniques give different values for some measurements gives rise to some uncertainty in the measurement value. IEEE Std 181-2011 [B7] provides several techniques for obtaining the amplitude (level) of a waveform, including the following:

a) Average above 50%

b) Level State 1 (IEEE Std 181-2011)

   1) Histogram analog-to-digital converter (ADC) values for binning (average bins)

   2) Histogram ADC values for binning (maximum bins)

   3) Histogram ADC values for binning (centroid)

   4) Minimum energy to determine number of bins (using average bins)

   5) Minimum energy to determine number of bins (using maximum bins)

   6) Minimum energy to determine number of bins (using centroid)

   7) Maximum and minimum values

All these methods return different values. Figure 49 illustrates the variation of the measurement techniques and the value expected utilizing the generic measurement technique for a tsf::SQUARE_WAVE reference signal.

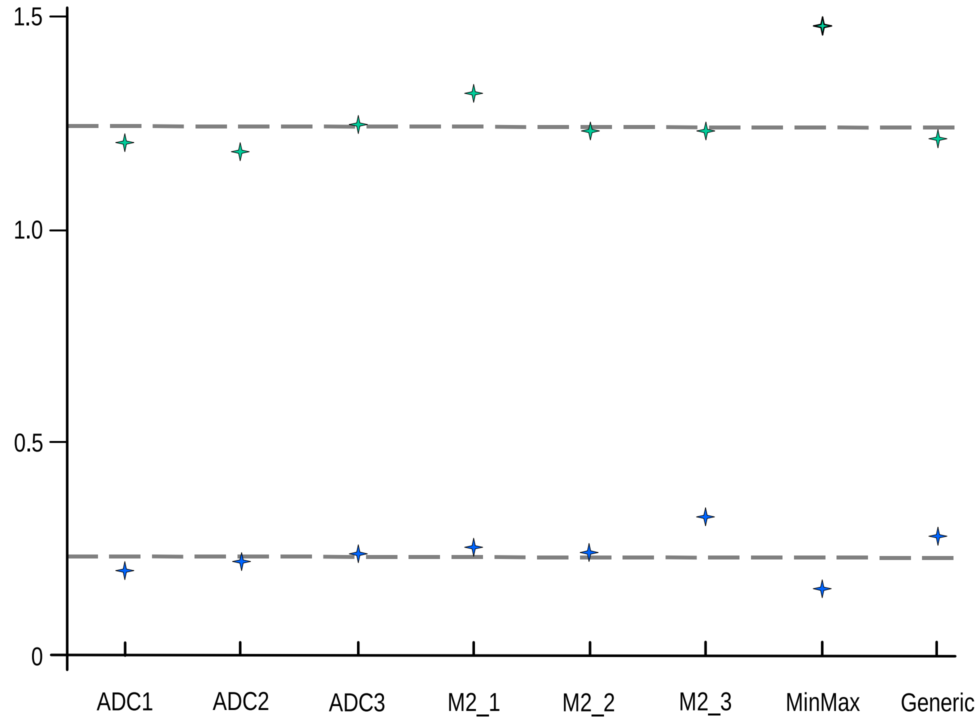


Figure 49—Distribution of values for amplitude levels

8. Describing Tests and Test Requirements

8.1 Structure—sequence and signals

Test requirements describe all the information needed to execute tests. They do not proscribe the methods or the test equipment to be used. Test requirements cover the following:

a) Stimuli required for a test

b) Measurements to be made

c) Test limits to determine whether the measured value is acceptable, i.e. whether the test has passed or failed.

d) Any calculations or transformations that need to be applied to stimuli, responses, or measurements

e) Any timing considerations for the tests

f) The sequence in which the tests should be applied, whether this is fixed or dependant upon previous results.

All these elements may be expressed within the two main components of an STD test requirement, i.e.

— The definition of the signals required to be applied and measured (signal definition part)

— The definition of the test sequencing and any calculations to be made (programmatic part)

The signal definition part will include items a) and b) from the list and the programmatic part will contain item f). Items c) to e) may be in either or both, but the resultant programs will be more easily ported from one environment to another if all items a) to e) are included in the signal part using BSCs and TSFs.

8.2 Using tools such as graphical environments

Creating test requirements directly in XML, TPL and carrier language code is a perfectly valid way of working. Most modern solutions will use tools to create valid XML TSFs and signals, together with test sequences in a carrier language. A typical solution may involve creating the test sequences using a graphical flow-chart style environment, together with a similar environment for creating signals and TSFs using graphical representations of BSCs (and TSFs) similar to those shown in Figure 13, Figure 14, Figure 15, and Figure 16.

Figure 50 shows a section from a graphical representation of an overall test plan. Each of the boxes represents a whole test sequence to verify a certain aspect of a UUT's functionality.

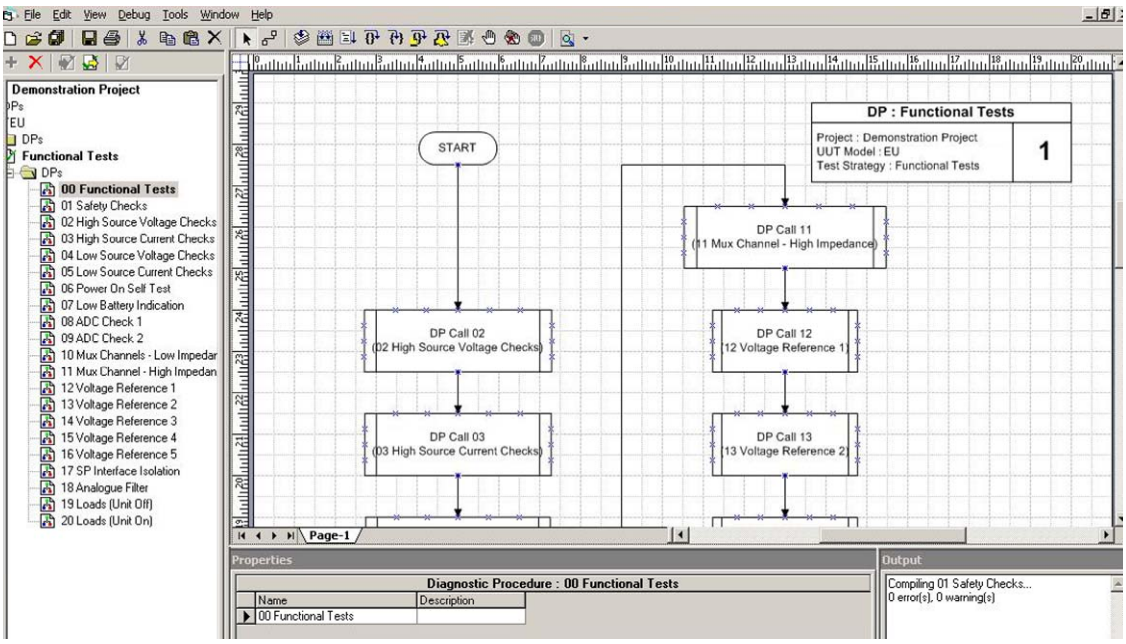


Figure 50 —Part of graphical test plan

Figure 51 shows part of a test sequence. The elements represent measurements that are to be undertaken and the subsequent program flow to be followed depending on the result of the test.

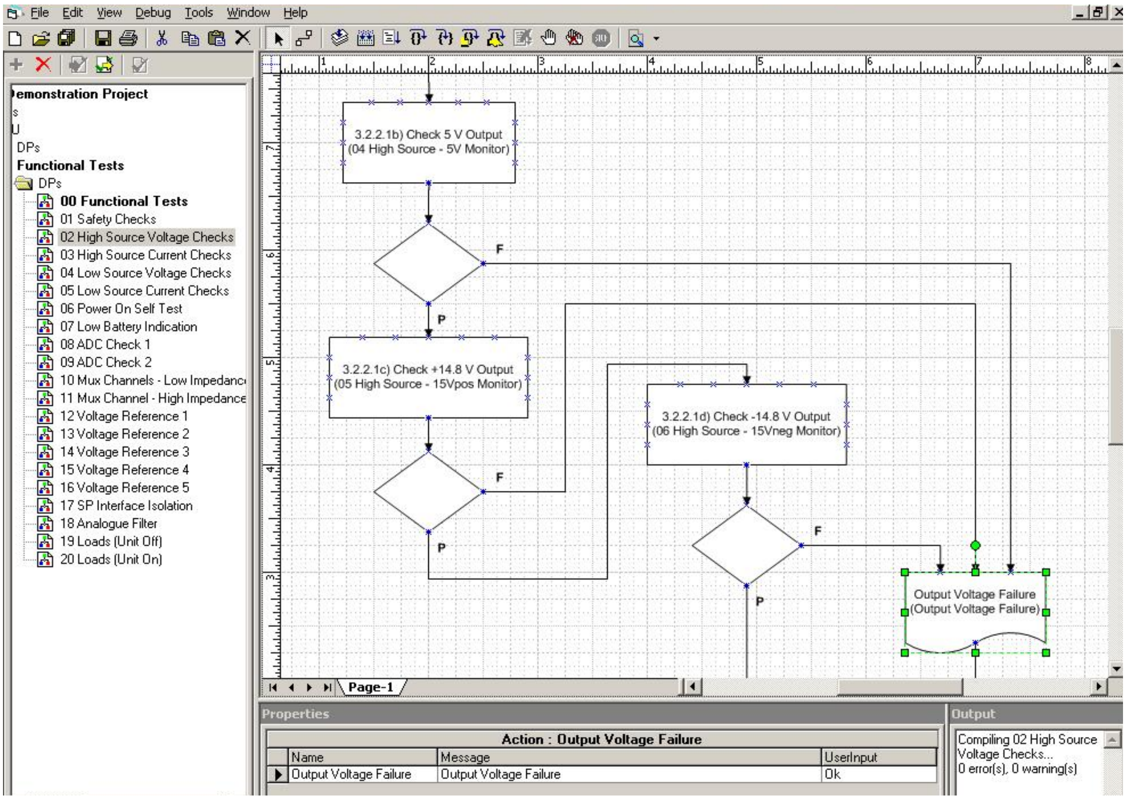


Figure 51 —Section of graphical program showing detail of program flow

The three rectangular boxes in Figure 51 represent voltage measurements. Each measurement could be described using a separate measurement TSF or a single re-usable TSF with the limits passed as attributes. Figure 52 shows a simple measurement TSF in graphical form.

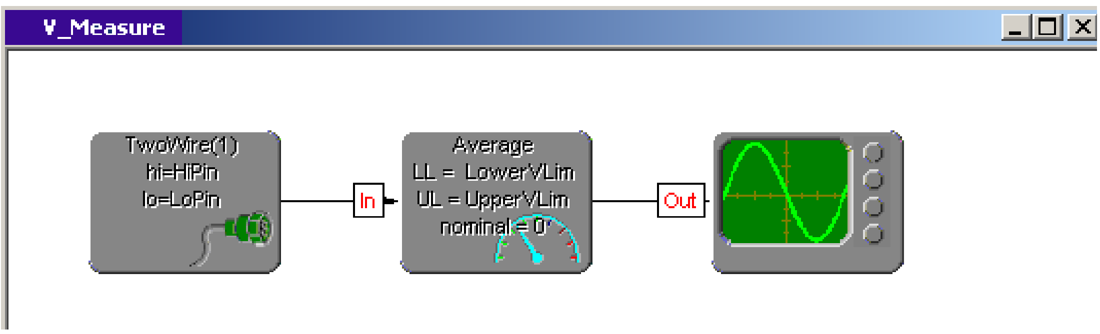


Figure 52 —Simple TSF for measuring voltage

The measurement TSF may be re-used by supplying different pin names to the connection attributes (HiPin and LoPin) and also by providing different measurement pass-fail limits (UpperVLim and LowerVLim).

Using graphical techniques in this way, test procedures may be rapidly created. The TSFs may be created as required or provided as a project wide library. The graphical toolset would then be used to create the XML signals and carrier language test sequences automatically. Information to create the test sequence would come from Figure 50and Figure 51. The XML signal information would come mainly from Figure 52 with some additional data from Figure 51.

8.3 Portable test requirements

For a test requirement to be portable and compliant with the STD standard, it requires both the signal definition part and the programmatic part to be available in formats compliant with the standard. So although graphical methods of creating test requirements are ideal for rapid program development and provide a user friendly environment, the key outputs are the XML signal definitions and the carrier language sequences. As the programmatic part is written directly in the chosen carrier language it does not require further explanation here. The signal definition part has to be extracted from the STD signal definitions and converted into code that can be used to program real physical signal sources and sensors, i.e. instrument drivers. The instrument driver may be derived directly from the XML signal descriptions or converted via carrier language code. The method to be used is dependant upon the actual automatic test system (ATS) to be used and its programming environment.

Up to this point, the discussion has been restricted to the presentation of signal data as BSC or TSF elements. This subclause of this guide illustrates how the information may appear in a typical carrier language (see Figure 53).

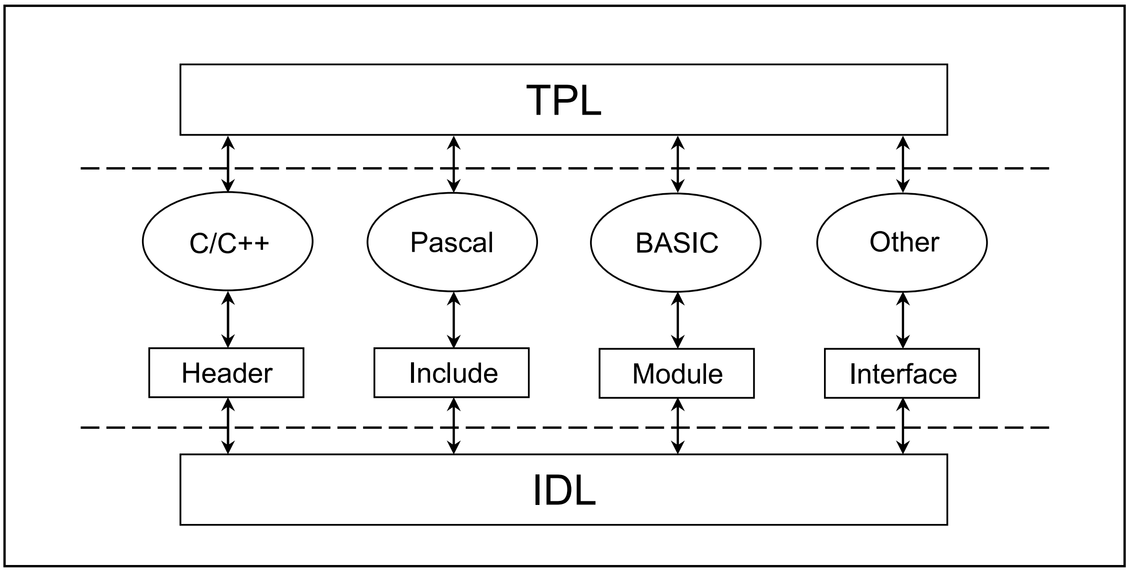


Figure 53—TPL used with any carrier language

The STD standard provides for both signal control and signal definition. Signal control is achieved programmatically. Signal definition is achieved in two ways by use of the signal building block templates:

- Using existing or predefined BSCs or TSF models from a library
- Building custom or unique signals directly using BSCs

8.3.1 Using BSC or existing TSF components

BSC and existing TSF components provide the simplest way of defining test requirements. They are used when the signal template has been defined. The test requirement becomes an instance of the TSF with the parameters entered to define a signal, a connection instance defining the location of the signal (at the UUT), and optionally an event instance defining any timing considerations.

This method is best illustrated by taking a simple example in which an ac signal with an amplitude of 1 mV and a frequency of 1 kHz is applied to the UUT. Within the carrier language, it is required to create an AC_SIGNAL object, instantiated with attributes and associated values (properties) of ampl="1 mV" and freq="1 kHz", and then the signal has to be connected to the UUT. This process can be shown in a block diagram design (see Figure 54).

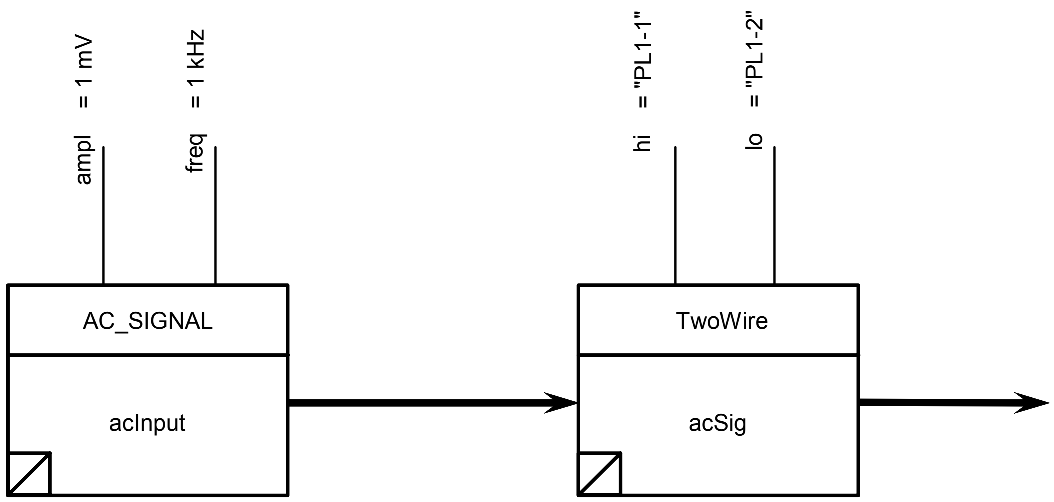


Figure 54—Simple block diagram design

8.3.1.1 Test requirement layer (using TPL)

```
Setup AC_SIGNAL ac_ampl 1 mV,
                freq 1 kHz
      as source acInput;

Connect acInput(TwoWire) to hi PL1-1 lo PL1-2
      as connection acSig;
```

8.3.1.2 Carrier language program (using IDL)

The test requirement can also be expressed directly in a carrier language using the IDL definitions from the STD standard. Here, the use of a ResourceManager object named STD is used, and it is assumed that it was previously declared and instantiated as a ResourceManager object in the carrier language.

```
'define acInput as an input
Set acInput = STD.Require("AC_SIGNAL")
       acInput.ac_ampl = "1 mV"
       acInput.freq = "1 kHz"

'define acSig as a UUT connection

Set acSig = STD.Require("TwoWire")
       acSig.hi = "PL1-1"
       acSig.lo = "PL1-2"

'assign the signal output to the connection input
Set acSig.In = acInput.Out

'turn the signal On
acSig.Out.Run
```

In the above example, the following statements apply:

a) Require is a method of the resource STD, which returns a SignalFunction object of the type specified, i.e., "TwoWire" or "AC_SIGNAL".

b) In and Out represent the signal input and the signal output to and from a SignalFunction.

c) Run is a signal control method that turns on the signal. Run has the meaning of apply because apply is both signal setup and connection switching. A signal can be explicitly turned off, using the Stop signal control method.

8.3.1.3 Carrier language program (using C)

This test requirement may also be written using the C language. The following is a typical C implementation of the same test requirement.

```
#include "STDTSF.h"
#include "STDBSC.h"
#include <my_rte.h>

#ifndef errChk
     #define errChk(fCall) if (error = (fCall), error < 0) \
     {goto Error;} else
     #endif

#ifndef CLEAR
     #define CLEAR(__handle, __func)         \
     if(__handle)                            \
     {                                       \
         __func(__handle);                   \
         __handle = 0;                       \
     }
     #endif

int main (int argc, char *argv[])
{

     int error = 0;
     ERRORINFO errorInfo;
     // Resource manager
```

```
    CAObjHandle STD = 0;
    // define acInput as an input
    CAObjHandle acInput = 0;
    CAObjHandle acAmpl = 0;
    CAObjHandle acFreq = 0;
    // define acSig as a UUT connection
    CAObjHandle acSig = 0;
    CAObjHandle iSignal = 0;

    if (Init MY_RTE (0, argv, 0) == 0)
         return -1;    /* out of memory */

    // Create a resource manager
    errChk(STDBSC_NewIResourceManager (NULL, 1, LOCALE_NEUTRAL, 0,
&STD));

    errChk( STDBSC_IResourceManagerRequire (STD, &errorInfo,
"AC_SIGNAL", CA_VariantEmpty (), &acInput));

    errChk( STDTSF_IAC_SIGNALGetac_ampl (acInput, &errorInfo,
&acAmpl));
    errChk( STDBSC_IPhysicalSetmagnitude (acAmpl, &errorInfo, 1));
    errChk( STDBSC_IPhysicalSetunits (acAmpl, &errorInfo, "mV"));
    errChk( STDTSF_IAC_SIGNALSetByRefac_ampl (acInput, &errorInfo,
acAmpl));

    errChk( STDTSF_IAC_SIGNALGetfreq (acInput, &errorInfo, &acFreq));
    errChk( STDBSC_IPhysicalSetmagnitude (acFreq, &errorInfo, 1));
    errChk( STDBSC_IPhysicalSetunits( acFreq, &errorInfo, "kHz"));
    errChk( STDTSF_IAC_SIGNALSetByReffreq (acInput, &errorInfo,
acFreq));

    errChk( STDBSC_IResourceManagerRequire (STD, &errorInfo,
"TwoWire", CA_VariantEmpty(), &acSig));
    errChk( STDBSC_ITwoWireSetHI (acSig, &errorInfo, "PL1-1"));
    errChk( STDBSC_ITwoWireSetLO (acSig, &errorInfo, "PL1-2"));

    //assign the signal output to the connection input
    errChk( STDBSC_ITwoWireSetByRefIn (acSig, &errorInfo, 0, acFreq));
    //turn the signal On
    errChk( STDBSC_ITwoWireGetOut (acSig, &errorInfo, 0, &iSignal));
    errChk( STDBSC_ISignalRun (iSignal, &errorInfo, 0));

Error:

    CLEAR (iSignal, CA_DiscardObjHandle);
    CLEAR (acSig, CA_DiscardObjHandle);
    CLEAR (acFreq, CA_DiscardObjHandle);
    CLEAR (acAmpl, CA_DiscardObjHandle);
    CLEAR (acInput, CA_DiscardObjHandle);
    CLEAR (STD, CA_DiscardObjHandle);

    return error;
}
```

8.3.1.4 XML version

The STD standard provides definitions so that the signal may be defined in XML format.

```
<Signal name="" Out="TW6" xmlns:tsf="STDTSF" xmlns="STDBSC" >
            <tsf:AC_SIGNAL name="AC6" ac_ampl="1mV" freq="1kHz" />
            <TwoWire name="TW6" lo="PL1-2" hi="PL1-1" In="AC6"/>
</Signal>
```

8.3.2 Building custom signals

At some point, a BSC or TSF library component that matches the required signal will not be available. If a noisy AC_SIGNAL was required because there is a need to measure the UUT tolerance to distorted signals, the AC_SIGNAL description used in 8.3.1 would not be suitable as it does not include a noise component. To overcome this problem, it is necessary only to define what a noisy signal is and then use it within the test requirement (see Figure 55). In this example, random or white noise will be assumed.

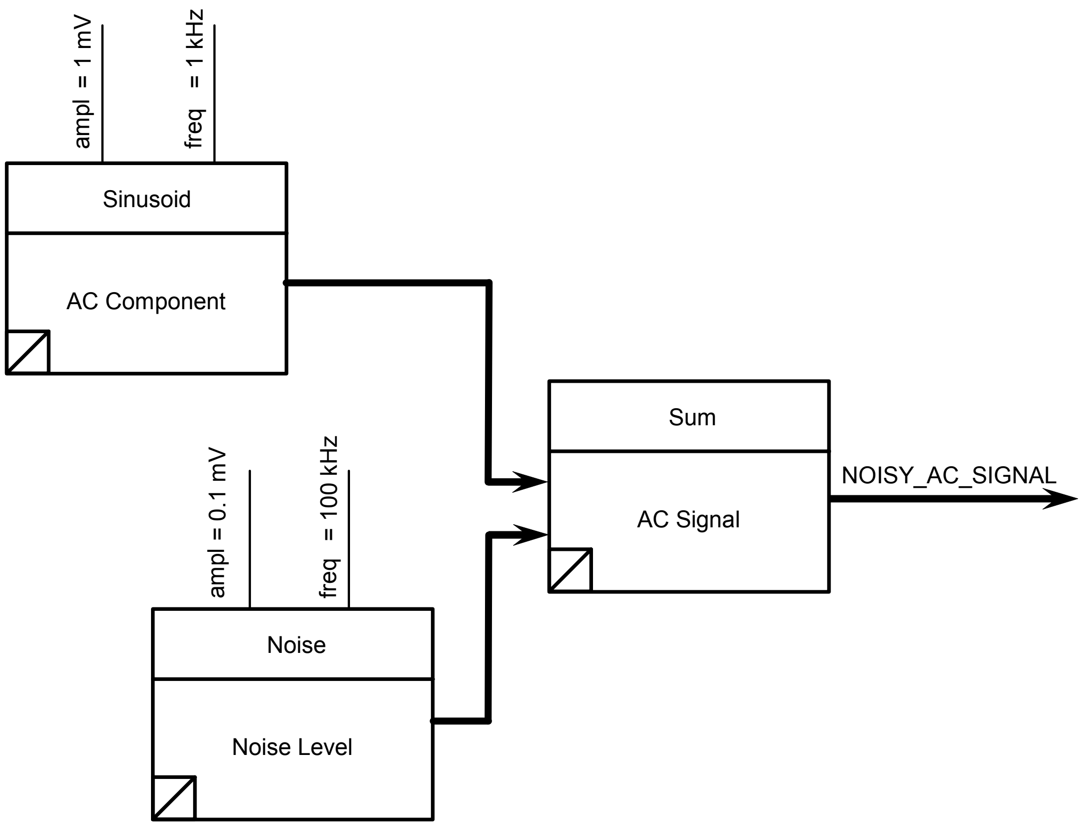


Figure 55—Noisy AC_Signal

Having defined such a signal, it can be either used directly in a test requirement or packaged into a TSF library for reuse. There is never any ambiguity of what the signal definition is because the mathematical definition in the STD standard provides a formal definition of the new signal.

8.3.2.1 Test requirement layer (using TPL and a new TSF)

```
Setup NOISY_AC_SIGNAL ampl 1 mV,
                freq 1 kHz,
                noise_ampl 0.1 mV,
                noise_freq 100 kHz,
      as source AcInput;

Connect AcInput(TwoWire) to hi PL1-1 lo PL1-2
      as connection acSig;

```

8.3.2.2 Carrier language program (using IDL)

The test requirement can also be expressed directly in a carrier language using the STD standard and its IDL definitions.

```
'define acComponent as an input
Set acComponent = STD.Require("AC_SIGNAL")
       acComponent.ac_ampl = "1 mV"
       acComponent.freq = "1 kHz"
'define NoiseLevel as an input
Set NoiseLevel = STD.Require("RANDOM_NOISE")
       NoiseLevel.ampl = "0.1 mV"
       NoiseLevel.freq = "100 kHz"

'define noisyAcSig as the sum of two components
Set noisyAcSig = STD.Require("SUM")
       noisyAcSig.In(1)= acComponent.Out
       noisyAcSig.In(2)= NoiseLevel.Out

'define acSig as a UUT connection

Set acSig = STD.Require("TwoWire")
       acSig.hi = "PL1-1"
       acSig.lo = "PL1-2"

'assign the signal output to the connection input
Set acSig.In = noisyAcSig.out

'turn the signal On
acSig.Out.Run

```

This can be expressed in an alternative (shorter) format using the XML format within the Require method (STD.Require):

```
'define Noisy AC_SIGNAL as sum of two components
Set noisyAcSig = STD.Require("SUM")
Set noisyAcSig.In(1) =STD.Require("tsf:<AC_SIGNAL ac_ampl='1mV'
freq='1kHz'/>")
Set noisyAcSig.In(2) =STD.Require("<tsf:RANDOM_NOISE ampl='0.1mV'
freq=’100kHz'/>")

'define UUT connection
Set acSig = STD.Require("<TwoWire hi='PL1-1' lo='PL1-2'/>")

```

```
'and connect signal to UUT input

Set acSig.In = noisyAcSig.Out

'Turn the signal On
acSig.Out.Run

```

8.3.2.3 XML version

XML can also be used to define the new signal.

```
<Signal name="noisyAcSig" Out="TW6" xmlns:tsf="STDTSF" xmlns=”STDBSC” >
            <tsf:AC_SIGNAL name="AC6" ac_ampl="1mV" freq="1kHz" />
            <tsf:RANDOM_NOISE name="NV8" freq="100kHz" ampl="0.1mV" />
            <Sum name="Sum8" In="AC6 NV8"/>
            <TwoWire name="TW6" lo="PL1-2" hi="PL1-1" In="Sum8"/>
</Signal>

```

This complete XML signal definition can be used by the Require method.

```
Set noisyAcSig = STD.Require(...)
noisyAcSig.Out.Run
```

8.3.2.4 C language version

The following is a typical C implementation of the same test requirement.

```
#include "STDTSF.h"
#include "STDBSC.h"
#include <my_rte.h>

#ifndef errChk
     #define errChk(fCall) if (error = (fCall), error < 0) \
     {goto Error;} else
     #endif

#ifndef CLEAR
     #define CLEAR(__handle, __func)         \
     if(__handle)                            \
     {                                       \
         __func(__handle);                   \
         __handle = 0;                       \
     }
     #endif

int main (int argc, char *argv[])
{
     int error = 0;
     ERRORINFO errorInfo;
     // Resource manager
     CAObjHandle STD = 0;
     // define acComponent as an input
     CAObjHandle acComponent = 0;
     CAObjHandle acAmpl = 0;
     CAObjHandle acFreq = 0;
```

```
     CAObjHandle noiseLevel = 0;
     CAObjHandle noiseLvlAmpl = 0;
     CAObjHandle noiseLvlFreq = 0;
     CAObjHandle iSignal = 0;
     CAObjHandle noisyACSig = 0;
     CAObjHandle acCompOut = 0;
     CAObjHandle noiseLvlOut = 0;
     CAObjHandle noisyACSigOut = 0;
     //define acSig as a UUT connection
     CAObjHandle acSig = 0;

     if (Init MY_RTE (0, argv, 0) == 0)
          return -1;    /* out of memory */

     // Create a resource manager
     errChk( STDBSC_NewIResourceManager (NULL, 1, LOCALE_NEUTRAL, 0,
&STD));

     // define acComponent as an input
     errChk( STDBSC_IResourceManagerRequire (STD, &errorInfo, "",
CA_VariantEmpty (), &acComponent));

     errChk( STDTSF_IAC_SIGNALGetac_ampl (acComponent, &errorInfo,
&acAmpl));
     errChk( STDBSC_IPhysicalSetmagnitude (acAmpl, &errorInfo, 1));
     errChk( STDBSC_IPhysicalSetunits (acAmpl, &errorInfo, "mV"));
     errChk( STDTSF_IAC_SIGNALSetByRefac_ampl (acComponent, &errorInfo,
acAmpl));

     errChk( STDTSF_IAC_SIGNALGetfreq (acComponent, &errorInfo,
&acFreq));
     errChk( STDBSC_IPhysicalSetmagnitude (acFreq, &errorInfo, 1));
     errChk( STDBSC_IPhysicalSetunits( acFreq, &errorInfo, "kHz"));
     errChk( STDTSF_IAC_SIGNALSetByReffreq (acComponent, &errorInfo,
acFreq));

     errChk( STDBSC_IResourceManagerRequire (STD, &errorInfo, "NOISE",
CA_VariantEmpty(), &noiseLevel));

     errChk( STDBSC_INoiseGetamplitude (noiseLevel, &errorInfo,
&noiseLvlAmpl));
     errChk( STDBSC_IPhysicalSetmagnitude (noiseLvlAmpl, &errorInfo,
0.1));
     errChk( STDBSC_IPhysicalSetunits (noiseLvlAmpl, &errorInfo,
"mV"));
     errChk( STDBSC_INoiseSetByRefamplitude (noiseLevel, &errorInfo,
noiseLvlAmpl));

     // define NoiseLevel as an input
     errChk( STDBSC_INoiseGetfrequency (noiseLevel, &errorInfo,
&noiseLvlFreq));
     errChk( STDBSC_IPhysicalSetmagnitude (noiseLvlFreq, &errorInfo,
100));
     errChk( STDBSC_IPhysicalSetunits( noiseLvlFreq, &errorInfo,
"kHz"));
     errChk( STDBSC_INoiseSetByReffrequency (noiseLevel, &errorInfo,
noiseLvlFreq));
```

```
     // define noisyAcSig as the sum of two components
     errChk( STDBSC_IResourceManagerRequire (STD, &errorInfo, "SUM",
CA_VariantEmpty(), &noisyACSig));
     errChk( STDTSF_IAC_SIGNALGetOut (acComponent, &errorInfo, 0,
&acCompOut));
     errChk( STDBSC_INoiseGetOut (noiseLevel, &errorInfo, 0,
&noiseLvlOut));

     errChk( STDBSC_ISumSetByRefIn (noisyACSig, &errorInfo, 1,
acCompOut));
     errChk( STDBSC_ISumSetByRefIn (noisyACSig, &errorInfo, 2,
noiseLvlOut));

     errChk( STDBSC_IResourceManagerRequire (STD, &errorInfo,
"TwoWire", CA_VariantEmpty(), &acSig));
     errChk( STDBSC_ITwoWireSetHI (acSig, &errorInfo, "PL1-1"));
     errChk( STDBSC_ITwoWireSetLO (acSig, &errorInfo, "PL1-2"));

     errChk  (STDBSC_ISumGetOut (noisyACSig, &errorInfo, 0,
&noisyACSigOut));
     //assign the signal output to the connection input
     errChk( STDBSC_ITwoWireSetByRefIn (acSig, &errorInfo, 0,
noisyACSigOut));
     //turn the signal On
     errChk( STDBSC_ITwoWireGetOut (acSig, &errorInfo, 0, &iSignal));
     errChk( STDBSC_ISignalRun (iSignal, &errorInfo, 0));


Error:

     CLEAR (iSignal, CA_DiscardObjHandle);
     CLEAR (acSig, CA_DiscardObjHandle);
     CLEAR (acCompOut, CA_DiscardObjHandle);
     CLEAR (noiseLvlOut, CA_DiscardObjHandle);
     CLEAR (noisyACSigOut, CA_DiscardObjHandle);
     CLEAR (acFreq, CA_DiscardObjHandle);
     CLEAR (acAmpl, CA_DiscardObjHandle);
     CLEAR (acComponent, CA_DiscardObjHandle);
     CLEAR (noiseLvlAmpl, CA_DiscardObjHandle);
     CLEAR (noiseLvlFreq, CA_DiscardObjHandle);
     CLEAR (noiseLevel, CA_DiscardObjHandle);
     CLEAR (noisyACSig, CA_DiscardObjHandle);
     CLEAR (STD, CA_DiscardObjHandle);

     return error;
}
```

8.4 ATML Test Description

The ATML Test Description standard [B11] provides a method of describing all the information required to implement a complete set of tests in a neutral format. It does not require any reference to specific test equipment. This is completely in line with the objectives of IEEE 1641. ATML uses IEEE 1641 as a method of defining signals. ATML Test Description is an updated replacement for the traditional test requirement document (TRD) standards that were maintained by the Department of Defense (DOD). Using

ATML with IEEE 1641 not only provides complete test requirement information, but also facilitates the exchange of data between the various components of an ATS.

It is outside of the scope of this document to describe in any detail how ATML is used, but information and examples of the use of IEEE 1641 with ATML are available in the ATML standards. For information on how IEEE 1641 may be used in ATML test descriptions, see IEEE Std 1671.1–2009, B3.3.

8.5 Examples of test requirements using TPL, ATLAS, and native languages

The following examples show different test requirements expressed in TPL, a COTS programming environment, and in ATLAS (specifically IEEE 716 C/ATLAS). The examples serve to illustrate how the STD standard can be implemented to define UUT test requirements using a variety of platforms and languages. The TPL examples include references to “preamble” and “procedural” parts, which have been included to assist with the comparison to the ATLAS versions.

8.5.1 DC amplifier example

In this example, the UUT test requirements are defined as follows:

a) Apply power 28 V dc between pins PL1-1 and PL1-2.

b) Apply power 20 V dc between pins PL1-3 and PL1-4.

c) Apply a dc voltage of 0.5 V between pins J1-1 and J1-2.

d) Measure the voltage across output pins J2-1 and J2-2.

e) Calculate the voltage gain between the 0.5 V input and the measured output.

Figure 56 represents a diagram of the UUT.

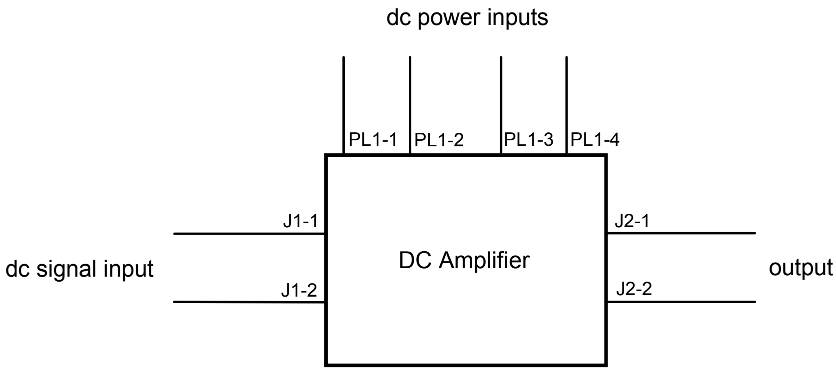


Figure 56 —DC amplifier example

8.5.1.1 TPL within a carrier language

This example shows how to express the dc amplifier requirement using an undefined carrier language. The lines of code in italic text would be expressed in a user-selected carrier language. The example test requirement shows how to mix TPL signal definitions with procedural and output statements using the carrier language.

```
/*********************************************************************/
/*     Test Methodology: TPL
       UUT Type:          DC Amplifier
       Model:             DC-01                                    */
/*********************************************************************/
/* In this Example:
     the Roman Text is Test Procedure Language Code
     the Italic Text is Carrier Language (including comments)       */
//
/*********************************************************************/
/* Begin the preamble part                                          */
/*********************************************************************/
//
single gain
  test PowerUp(single Vin-1, Vin-2)
 {
  /*<TPL>
  Setup DC_SIGNAL dc_ampl Vin1 range 16 V to 32 V
      as source Power1 ;
  Setup DC_SIGNAL dc_ampl Vin2 range 16 V to 32 V
      as source Power2 ;
            Connect Power1 to hi PL1-1 lo PL1-2;
            Connect Power2 to hi PL1-3 lo PL1-4;
  </TPL> */
  return() ;
}
//
  test GainTest(single Vin)
{
  single Vout;
  /*<TPL>
  Setup DC_SIGNAL ampl Vin range 0 V to 20 V
      as source SignalIn ;
  Setup DC_SIGNAL ampl range 0 V to 30 V
      as sensor dcVoltMeter ;
  Connect SignalIn to hi J1-1 lo J1-2;
  Connect hi J2-1 lo J2-2 to dcVoltMeter;
  Read dcVoltMeter into Vout;
  </TPL> */
return(Vout/Vin);
}
//
/*******************************************************************/
/* Begin the procedural part
   Start testing the UUT                                          */
/*******************************************************************/
//
output("Start of TPL Program 'DC_AMP' " ) ;
powerUp(28.0, 20.0);
gain=GainTest(0.5);
output("Gain is",gain);
/*<TPL>
remove all;
</TPL>*/
```

```
output("End of TPL Program 'DC_AMP' " ) ;
//
/*******************************************************************/
/* End of testing
*/
/*******************************************************************/
```

8.5.1.2 Embedded TPL in a native carrier language

This example shows an optimized requirement for the dc amplifier using a carrier language. It illustrates how Disconnect or Reset statements may be omitted at the end of a program. The TPL verbs create SignalFunctions and control them through their signal control interface. Within the TPL, signals are turned on with an internal statement like "<object>.Out.Run". To remove or disconnect the signal, an equivalent "<object>.Out.Stop" may be used. All SignalFunction objects are deleted when a program exits. The effect is that as a program exits, it tidies up all variables and references to SignalFunctions. This step, in turn, informs the SignalFunction implicitly that it is no longer used, then resets any signals, and thus disconnects or removes the signal or connection.

The TPL statements are introduced and annotated via carrier language comments, which for this carrier language are prefixed by the single quote character (').

```
'*******************************************************************
'     Test Methodology: TPL
'     UUT Type:         DC Amplifier
'     Model:            DC-01
'*******************************************************************
'  In this Example:
'    the Roman Text is Test Procedure Language Code
'    the Italic Text is Carrier Language (including comments)
'*******************************************************************
'  Begin the preamble part (containing declarative statements)
'*******************************************************************
'
Subroutine PowerUp Vin-1, Vin-2
   '<TPL>
   'Setup DC_SIGNAL ampl Vin-1 range 16V to 32V as source Power1;
   'Setup DC_SIGNAL ampl Vin-2 range 16V to 32V as source Power2;
   'Connect Power1 to hi PL1-1 lo PL1-2;
   'Connect Power2 to hi PL1-3 lo PL1-4;
   '</TPL>
End Subroutine
'
Subroutine PowerDown
   '<TPL>
   'Disconnect Power1;
   'Disconnect Power2;
   'Reset Power1;
   'Reset Power2;
   '</TPL>
End Subroutine
'
Function GainTest(Vin) as Double
   '<TPL>
   'Setup DC_SIGNAL ampl Vin range 0V to 20V as source Signal-In;
   'Setup DC_SIGNAL ampl range 0V to 30V as sensor DC-VoltMeter;
```

```
   'Connect SignalIn to hi J1-1 lo J1-2;
   'Connect hi J2-1 lo J2-2 to dcVoltMeter;
   'Read dcVoltMeter into Vout;
   '</TPL>
   GainTest = Vout/Vin
End Function
'
'********************************************************************
'  End the preamble part
'********************************************************************
'
'********************************************************************
'  Begin the procedural part (containing the UUT test statements)
'  that calls the declarative statements in the preamble part
'********************************************************************
'
'********************************************************************
'  Start testing the UUT
'********************************************************************
'
'     Inform user: Begin testing
Output " Begin STD Program DC_AMP "
'
'     Power up the UUT
PowerUp  Vin1:=28.0, Vin2:=20.0

'     Start the amplifier test
Gain = GainTest(Vin:=0.5);
'     Output the result information to the test operator
Output " Gain is Vout/Vin " & Gain
'     Remove all connections and
'     settings of the test instrumentation.
PowerDown
'     Inform user:  End of UUT testing
Output " End STD Program DC_AMP "
'
'********************************************************************
'  End of Testing
'********************************************************************
```

8.5.1.3 Native language program using STD components

This example builds a complete program that defines the dc amplifier test requirement. The program defines a signal model and then controls the model using the Run methods. The style of the program is not optimal, but follows the same layout of previous examples to allow direct comparison and mapping of TPL components. Additional non-signal statements such as user interaction are also included.

The code is written without the support of the TPL. It makes use of a resource object called STD. The reason for the need of a resource object is that any development environment that uses the STD standard cannot restrict itself to only one source that can create building block objects. It would be like locking oneself into a single implementer and then never being able to change. Therefore, by having all building blocks created through some resource object, the user can have multiple STD implementations on the same machine.

```
'********************************************************************
```

```
'      Test Methodology: TPL
'      UUT Type:         DC Amplifier
'      Model:            DC-01
'*********************************************************************
'   Begin the preamble part (containing declarative statements)
'*********************************************************************
'
Set STD = CreateObject("STD.Resource")
'
'Setup DC_SIGNAL ampl Vin1 range 16V to 32V as source Power1;
Set Power1 = STD.Require ("DC-SIGNAL")
    Power1.dc_ampl = Vin1 & " range 16 V to 32 V"
'Setup DC_SIGNAL ampl Vin2 range 16V to 32V as source Power2;
Set Power2 = STD.Require ("DC-SIGNAL")
    Power2.dc_ampl = Vin2 & " range 16 V to 32 V"
'
'Connect Power1 to hi PL1-1 lo PL1-2;
Set Power1Cnx = STD.Require ("TwoWire")
    Power1Cnx.hi = "PL1-1"
    Power1Cnx.lo = "PL1-2"
Power1Cnx.In = Power1.Out
'
'Connect Power2 to hi PL1-3 lo PL1-4;
Set Power2Cnx = STD.Require ("TwoWire")
    Power2Cnx.hi = "PL1-3"
    Power2Cnx.lo = "PL1-4"
Power2Cnx.In = Power2.Out
'
'Setup DC_SIGNAL ampl Vin range 0V to 20V as source Signal-In;
Set SignalIn = STD.Require ("DC-SIGNAL")
    SignalIn.dc_ampl = Vin & " range 0 V to 20 V"
'
'Connect SignalIn to hi J1-1 lo J1-2;
Set SignalInCnx = STD.Require ("TwoWire")
    SignalInCnx.hi = "J1-1"
    SignalInCnx.lo = "J1-2"
SignalInCnx.In = SignalIn.Out
'
'Setup DC_SIGNAL ampl range 0V to 30V as sensor DC-VoltMeter;
Set dcVoltMeter = STD.Require ("Measure")
    dcVoltMeter.attribute = "ampl"
Set dcVoltMeter.As = STD.Require ("DC_SIGNAL")
    dcVoltMeter.As.ampl = "av range 0 V to 30 V"
'
'Connect hi J2-1 lo J2-2 to DC-VoltMeter;
Set dcVoltMeterCnx = STD.Require ("TwoWire")
    dcVoltMeterCnx.hi = "J2-1"
    dcVoltMeterCnx.lo = "J2-2"
dcVoltMeter.In = dcVoltMeterCnx.Out
'
'*********************************************************************
'   End the preamble part
'*********************************************************************
'
'*********************************************************************
'   Begin the procedural part (containing the UUT test statements)
'   that calls the declarative statements in the preamble part
```

```
'********************************************************************
'
'********************************************************************
'  Start testing the UUT
'********************************************************************
'
'      Inform user: Begin testing
Output " Begin STD Program DC_AMP "
'
'      Power up the UUT
Vin1 = 28.0
Vin2 = 20.0
Power1.Out.Run
Power2.Out.Run
'
'      Start the amplifier test
Vin = 0.5
SignalIn.Out.Run
'
'Read dcVoltMeter into Vout;
dcVoltMeter.Out.Run
Vout = dcVoltMeter.Measurement
Gain = Vout/Vin
'
'      Output the result information to the test operator
Output " Gain is " & Gain
'
'      Remove the signals
'
SignalIn.Out.Stop
Power2.Out.Stop
Power1.Out.Stop
'
'      Inform user:  End of UUT testing
Output " End STD Program DC_AMP "
'
'********************************************************************
'  End of Testing
'********************************************************************
```

8.5.1.4 C language version

The following is a typical C implementation of the same test requirement.

```
#include "STDTSF.h"
#include "STDBSC.h"
#include <ansi_c.h>
#include <my_rte.h>

// ******************************************************************
//      Test Methodology: TPL
//      UUT Type: DC Amplifier
//      Model: DC-01
//
********************************************************************
```

```
int main (int argc, char *argv[])
{
     int error = 0;
     ERRORINFO errorInfo;
     // Resource manager
     CAObjHandle STD = 0;

     CAObjHandle iSignal = 0;
     CAObjHandle power1 = 0;
     CAObjHandle power1Out = 0;
     CAObjHandle power1Ampl = 0;
     CAObjHandle power2 = 0;
     CAObjHandle power2Out = 0;
     CAObjHandle power2Ampl = 0;
     CAObjHandle signalIn = 0;
     CAObjHandle signalInAmpl = 0;
     CAObjHandle signalInOut = 0;
     CAObjHandle dcVoltMeter = 0;
     CAObjHandle dcVoltMeterAmpl = 0;
     CAObjHandle dcVoltMeterOut = 0;
     CAObjHandle power1Cnx = 0;
     CAObjHandle power2Cnx = 0;
     CAObjHandle signalInCnx = 0;
     CAObjHandle dcVoltMeterCnx = 0;

     VARIANT power1Val = CA_VariantNULL();
     VARIANT power2Val = CA_VariantNULL();
     VARIANT signalInVal = CA_VariantNULL();
     VARIANT dcVoltMeterVal = CA_VariantNULL();
     VARIANT measValVT = CA_VariantNULL();
     double  measVal = 0;

     if (InitMY_RTE (0, argv, 0) == 0)
          return -1;    /* out of memory */

     CA_VariantSetCString (&power1Val, "Vin1 range 16 V to 32 V");
     CA_VariantSetCString (&power2Val, "Vin2 range 16 V to 32 V");
     CA_VariantSetCString (&signalInVal, "Vin range 0 V to 20 V");
     CA_VariantSetCString (&dcVoltMeterVal, "av range 0 V to 30 V");

     // Create a resource manager
     STDBSC_NewIResourceManager (NULL, 1, LOCALE_NEUTRAL, 0, &STD);

     // Setup DC_SIGNAL ampl Vin1 range 16V to 32V as source Power1;
     STDBSC_IResourceManagerRequire (STD, NULL, "DC-SIGNAL",
CA_VariantEmpty(), &power1);
     STDTSF_IDC_SIGNALGetac_ampl (power1, NULL, &power1Ampl);
     STDBSC_IPhysicalSetvalue (power1Ampl, NULL, power1Val);
     STDTSF_IDC_SIGNALSetByRefac_ampl (power1, NULL, power1Ampl);
     STDTSF_IDC_SIGNALGetOut (power1, NULL, 0, &power1Out);

     // Setup DC_SIGNAL ampl Vin2 range 16V to 32V as source Power2;
     STDBSC_IResourceManagerRequire (STD, NULL, "DC-SIGNAL",
CA_VariantEmpty(), &power2);
     STDTSF_IDC_SIGNALGetac_ampl (power2, NULL, &power2Ampl);
     STDBSC_IPhysicalSetvalue (power2Ampl, NULL, power2Val);
```

```
    STDTSF_IDC_SIGNALSetByRefac_ampl (power2, NULL, power2Ampl);
    STDTSF_IDC_SIGNALGetOut (power2, NULL, 0, &power2Out);

    // Connect Power1 to hi PL1-1 lo PL1-2;
    STDBSC_IResourceManagerRequire (STD, NULL, "TwoWire",
CA_VariantEmpty(), &power1Cnx);
    STDBSC_ITwoWireSetHI (power1Cnx, NULL, "PL1-1");
    STDBSC_ITwoWireSetLO (power1Cnx, NULL, "PL1-2");
    STDBSC_ITwoWireSetByRefIn (power1Cnx, NULL, 0, power1Out);

    // Connect Power2 to hi PL1-3 lo PL1-4;
    STDBSC_IResourceManagerRequire (STD, NULL, "TwoWire",
CA_VariantEmpty(), &power2Cnx);
    STDBSC_ITwoWireSetHI (power2Cnx, NULL, "PL1-3");
    STDBSC_ITwoWireSetLO (power2Cnx, NULL, "PL1-4");
    STDBSC_ITwoWireSetByRefIn (power2Cnx, NULL, 0, power2Out);

    // Setup DC_SIGNAL ampl Vin range 0V to 20V as source Signal-In;
    STDBSC_IResourceManagerRequire (STD, NULL, "DC-SIGNAL",
CA_VariantEmpty(), &signalIn);
    STDTSF_IDC_SIGNALGetac_ampl (signalIn, NULL, &signalInAmpl);
    STDBSC_IPhysicalSetvalue (signalInAmpl, NULL, signalInVal);
    STDTSF_IDC_SIGNALSetByRefac_ampl (signalIn, NULL, signalInAmpl);
    STDTSF_IDC_SIGNALGetOut (signalIn, NULL, 0, &signalInOut);

    // Connect SignalIn to hi J1-1 lo J1-2;
    STDBSC_IResourceManagerRequire (STD, NULL, "TwoWire",
CA_VariantEmpty(), &signalInCnx);
    STDBSC_ITwoWireSetHI (signalInCnx, NULL, "J1-1");
    STDBSC_ITwoWireSetLO (signalInCnx, NULL, "J1-2");
    STDBSC_ITwoWireSetByRefIn (signalInCnx, NULL, 0, signalInOut);

    // Setup DC_SIGNAL ampl range 0V to 30V as sensor DC-VoltMeter;
    STDBSC_IResourceManagerRequire (STD, NULL, "DC-SIGNAL",
CA_VariantEmpty(), &dcVoltMeter);
    STDTSF_IDC_SIGNALGetac_ampl (dcVoltMeter, NULL, &dcVoltMeterAmpl);
    STDBSC_IPhysicalSetvalue (dcVoltMeterAmpl, NULL, dcVoltMeterVal);
    STDTSF_IDC_SIGNALSetByRefac_ampl (dcVoltMeter, NULL,
dcVoltMeterAmpl);
    STDTSF_IDC_SIGNALGetOut (dcVoltMeter, NULL, 0, &dcVoltMeterOut);

    // Connect hi J2-1 lo J2-2 to DC-VoltMeter;
    STDBSC_IResourceManagerRequire (STD, NULL, "TwoWire",
CA_VariantEmpty(), &dcVoltMeterCnx);
    STDBSC_ITwoWireSetHI (dcVoltMeterCnx, NULL, "J2-1");
    STDBSC_ITwoWireSetLO (dcVoltMeterCnx, NULL, "J2-2");
    STDBSC_ITwoWireSetByRefIn (dcVoltMeterCnx, NULL, 0,
dcVoltMeterOut);

    // Inform user: Begin testing
    printf(" Begin STD Program DC_AMP \n");

    // Power up the UUT
    STDBSC_IPhysicalSetmagnitude (power1Ampl, NULL, 28.0);
    STDTSF_IDC_SIGNALSetByRefac_ampl (power1, NULL, power2Ampl);
    STDBSC_ISignalRun (power1Out, NULL, 0);
    STDBSC_IPhysicalSetmagnitude (power2Ampl, NULL, 20.0);
```

```
    STDTSF_IDC_SIGNALSetByRefac_ampl (power2, NULL, power2Ampl);
    STDBSC_ISignalRun (power2Out, NULL, 0);

    // Start the amplifier test
    STDBSC_IPhysicalSetmagnitude (signalInAmpl, NULL, 0.5);
    STDTSF_IDC_SIGNALSetByRefac_ampl (signalIn, NULL, signalInAmpl);
    STDBSC_ISignalRun (signalInOut, NULL, 0);

    // Read dcVoltMeter into Vout;
    STDBSC_ISignalRun (dcVoltMeterOut, NULL, 0);
    STDBSC_ISensorGetmeasurement (dcVoltMeterOut, NULL, &measValVT);
    CA_VariantSetDouble (&measValVT, measVal);

    printf("Gain is %d\n", measVal/0.5);

    STDBSC_ISignalStop (signalInOut, NULL, 0);
    STDBSC_ISignalStop (power1Out, NULL, 0);
    STDBSC_ISignalStop (power2Out, NULL, 0);

    // Inform user: End of UUT testing
    printf(" End STD Program DC_AMP \n");

Error:
    // Clear declared variables

    return error;
}
```

8.5.1.5 DC amplifier example in C/ATLAS

The following version of the example is in C/ATLAS and is provided as a reference to users who are familiar with test requirements written in ATLAS.

```
 000010 BEGIN, ATLAS PROGRAM 'DC_AMP' $
C$
C*****************************************************************
*     Test Methodology: C/ATLAS (716-1995)                        *
*     UUT Type:         DC Amplifier                              *
*     Model:            DC-01                                     $
C*****************************************************************
*  Begin the preamble part (containing declarative statements)    *
*****************************************************************$
C$
 001000 DECLARE, VARIABLE, 'Gain' IS DECIMAL $
C$
 005000 DEFINE, 'POWER-UP', PROCEDURE ('VIN-1','VIN-2' IS DECIMAL)$
    10    APPLY, DC SIGNAL,
             VOLTAGE 'VIN-1' RANGE 16 V TO 32 V,
             CNX HI PL1-1 LO PL1-2 $
    20    APPLY, DC SIGNAL,
             VOLTAGE 'VIN-2' RANGE 16 V TO 32 V,
             CNX HI PL1-3 LO PL1-4 $
    30 END, 'POWER-UP' $
C$
 006000 DEFINE, 'GAIN TEST', PROCEDURE ('VIN' IS DECIMAL)
```

```
                                RESULT ('GAIN' IS DECIMAL) $
     10    DECLARE, VARIABLE, 'VOUT' IS DECIMAL $
     20    APPLY, DC SIGNAL,
              VOLTAGE 'VIN' RANGE 0 V TO 20 V,
              CNX HI J1-1 LO J1-2 $
     30    MEASURE, (VOLTAGE INTO 'VOUT'), DC SIGNAL,
              VOLTAGE RANGE 0 V TO 30 V,
              CNX HI J2-1 LO J2-2 $
     40    CALCULATE, 'Gain' = 'VOUT' / 'VIN' $
     50 END, 'GAIN TEST' $
C$
C******************************************************************
*  End the preamble part
******************************************************************$
C$
C******************************************************************
*  Begin the procedural part (containing the UUT test statements)
*  that calls the declarative statements in the preamble part
******************************************************************$
C$
E100000 PERFORM, 'POWER-UP' (28, 24) $
     10 PERFORM, 'GAIN TEST' (0.5) RESULT ('GAIN') $
     20 OUTPUT, C' GAIN IS ', 'Gain' $
 900000 REMOVE, ALL $
     90 FINISH $
C$
C******************************************************************
*   End of procedural part
******************************************************************$
C$
 999999 TERMINATE, ATLAS PROGRAM 'DC_AMP' $
```

8.5.2 AC amplifier example

The following example is a UUT test requirement for a simplified ac amplifier. The test requirements are defined as follows (note that this is from a legacy program and that the “voltmeter” may be taken to mean an “ac signal sensor”):

a) Apply a 20 mV, 100 Hz ac signal to the input of the UUT.

b) Connect an ac voltmeter (capable of measuring up to 5 V in the range of 100 Hz to 1000 Hz) to the output of the UUT.

c) Increment the frequency of the input signal from 100 Hz to 1000 Hz in steps of 100 Hz.

d) At each frequency, measure the output voltage and calculate the gain, which should be between 195 and 205.

e) If (at any frequency) the gain is out of range, instruct the operator to increase or decrease the gain control and stop testing.

8.5.2.1 TPL within carrier language

The following example shows a dynamic test requirement. The lines of code in italic text would be expressed in a user-selected carrier language.

```
/**********************************************************************/
/*    Test Language: TPL
      UUT Type:      AC Amplifier
      Model:         AC-01                                            */
/**********************************************************************/
/**********************************************************************/
/* In this Example:
     the Roman Text is Test Procedure Language Code
     the Italic Text is Carrier Language (including comments)         */
//
/**********************************************************************/
/ * Begin the preamble part                                            */
/**********************************************************************/
//
function Diagnosis1()
output("Adjust Gain Control – Lower");
//
function Diagnosis2()
output("Adjust Gain Control – Higher");
//
/**********************************************************************/
/*    End the preamble part                                            */
/**********************************************************************/
//
output("Start Testing") ;
/*<TPL>
Setup AC_SIGNAL ac_ampl 20.0 mV errlmt +-0.5%,
                freq   FreqVal  errlmt +-5.0% range 100Hz to 1000Hz
      as source AC1;
Connect  AC1  to hi SKT1-1 lo SKT1-2;
Setup AC_SIGNAL ac_ampl range MAX 5 V,
      as sensor rmsVolts;
Connect  hi SKT2-2 lo SKT2-4 to rmsVolts;
  <TPL/> */
//
for FreqVal= 100.0; 100.0 < FreqVal > 1000.0; FreqVal ++100.0
{
      /*<TPL>
      Change AC1 freq FreqVal;
      Read rmsVolts into Vout;
        <TPL/> */
      Gain = Vout/0.02;
      if Gain >= 205
      {
         Diagnosis1();
         break;
      }
      if Gain <= 195
      {
         Diagnosis2();
         break;
      }
}
/*<TPL>
Disconnect all;
Reset all;
  <TPL/> */
```

```
output("End of AC Amplifier Gain Test");
//
//
/*******************************************************************/
/*      End of testing                                             */
/*******************************************************************/
```

8.5.2.2 Embedded TPL in native language

Any test requirement capture will need to compare measured values against expected values. The Compare statement is interesting because the Compare method acts not on the measurement variable but on the measurement object.

The As clause is used to identify the signal object. Remember that the signal graph remains regardless of whether a signal is present. The graph does not define the signal at that time; only at some time in the future when it is run will the signal exist.

Note that the TPL statements are embedded using a carrier language comment, in this case the single quote character (').

```
'*******************************************************************
'     Test Methodology: TPL
'     UUT Type:         AC Amplifier
'     Model:            AC-01
'*******************************************************************
'  In this Example:
'    the Roman Text is Test Procedure Language Code
'    the Italic Text is Carrier Language (including comments)
'*******************************************************************
'  Begin the preamble part (containing declarative statements)
'*******************************************************************
'
Subroutine Diagnosis1
            Output "Adjust Gain Control – Lower"
End Subroutine

Subroutine Diagnosis2
            Output "Adjust Gain Control – Higher"
End Subroutine
'
'*******************************************************************
'     End the preamble part
'*******************************************************************
'
'*******************************************************************
'     Begin the procedural part (containing the UUT test statements)
'     that calls the declarative statements in the preamble part
'*******************************************************************
'
'*******************************************************************
'     Start testing the UUT                     *
'*******************************************************************
'
'     Inform user: Begin testing
Output " Begin AC Amplifier Gain Test "
```

```
   '<TPL>
   'Setup AC_SIGNAL ac_ampl 20.0 mV errlmt +-0.5%,
                    freq   FreqVal  errlmt +-5.0% range 100Hz to 1000Hz
                as source AC1;
   'Connect AC1 to hi SKT1-1 lo SKT1-2;

   'Setup AC_SIGNAL ac_ampl range MAX 5 V,
          as sensor rmsVolts;
   'Connect hi SKT2-2 lo SKT2-4 to rmsVolts;
   '<TPL/>
'
' Calculate measurement values for required Gain limits
'Measurement Upper Limit = Gain Upper Limit * Input
MeasUpperLimit = 205 * 0.02
'Measurement Lower Limit = Gain Lower Limit * Input
MeasLowerLimit = 195 * 0.02
'
For FreqVal = 100 To 1000 Step 100
   '<TPL>
   'Change AC1 freq FreqVal;
   'Read rmsVolts into Vout;
   'Compare rmsVolts UL MeasUpperLimit LL MeasLowerLimit;
   '<TPL/>
   If rmsVolts.NOGO Then
      If rmsVolts.HI Then
         Output "Gain too High"
         Diagnosis1
      Else
         Output "Gain too Low"
         Diagnosis2
      End if
      Exit Next
   End If
Next
'
' Remove the signals
'<TPL>
'Disconnect all;
'Reset all;
'<TPL/>
output("End of AC Amplifier Gain Test");
'
'*******************************************************************/
'*      End of testing                                             */
'*******************************************************************/
```

8.5.2.3 Native language program using STD basic components

If the requirement were to match an original control language rather than define the test specification, it is still possible to do so by using the **Conn** reference of the SignalFunction. **Conn** references are similar to **In** references in that they provide connection information for signal statements. Nevertheless, it does not infer any signal control; therefore, the programmer is fully responsible for making sure that everything is connected in the correct order. This feature enables the mapping of existing dynamic test requirement semantics.

The previous example and this one show how, when using dynamic test requirements, test semantics can be subtly changed, in this case, to optimize the usage of connection devices by changing the signal models.

Note the sense object rmsVolts must use either **In** or **Conn** in order to take the measurement.

a) Using **In**

```
rmsVolts.In = rmsVoltsCnx.Out

rmsVolts.Out.Run                    ' Connect (SKT2-2 SKT2-4) and take measurement
```

b) Using **Conn**

```
rmsVolts.Conn = rmsVoltsCnx.Out

rmsVoltsCnx.Out.Run                 ' Connect (SKT2-2 SKT2-4) for measurement

rmsVolts.Run (maxTime)              ' Take measurement
```

This example uses **Conn**.

```
'*********************************************************************
'     Test Methodology: Basic
'     UUT Type:         AC Amplifier
'     Model:            AC-01
'*********************************************************************
'  Begin the preamble part (containing declarative statements)
'*********************************************************************
'
Subroutine Diagnosis1
            Output "Adjust Gain Control – Lower"
End Subroutine
'
Subroutine Diagnosis2
            Output "Adjust Gain Control – Higher"
End Subroutine
'
'*********************************************************************
'  Define Model
'*********************************************************************
Set STD = CreateObject("STD.Resource")
'
'Setup AC_SIGNAL ac_ampl 20.0 mV errlmt +-0.5%,
'                 freq    FreqVal errlmt +-5.0% range 100Hz to 1000Hz
'            '   as source AC1;
Set AC1 = STD.Require ("AC_SIGNAL")
    AC1.ac_ampl = " 20 mV  errlmt 0.5%"
    ACl.freq    = "FreqVal errlmt 5% range 100Hz to 1000Hz"
'
'Connect AC1 to hi SKT1-1 lo SKT1-2;
Set AC1Cnx = STD.Require ("TwoWire")
    AC1Cnx.hi = "SKT1-1"
    AC1Cnx.lo = "SKT1-2"
AC1Cnx.In = AC1.Out
```

```
Set freqSource = AC1Cnx
'
'Setup AC-SIGNAL ac_ampl range MAX 5 V,
'         as sensor rmsVolts;
Set rmsVolts = STD.Require ("Measure")
    rmsVolts.attribute = "ac_ampl"
Set rmsVolts.As = STD.Require ("AC_SIGNAL")
    rmsVolts.As.ac_ampl = "trms range 0 V to 5 V"
'
'Connect hi SKT2-2 lo SKT2-4 to rmsVolts;
Set rmsVoltsCnx = STD.Require ("TwoWire")
    rmsVoltsCnx.hi = "SKT2-2"
    rmsVoltsCnx.lo = "SKT2-4"
rmsVolts.Conn = rmsVoltsCnx.Out
'
'********************************************************************
'     End the preamble part
'********************************************************************
'
'********************************************************************
'     Begin the procedural part (containing the UUT test statements)
'     that calls the declarative statements in the preamble part
'********************************************************************
'
'********************************************************************
'     Start testing the UUT
'********************************************************************
'
'     Inform user: Begin testing
Output " Begin AC Amplifier Gain Test "
freqSouce.Out.Run              'Apply Frequency Source
rmsVoltsCnx.Out.Run            'Connect SKT2-2 SKT2-4 for measurement
'
' Calculate measurement values for required Gain limits
'Measurement Upper Limit = Gain Upper Limit * Input
MeasUpperLimit = 205 * 0.02
'Measurement Lower Limit = Gain Lower Limit * Input
MeasLowerLimit = 195 * 0.02
'
'timeout value for waiting for measurements
maxTime = 2000 '2secs
'
For FreqVal = 100 To 1000 Step 100
   freqSource.freq = FreqVal;
   freqSource.Out.Change        'Change output signal of Freq Source
   rmsVolts.Run(maxTime)        'Take measurement
   'Read rmsVolts into Vout;
   Vout = rmsVolts.Measurement
   Gain = Vout/0.02
   ' Output the Gain for the user
   Output "Amplifier Gain = " & Gain

   'Compare rmsVolts UL MeasUpperLimit LL MeasLowerLimit;
   rmsVolts.UL = MeasUpperLimit
   rmsVolts.LL = MeasLowerLimit
   '
   If rmsVolts.NOGO Then
```

```
      If rmsVolts.HI Then
         Output "Gain too High"
         Diagnosis1
      Else
         Output "Gain too Low"
         Diagnosis2
      End if
      Exit Next
   End If
Next
'
rmsVoltsCnx.Out.Stop              'Disconnect SKT2-2 SKT2-4
freqSouce.Out.Stop                'Remove Frequency Source
'
'     Inform user: End of testing
Output "End of AC Amplifier Gain Test"
'
'*******************************************************************
'      End of testing
'*******************************************************************
```

8.5.2.4 C language version

The following is a typical C implementation of the same test requirement.

```
#include "STDTSF.h"
#include "STDBSC.h"
#include <ansi_c.h>
#include <my_rte.h>

// *******************************************************************
//      Test Methodology: TPL
//      UUT Type: DC Amplifier
//      Model: DC-01
//
*******************************************************************

void Diagnosis1()
{
     printf ("Adjust Gain Control Lower\n");
}
void Diagnosis2()
{
     printf ("Adjust Gain Control Higher\n");
}


int main (int argc, char *argv[])
{
     int error = 0;
     ERRORINFO errorInfo;
     // Resource manager
     CAObjHandle STD = 0;
     CAObjHandle power1 = 0;
     CAObjHandle power1Ampl = 0;
```

```
      CAObjHandle AC1 = 0;
      CAObjHandle AC1Ampl = 0;
      CAObjHandle AC1Freq = 0;
      CAObjHandle AC1Cnx = 0;
      CAObjHandle AC1Out = 0;
      CAObjHandle rmsVolts = 0;
      CAObjHandle rmsVolts_Sig = 0;
      CAObjHandle rmsVolts_Sig_Ampl = 0;
      CAObjHandle rmsVoltsCnx = 0;
      CAObjHandle rmsVoltsCnxOut = 0;
      CAObjHandle freqSource = 0;
      CAObjHandle freqSource_Freq = 0;
      CAObjHandle rmsVolts_UL = 0;
      CAObjHandle rmsVolts_LL = 0;
      VARIANT power1Val = CA_VariantNULL();
      VARIANT AC1_Ampl_Val = CA_VariantNULL();
      VARIANT AC1_Freq_Val = CA_VariantNULL();
      VARIANT rmsVolts_Sig_Ampl_Val = CA_VariantNULL();
      VARIANT measValVT = CA_VariantNULL();
      VBOOL vBoolNOGO = VARIANT_FALSE;
      VBOOL vBoolHi = VARIANT_FALSE;

      double MeasUpperLimit = 0.0;
      double MeasLowerLimit = 0.0;
      int FreqVal = 0;
      double measVal = 0.0;
      double Gain = 0.0;

      if (InitMY_RTE (0, argv, 0) == 0)
           return -1;    /* out of memory */

      // Create a resource manager
      STDBSC_NewIResourceManager (NULL, 1, LOCALE_NEUTRAL, 0, &STD);

      CA_VariantSetCString (&power1Val, "Vin-1 range 16 V to 32 V");
      CA_VariantSetCString (&AC1_Ampl_Val, "20 mV errlmt 0.5%");
      CA_VariantSetCString (&AC1_Freq_Val, "FreqVal errlmt 5% range
100Hz to 1000Hz");
      CA_VariantSetCString (&rmsVolts_Sig_Ampl_Val, "trms range 0 V to 5
V");

      // Setup DC_SIGNAL ampl Vin-1 range 16V to 32V as source Power1;
      STDBSC_IResourceManagerRequire (STD, NULL, "DC-SIGNAL",
CA_VariantEmpty(), &power1);
      STDTSF_IDC_SIGNALGetac_ampl (power1, NULL, &power1Ampl);
      STDBSC_IPhysicalSetvalue (power1Ampl, NULL, power1Val);
      STDTSF_IDC_SIGNALSetByRefac_ampl (power1, NULL, power1Ampl);

      // Setup AC_SIGNAL ac_ampl 20.0 mV errlmt +-0.5%, freq FreqVal
errlmt +-5.0% range 100Hz to 1000Hz
      // as source AC1;
      STDBSC_IResourceManagerRequire (STD, NULL, "AC-SIGNAL",
CA_VariantEmpty(), &AC1);
      STDTSF_IDC_SIGNALGetac_ampl (power1, NULL, &AC1Ampl);
      STDBSC_IPhysicalSetvalue (AC1Ampl, NULL, AC1_Ampl_Val);
      STDTSF_IDC_SIGNALSetByRefac_ampl (AC1, NULL, AC1Ampl);
      STDTSF_IDC_SIGNALGetfreq (AC1, NULL, &AC1Freq);
```

```
     STDBSC_IPhysicalSetvalue (AC1Freq, NULL, AC1_Freq_Val);
     STDTSF_IDC_SIGNALSetByReffreq (AC1, NULL, AC1Freq);
     STDTSF_IDC_SIGNALGetOut (AC1, NULL, 0, &AC1Out);

     // Connect AC1 to hi SKT1-1 lo SKT1-2;
     STDBSC_IResourceManagerRequire (STD, NULL, "TwoWire",
CA_VariantEmpty(), &AC1Cnx);
     STDBSC_ITwoWireSetHI (AC1Cnx, NULL, "SKT1-1");
     STDBSC_ITwoWireSetLO (AC1Cnx, NULL, "SKT1-2");
     STDBSC_ITwoWireSetByRefIn (AC1Cnx, NULL, 0, AC1Out);

     freqSource = AC1Cnx;

     // Setup AC-SIGNAL ac_ampl range MAX 5 V, as sensor rmsVolts;
     STDBSC_IResourceManagerRequire (STD, NULL, "Measure",
CA_VariantEmpty(), &rmsVolts);
     STDBSC_IMeasureSetattribute (rmsVolts, NULL, "ac_ampl");
     STDBSC_IResourceManagerRequire (STD, NULL, "AC_SIGNAL",
CA_VariantEmpty(), &rmsVolts_Sig);
     STDTSF_IAC_SIGNALGetac_ampl (rmsVolts_Sig, NULL,
&rmsVolts_Sig_Ampl);
     STDBSC_IPhysicalSetvalue (rmsVolts_Sig_Ampl, NULL,
rmsVolts_Sig_Ampl_Val);
     STDTSF_IAC_SIGNALSetByRefac_ampl (rmsVolts_Sig, NULL,
rmsVolts_Sig_Ampl);
     STDBSC_IMeasureSetAs (rmsVolts, NULL, rmsVolts_Sig);

     // Connect hi SKT2-2 lo SKT2-4 to rmsVolts;
     STDBSC_IResourceManagerRequire (STD, NULL, "TwoWire",
CA_VariantEmpty(), &rmsVoltsCnx);
     STDBSC_ITwoWireSetHI (rmsVoltsCnx, NULL, "SKT2-2");
     STDBSC_ITwoWireSetLO (rmsVoltsCnx, NULL, "SKT2-4");
     STDBSC_ITwoWireGetOut (rmsVoltsCnx, NULL, 0, &rmsVoltsCnxOut);
     STDBSC_IMeasureSetByRefConn (rmsVolts, NULL, 0, rmsVoltsCnxOut);

     // Inform user: Begin testing
     printf (" Begin AC Amplifier Gain Test \n");

     // Apply Frequency Source
     STDBSC_ISignalRun (AC1Out, NULL, 0);
     // Connect SKT2-2 SKT2-4 for measurement
     STDBSC_ISignalRun (rmsVoltsCnxOut, NULL, 0);

     //  Calculate measurement values for required Gain limits
     // Measurement Upper Limit = Gain Upper Limit * Input
     MeasUpperLimit = 205 * 0.02;
     // Measurement Lower Limit = Gain Upper Limit * Input
     MeasLowerLimit = 195 * 0.02;

     for (FreqVal = 100; FreqVal < 1000; FreqVal += 100)
     {
          STDTSF_IAC_SIGNALGetfreq ( freqSource, NULL,
&freqSource_Freq);
          STDBSC_IPhysicalSetmagnitude (freqSource_Freq, NULL,
FreqVal);
          STDTSF_IAC_SIGNALSetByReffreq (freqSource, NULL,
freqSource_Freq);
```

```
          STDTSF_ISignalChange (AC1Out, NULL, 0);
          STDBSC_ISignalRun (rmsVolts, NULL, 0);
          // Read rmsVolts into Vout;
          STDBSC_IMeasureGetmeasurement (rmsVolts, NULL, &measValVT);
          CA_VariantSetDouble (&measValVT, measVal);
          Gain = measVal / 0.02;
          // Output the Gain for the user
          printf ("Amplifier Gain = %d\n", measVal);
          // Compare rmsVolts UL MeasUpperLimit LL MeasLowerLimit;
          STDBSC_ISensorGetUL (rmsVolts, NULL, &rmsVolts_UL);
          STDBSC_ISensorGetLL (rmsVolts, NULL, &rmsVolts_LL);
          STDBSC_IPhysicalSetmagnitude (rmsVolts_UL, NULL,
MeasUpperLimit);
          STDBSC_IPhysicalSetmagnitude (rmsVolts_LL, NULL,
MeasLowerLimit);
          STDBSC_ISensorSetByRefUL (rmsVolts, NULL, rmsVolts_UL);
          STDBSC_ISensorSetByRefLL (rmsVolts, NULL, rmsVolts_LL);
          STDBSC_ISensorGetNOGO (rmsVolts, NULL, &vBoolNOGO);
          if (vBoolNOGO == VARIANT_TRUE)
          {
               STDBSC_ISensorGetHI (rmsVolts, NULL, &vBoolHi);
               if (vBoolHi == VARIANT_TRUE)
               {
                    printf ("Gain too High\n");
                    Diagnosis1();
               }
               else
               {
                    printf ("Gain too Low\n");
                    Diagnosis2();
               }
               break;
          }
     }

     // Inform user: End of testing
     printf ("End of AC Amplifier Gain Test\n");

Error:
     return error;

}
```

8.5.2.5 AC amplifier example in C/ATLAS

The following version of the example is in C/ATLAS and is provided as a reference to users who are familiar with test requirements written in ATLAS.

The control of the testing flow in C/ATLAS is determined by the ordering of the C/ATLAS statements and the first-order predicate logic IF-THEN-ELSE.

```
000010 BEGIN, ATLAS PROGRAM 'AC_AMP'
C*******************************************************************
*     Test Methodology: C/ATLAS (IEEE Std 716-1995)                *
*     UUT Type:         AC Amplifier                               *
*     Model:            AC-01                                      *
```

```
*******************************************************************$
C$
C*******************************************************************
*  Begin the preamble part (containing declarative statements)      *
*******************************************************************$
C$
 000100 REQUIRE,  'AC1', SOURCE, AC SIGNAL,
           CONTROL,
              VOLTAGE 20 MV ERRLMT +-0.5 PC $
              FREQ RANGE 100 HZ TO 1000 HZ BY 100 HZ ERRLMT +-5 PC,
           CNX HI LO $
C$
     10 REQUIRE,  'rmsVolts', SENSOR, (VOLTAGE) AC SIGNAL,
           CAPABILITY,
              VOLTAGE MAX 5 V $
              FREQ RANGE 100 HZ TO 1000 HZ BY 100 HZ ERRLMT +-5 PC,
           CNX HI LO $
 000200 DECLARE, VARIABLE, 'STIM-F', is DECIMAL $
     10 DECLARE, VARIABLE, 'Vout', is DECIMAL $
     20 DECLARE, VARIABLE, 'GAIN', is DECIMAL $
C$
 000300 DEFINE, 'DIAGNOSIS-1', PROCEDURE $
     10    OUTPUT, C'ADJUST GAIN CONTROL - LOWER' $
     20 END, 'DIAGNOSIS-1' $
C$
 000400 DEFINE, 'DIAGNOSIS-2', PROCEDURE $
     10    OUTPUT, C'ADJUST GAIN CONTROL - HIGHER' $
     20 END, 'DIAGNOSIS-2' $
C$
C*******************************************************************
*  End the preamble part
*******************************************************************$
C$
C*******************************************************************
*  Begin the procedural part (containing the UUT test statements)
*  that calls the declarative statements in the preamble part
*******************************************************************$
C$
C*******************************************************************
*  Start testing the UUT
*******************************************************************$
C$
C       Apply initial signal $
 001000 APPLY, AC SIGNAL USING 'AC1',
            VOLTAGE 20MV ERRLMT +-0.5 PC,
            FREQ 100 HZ ERRLMT +-5 PC,
            CNX HI SKT1-1 LO SKT1-2 $
C$
C       Prepare measurement resource for subsequent READs $
     20 SETUP, (VOLTAGE), AC SIGNAL USING 'rmsVolts',
            VOLTAGE MAX 5 V,
            CNX HI SKT2-2 LO SKT2-4 $
     30 CONNECT, (VOLTAGE), AC SIGNAL USING 'rmsVolts',
            VOLTAGE MAX 5 V,
            CNX HI SKT2-2 LO SKT2-4 $
C$
B       Start of loop $
```

```
 001100 FOR, 'STIM-F' = 100 THRU 1000 BY 100, THEN $
     10    CHANGE, AC SIGNAL USING 'AC1',
               VOLTAGE 20 MV ERRLMT +-0.5 PC,
               FREQ 'STIM-F' RANGE 100HZ TO 1000HZ ERRLMT +- 5 PC,
               CNX HI SKT1-1 LO SKT1-2 $
     20    READ, (VOLTAGE INTO 'Vout'), AC SIGNAL USING 'rmsVolts',
               VOLTAGE MAX  5V,
               CNX HI SKT2-2 LO SKT2-4 $
     30    CALCULATE, 'GAIN' = 'Vout'/0.02 $
     40    COMPARE, 'GAIN', UL 205 LL 195 $
 001200    IF, NOGO, THEN $
     10       IF, HI, THEN $
     20          OUTPUT, C'GAIN TOO HIGH' $
     30          PERFORM, 'DIAGNOSIS-1' $
     40       ELSE $
     50          OUTPUT, C'GAIN TOO LOW' $
     60          PERFORM, 'DIAGNOSIS-2' $
     70       END, IF $
     80       LEAVE, FOR STEP 001100 $
     90    END, IF $
     95 END, FOR $
C       End of loop $
C$
 001300 REMOVE, (VOLTAGE), AC SIGNAL USING 'rmsVolts',
            VOLTAGE MAX 5 V,
            CNX HI SKT2-2 LO SKT2-4 $
     10 REMOVE, AC SIGNAL USING 'AC1',
            VOLTAGE 20 MV ERRLMT +-0.5 PC,
            FREQ 'STIM-F' RANGE 100HZ TO 1000HZ ERRLMT +- 5 PC,
            CNX HI SKT1-1 LO SKT1-2 $
 999900 FINISH $
C$
C*********************************************************************
*  End of testing
*********************************************************************$
C$
 999990 TERMINATE, ATLAS PROGRAM 'AC_AMP'
```

8.5.2.6 AC amplifier example using static model

Apart from using a text-based method of producing test requirements with TPL statements embedded in a carrier language, it is quite acceptable to present test requirements in other formats. For example, the STD standard supports graphically produced and tabular-based environments. This example not only demonstrates a graphical method of performing the test but also demonstrates an alternative (graphical) method of presenting test requirement and signal model information.

The original C/ATLAS specification had been written using a For/Next Loop (see 8.5.2.5) as have the STD examples. This example defines a dynamic test requirement. The term dynamic refers to the signal definitions that are being dynamically changed over time by the test program. The speed or timing of each loop is undefined and, therefore, implementation dependent.

An alternative way to define this test requirement would have been through a frequency sweep description. This represents a static test requirement. The term static is used because, even though the signal changes over time, the signal model remains constant. In static test requirements, there is no timing ambiguity because both the frequency value and the time are defined.

Historically, test requirements have been produced using procedural or dynamic descriptions, and consequently there are not many static test requirements available. This practice has lead to a situation where requirements are seen as linear and procedural in nature, rather than being parallel and functional. In the case of the ac amplifier, the test requirement has not necessarily been captured. It may be that a particular implementation has been captured.

The choice of using static or dynamic descriptions for test requirements depends on the final user. What is important is that the STD standard supports both approaches. Figure 57 shows a block diagram design for a frequency swept test of an ac amplifier. It shows how BSCs and TSFs may be combined to perform the test in a single operation. This example shows a continuous sweep, although a discontinuous (stepped) sweep could equally well have been defined. In that case, the timing between steps could be defined, if that is important, and not left to be determined by the operating system.

The sinusoid BSC provides the initial 0.02 V, 100 Hz signal. This signal passes through the sweep conditioner, which is a TSF model that conditions the input fixed frequency signal to provide a frequency sweep. The swept signal is then passed into the TwoWire BSC to connect it to the UUT. The output of the UUT is interfaced via another TwoWire BSC, and the output of that is observed by the RMS BSC.

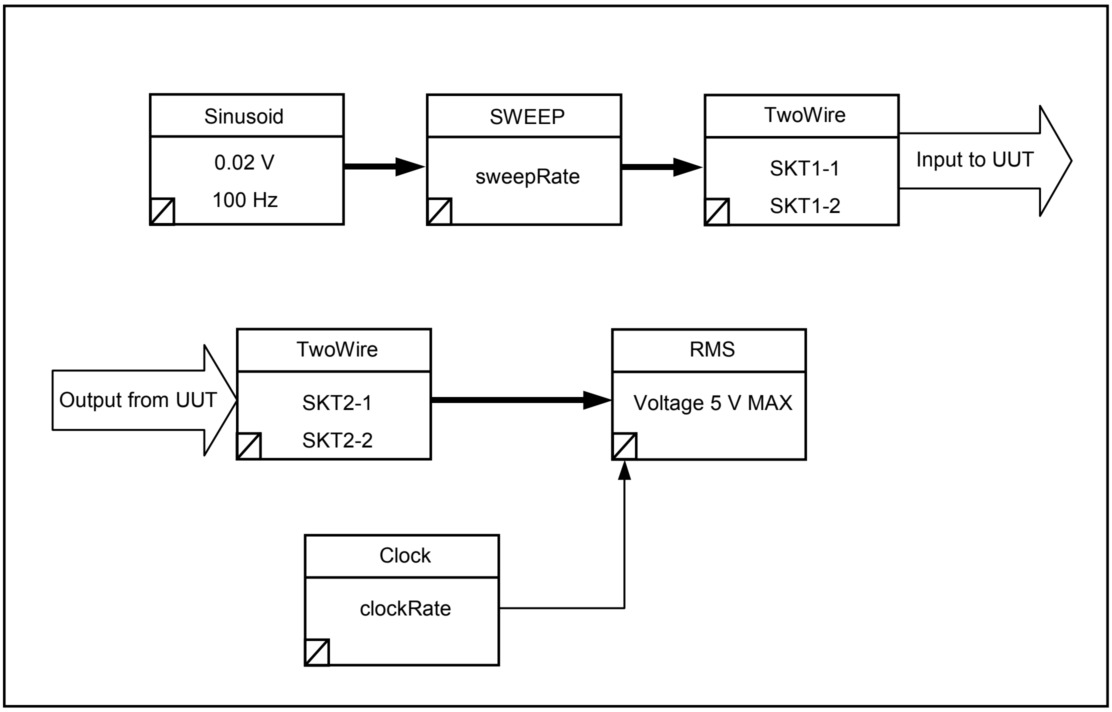


Figure 57 —AC amplifier test block diagram design

The sweep TSF model contains standard BSCs to define a frequency sweep. This TSF model is discussed in more detail in 10.8.

When the test is initiated, the sinusoid BSC provides a 20 mV, 100 Hz signal. This signal is conditioned by the sweep TSF model to provide a sweep signal and is connected to the UUT via the TwoWire BSC. The UUT response is monitored via the second TwoWire BSC. The clock BSC starts at the same time as the sinusoid BSC and in turn provides a series of Gate events to the RMS BSC. This RMS BSC will acquire the number of results that it has been instructed to take. The clockRate must be set so that it coincides with the points at which the sweeping signal achieves the required frequencies. When the RMS BSC has acquired all the results required, it will output a Done event to indicate that it has completed the test. The complete set of results may then be retrieved from the RMS BSC.

Figure 58 shows the start of the sweep signal to be applied to the UUT. This signal, which has an increasing frequency starting at 100 Hz, is shown against the initial 100 Hz sinusoidal signal.

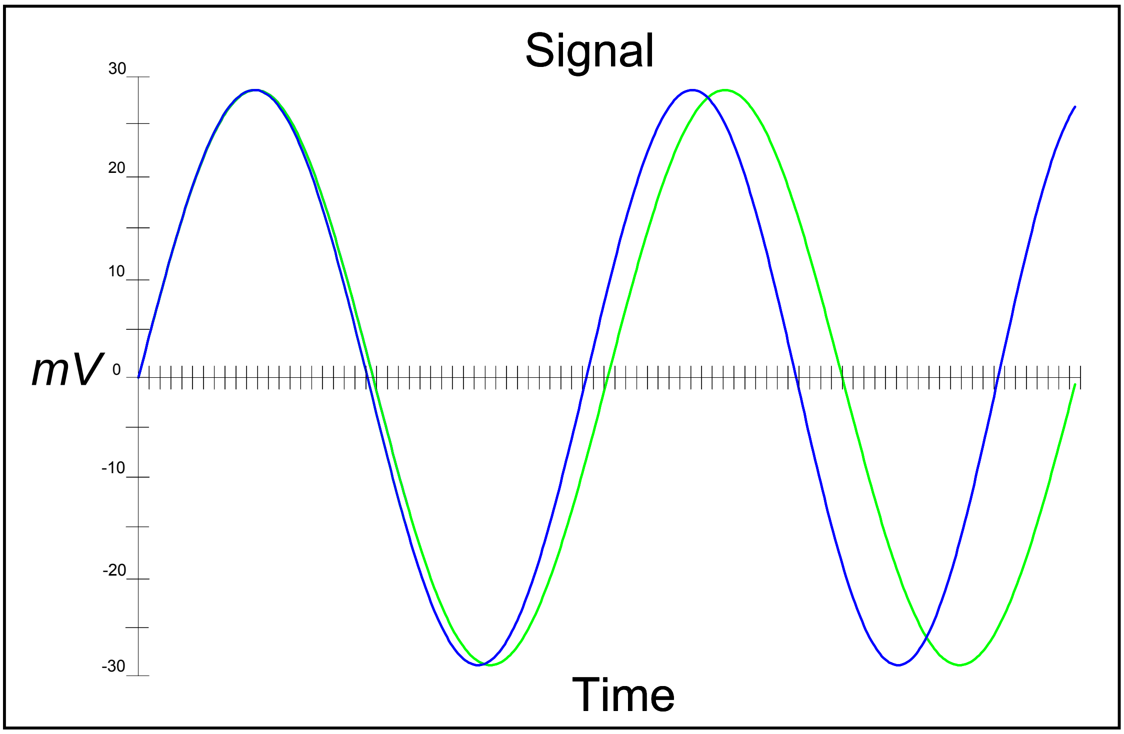


Figure 58—Start of sweep signal

Figure 59 shows the end of the sweep signal as it reaches its final frequency of 1 kHz. This signal is also shown against the original 100 Hz sinusoidal signal.

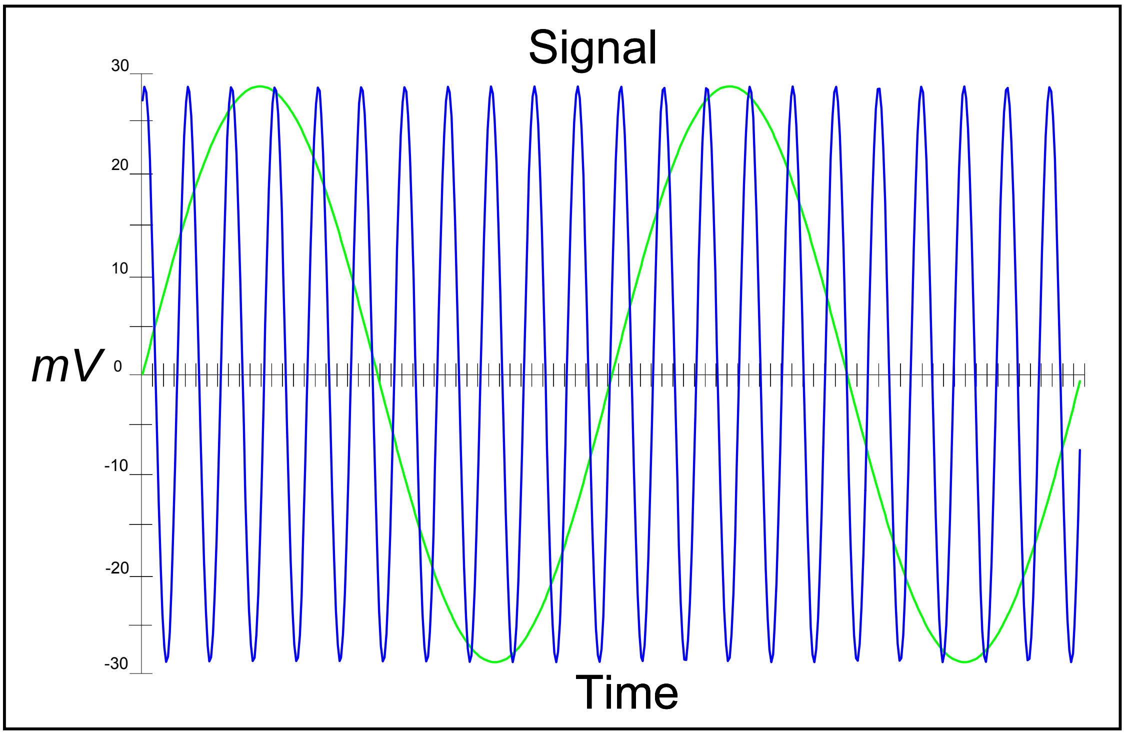


Figure 59—End of sweep signal

The method described above is only one of several possible scenarios for performing such a test. Once such a test has been designed and configured, it may be encapsulated as a TSF model. This step results in a completely self-contained test, including any timing required, that is then fully portable to any ATS that has a suitable capability.

9. Basic signal components (BSCs)

9.1 Introduction of new BSCs

There should not be a requirement to add further BSCs for the foreseeable future as it is believed that the current set of BSCs will provide all the necessary lower level support for users to create their own TSF models for electrical and electronic applications. Therefore, this guide does not attempt to explain how to create a BSC. Should users believe that there is a need for an additional BSC, they should send their comments to the IEEE, which will forward the comments to the appropriate working group.

9.2 BSC interfaces

This subclause provides further information about the interaction of BSCs using some illustrated examples. It also highlights the information required when using BSCs to create new or modified TSF signal models. The STD standard already provides a detailed description of the BSC layer, but a part of it is reproduced here for convenience.

A BSC defines specific signal characteristics through its attributes. BSCs can be grouped together into models, called *signal models*, to define complex signals. BSCs are interconnected through their properties of **In**, **Out**, and so on. 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 define their behavior in terms of what happens to their Signal properties and attributes. For testing purposes, the ultimate interest is in what happens with signals at the UUT or test interface. Where a signal defined by a BSC model passes through a connection, it becomes such a signal and may be defined in terms such as physical signal or real signal, both of which are nothing more than the signals used to perform the testing of the test subject. The corollary of this statement is that the items defined as Signal are in some way virtual and only become a physical entity that can be used for testing when they are connected. 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 control attributes, which are used to define the signal characteristics of the output signal, and values, which are attributes that 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 and, 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.

The BSC's **In** property consists of signal inputs that it uses. Where multiple inputs are required (e.g., Sum, Diff, Product, Or, And, Xor, and EventedEvent), the behavior of a BSC with multiple inputs shall be the same as multiple, dual (two) input. BSCs are cascaded together with their output being the first input of the next BSC and with the second input being the next input signal.

9.3 Diagrammatic representation of BSCs

Figure 60 represents a generalized form of a BSC and shows all possible interfaces and properties.

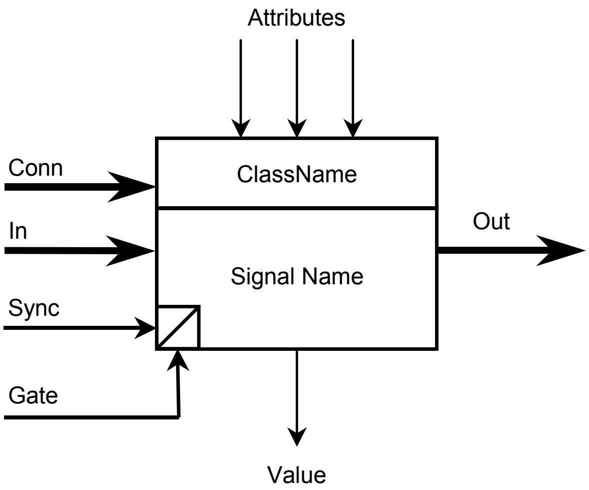


Figure 60—BSC diagram

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.

These diagrammatic representations of a BSC are of particular use when combining BSCs to create a signal model in a TSF or when describing a test requirement.

9.3.1 Model diagram for typical BSC

As an example, the sinusoid BSC may be used as a typical BSC. In the STD standard, the sinusoid BSC is expressed in a textual format as follows:

Sinusoid<type: Physical> ::Periodic

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

$$e = A\sin(\omega t+\varphi) \tag{1}$$

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.

9.3.2 Model diagram for sinusoid BSC

Figure 61 shows the sinusoid BSC in diagrammatic form. The class name is Sinusoid, and the signal name is provided by the user, in this case "User Signal Name."[7] If this BSC is used as part of a more complex signal, there may be more than one sinusoid in the model. The user-supplied signal name allows the user to identify individual instances of sinusoids. The three attributes defined for a sinusoid (amplitude, frequency and phase) are shown together with the **Out** signal. This diagram indicates that a sinusoid signal of given amplitude, frequency, and initial phase is being produced. There is no **In** signal, and there are no measurements to be taken. As no inputs are shown into the **Sync** and **Gate** terminals, this signal will appear the instant the Sinusoid class is created and will remain constant until it is destroyed. The convention is to use uppercase characters for the **Out** signal, which in this case has been given the name OUT_SIGNAL.

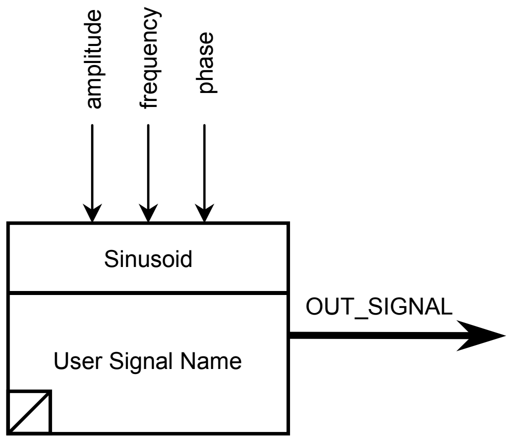


Figure 61—Sinusoid BSC

It is also a convention to omit attributes from the diagram if they assume the default value. For example, if the initial phase angle was 0 rad (or 0°), then the phase attribute could be omitted from the diagram. An attribute with a "don't care" value could also be omitted as the model would assume the default value.

9.4 SignalFunctions and events

The following paragraphs illustrate how synchronizing and gating events affect BSCs. It is convenient to restate a few facts about events and their effects.

The term *event stream* is used to indicate the output of an event source or conditioner, which represents the stream of events.

The EventFunction class contains EventSources and EventConditioners. EventSources define events while EventConditioners allow event definitions to be modified, based on the action of other events and signals.

[7] Throughout the remainder of this guide, the class name of the BSC is written without quotation marks, and the signal name provided by the user is enclosed in quotation marks.

An EventFunction must have an **Out** property and will possess attributes. It may have a signal **In** and will support **Gate** and **Sync** events, but does not possess values.

All SignalFunctions have two input event references (**Sync** and **Gate**). If the **Sync** reference is assigned, the behavior of the SignalFunction is restarted whenever the **Sync** event enters the *Active* state. If no **Sync** reference is assigned, the behavior of the SignalFunction is not restarted. If the **Gate** reference is assigned, the behavior is operational only while the **Gate** event is in the *Active* state so that the real characteristics of the signal are available only while the **Gate** event is active. If no **Gate** reference is assigned, the SignalFunction is operational while the **In** signal is in the *Active* state.

Once the **Out** signal of the SignalFunction is active, the SignalFunction is 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 real signal is present only while the **Gate** event is active (gated on). When the **Gate** event is inactive, the SignalFunction produces a null value. For signals, a null signal value could be 0 V or noise, but for events, a null value would be inactive (i.e., not on).

In summary:

For SignalFunctions (sources)

— A **Sync** event restarts the source at Time = 0.

— The output is available only when the **Gate** event is active.

For SignalFunctions (sensors – measurement and compare)

— A **Sync** event initializes the sensor object.

— A **Gate** event initiates a measurement. Each **Gate** event causes a measurement to be made. Only one measurement per **Gate** event is captured.

For sensors (monitor)

— A **Sync** event initializes the sensor object.

— A sensor only monitors while the **Gate** event is active. If the sensor is not monitoring, then the output event stream is inactive.

— Sensors take a measurement when the **Gate** event is active. After the measurement(s) are made, the output event stream becomes active.

For conditioners

— A **Sync** event restarts the conditioning at Time = 0.

— The output is available only when the **Gate** event is present. The **Gate** event is equivalent to gating all conditioner inputs.

For connections

— A **Sync** event restarts the connection at Time = 0.

— The output is available only when **Gate** event is present.

For EventFunction

— A **Sync** event restarts the EventFunction at Time = 0.

— A **Gate** event has the same meaning as ANDing the gate with the event outputs.

In the following examples, the event stream illustrated in Figure 62 is used to show the effect of **Sync** and **Gate** events on various signals.

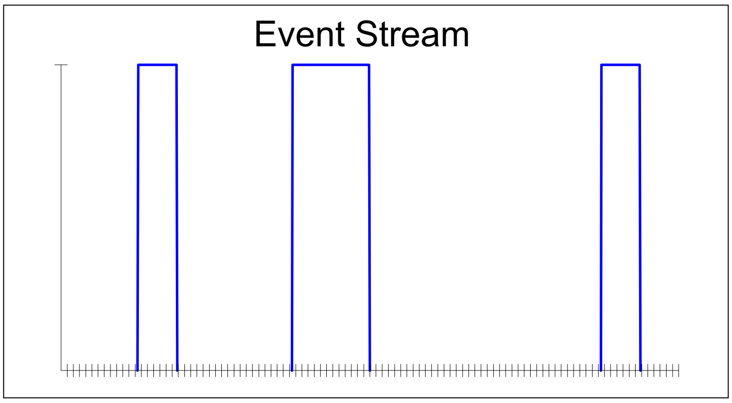


Figure 62—Event stream only

9.4.1 Signal source example

A source has to consider only event inputs (see Figure 63). The **In** property is explicitly ignored for both signal and event sources.

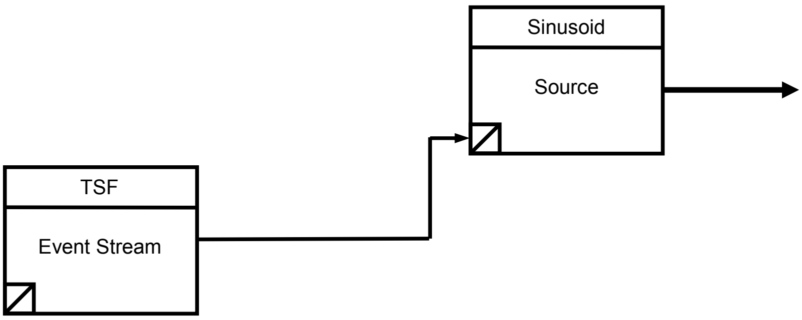


Figure 63— Source with Sync event

A source starts its signal from time zero at the arrival of each event. The signal continues until the next event arrives. Prior to the arrival of a **Sync** event, the signal is inactive.

Figure 64 shows the effect of the **Sync** events on the output signal. The sinusoid restarts at zero at the arrival of each **Sync** event.

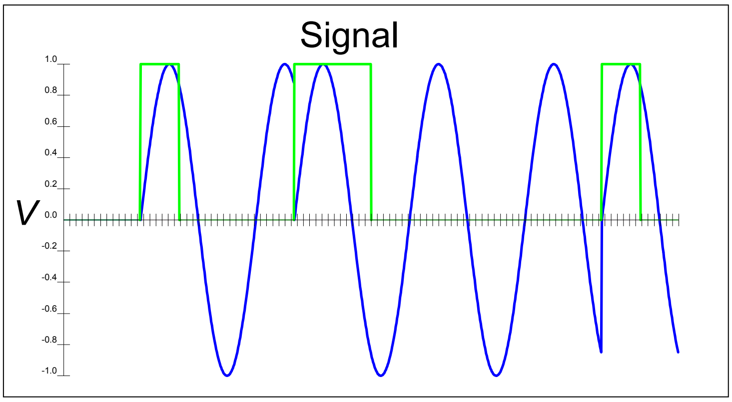


Figure 64—Effect of Sync event on sine wave

If the same event stream is used to gate the sinusoid BSC, the output will be active only when the **Gate** event is active (see Figure 65 and Figure 66). The **Gate** event does not resynchronize the signal.

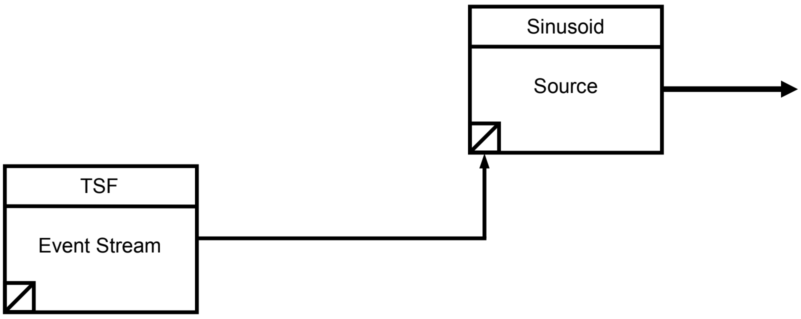


Figure 65—Source with Gate event

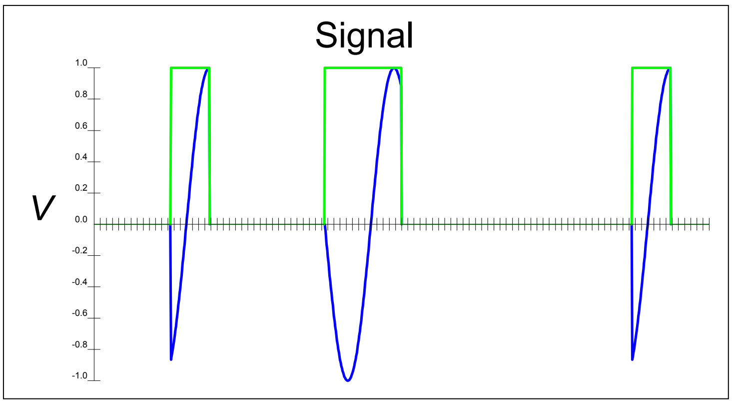


Figure 66—Effect of Gate event on sine wave

If an event stream is used to gate and synchronize the sinusoid BSC, the output will be active only when the **Gate** event is active, and it will be restarted when the event arrives at the sync input (see Figure 67 and Figure 68).

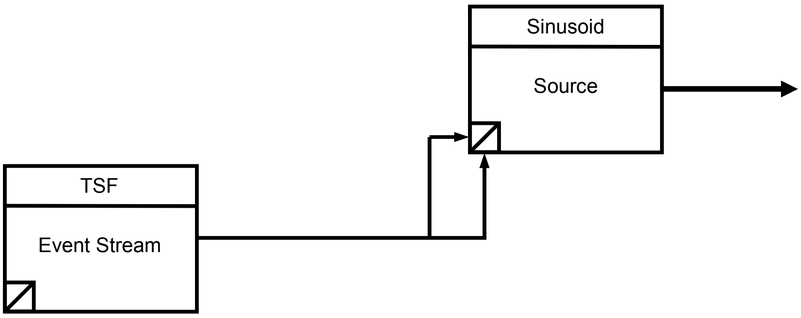


Figure 67—Source with Gate and Sync events

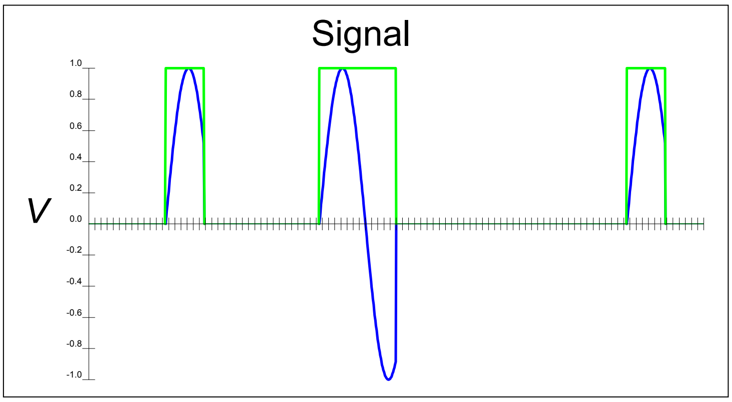


Figure 68—Effect of Gate and Sync events on sine wave

If the **Sync** events and **Gate** events come from different sources, there will be no relationship between the appearance of the signal at the output and the start of the signal (see Figure 69 and Figure 70).

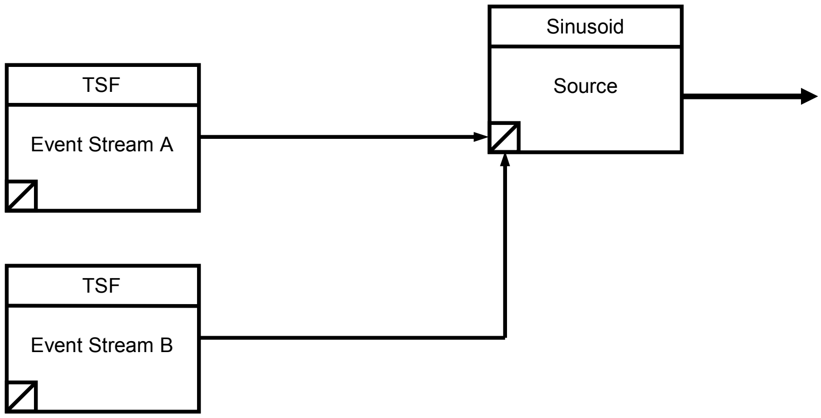


Figure 69—Source with separate Gate and Sync events

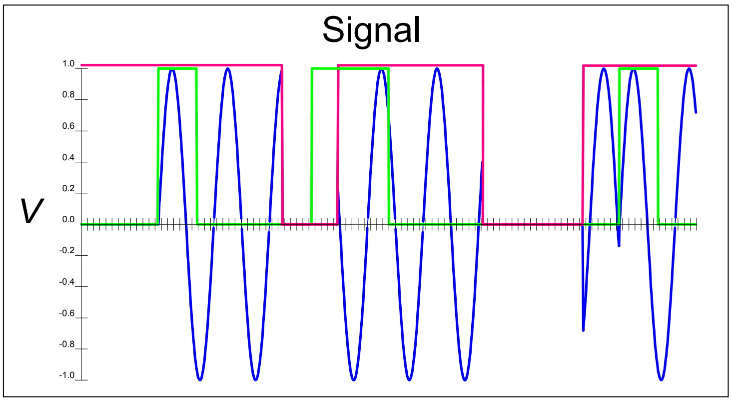


Figure 70—Effect of different Gate and Sync events on sinusoid wave

The signal may appear only at the output when the **Gate** event is active as shown in Figure 70 where three bursts of the sine wave occur during periods when the **Gate** event is active. Note that the first burst does not start immediately when the **Gate** event becomes active. The signal appears when the first **Sync** event occurs. Subsequent **Sync** events resynchronize the sine wave independently of the operation of the gate input, as shown by the second and third bursts.

If a component has a sync input connected, the signal cannot start until the first **Sync** event occurs, irrespective of the events on the gate input.

9.4.2 Signal conditioner examples

A conditioner uses both the **In** and **Sync** properties. Measurements may be considered as conditioners because of their monitor capability, which turns signal values into a gated event stream.

A conditioner has inputs via its **In** property and outputs via its **Out** property. The effect of **Sync** and **Gate** events on the conditioner is the same as it is for the source. In other words, the signal will appear only on the output of the conditioner when the **Gate** event is present, and the conditioner will be restarted each time a **Sync** event arrives. This activity is independent of the input signal, except that if there is no input signal, there will be no output signal.

This situation can be illustrated with a set of models diagrams showing the use of gate and sync inputs on a source and a conditioner, together with their corresponding output signals. For these examples, the source is a sinusoid BSC named “Source,” and the conditioner is an exponential BSC named “Conditioner.” The same event stream is used.

The first three examples show the effect of **Sync** and **Gate** events on the conditioner with a constant sinusoid on its input.

9.4.2.1 Constant signal with synchronized conditioner

The model in Figure 71 shows a sinusoid BSC (“Source”) with its output signal being conditioned by an exponential BSC (“Conditioner”). The conditioner will not produce any output until the first **Sync** event occurs. For each subsequent time, the conditioner is resynchronized, and the exponential decay is restarted.

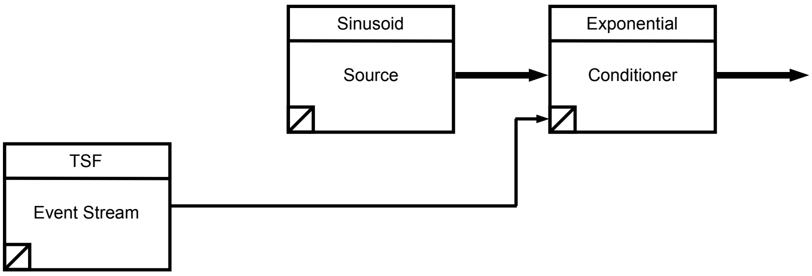


Figure 71—Conditioner with sync input

Figure 72 shows that the sine wave is continuous and that it was already running before the conditioner was triggered.

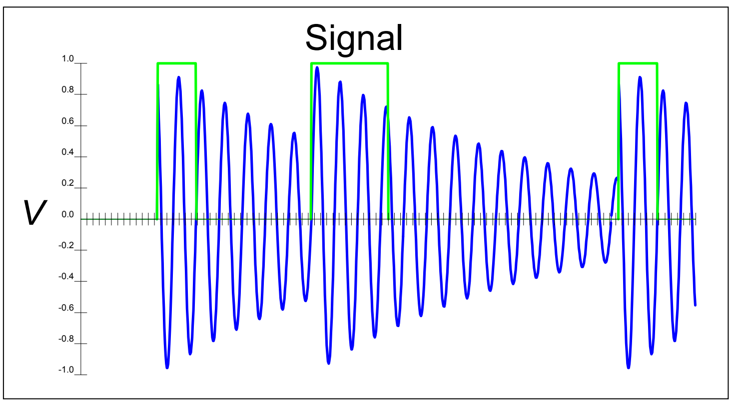


Figure 72—Effect of Sync event on conditioner

9.4.2.2 Constant signal with gated conditioner

The model in Figure 73 shows the same sinusoid BSC with its output signal being conditioned by the same exponential BSC. In this case, the conditioner is gated by the event stream. The conditioner will produce an output signal while the **Gate** event is present. As there is no **Sync** event present, the conditioner is not restarted. Eventually, there will be no output even if the **Gate** event is present as the signal will have decayed to an insignificant value.

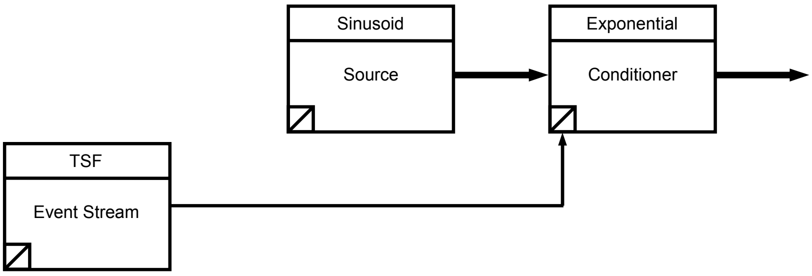


Figure 73—Conditioner with gate input

The exponential decay starts when the sine wave first appears at the input of the conditioner. The signal is already decaying when the first **Gate** event occurs (see Figure 74).

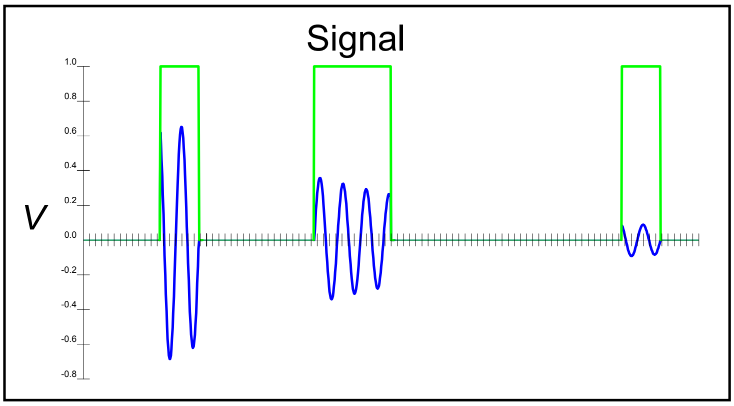


Figure 74—Effect of Gate event on a conditioner

9.4.2.3 Constant signal with gated and synchronized conditioner

The model in Figure 75 also shows the same sinusoid BSC with its output signal being conditioned by the exponential BSC. In this case, the conditioner is gated and synchronized by the event stream. The conditioner will produce an output signal while the **Gate** event is present. Not only will an output appear each time a **Gate** e

vent occurs, but the **Sync** event will restart the exponential conditioner.

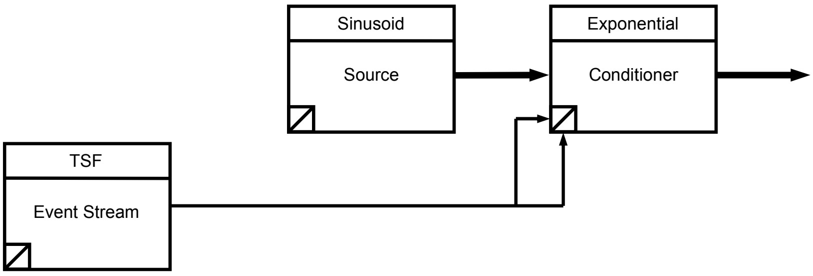


Figure 75—Conditioner with sync and gate inputs

Figure 76 shows the exponential decay starting at its initial maximum value at the beginning of each burst.

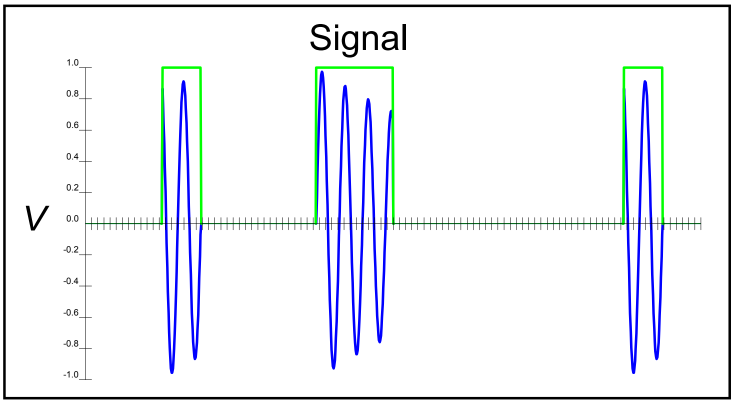


Figure 76—Effect of Sync and Gate events on conditioner

9.4.2.4 Synchronizing the source

The model in Figure 77 shows the sinusoid BSC being synchronized by an event stream and its output signal being conditioned by the exponential BSC. The conditioner does not have any **Sync** or **Gate** events. The sinusoid will not produce a sine wave until the first **Sync** event occurs, and for each subsequent event the sine wave is resynchronized (see Figure 78). The exponential decay is not restarted.

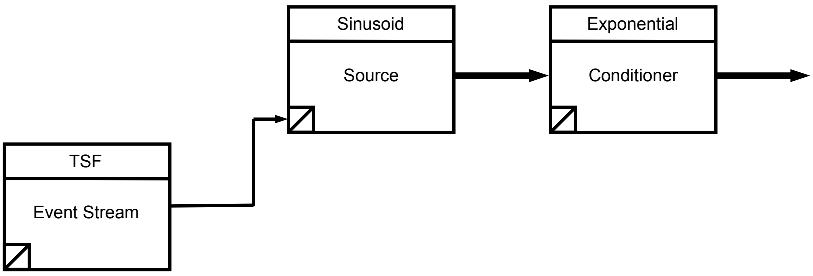


Figure 77—Source with sync input (followed by conditioner)

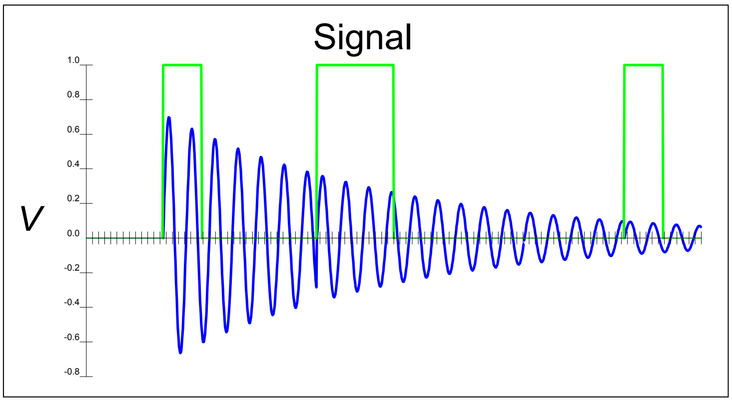


Figure 78—Effect of Sync event on source (followed by conditioner)

The models in the subsequent subclauses show the effects of gating and synchronizing both the source and the conditioner in different combinations.

9.4.2.5 Synchronizing the source and conditioner

The model in Figure 79 shows the sinusoid BSC and the exponential BSC both being triggered by the same event stream. Both BSCs will be started with the same initial event at their sync inputs. Both will be restarted (or resynchronized) with each subsequent event.

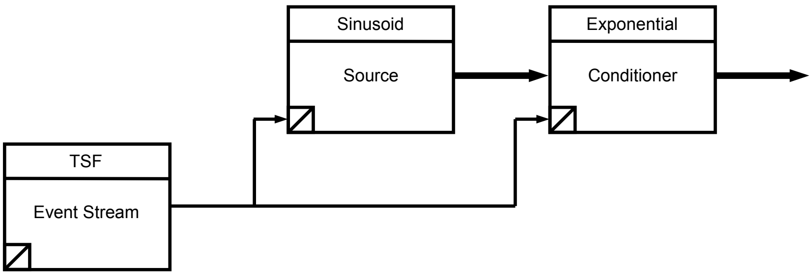


Figure 79—Source and conditioner with sync inputs

Figure 80 shows both the sine wave and the exponential decay starting at the leading edge of each of the three events.

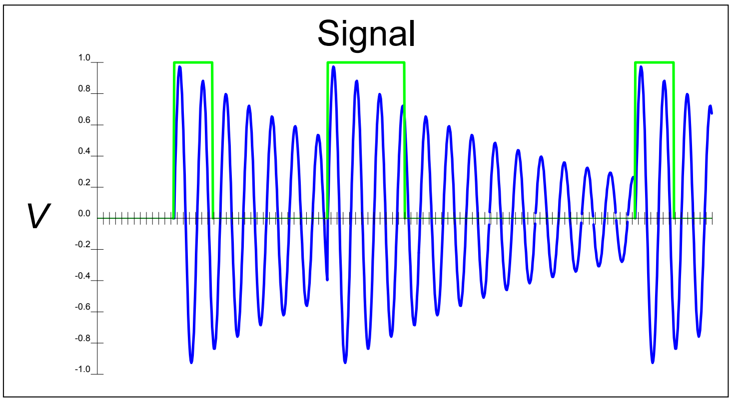


Figure 80 —Effect of Sync event on both source and conditioner

9.4.2.6 Gating the source and synchronizing the conditioner

The model in Figure 81 shows the sinusoid BSC being gated and the exponential BSC being synchronized by the same event stream.

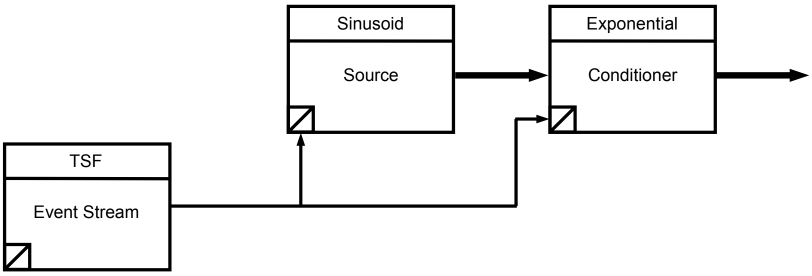


Figure 81 —Source with gate input and conditioner with sync input

Figure 82 shows the exponential decay starting at the leading edge of each of the three events. The sine wave is gated on and off by the same three events.

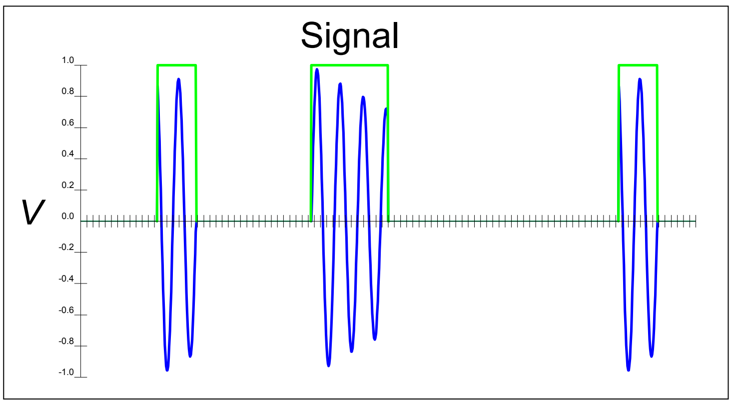


Figure 82—Effect of Gate event on source and Sync event on conditioner

9.4.2.7 Gating and synchronizing the source

The model in Figure 83 shows the sinusoid BSC being gated and synchronized and the exponential BSC being synchronized by the same event stream.

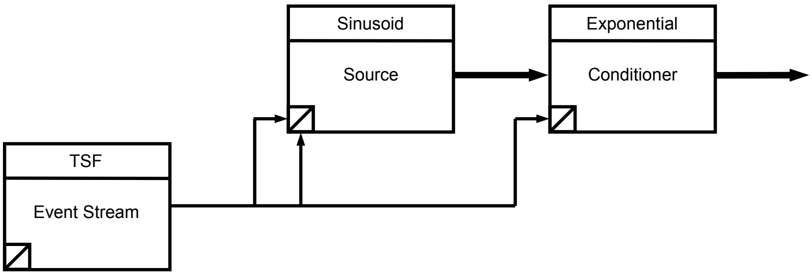


Figure 83—Source with gate and sync input plus conditioner with sync input

Figure 84 shows the sine wave and the exponential decay starting at the leading edge of each of the three events. The sine wave is gated on and off by the same three events.

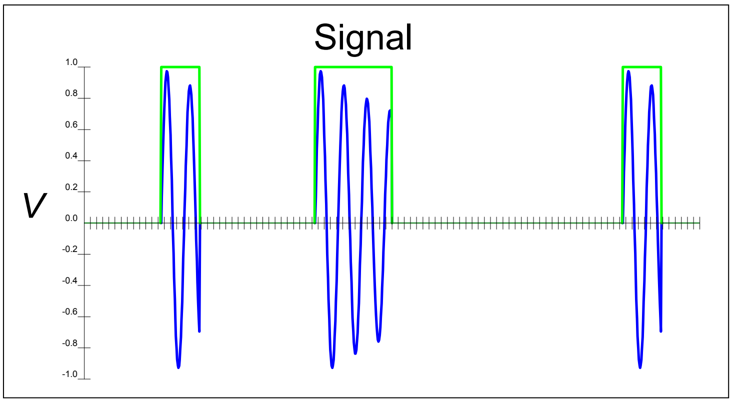


Figure 84—Effect of Gate and Sync events on source plus Sync event on conditioner

9.4.2.8 Gating and synchronizing the conditioner

The model in Figure 85 shows the sinusoid BSC being synchronized and the exponential BSC being gated and synchronized by the same event stream.

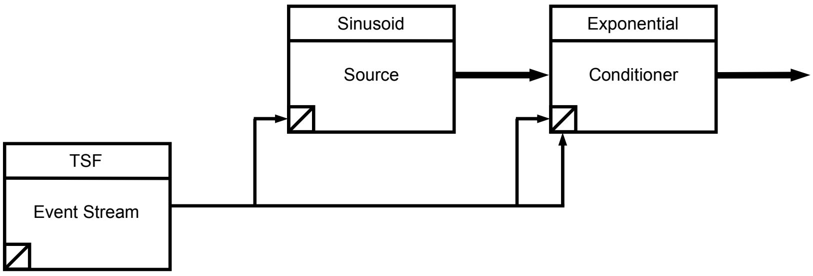


Figure 85—Source with sync input plus conditioner with gate and sync input

Figure 86 shows the sine wave and the exponential decay starting at the leading edge of each of the three events. The conditioner is gated on and off by the same three events.

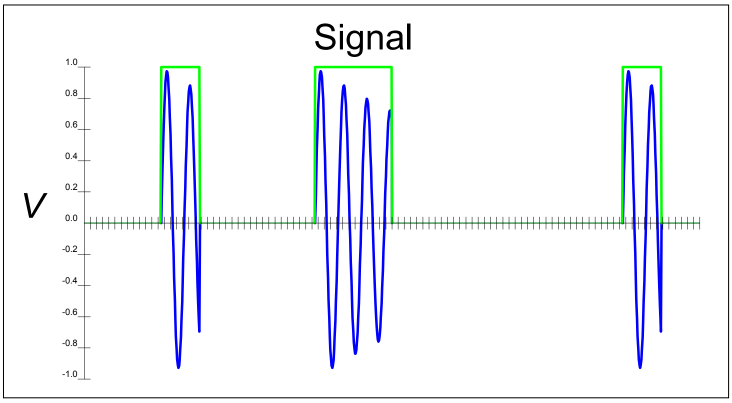


Figure 86—Effect of Sync event on source plus Gate and Sync events on conditioner

Comparing Figure 84 and Figure 86 shows that they are identical. It does not appear to matter whether the source or the conditioner is gated. In most cases, for a linear series of BSCs, only one needs to be gated. With few exceptions, to gate the output of any complex arrangement of BSCs, it is only necessary to gate the final BSC.

This effect is also shown by comparing the signal model diagrams in Figure 76 and Figure 82 for a constant signal with a gated and synchronized conditioner (see 9.4.2.3) and for a gated signal and synchronized conditioner (see 9.4.2.6).

This rule does not apply if the collection of BSCs includes a delay BSC with the delay attribute not set to zero.

9.4.2.9 Using different Sync event signals

The model in Figure 87 shows both the sinusoid BSC and the exponential BSC being synchronized by different event streams.

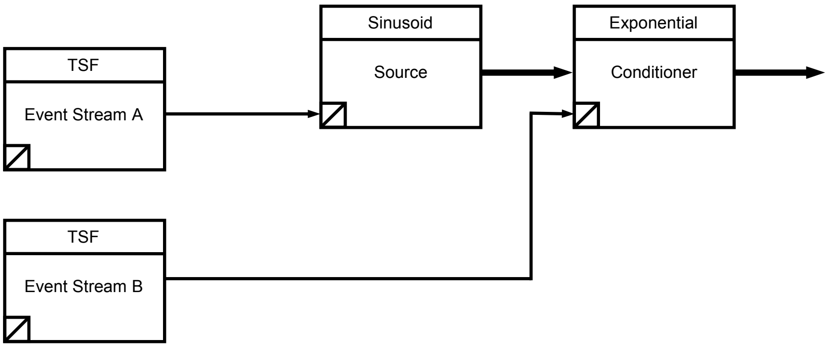


Figure 87 —Source and conditioner with separate sync inputs

Figure 88 shows the effect of different event streams being used to trigger the source and conditioner components. The exponential BSC is triggered first, and this step causes the exponential decay to start. The signal appears when the sinusoid BSC is triggered. Although the exponential BSC is started first, no output is possible unless there is a signal at the input. Both components are restarted separately when an event occurs at their respective sync inputs.

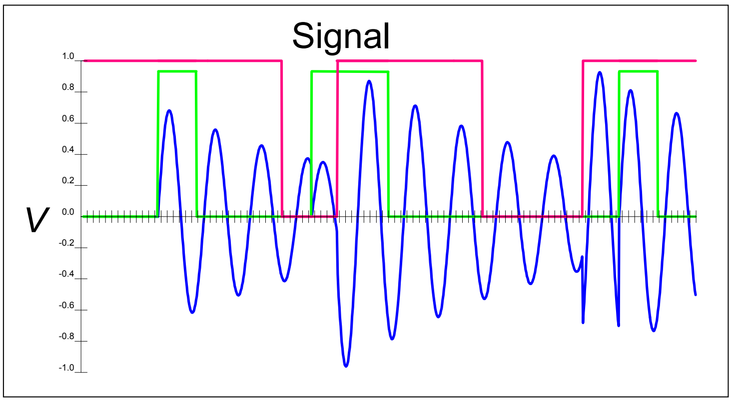


Figure 88—Effect of different Sync events on source and conditioner

10. Test Signal Framework (TSF)

10.1 Introduction to TSF

A TSF library is a collection of signal definitions relating to a particular test technology or other logical grouping. Annex E of IEEE Std 1641-2010 provides a set of signals based upon the signals described by the Nouns and Noun-Modifiers listed in the C/ATLAS standard (i.e., IEEE Std 716-1995 [B9]). That annex illustrates many of the most commonly used signals used in electronic test, plus a few signals that are more complex.

Annex E of IEEE Std 1641-2010 is not intended to be definitive or comprehensive. One reason for this approach is that the C/ATLAS standard [B9] does not provide enough information to completely specify all the signals listed. It is expected that users would wish or need to modify, expand, redefine, and add signals.

These modified and expanded sets of signals may be considered a TSF library for use with a particular set of UUTs, a particular contract, or a particular test facility. It is essential that each TSF model be built upon the SML and the BSC definitions. As a result, the TSF library and test requirements will be fully defined. They will then be portable across different designs of host systems.

This clause provides guidance on the creation of TSF models using BSCs and already defined TSF signals.

10.2 Building TSF signal models using BSCs

A signal model may comprise one or more BSCs. The signal model shall be documented using the XML format defined in the standard. To improve human readability, the tabular method used in Annex E of IEEE Std 1641-2010 may be used when first designing TSFs.. This method provides a more reliable definition than the pictorial method. The pictorial method shows a diagram with BSCs linked with signals that pass through the model until the desired signal is achieved. This pictorial method helps the visualization and understanding of a complex signal. Ultimately, he XML format described in Annex E of IEEE Std 1641-2010 is the required formal format for defining TSF models and providing interchangeable TSF libraries and should be used as the definitive interchange medium.

When creating a new signal model, it is useful to sketch the required signal model using the pictorial method, showing the flow of signals from and through BSCs, until the required signal is created. Having designed the new signal, it should then be specified in tabular form. As each BSC is included in the table, it is then simple to check that no attributes have been accidentally ignored. Once all the TSFs have been created, a TSF library should be created using the XML format. This process assumes that the user is creating a model without the aid of a proprietary tool. Tools are available that allow the user to create the model graphically and then check that the output waveform is exactly as expected.

10.3 Examples of source signal models

10.3.1 Simple ac signal model

The simplest form of signal model is one that comprises a single BSC.

10.3.1.1 Model diagram

Figure 89 shows a simple ac signal using the sinusoid BSC. This signal is named AC_SIGNAL. Diagrammatically, this figure comprises the sinusoid BSC diagram (see Figure 61 from 9.3.2) with user-specified external signal and attribute names (a BSC in a box). In this signal model, it is assumed that the initial phase angle is not important, and the phase attribute has been omitted from the diagram.

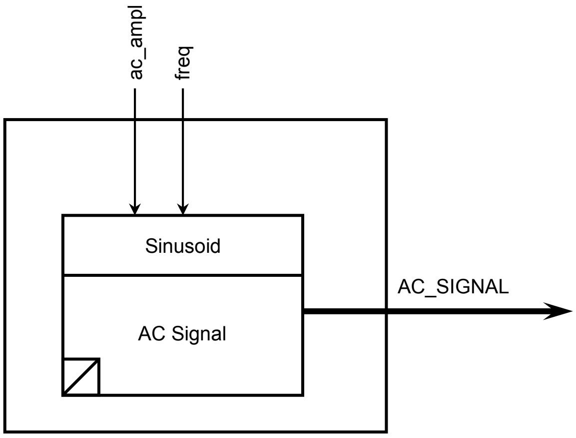


Figure 89—AC_SIGNAL model

10.3.1.2 Interface properties

The interface properties table showing the external interfaces may now be drawn up. For this simple model, Table 9 presents an interface properties table with only two entries.

Table 9—AC_SIGNAL interface

Description	Name	Type	Default	Range
AC Signal amplitude	ac_ampl	Physical	—	—
AC Signal frequency	freq	Frequency	—	—

This interface properties table provides the following information:

— The Description column contains a short textual description of each attribute or input signal.

— The Name column specifies the external attribute name for the model.

— The Type column specifies the type of the attribute. The freq attribute can be only of type Frequency, but the ac_ampl attribute may be any Physical type such as Voltage and Current.

— The Default column has no entries. Therefore, the user must provide a value for both the amplitude and the frequency. The BSC has its own default values of 0 for amplitude and 1 Hz for frequency. The BSC default values are part of the BSC definition and should not be considered when creating a signal model.

— The Range column has no entries. Therefore, there is no restriction on the values that may be used. In other words, the AC_SIGNAL model could be used for a radio frequency carrier where the amplitude is likely to be small and the frequency very high, or it could be an ac power line where the amplitude could be large and the frequency comparatively low.

10.3.1.3 Model description

The model description table is very simple for this signal as only one BSC needs to be defined and no BSC interconnections need to be defined (see Table 10).

Table 10 —AC_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
AC Signal	Sinusoid	Signal [Out]	—	AC_SIGNAL	—
		amplitude	ac_ampl	—	—
		frequency	freq	—	—
		phase	—	—	0 rad

This model description table is organized as a network list and provides the following information:

— The Name column has an entry for each BSC element. In this case, there is only one named AC Signal.

— The Type column also has a single entry for each BSC element and specifies the class of the BSC, which, in this case, is Sinusoid.

— The Terminal column has an entry for all the terminals of each BSC, including those not associated with a signal interface attribute or not in the model diagram. This example shows four terminals, three inputs and one output. The first terminal specified in the list is normally the output followed by attributes, [In] signals, sync inputs, and gate inputs. Sync and gate inputs may be omitted from the table unless they have a specific internal or external connection. This practice is explained in more detail in 10.8 and 10.13.

— The remaining three columns are associated with the terminals and define where the terminals are connected. Input terminals must have an entry in either the Input column or the Formula column, and Output terminals must have an entry in the Output column.

In the example the first terminal is the **Out** signal, and it is shown connected to the AC_SIGNAL output, which, in this case, is the output signal created by the model.

The other three terminals are the Sinusoid attributes. The Input column specifies the amplitude attribute from the interface terminal "ac_ampl" and the frequency attribute from the interface terminal "freq". The phase attribute does not have an input so that the initial phase angle cannot be specified by the user, but it would be incorrect to leave an attribute completely unspecified. The Formula column is used to provide a value for the attribute, which, in this case, is a fixed default value of 0 rad.

10.3.1.4 Equations, notes, rules, and examples

Equations, notes, rules, and examples may accompany a TSF model if they provide additional clarity and information about the model and its operation.

10.3.1.4.1 Equations

The AC_SIGNAL model does not really require any additional equations as the BSC Sinusoid is fully defined. An ac signal is a cyclic time-varying signal, which alters the amplitude of the dependant variable. If an equation was required, it could be presented as shown in this subclause.

The output is given by Equation (2):

$$e = E\sin(\omega t+\varphi) \qquad (2)$$

where

E is the amplitude
ω is $2\pi \times$ the frequency
t is time
φ is the initial phase angle.

10.3.1.4.2 Notes

Notes are used to provide information about the model, e.g., any limitations. An example of a note for the simple AC_SIGNAL could be the following:

This model has limited functionality. It does not provide for the variation of the phase parameter.

10.3.1.4.3 Rules

Rules are used to provide information about the model, e.g., any restrictions in its use. For example, if the AC_SIGNAL model is intended only for use with voltage or current signals and not for power signals, a rule could be added stating the following:

For this signal, the allowable types are Voltage and Current.

10.3.1.4.4 Example

An example may be provided to give a graphical example of what a typical signal may look like. For an AC_SIGNAL with a fixed initial phase angle of 0 rad, this example would be as shown in Figure 90.

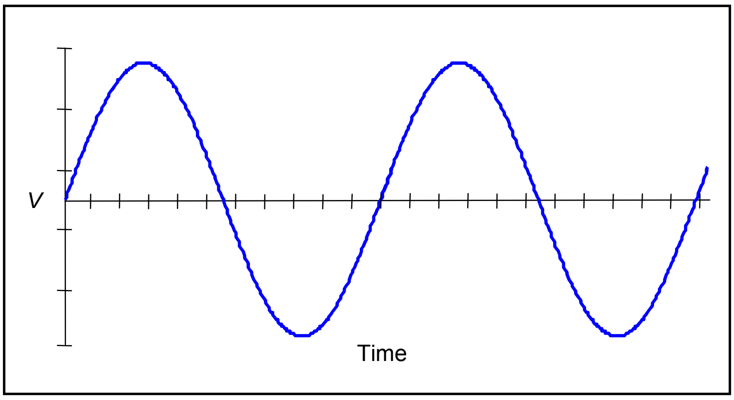


Figure 90—AC_SIGNAL example

10.3.2 Naming conventions used with signal models

The naming convention used with signal models has arisen from previous methods of naming signals.

The name of the output signal of the signal model is given in uppercase characters with underscores used where spaces may be expected. This convention is to retain some compatibility with the C/ATLAS Nouns, which are in uppercase. The use of the underscore instead of a space was agreed upon as it makes the parsing of signals much easier.

The names of the signal interface attributes are in lowercase characters, also with underscore characters where spaces may be expected.

The user-provided signal name for the BSC is usually in title case. This convention works well with names such as "Modulating Signal" or "Interrogation Event," but less so with names that include abbreviations. The convention for abbreviations, e.g., ac for alternating current and dc for direct current, is to use lowercase characters (refer to *The Authoritative Dictionary of IEEE Standards Terms* [B5]). This practice gives rise to "Ac Signal" in title case; therefore, it was decided to make all abbreviations uppercase in BSC signal names (hence "AC Signal").

10.3.3 A more complex ac signal model

10.3.3.1 Model diagram

This model of an ac signal in Figure 91 is more complex and includes the ability to specify a dc offset and some level of noise. There is no reason why a TSF should not contain more than one version of the same signal. The models must be given different names; therefore, this model is called NOISY_AC_SIGNAL.

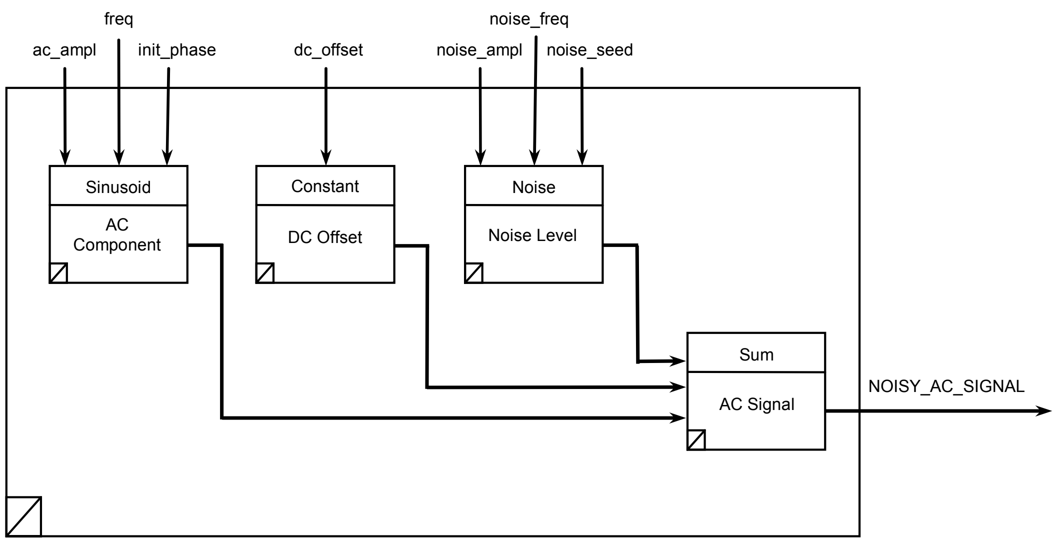


Figure 91—NOISY_AC_SIGNAL model

10.3.3.2 Interface properties

The interface properties table showing the external interfaces may now be drawn up. For this model, the interface properties table is more complex, with seven entries. When using this signal model, the user may not always wish to provide all seven attributes. The example in Table 11 shows how some may be set up with default values.

Table 11—NOISY_AC_SIGNAL interface

Description	Name	Type	Default	Range
AC Signal amplitude	ac_ampl	Physical	—	—
AC Signal frequency	freq	Frequency	—	—
AC Signal phase angle	init_phase	PlaneAngle	0 rad	0 rad to 2π rad
DC Offset	dc_offset	Physical	0	—
Noise amplitude	noise_ampl	Physical	0	—
Noise frequency	noise_freq	Frequency	0 Hz	—
Noise seed	noise_seed	Integer	0	—

This interface properties table provides the following information:

- The Description, Name, and Type columns provide the same (or equivalent) information as for the simple AC_SIGNAL model.
- The Default column has several entries, each indicating that if the user does not provide a value for that attribute, the specified default value will be assumed. The user still has to supply the minimum information of the ac signal amplitude and frequency. If the user supplies only the ac signal amplitude and frequency, the default values for all the other attributes will be used and will result in zero initial phase angle, zero dc offset, zero noise amplitude, zero noise frequency, and the noise seed set to zero. This more complex model may be used in place of the previous simple model because if only ac amplitude and frequency are provided, the same signal will be created.
- The Range column has a single entry for AC Signal phase angle, which is limited between 0 rad and 2π rad.

This example would also require a rule to ensure that the ac signal amplitude, the dc offset, and the noise amplitude had the same type. This rule could be worded as follows:

All types must be consistent. Thus, for example, if the AC Signal amplitude is specified in volts, then the DC Offset and the Noise amplitude must also be specified in volts.

10.3.3.3 Model description

The model description table is more detailed for this signal. Four BSCs are interconnected as defined in Table 12. By convention, the final BSC that produces the required signal is always placed at the head of the table.

Table 12—NOISY_AC_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
AC Signal	Sum	Signal [Out]	—	NOISY_AC_SIGNAL	—
		Signal [In]	Noise Level	—	—
		Signal [In]	DC Offset	—	—

		Signal [In]	AC Component	—	—
Noise Level	Noise	Signal [Out]	—	AC Signal	—
		amplitude	noise_ampl	—	—
		frequency	noise_freq	—	—
		seed	noise_seed	—	—
DC Offset	Constant	Signal [Out]	—	AC Signal	—
		amplitude	dc_offset	—	—
AC Component	Sinusoid	Signal [Out]	—	AC Signal	—
		amplitude	ac_ampl	—	—
		frequency	freq	—	—
		phase	init_phase	—	—

This model description table provides the following information:

— The Name and Type columns show the four BSCs that compose this model: “AC Signal” of class Sum, “Noise Level” of class Noise, “DC Offset” of class Constant, and “AC Component” of class Sinusoid. The first BSC in the table is “AC Signal,” which has an **Out** signal (NOISY_AC_SIGNAL) created by summing its three input signals. The input terminals to the BSC “AC Signal” are each linked to one of the other BSCs by a reference to its name.

— The noise BSC (“Noise Level”) has three input terminals (amplitude, frequency, and seed) that are linked to the three interface attributes (noise_ampl, noise_freq, and noise_seed, respectively). The input attributes are listed in the order they appear under the Attributes heading of the BSC definition in Annex B of IEEE Std 1641-2010. The Output and Formula columns for these inputs are left blank. All the required information is supplied by the interface via the default values and ranges defined in the interface properties table so they do not need to be duplicated in the model description table. The output is linked to the BSC “AC Signal.”

— The constant BSC (“DC Offset”) has its single input terminal linked to the interface attribute dc_offset. This attribute also has its default value defined in the interface properties table. The output is linked to the BSC “AC Signal.”

— The sinusoid BSC (“AC Component”) has its three inputs terminals (amplitude, frequency and phase) linked to the three interface attributes (ac_ampl, freq, and init_phase, respectively). Only one default value is defined in the interface properties table, i.e., init_phase. It is essential that the values of amplitude and frequency be supplied by the user; therefore, there are no default values for these attributes, either in the interface properties table or the model description table. The output is linked to the BSC “AC Signal.”

10.3.4 Model for dc signal

There is very little difference between a model for an ac signal with a dc offset and a dc signal with ripple.

10.3.4.1 Model diagram

The model represented in Figure 92, NOISY_DC_SIGNAL, may be seen to be the same fundamental model as NOISY_AC_SIGNAL, but with different names for attributes and with different default values. A comparison between the signal models for AC_SIGNAL and DC_SIGNAL in Annex E of IEEE Std 1641-2010 illustrates this point.

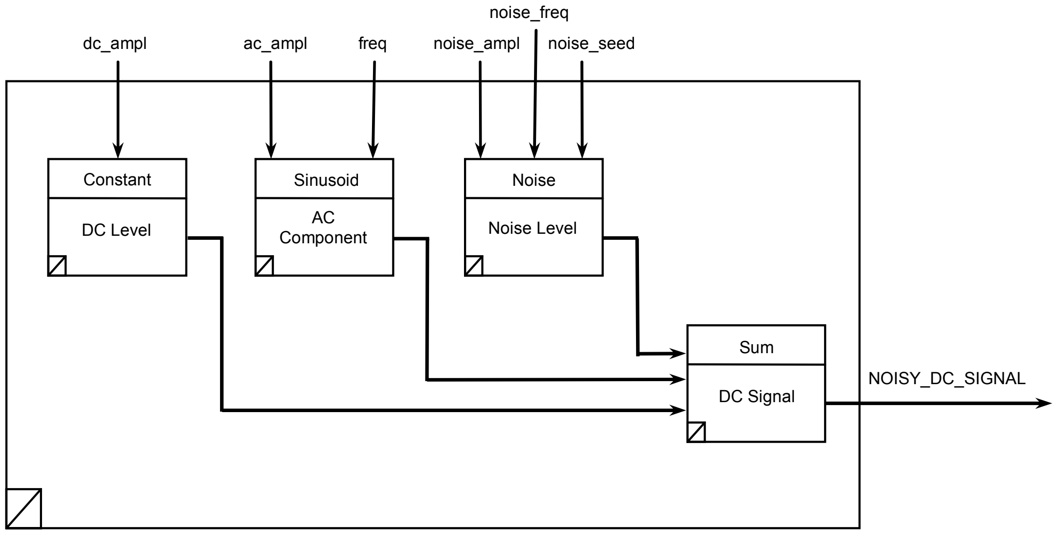


Figure 92—NOISY_DC_SIGNAL model

10.3.4.2 Interface properties

The interface properties table is very similar to the NOISY_AC_SIGNAL table in 10.3.3. The main differences are in the default values and the names given to some of the attributes (see Table 13).

Table 13—NOISY_DC_SIGNAL interface

Description	Name	Type	Default	Range
DC amplitude	dc_ampl	Physical	—	—
AC Component amplitude	ac_ampl	Physical	0	—
AC Component frequency	freq	Frequency	0 Hz	—
Noise amplitude	noise_ampl	Physical	0	—
Noise frequency	noise_freq	Frequency	0 Hz	—
Noise seed	noise_seed	Integer	0	—

This interface properties table does not show a default value for the dc amplitude. As the model is for a dc signal, it is essential that the user supply the required dc amplitude. The ac component amplitude and frequency default to zero. In other words, if no ac component amplitude and frequency are provided, the model outputs a signal with no ripple component. Similarly, the noise component also defaults to zero (no noise element).

10.3.4.3 Model description

The model description table could be almost identical to the NOISY_AC_SIGNAL version with the exception of some of the names used, but the order of the BSCs in the table has changed (see Table 14).

Table 14—NOISY_DC_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
DC Signal	Sum	Signal [Out]	—	NOISY_DC_SIGNAL	—
		Signal [In]	Noise Level	—	—
		Signal [In]	AC Component	—	—
		Signal [In]	DC Level	—	—
Noise Level	Noise	Signal [Out]	—	DC Signal	—
		amplitude	noise_ampl	—	—
		frequency	noise_freq	—	—
		seed	noise_seed	—	—
AC Component	Sinusoid	Signal [Out]	—	DC Signal	—
		amplitude	ac_ampl	—	—
		frequency	freq	—	—
		phase	—	—	0 rad
DC Level	Constant	Signal [Out]	—	DC Signal	—
		amplitude	dc_ampl	—	—

The only other significant difference between this model and the model for the NOISY_AC_SIGNAL is that the ac component (ripple) has a default initial phase angle of 0 rad. This value is not accessible to the user because, in the case of a ripple voltage, the initial phase angle is a “don’t care” condition.

10.4 Dual or multiple use TSF models

It is sometimes efficient to create a TSF that may be used in more than one way. An example of this is a model for an AM signal where the amount of modulation may be specified by one of two different methods; either by the depth of modulation or by the amplitude of the modulating signal. The model used as an example is based on the AM_SIGNAL as specified in Annex E of IEEE Std 1641-2010.

10.4.1 Signal model for AM_SIGNAL

10.4.1.1 Model diagram

The signal model diagram (see Figure 93) shows attributes for both modulation amplitude (mod_ampl) and depth of modulation (mod_depth), both of which will affect the output modulated signal.

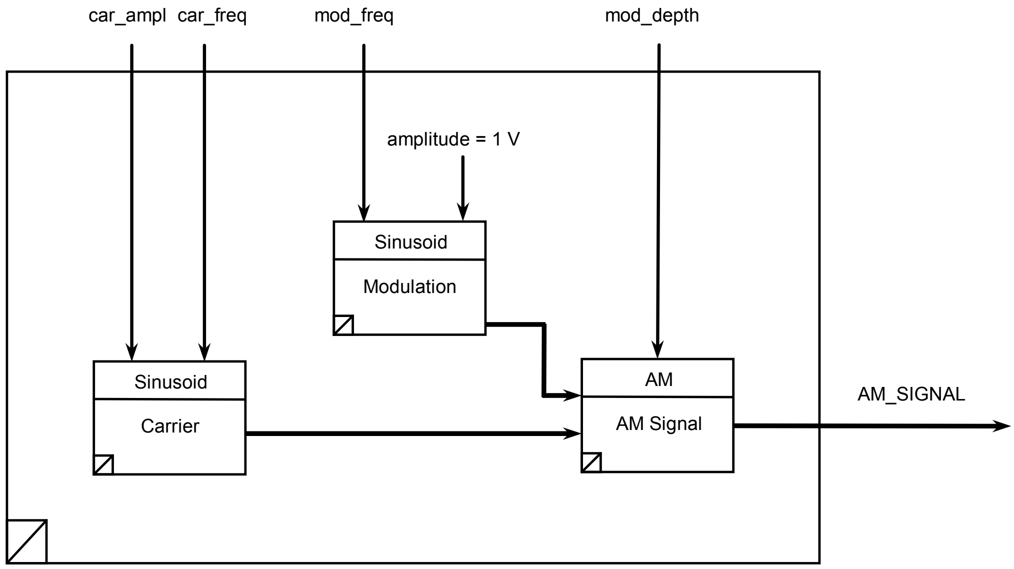


Figure 93—AM_SIGNAL model

10.4.1.2 Interface properties

The interface properties table shows all the attributes that are available to the user (see Table 15).

Table 15—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	1	—
Modulation amplitude	mod_ampl	Voltage	1 V	—

This model restricts the Physical type to Voltage only. A user could easily create a model with a different Physical type. The carrier amplitude would have to be changed to the required Physical type in the Type column.

For modulation signals, it is normal to expect that the carrier frequency will be much higher than the modulating frequency. There are no range restrictions for either frequency in the interface properties table as the model will operate correctly for any modulation and carrier frequencies.

10.4.1.3 Model description

The initial phase angle for the sinusoids are set to zero as they are “don’t care” values in this model (see Table 16).

Table 16—AM_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
AM Signal	AM	Signal [Out]	—	AM_SIGNAL	—
		modIndex	mod_depth	—	—
		Carrier [In]	Carrier	—	—
		Signal [In]	Modulation	—	—
Carrier	Sinusoid	Signal [Out]	—	AM Signal	—
		amplitude	car_ampl	—	—
		frequency	car_freq	—	—
		phase	—	—	0 rad
Modulation	Sinusoid	Signal [Out]	—	AM Signal	—
		amplitude	mod_ampl	—	—
		frequency	mod_freq	—	—
		phase	—	—	0 rad

The output is given by following Equation (3):

$$e = E_c(1+m_a E_m \sin(\omega_m t))\sin(\omega_c t) \tag{3}$$

where

E_c is the carrier amplitude (unmodulated)
E_m is the modulation amplitude
m_a is the depth of modulation (≡ modulation index)
ω_m is $2\pi \times$ modulating frequency
ω_c is $2\pi \times$ carrier frequency
t is the time

This model is designed to produce an amplitude modulated signal with constant modulation where the amount of modulation is specified by the user:

- If a value of unity (1 V) is used for the amplitude of the modulating signal (mod_ampl), then the depth of modulation will be correct as specified in terms of the attribute mod_depth.
- If a value of unity (1) is used for the depth of the modulation (mod_depth), then the amount of modulation will be determined by the amplitude of the modulating signal (mod_ampl). In this case an amplitude of 1 V will result in 100% modulation.
- For any other values for mod_ampl and mod_depth, the depth of modulation will be determined by the product of the mod_ampl (in volts) and the mod_depth.

If the default values for mod_ampl and mod_depth (as listed in the interface table) are used, the model will output an amplitude modulated signal with 100% modulation.

10.5 Signal models with preset internal attributes

Some signals include fixed preset attributes that are not default values, but are required either for the correct operation of the model or because the signal specifically includes set parameters.

10.5.1 Signal model for PULSED_DC_TRAIN

This example uses the signal model for PULSED_DC_TRAIN from Annex E of IEEE Std 1641-2010.

10.5.1.1 Model diagram

The signal model diagram (see Figure 94) shows an attribute for the pulse train that has a preset value that is not available to the user. The convention for this situation is to show the attribute for the BSC, but put it inside the box for the model.

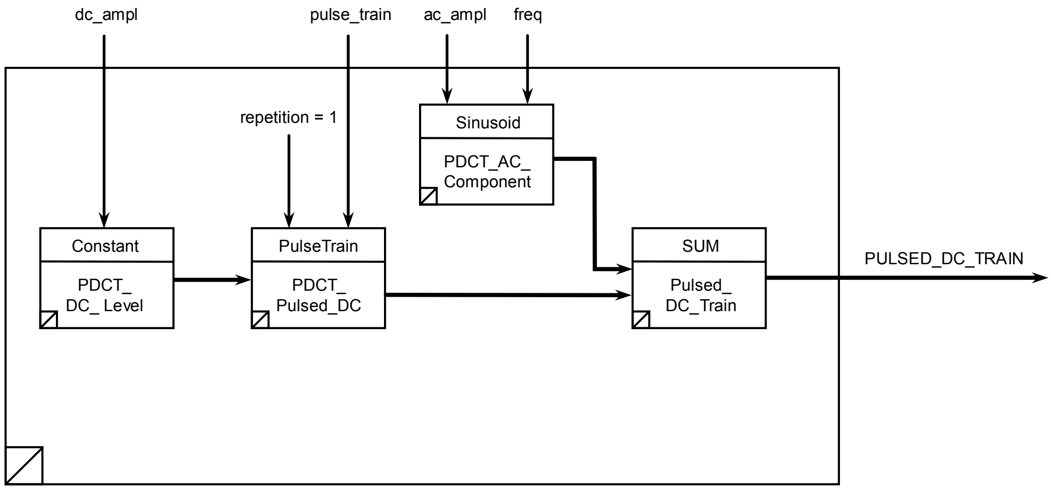


Figure 94—PULSED_DC_TRAIN model

This fixed attribute is used because the PULSED_DC_TRAIN model only outputs a single train of pulses that is not repeated.

10.5.1.2 Interface properties

The interface properties table will show only the attributes that appear outside of the model box (see Table 17).

Table 17—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	—

The model allows the dc train to have ripple superimposed on the signal. In many applications a user may not require to use this facility, so the interface definition includes default values that would give a pulse train without ripple. The minimum information that the user has to provide is the nominal dc level and the pulse definitions.

10.5.1.3 Model description

The pulse amplitude is specified in the model description table (see Table 18) in the Inputs column by the interface attribute dc_ampl, which provides the amplitude for the Constant BSC "PDCT_DC_Level." Ripple (AC Component) is specified by the attributes ac_ampl and freq in the Inputs column. If no values are specified the default values of zero for amplitude and zero for frequency are used, which means that there is no output from the Sinusoid BSC "PDCT_AC_Component." The PulseTrain BSC "PDCT_Pulsed_DC" has two attributes; pulse_train and repetition. The form of the pulses is defined in the attribute pulse_train and the repetition attribute is fixed at unity because the value 1 is inserted in the Formula column. The base level for the pulses in the PulseTrain BSC is determined by the **In** signal, which is provided by the Constant BSC. Finally, the overall TSF output is obtained by summing the outputs from the PulseTrain and Sinusoid BSCs.

Table 18— PULSED_DC_TRAIN model

Name	Type	Terminal	Inputs	Output	Formula
Pulsed_DC_Train	Sum	Signal [Out]	—	PULSED_DC_ TRAIN	—
		Signal [In]	PDCT_Pulsed_DC	—	—
		Signal [In]	PDCT_AC_Component	—	—
PDCT_Pulsed_DC	PulseTrain	Signal [Out]	—	Pulsed_DC_Train	—
		pulses	pulse_train	—	—
		repetition	—	—	1
		Signal [In]	PDCT_DCLevel	—	—
PDCT_AC_Component	Sinusoid	Signal [Out]	—	Pulsed_DC_Train	—
		amplitude	ac_ampl	—	—
		frequency	freq	—	—
		phase	—	—	0 rad
PDCT_DC_Level	Constant	Signal [Out]	—	PDCT_Pulsed_DC	—
		amplitude	dc_ampl	—	—

This model is designed to produce a single train of dc pulses. The model could easily be changed to provide repeated pulse streams by removing the fixed value (1) from the Formula column of the model for the attribute repetition and adding a repetition attribute to the interface table.

10.5.2 Signal model for DME_INTERROGATION

This example uses the signal model for interrogation by distance measuring equipment (DME), i.e., DME_INTERROGATION provided in Annex E of IEEE Std 1641-2010.

10.5.2.1 Model diagram

This simplified model for DME_INTERROGATION includes further examples of internally specified attributes (see Figure 95). The signal model diagram and tables are repeated here from the STD standard without any changes.

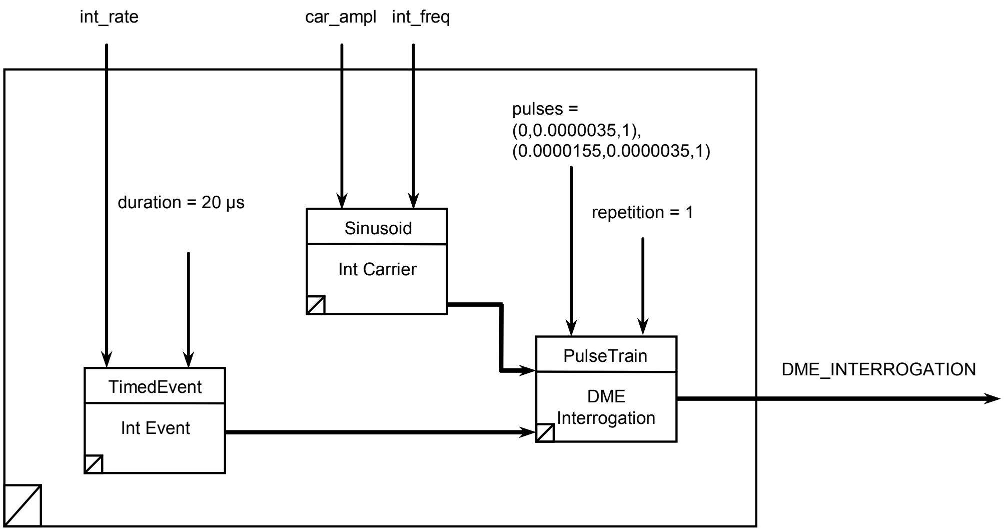


Figure 95—DME_INTERROGATION signal model

The signal model diagram shows three attributes that have fixed values: the duration of the timed event; the timing, duration, and magnitude of the pulses; and the number of repetitions of the pulses.

10.5.2.2 Interface properties

The interface properties table further restricts the interrogator frequencies and interrogation rates to those normally associated with the DME signals. The only value that a user must provide is the carrier amplitude (see Table 19).

Table 19—TSF DME_INTERROGATION interface

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Voltage	—	—
Interrogation frequency	int_freq	Frequency	1025 MHz	1025–1150 MHz

Description	Name	Type	Default	Range
Interrogation rate	int_rate	Frequency	27 Hz	27 Hz | 150 Hz

In this example, the attribute for interrogation rate (int_rate) may take only one of two values: 27 Hz or 150 Hz. This restriction is specified in the interface properties table in which the Range column shows the two possible values.

10.5.2.3 Model description

Table 20 presents the model description table for DME_INTERROGATION.

Table 20—TSF DME_INTERROGATION model

Name	Type	Terminal	Inputs	Output	Formula
DME Interrogation	PulseTrain	Signal [Out]	—	DME_ INTERROGATION	—
		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

This model description table illustrates several features not shown in earlier examples:

- — Pulse definitions (in this case a fixed definition for two pulses)
- — Use of the Sync[In] terminal
- — A calculation required to obtain the required attribute value

The pulse definition is for two pulses. The first commences immediately (after 0 s) with a duration of 3.5 µs and a level factor of unity, and the second commences after 15.5 µs with the same duration of 3.5 µs and the same level factor of unity. The unity level factor means that the amplitude of the signal being processed remains unchanged.

The output of the TimedEvent BSC (“Int Event”) is associated with sync input of the PulseTrain BSC (“DME Interrogation”). In other words, the pulse train is activated each time the BSC “Int Event” produces an output event. The duration parameter of the BSC “Int Event” is arbitrarily set at 20 µs.

The BSC “Int Event” has period as an attribute, but DME expects the value to be given as an interrogation rate. The period is calculated from the interrogation rate (int_rate) as shown in the Formula column. In STD, period and frequency may be used interchangeably.

Other default values given in the Formula column include a value for repetition of 1. In other words, only one train of pulses is output each time the BSC “DME Interrogation” is triggered. The BSC “Int Event” has a value for repetition of 0, and this value means that the events will occur continuously at a rate determined by the attribute int_rate.

This model is a simplified model to produce DME interrogation pulses with the correct timings at a constant rate. If the user wishes to produce interrogation pulses with different timings and different amplitudes (or with missing pulses, etc.), the user will need to modify the model extensively. When testing UUTs, it is often required to provide inputs that do not conform to the ideal signal in order to check the response of the UUT to such bad signals. This testing can easily be achieved by extending or modifying the signal models. Each parameter that needs to be modified under the control of the user will need to be accessible externally via an entry in the interface properties table.

10.6 Examples of signal models that process input signal

The signal models considered so far have been sources that create a signal using BSCs with specified attributes. Also, there are signal models that create a signal as a result of processing a previously established signal. Examples are SSR_RESPONSE, DME_RESPONSE and RADAR_RX_SIGNAL, all of which are supported by examples in Annex E of IEEE Std 1641-2010. Some of these models are quite complex even in their simplified form.

The first example uses the RADAR_RX_SIGNAL from Annex E of IEEE Std 1641-2010. This signal shows the incident RADAR_TX_SIGNAL being processed and returned as if the signal were being reflected by a distant object.

10.6.1 RADAR_RX_SIGNAL

The RADAR_RX_SIGNAL model takes an input radar signal and delays the signal response. A response is not created for every input transmitted radar pulse, and this phenomenon gives rise to a reply efficiency.

To achieve this reply efficiency, the RADAR_RX_SIGNAL model must detect the incoming radar pulses and suppress some individual pulses. To detect a radar pulse, an RMS monitor BSC is used with a selected gateTime. This RMS monitor BSC provides an event while the continuous rms value is greater than a nominal threshold value. This RMS monitor BSC “Event Train” is used solely to detect a signal.

10.6.1.1 Model diagram

Figure 96 presents the signal model diagram for RADAR_RX_SIGNAL.

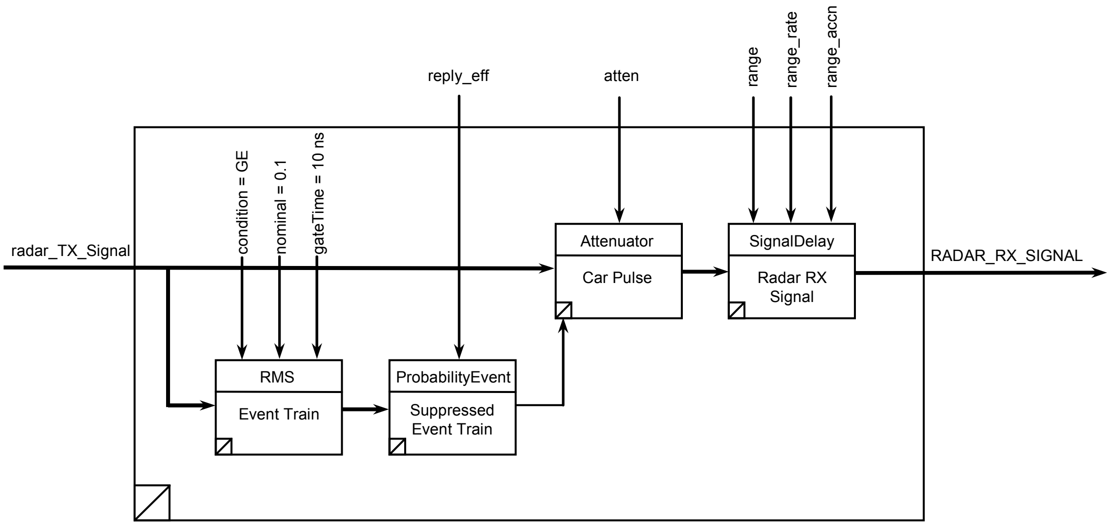


Figure 96—TSF RADAR_RX_SIGNAL

This signal model diagram shows the input radar signal (from the radar transmitter) being processed in two ways. The lower path shows the signal being linked to the Signal [In] terminal of the rms detector. The [Out] terminal is active when an input signal is present. The [Out] terminal is associated with the Signal [In] terminal of the ProbabilityEvent BSC ("Suppressed Event Train"). Depending upon the value of the probability attribute (reply_eff), an event will be generated for some of the incoming radar signals. The higher the value provided for reply efficiency (reply_eff), the higher the proportion of input radar signals that will produce events.

The main path for the incoming radar signal is associated with the input Signal [In] terminal of the attenuator BSC ("Car Pulse"). The incoming signal is gated by the events from the BSC "Suppressed Event Train," and this gating allows only a proportion of the incoming signals to pass. The signals that do pass through will be attenuated according to the value provided for the atten attribute, and this attenuation will simulate the quality of the reflected radar signal.

The signal then passes through the SignalDelay BSC ("Radar RX Signal"). This delay simulates the distance, speed, and acceleration of a target.

10.6.1.2 Interface properties

The default value for both range_rate and range_accn is zero, and this value represents a stationary target.

The attribute atten represents the attenuation of the reflected signal. This attribute has a default value of unity, which indicates that there is no loss in the reflected signal. The default value of 100% for reply_eff indicates that every transmitted radar signal is reflected (see Table 21).

Table 21—TSF RADAR_RX_SIGNAL Interface

Description	Name	Type	Default	Range
Atten	atten	Ratio	1	—
Range of simulated target	range	Distance	—	—

Description	Name	Type	Default	Range
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% to 100%
Transmitted Radar Signal	radar_TX_Signal	SignalFunction	—	—

10.6.1.3 Model description

The values for range, range_rate, and range_accn in radar terms would normally be in terms of distance (meters), speed (meters per second), and acceleration (meters per second per second). The model for the SignalDelay BSC is expecting the values to be in terms of time, ratio, and frequency, respectively. Therefore, the Formula column in Table 22 shows the conversion factor of 2/3.0e8. This factor is to convert the distance in meters to a target into the time that the radar signal would need to travel that distance twice. If the distance is specified in, e.g., nautical miles, then further mathematical conversion would be needed.

Table 22—TSF RADAR_RX_SIGNAL Model

Name	Type	Terminal	Inputs	Output	Formula
Radar RX Signal	SignalDelay	Signal [Out]	—	RADAR_RX_SIGNAL	—
		acceleration	—	—	range_accn×2/3.0e8
		delay	—	—	range×2/3.0e8
		rate	—	—	range_rate×2/3.0e8
		Signal [In]	Car Pulse	—	—
Car Pulse	Attenuator	Signal [Out]	—	Radar RX Signal	—
		gain	atten	—	—
		Signal [In]	radar_TX_Signal	—	—
		Gate[In]	Suppressed Event Train	—	—
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

Name	Type	Terminal	Inputs	Output	Formula
		condition	—	—	GE
		NOGO	—	—	—
		GO	—	—	—
		UL	—	—	—
		LL	—	—	—
		Signal [In]	radar_TX_Signal	—	—

10.6.2 Example of modulation using incoming signal

This example shows how an incoming signal may be used as a modulation signal. This type of model is relevant in situations where a signal output from the UUT is used as the modulation source for a signal, which may then be fed back to the UUT. Alternatively, it can be used where a modulator is using an external source for its modulation input.

This model is simple, yet it highlights issues that must be considered when implementing models of this type. It is a variation of the model used for AM_SIGNAL in Annex E of IEEE Std 1641-2010. The modulating signal has been removed from being inside of the model to being an input signal to the model. To differentiate this model from the previous AM_SIGNAL, this new model has been called MODULATED_SIGNAL.

10.6.2.1 Model diagram

Figure 97 presents the signal model diagram for MODULATED_SIGNAL with an incoming SignalFunction.

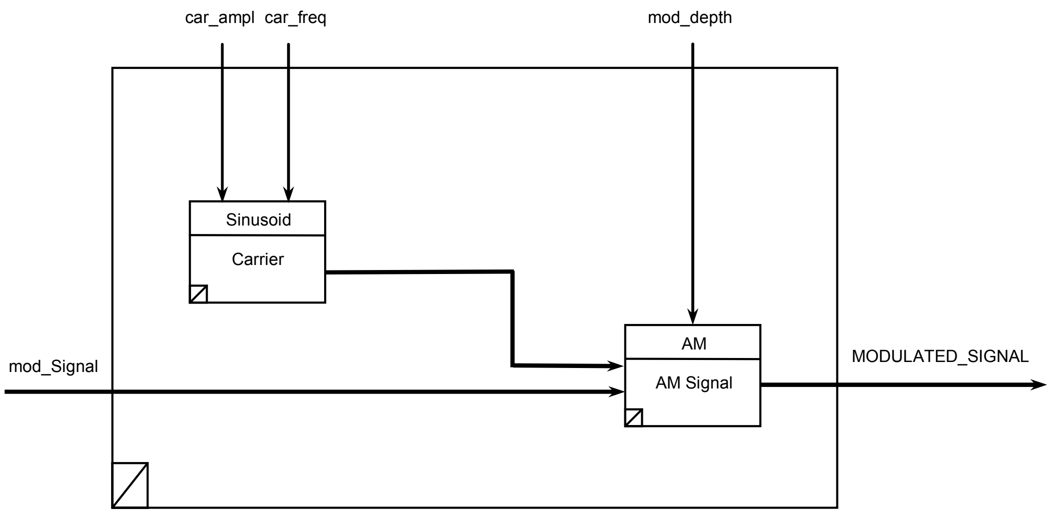


Figure 97 —Signal model with incoming SignalFunction

10.6.2.2 Interface properties

The modulation signal is provided as a SignalFunction (see Table 23).

Table 23—MODULATED_SIGNAL interface

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Voltage	—	—
Carrier frequency	car_freq	Frequency	—	—
Depth of modulation	mod_depth	Ratio	—	0 to 1
Modulation Signal	mod_Signal	SignalFunction	—	—

10.6.2.3 Model description

The modulation depth is specified by the user in the same way as it would for the AM_SIGNAL (see Table 24).

Table 24—MODULATED_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
AM Signal	AM	Signal [Out]	—	MODULATED_SIGNAL	—
		modIndex	mod_depth	—	—
		Carrier [In]	Carrier	—	—
		Signal [In]	mod_Signal	—	—
Carrier	Sinusoid	Signal [Out]	—	AM Signal	—
		amplitude	car_ampl	—	—
		frequency	car_freq	—	—
		phase	—	—	0 rad

If the input modulation signal is a sinusoid with an amplitude of 1 V, the resultant signal will be identical to that produced by the signal model AM_SIGNAL. If the input modulation signal is a sinusoid with any other amplitude, the output signal will have a different depth of modulation from that specified by the mod_depth attribute. It may even be more than 100% modulated and distort the expected signal.

A non-sinusoidal signal may be used, and the output will be modulated accordingly. In this case, the value for the depth of modulation must be carefully chosen to be compatible with the amplitude of the input signal. This caution is especially true if the amplitude of the signal varies considerably. The value for the depth of modulation must be considered with respect to the peak amplitude in order to avoid distortion.

10.7 Example of signal models that include connection BSCs

An example of a signal model that includes a connection BSC is SYNCHRO as defined in Annex E of IEEE Std 1641-2010. This signal produces three interdependent outputs. These are connected via the special connector ThreePhaseSynchro with a channelWidth of 3.

10.7.1 Definition of SYNCHRO

The SYNCHRO signal refers to three ac sinusoidal voltages whose relationships of amplitude represent the rotational shaft position of an electromechanical transducer.

10.7.2 Model diagram

The specific point of interest in the signal model diagram for SYNCHRO (see Figure 98) is the final BSC “Three Phase Synchro” (of class ThreePhaseSynchro). This BSC has three input signals, which it passes via a single Signal [Out] port that has a channelWidth of 3.

It is difficult to show formulae or expressions on a model diagram. Figure 98 does not show the correct phase relationships, i.e. it appears on the diagram that the identical shaft_angle is used as an input for all three Field BSCs. In fact the signals all differ by 2π/3 rad (120°) (see Figure 20).

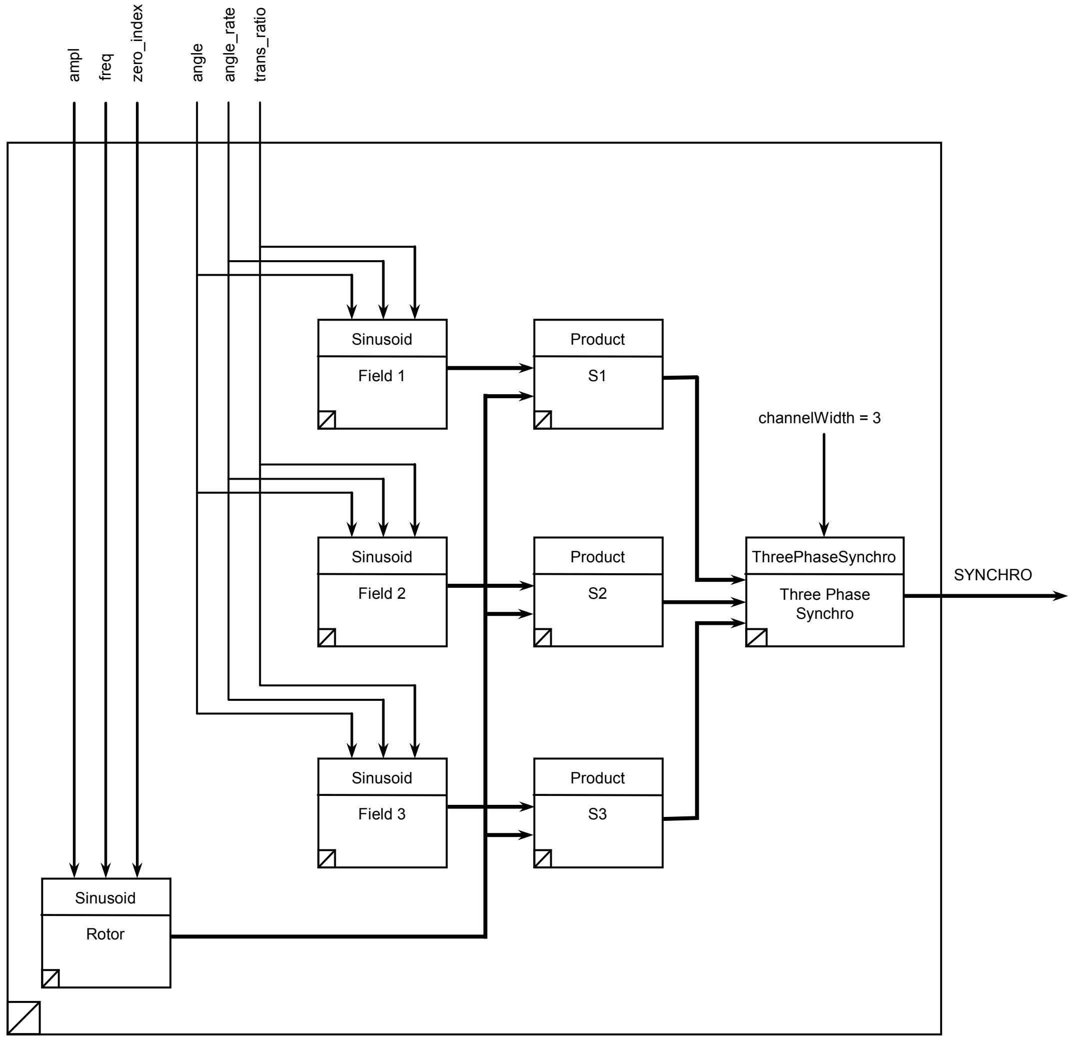


Figure 98—SYNCHRO signal model

10.7.3 Interface properties

The interface properties for the SYNCHRO signal are shown in Table 25.

Table 25—SYNCHRO interface

Description	Name	Type	Default	Range
Shaft angle	angle	PlaneAngle	0	—
Reference amplitude	ampl	Voltage	26 V	26 V to 119 V
Reference frequency	freq	Frequency	400 Hz	30 Hz to 54 kHz
Zero index	zero_index	PlaneAngle	0 rad	0 rad to 2π rad
Shaft angle rate	angle_rate	Frequency	0 Hz	—

Description	Name	Type	Default	Range
Transformer Ratio	trans_ratio	Ratio	1	—

This model does not consider the effects of angular velocity of the rotor and the quadrature voltages generated in the stator windings.

10.7.4 Model description

The connection BSC is the first BSC in the model description table (see Table 26). The three separate signals S1, S2, and S3 are inputs to the ThreePhaseSyncro BSC (“Three Phase Syncro”). The channelWidth attribute confirms that there are only three signal channels.

Table 26—TSF SYNCHRO model

Name	Type	Terminal	Inputs	Output	Formula
Three Phase Syncro	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 - (2π /3)
Field 2	Sinusoid	Signal [Out]	—	S2	—
		amplitude	—	—	trans_ratio
		frequency	—	—	angle_rate
		phase	angle	—	—
Field 3	Sinusoid	Signal [Out]	—	S3	—

Name	Type	Terminal	Inputs	Output	Formula
		amplitude	—	—	trans_ratio
		frequency	—	—	angle_rate
		phase	—	—	angle + (2π/3)
Rotor	Sinusoid	Signal [Out]	—	S2, S1, S3	—
		amplitude	ampl	—	—
		frequency	freq	—	—
		phase	zero_index	—	—

10.8 SignalDelay TSF model for SWEEP

10.8.1 Model diagram

This TSF model shows how the SWEEP element used in the ac amplifier test described in 8.5.2.6 may be created using a SignalDelay BSC (see Figure 99).

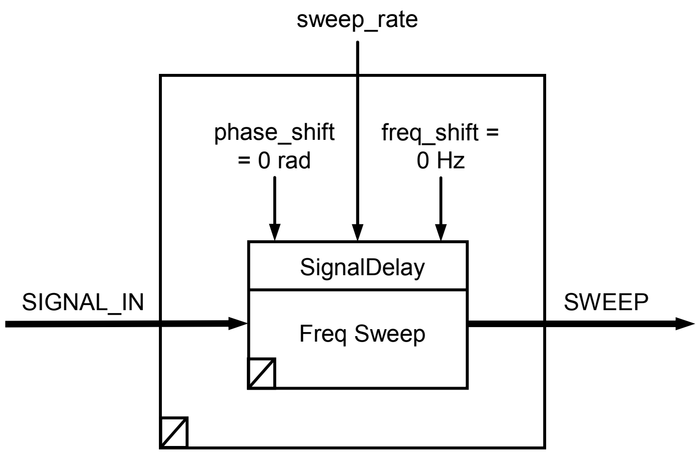


Figure 99—SWEEP signal model

10.8.2 Interface properties

The interface properties for the SWEEP signal are shown in Table 27.

Table 27—SWEEP interface

Description	Name	Type	Default	Range
Sweep Rate	sweep_rate	Frequency	0	—
Input Signal	signal_in	SignalFunction	—	—

10.8.3 Model description

In this model, the mappings are as follows:

— phase_shift maps to the delay attribute because any delay in the signal appears as a phase shift with no change in the output frequency.

— freq_shift maps to the rate attribute because it represents a fixed frequency shift to the input signal frequency.

— sweep_rate maps to the acceleration attribute because the change of frequency of the input signal varies with time.

In the specific example required for the amplifier test, the phase_shift and freq_shift attributes are not used, and their values are preset to zero in the model (see Table 28).

Table 28—SWEEP model

Name	Type	Terminal	Inputs	Output	Formula
Freq Sweep	SignalDelay	Signal [Out]	—	SWEEP	—
		acceleration	—	—	-(sweep_rate)
		delay	—	—	0 s
		rate	—	—	0
		Signal [In]	signal_in	—	—

The acceleration attribute is equal to the negated value of the sweep_rate because the delay BSC is being used to provide a sweep of increasing frequency.

10.9 TSF model for linear sweep using frequency modulation (FM) BSC

The FM BSC forms a useful basis for a sweep, which may be used in place of the delay BSC. The FM BSC varies its output frequency in proportion to the input amplitude. Changing this input with the output from a SingleRamp BSC generates a corresponding change in the output frequency. The TSF illustrated in Figure 100 uses this principle to provide a linear frequency sweep.

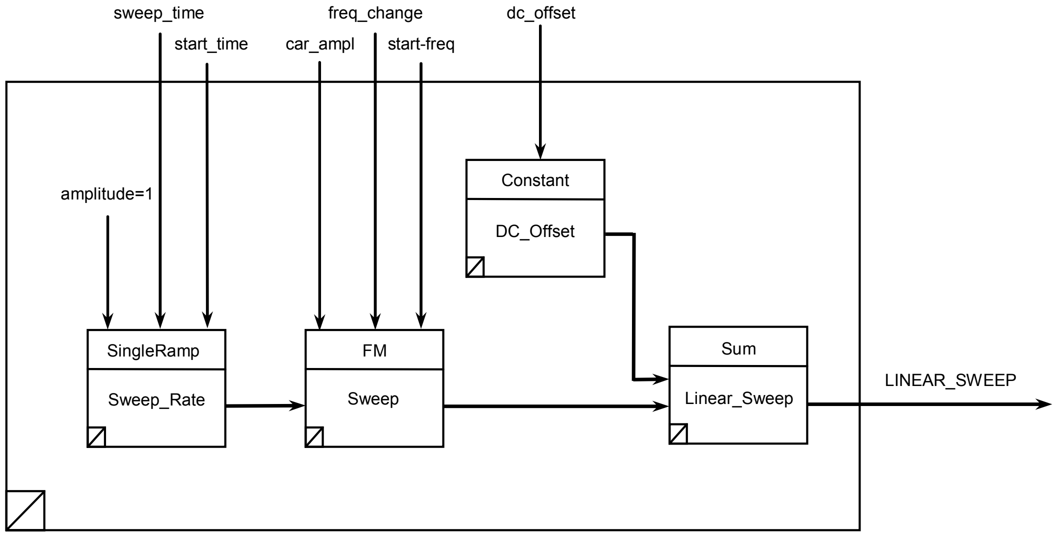


Figure 100 —Linear sweep using FM BSC

The interface properties table is shown in Table 29. The model information for this TSF is provided in XML format (see 10.14.3).

Table 29—LINEAR_SWEEP Interface

Description	Name	Type	Default	Range
Sweep time	sweep_time	Time	—	—
Start time	start_time	Time	0 s	—
Carrier amplitude	car_ampl	Physical	—	—
Start frequency	start_freq	Frequency	—	—
Frequency change	freq_change	Frequency	—	—
DC Offset	dc_offset	Physical	0	—

10.10 TSF model for logarithmic sweep using FM BSC

The FM BSC may also be used to produce a logarithmic sweep. In this case, the linear ramp used at the FM BSC's input for the linear sweep is replaced by an exponential input. The TSF model illustrated in Figure 101 will provide a logarithmic frequency sweep.

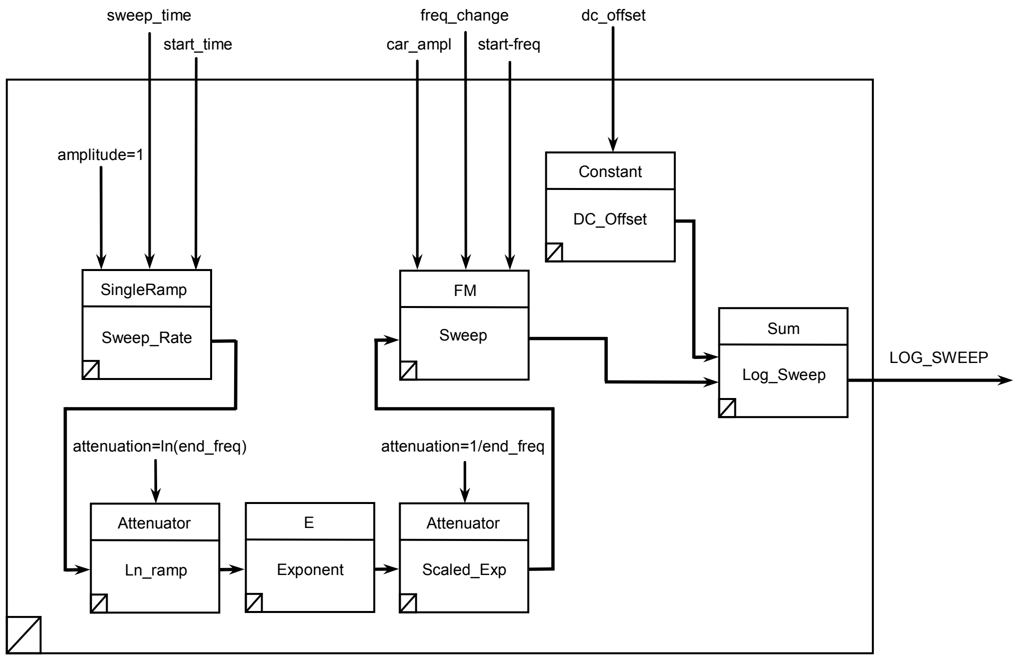


Figure 101 —Logarithmic sweep using FM BSC

An E BSC is used to convert the linear ramp into an exponential ramp. The two attenuation BSCs used each side of the E BSC are used to achieve the correct shape and scaling of the exponential signal applied to the FM BSC. The reason for these BSCs may be explained as follows. The FM BSC requires an input changing from (near) zero to one to achieve the correct start frequency (start_freq) and end frequency (end_freq), where end_freq = start_freq + freq_change. If an E BSC were simply inserted between the SingleRamp and FM BSCs, the input to the FM BSC would change from (near) zero to e (2.718). Inserting the attenuator BSC before the E BSC with a gain of ln(end_freq) provides an exponential ramp from (near) zero to the end frequency value. This ramp is then passed through another attenuator, which divides by the end frequency to provide an exponential ramp going between (near) zero and one. The TSF model includes the calculations required to provide the correct values for different frequencies.

The interface properties table is shown in Table 30. The model information for this TSF is provided in XML format (see 10.14.4).

Table 30 —LOG_SWEEP Interface

Description	Name	Type	Default	Range
Sweep time	sweep_time	Time	—	—
Start time	start_time	Time	0 s	—
Carrier amplitude	car_ampl	Physical	—	—
Start frequency	start_freq	Frequency	—	—
Frequency change	freq_change	Frequency	—	—
DC Offset	dc_offset	Physical	0	—

10.11 TSF attributes mappings and formulae

Many of the previous examples illustrate the fact that TSF attributes are mapped to their internal model attributes via formulae. Examples of these attributes within the STD standard are 2π or 1/prf. The expressions tend to be quite simple and obvious so that implementation is not difficult or ambiguous.

The use of these attribute mapping formulae is not limited just to simple expressions. The use of formulae with the STD standard provides a powerful tool for TSFs to provide the following:

— Type conversion
— Text manipulation
— Measurement analysis

Because the BSCs in the STD standard are defined utilizing Haskell 98 [B15], it would seem reasonable that these mathematical expressions can also be defined in that syntax. The problem with this approach is that although such expressions are written to a standard that is nonproprietary, these definitions are not widely recognized or used across the test and measurement industry. The STD standard does not exclude the use of different language expressions for attribute mapping, but may prevent TSFs being portable across different implementations because not all implementations may support proprietary language expressions. Care must be used when selecting the expression language to ensure that it is portable across the maximum number of implementations. The examples here use Haskell 98.

10.11.1 Text manipulation example

As an example of the type of expressions that are supported by the STD standard, consider the TSF RS232 where ***data_word*** is the string data wishing to be sent over the bus. There currently is no BSC model relying on the TIA-232 [B16] definition, which specifies its behavior with reference to that standard. An RS232 TSF model could have been defined utilizing a SerialDigital BSC and using an attribute expression, mapping the data and ***data_word*** attributes with a formula that converts a string value such as 'Hello World' into its digital logic values 'LLLHLLHL HLHLLHHL LLHHLHHL LLHHLHHL HHHHLHHL LLLLLHLL HHHLHLHL HHHHLHHL LHLLHHHL LLHHLHHL LLHLLHHL'. The following code would achieve the required conversion:

NOTE—This example assumes odd parity, two start and stop bits.

```
>map(map(\x->if x then 'H' else 'L'))$
>map(\y->[True,False]++
        (map(testBit(intToInt8 $ ord y)) [0,1,2,3,4,5,6,7])++
                                         [odd(ord y), True, False])
>   data_word
```

The above code does the following (line by line):

— Converts true and false to H and L.
— Creates the start bits.
— Inserts the ASCII[8] data bits in least significant bit order.
— Adds the odd parity and two stop bits.

[8] American Standard Code for Information Interchange.

The attribute mappings provide for complex mappings between attributes and converting between different attribute types. When it is required to receive or measure a value, the inverse mapping is applied; in this example, the digital string is turned back into an ASCII string.

10.11.2 Measurement analysis example

If it was required to extract the median of a set of measurement values, the following formula could be used (for an odd set of values):

```
(qsort measvalues)!!((length+1)/2)

where qsort orders values in the order smallest to largest:

qsort []     = []
qsort (x:xs) = qsort elts_lt_x ++ [x] ++ qsort elts_greq_x
       where

elts_lt_x   = [y | y <- xs, y < x]
            elts_greq_x = [y | y <- xs, y >= x]
```

NOTE—The median is the value of the point that has half the data smaller than that point and half the data larger than that point.

10.12 Synchronization of signal model

A TSF signal model may be synchronized in the same way as a BSC. When a **Sync** event arrives at the sync input of a TSF model, the whole model is started or restarted. This behavior may be achieved in the TSF model by applying the **Sync** event to all the BSCs within the model.

When a signal model is synchronized by an external event, all the BSCs within the TSF model will not become active until the event occurs, and then the component BSCs are all started together. Any subsequent events arriving at the TSF model's sync input will have the effect of restarting the component BSCs.

This synchronization is illustrated in Figure 102, which shows an AM_SIGNAL TSF model with all the sync inputs connected to the external sync input.

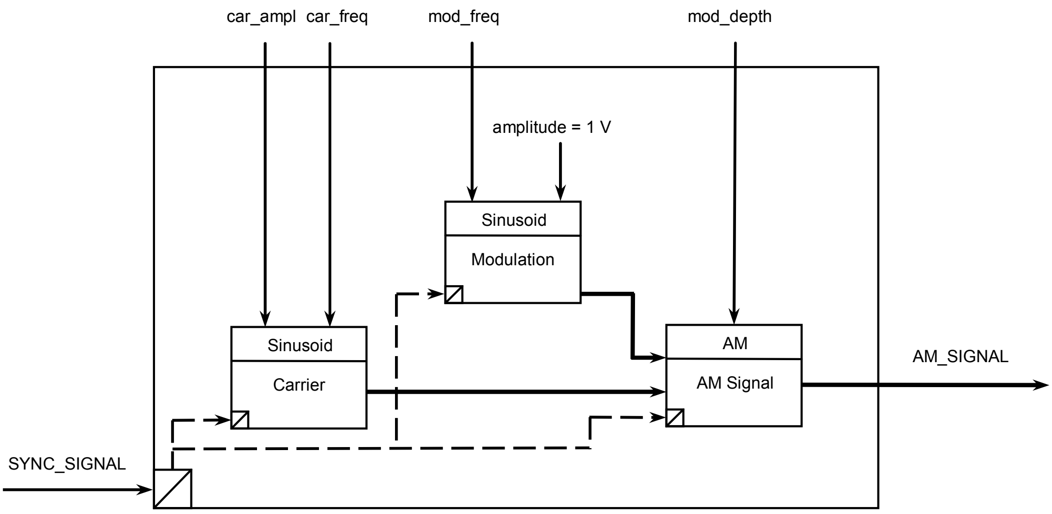


Figure 102 —AM_SIGNAL model with all sync inputs connected

10.12.1 Signal model without explicit external-to-internal synchronization

To illustrate the operation of a signal model with an internal synchronization path, consider the signal model for DME_Interrogation as described in 10.5.2. Figure 103 shows an internal sync connection from BSC “Int Event” to the sync input of the BSC “DME Interrogation.”

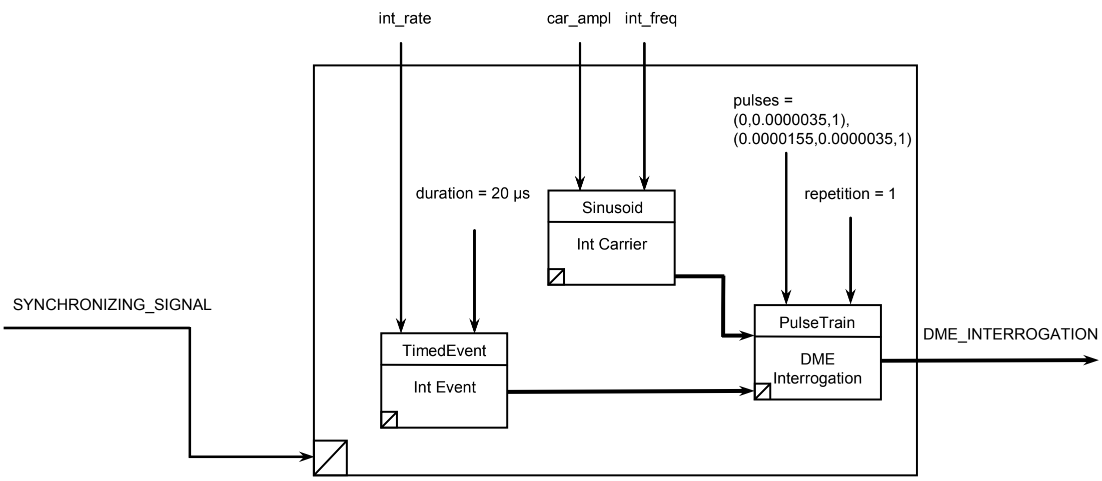


Figure 103 —Signal model with sync input

BSCs can accept only a single **Sync** and **Gate** signal. As the BSC “DME Interrogation” already has a sync input connection, it cannot receive a sync input from the TSF model external sync input. In this case, the convention with TSF model diagrams is that all unused BSC **Sync** references are connected to the external TSF sync input. This has been chosen for convenience as it allows the omission of many lines from diagrams and entries in TSF model description tables. The real situation could be shown as in Figure 104, in which the implicit sync connections are shown as dotted lines.

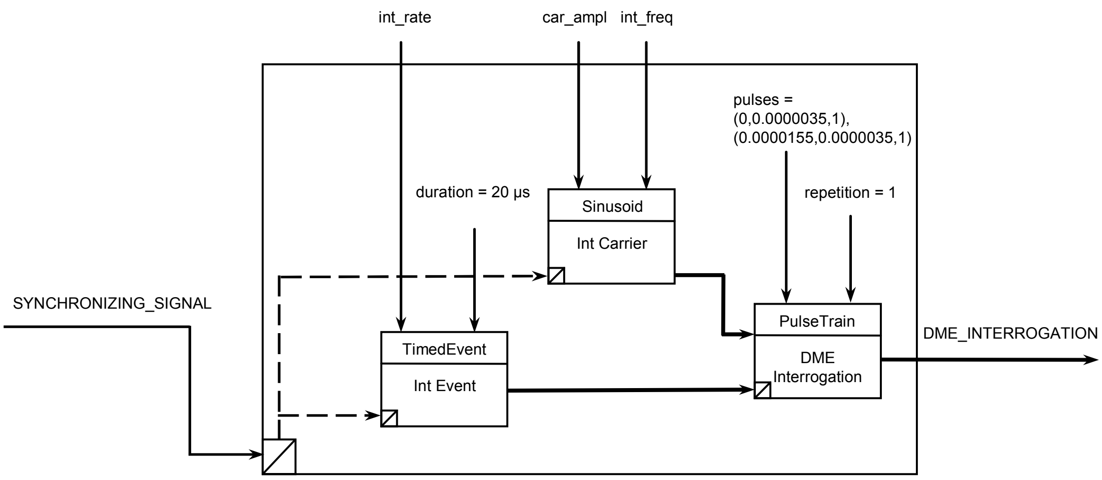


Figure 104 —Signal model showing implicit sync connections

This detail could be represented in the model description table by adding a Sync[In] entry for every BSC in the model as shown in Table 31. For the sake of brevity, these entries are normally omitted.

Table 31 —Modified DME interrogation model

Name	Type	Terminal	Inputs	Output	Formula
DME Interrogation	PulseTrain	Signal [Out]	—	DME_ INTERROGATION	—
		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
		Sync[In]	Event	—	—
Int Event	TimedEvent	Event [Out]	—	DME Interrogation	—
		delay	—	—	0 s
		duration	—	—	20 µs

Name	Type	Terminal	Inputs	Output	Formula
		period	—	—	1/int_rate
		repetition	—	—	0
		Sync[In]	Event	—	—

To better understand the operation of the TSF model, it is convenient to consider the model without the TSF surrounding box and consider the operation as a collection of interconnected BSCs as illustrated in Figure 105. Removing the box does result in the loss of some information, such as which attributes are available to the user, but this detail is not important to the description of the operation.

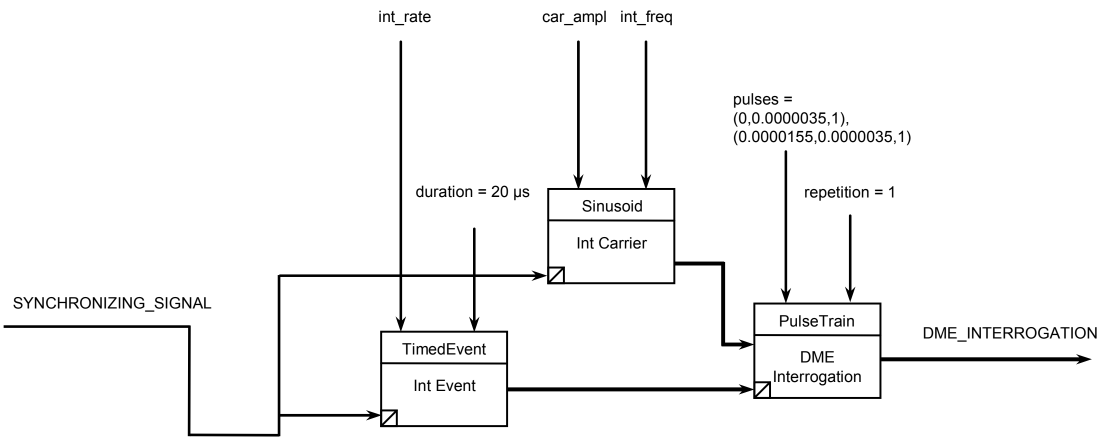


Figure 105 —Signal model as collection of BSCs

The operation of the TSF may be explained by referring to the description of the operation of BSCs (see 0). The TimedEvent and sinusoid BSCs start immediately when the SYNCHRONIZING_SIGNAL event occurs. The PulseTrain BSC starts as soon as it receives an event at its **Sync** reference from the TimedEvent BSC (see Table 32). The sinusoid BSC will start with a phase angle of zero because the sinusoid BSC always defaults to an initial phase angle of zero if it is not otherwise specified. The model outputs two pulses each time the TimedEvent BSC generates an event. These events occur at a frequency specified by int_rate.

Table 32—State of BSCs around first Sync event

	TimedEvent BSC		Sinusoid BSC		PulseTrain BSC	
	"Int Event"		"Int Carrier"		"DME Interrogation"	
At T=0	Inactive	Waiting for SYNCHRONIZING _SIGNAL	Inactive	Waiting for SYNCHRONIZING _SIGNAL	Inactive	Waitng for "Int Event"
Before Sync	Inactive	Waiting for SYNCHRONIZING _SIGNAL	Inactive	Waiting for SYNCHRONIZING _SIGNAL	Inactive	Waitng for "Int Event"
At Sync	Active	Pulse stream starts with 20 µs pulse	Active	Sinusoid starts at phase angle = 0	Active (on receipt of "Int Event" signal)	Pulse stream starts
After Sync	Active	Pulse stream continues with 20 µs pulses at int_rate	Active	Sinusoid continues	Active	Stream of 2 pulses on every "Int Event" signal

Subsequent events occurring at the external sync input will restart both the TimedEvent and sinusoid BSCs immediately (see Table 33). This results in a new stream of pulses with the initial phase of the carrier reset to zero for each event.

Table 33—State of BSCs around subsequent Sync events

	TimedEvent BSC		Sinusoid BSC		PulseTrain BSC	
	"Int Event"		"Int Carrier"		"DME Interrogation"	
Before Sync	Active	Pulse stream continues with 20 µs pulses at int_rate	Active	Sinusoid continues	Active	Stream of 2 pulses on every "Int Event" signal
At Sync	Active	Pulse stream restarts with 20 µs pulse (irrespective of current state)	Active	Sinusoid restarts at phase angle = 0	Active	Pulse stream restarts (irrespective of current state)
After Sync	Active	Pulse stream continues with 20 µs pulses at int_rate	Active	Sinusoid continues	Active	Stream of 2 pulses on every "Int Event" signal

10.12.2 Signal model with explicit external-to-internal synchronization

It is possible to include explicit external-to-internal synchronization. This situation is illustrated in the modified version of the DME_Interrogation signal model shown in Figure 106. The model includes an external synchronization connection to only one BSC. By convention, only the specific BSCs connected to the external sync input are synchronized by the external event. The other BSCs start at time zero when the Out.Run method is called unless their sync input is connected to an internal event.

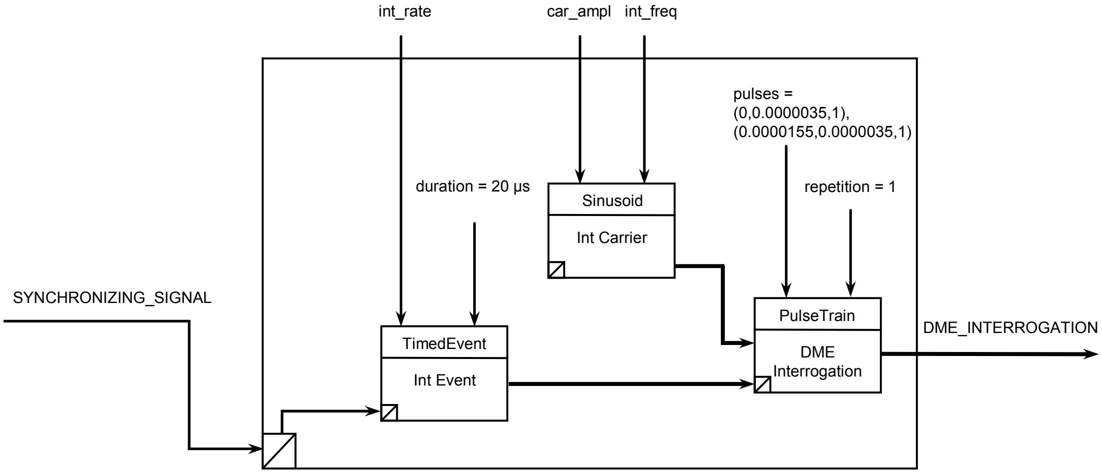


Figure 106 —Signal model with explicit sync input to TimedEvent BSC

The model in Figure 106 is indicating that only the TimedEvent BSC is synchronized each time an external event arrives at the sync input of the signal model.

When the signal runs, the sinusoid BSC will start immediately and output a sinusoid with an initial phase angle of zero. The first time an external event occurs at the sync input of the TSF model, the TimedEvent BSC starts. The PulseTrain BSC starts when it receives a **Sync** event from the TimedEvent BSC (see Table 34). The effect is to delay the start of the output pulse train starts until the SYNCHRONIZING_SIGNAL event occurs.

Table 34—State of BSCs around first Sync event

	TimedEvent BSC		**Sinusoid BSC**		**PulseTrain BSC**	
	“Int Event”		**“Int Carrier”**		**“DME Interrogation”**	
At T=0	Inactive	Waiting for SYNCHRONIZING _SIGNAL	Active	Sinusoid starts at phase angle = 0	Inactive	Waitng for “Int Event”
Before Sync	Inactive	Waiting for SYNCHRONIZING _SIGNAL	Active	Sinusoid continues	Inactive	Waitng for “Int Event”
At Sync	Active	Pulse stream starts with 20 µs pulse	Active	Sinusoid continues	Active (on receipt of “Int Event” signal)	Pulse stream starts
After Sync	Active	Pulse stream continues with 20 µs pulses at int_rate	Active	Sinusoid continues	Active	Stream of 2 pulses on every “Int Event” signal

When subsequent external events arrive at the sync input of the TSF model, the sinusoid BSC continues to run unchanged, i.e., it is not resynchronized. The TimedEvent BSC is restarted, and as a result, the pulse train is restarted (see Table 35).

Table 35—State of BSCs around subsequent Sync events

	TimedEvent BSC		Sinusoid BSC		PulseTrain BSC	
	“Int Event”		“Int Carrier”		“DME Interrogation”	
Before Sync	Active	Pulse stream continues with 20 µs pulses at int_rate	Active	Sinusoid continues	Active	Stream of 2 pulses on every “Int Event” signal
At Sync	Active	Pulse stream restarts with 20 µs pulse (irrespective of current state)	Active	Sinusoid continues	Active	Pulse stream restarts (irrespective of current state)
After Sync	Active	Pulse stream continues with 20 µs pulses at int_rate	Active	Sinusoid continues	Active	Stream of 2 pulses on every “Int Event” signal

The lack of a specific sync input connection to the sinusoid BSC indicates that the synchronization of the sinusoid is not important because it is a “don’t care” condition. In practice, the sinusoid starts with a zero initial phase angle when it is activated because it starts with a default of zero. As only the TimedEvent BSC (“Int Event”) is synchronized, the sinusoid BSC (“Int Carrier”) will continue to run without being resynchronized. So, for each SYNCHRONIZING_SIGNAL event, the carrier of the new pulse train will appear to start with a random initial phase angle.

Table 36 shows the entry for the explicit connection from the external **Sync** to the **Sync** reference of the TimedEvent BSC in the last entry in the table.

Table 36—Model with explicit internal Sync

Name	Type	Terminal	Inputs	Output	Formula
DME Interrogation	PulseTrain	Signal [Out]	—	DME_ INTERROGATION	—
		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

Name	Type	Terminal	Inputs	Output	Formula
		Sync[In]	Event	—	—

10.12.3 Signal model with synchronization and random start of BSC

If the requirement is for a BSC to start with random initial conditions, then this requirement must be built into the signal model.

To illustrate this case, a further modification is required to the DME_INTERROGATION signal model. The model now includes a randomizing input to the phase attribute of the sinusoid BSC (see Figure 107).

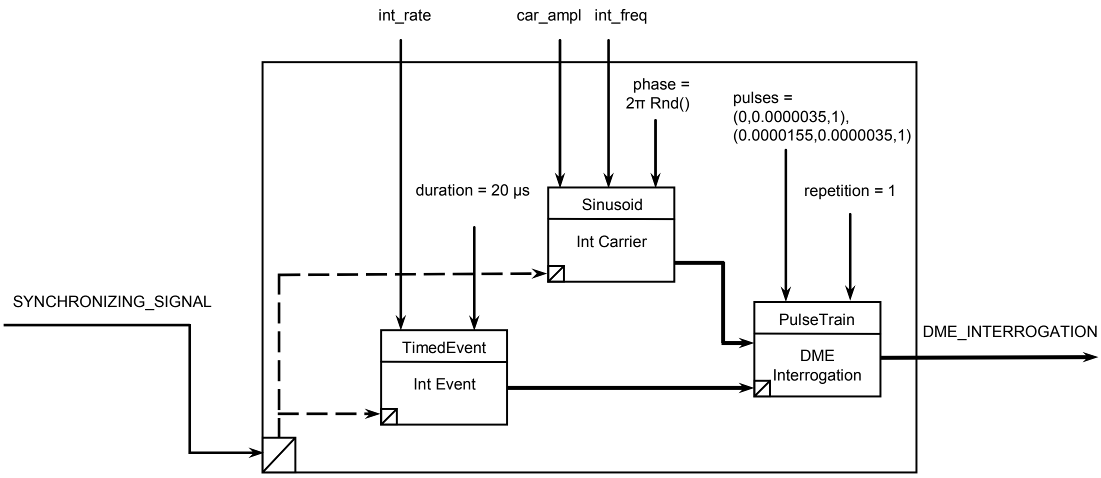


Figure 107 —Signal model with randomizing input to sinusoid BSC

As this case involves an equation, the model description table is also included (see Table 37). Figure 107 also shows the implicit sync connections as dotted lines.

Table 37 —DME_INTERROGATION model with randomized input

Name	Type	Terminal	Inputs	Output	Formula
DME Interrogation	PulseTrain	Signal [Out]	—	DME_ INTERROGATION	—
		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	—	—

Name	Type	Terminal	Inputs	Output	Formula
		frequency	int_freq	—	—
		phase	—	—	$2\pi \times$ Rnd() rad
Int Event	TimedEvent	Event [Out]	—	DME Interrogation	—
		delay	—	—	0 s
		duration	—	—	20 µs
		period	—	—	1/int_rate
		repetition	—	—	0
NOTE— Rnd() indicates a function that provides a random number between 0 and 1.					

This model operates in a similar manner to the version described in 9.4. When the first external event occurs, both the sinusoid and TimedEvent BSCs become active (see Table 38). As the phase attribute has a random function as its input value, the sinusoid BSC does not always start with zero initial phase angle. The random function generates a value between 0 rad and 2π rad (0° to 360°). This function ensures that the first pulse train has a carrier with random initial phase angle.

Table 38—State of BSCs around first Sync event

	TimedEvent BSC		Sinusoid BSC		PulseTrain BSC	
	"Int Event"		"Int Carrier"		"DME Interrogation"	
At T=0	Inactive	Waiting for SYNCHRONIZING _SIGNAL	Inactive	Waiting for SYNCHRONIZING _SIGNAL	Inactive	Waitng for "Int Event"
Before Sync	Inactive	Waiting for SYNCHRONIZING _SIGNAL	Inactive	Waiting for SYNCHRONIZING _SIGNAL	Inactive	Waitng for "Int Event"
At Sync	Active	Pulse stream starts with 20 µs pulse	Active	Sinusoid starts with random phase angle	Active (on receipt of "Int Event" signal)	Pulse stream starts
After Sync	Active	Pulse stream continues with 20 µs pulses at int_rate	Active	Sinusoid continues	Active	Stream of 2 pulses on every "Int Event" signal

When subsequent external events occur, both the TimedEvent and sinusoid BSCs are restarted. This step not only results in a new pulse train, but ensures that the initial phase angle of the carrier is randomized again (see Table 39).

Table 39—State of BSCs around subsequent Sync events

	TimedEvent BSC		Sinusoid BSC		PulseTrain BSC	
	"Int Event"		"Int Carrier"		"DME Interrogation"	
Before Sync	Active	Pulse stream continues with 20 µs pulses at int_rate	Active	Sinusoid continues	Active	Stream of 2 pulses on every "Int Event" signal
At Sync	Active	Pulse stream restarts with 20 µs pulse (irrespective of current state)	Active	Sinusoid restarts with random phase angle	Active	Pulse stream restarts (irrespective of current state)
After Sync	Active	Pulse stream continues with 20 µs pulses at int_rate	Active	Sinusoid continues	Active	Stream of 2 pulses on every "Int Event" signal

10.13 Gating a signal model

10.13.1 Signal models with implicit internal Gate

A TSF signal model may be gated in the same way as a BSC. When a **Gate** event arrives at the gate input of a TSF model the whole model is gated on for the duration of the event. This behavior may be achieved in the TSF model by applying the **Gate** event to final (output) BSCs within the model. Figure 108 shows the AM_SIGNAL TSF with a single implicit connection between the external gate input and the final BSC gate input.

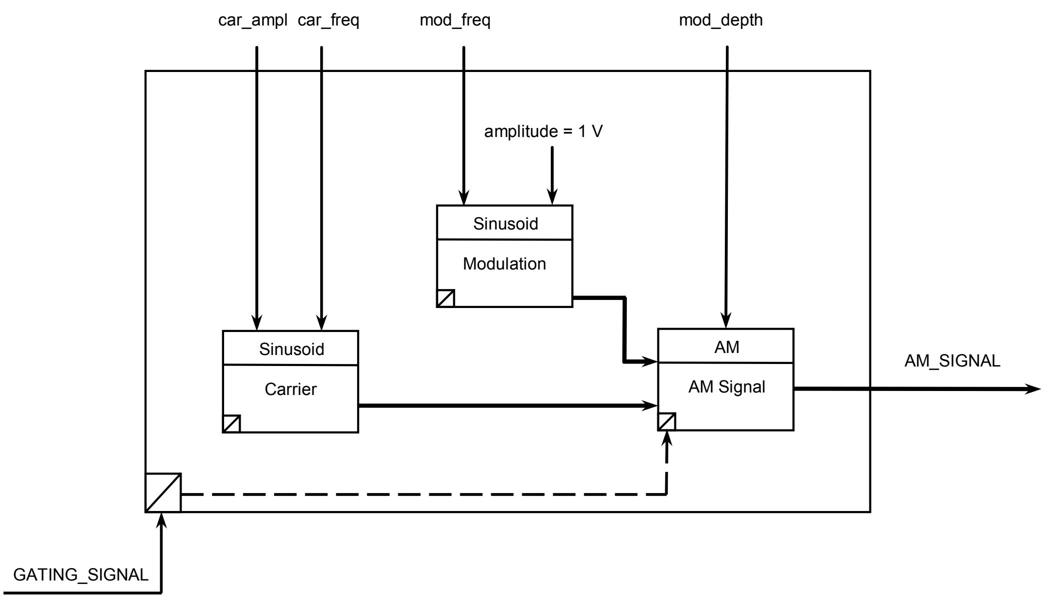


Figure 108 —Signal model with single implicit internal Gate

When the Out.Run method is called, the two sinusoid BSCs become active. The signals arriving from the sinusoid BSCs immediately start the AM BSC. The signal will only appear at the output when the GATING_SIGNAL event is present at the gate of the AM BSC. This representation of behavior breaks down if the TSF model already has an internal gate input to the final BSC, as shown by the TSF models for RADAR_TX_SIGNAL and TACAN in Annex E of IEEE Std 1641-2010. Only one event connection may be made to a gate input at any time.

A more accurate way of describing the behavior is to imagine that every TSF model has an additional BSC of class Sum between the last BSC in the model and the output. This Sum BSC has only one signal input and is gated by the external gating signal. This is illustrated in Figure 109.

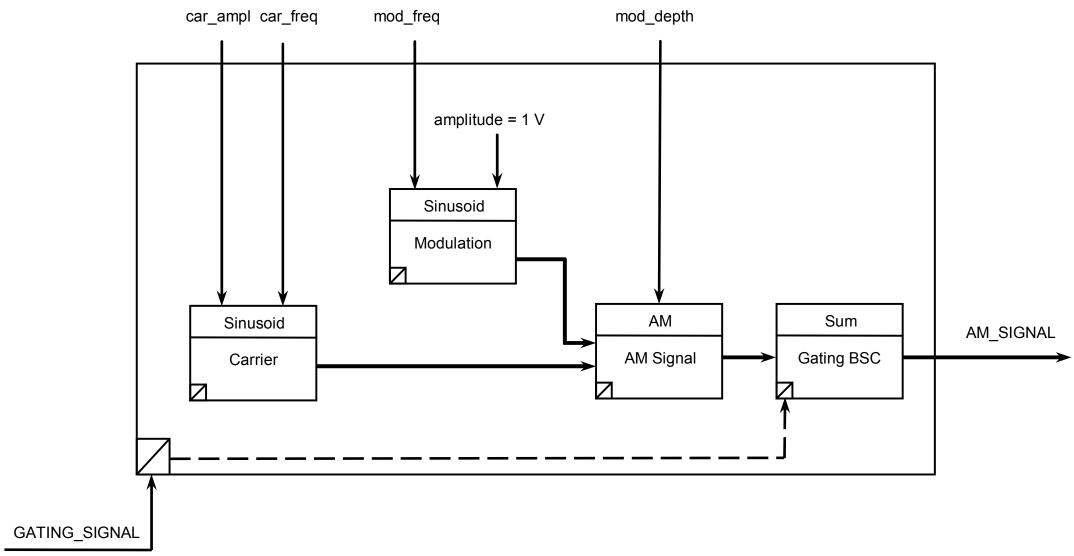


Figure 109 —Signal model with additional gating BSC

The additional Sum BSC has no effect on the model when there is no external event connected. When an external event is used to gate the TSF model it only gates the output signal without having any further effect on the operation of the model.

10.13.2 Signal model with explicit external-to-internal gating

Explicit connections may be used between the external gate and specific BSC gate inputs. By convention, only the specific BSCs connected to the external gate input are gated by the external event. The other BSCs are gated from time zero when the Out.Run method is called unless their gate input is connected to an internal event. This can be illustrated by considering what happens when a gating signal is applied to the TSF shown in Figure 110.

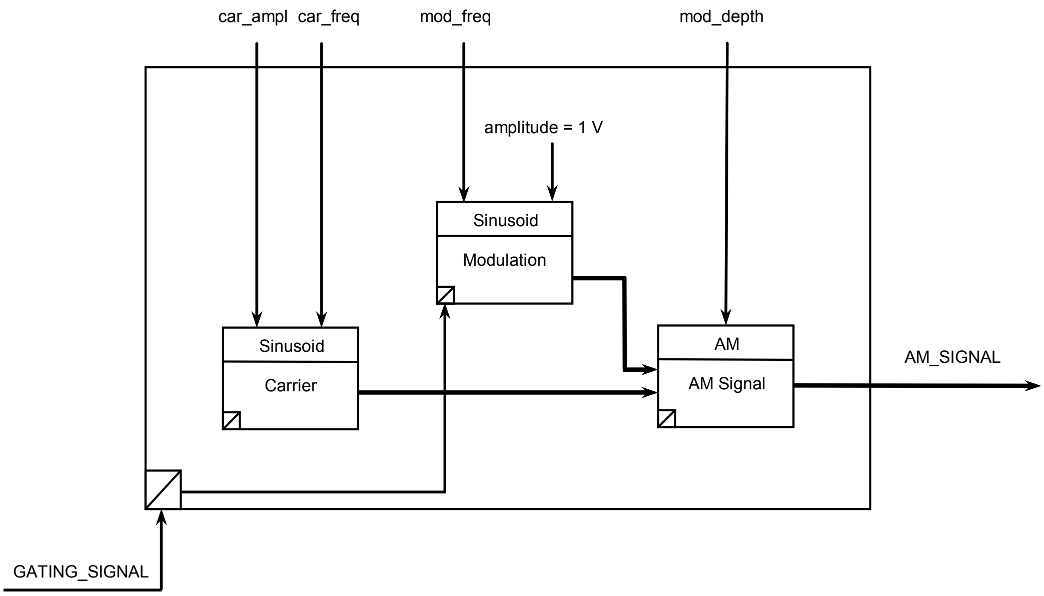


Figure 110 —Signal model showing explicit internal Gate

When the Out.Run method is called the BSCs become active, but the output from the sinusoid BSC (“Modulation”) is inhibited until a **Gate** event is present. The external GATING_SIGNAL will have the effect of switching the modulation signal on and off. The output of the TSF will be present the whole time, but there will be periods of plain carrier and modulated carrier according to the state of GATING_SIGNAL.

10.13.3 Signal model with explicit internal gating only

A TSF model which only has gate connections within the model may be modeled exactly as described in the section describing the SignalFunctions and events (see 9.4).

10.14 Use of XML to specify TSF signal model information

The XML is used in many environments and situations to store self documented information. Signal information is ideally suited for this method of data representation. The STD standard provides the details of the BSCs and the TSF models in XML.

Annex I of IEEE Std 1641-2010 provides an XML representation of the BSCs defined in Annex B of IEEE Std 1641-2010. As it is not expected that users will be required to create further BSC models no further explanation is given in this guide.

The contents of the TSF library in Annex E of IEEE Std 1641-2010 is provided in XML format in Annex J of IEEE Std 1641-2010. This annex also includes the Schema to be used when creating XML representations of new TSF models.

As an example, the relationship between the tabular definitions for RADAR_RX_SIGNAL in Annex E of IEEE Std 1641-2010 and the XML description in Annex J of IEEE Std 1641-2010 is provided.

10.14.1 XML for RADAR_RX_SIGNAL

This is not intended to be a tutorial for the use of XML, but merely to show the relationship of the entries in the tabular format and the XML format. Extracts from the XML description are given together with the equivalent extract from the tabular description.

10.14.1.1 General Information

The TSF Class name is retained from Annex E of IEEE Std 1641-2010, i.e., RADAR_RX_SIGNAL.

```
<TSF name="RADAR_RX_SIGNAL" uuid="{7D866851-EDFA-4A99-9801-658439350C9C}">
```

The major elements in the XML description are interface, model and description. The interface includes the TSF Class name RADAR_RX_SIGNAL and the textual definition of the signal.

```
<xs:element name="RADAR_RX_SIGNAL">
<xs:annotation>
   <xs:documentation>The Radar_RX_Signal provides an appropriate delayed
   signal response to an input Radar signal.
   </xs:documentation>
</xs:annotation>
```

10.14.1.2 Interface data

The five attributes from the interface properties table (see Table 21) are each listed between <xs:attribute> tags. The description is given as text between <xs:documentation> tags. The attribute name is taken from the Name column. The type is taken from the Type column. The default is taken from the Default column. The range details are presented between the <xs:restriction> tags. If the column has no entry, the equivalent XML tags are omitted.

Description	Name	Type	Default	Range
Atten	atten	Ratio	1	

```
<xs:attribute name="atten" type="Ratio" default="1">
   <xs:annotation>
      <xs:documentation>Atten</xs:documentation>
   </xs:annotation>
</xs:attribute>
```

Range of simulated target	range	Distance		

```
<xs:attribute name="range" type="Distance">
   <xs:annotation>
      <xs:documentation>Range of simulated target
      </xs:documentation>
   </xs:annotation>
</xs:attribute>
```

Rate of change of rate change	range_accn	Acceleration	0	

```
<xs:attribute name="range_accn" type="Acceleration" default="0">
   <xs:annotation>
      <xs:documentation>Rate of change of rate change
      </xs:documentation>
   </xs:annotation>
</xs:attribute>
```

Rate of change of target range	range_rate	Speed	0	

```
<xs:attribute name="range_rate" type="Speed" default="0">
   <xs:annotation>
      <xs:documentation>Rate of change of target range
      </xs:documentation>
   </xs:annotation>
</xs:attribute>
```

Proportion of Tx pulses returned	reply_eff	Ratio	100%	0% to 100%

```
<xs:attribute name="reply_eff" default="100%">
   <xs:annotation>
      <xs:documentation>Proportion of Tx pulses returned
      </xs:documentation>
   </xs:annotation>
   <xs:simpleType>
      <xs:restriction base="Ratio">
         <xs:minInclusive value="0 "/>
         <xs:maxInclusive value=" 100%"/>
      </xs:restriction>
   </xs:simpleType>
</xs:attribute>
```

The final entry in the interface properties table is not an attribute but an input SignalFunction that is included in the signal model by making RADAR_RX_SIGNAL an extension of the baseclass SIgnalFunctionType. The model description table contains four BSCs and their relationships.

10.14.1.3 Signal model

The XML signal model is a more compact form of the model description table. Cells in the model description table that do not contain any data are omitted from the XML description. The full table is shown in Table 22, and the segments shown below are taken from that table BSC by BSC. The first BSC entry to be considered is actually the last BSC in the table, and so on.

The first entry is for the BSC “Event Train,” which is of class RMS.

Name	Type	Terminal	Inputs	Output	Formula
Event Train	RMS	[Out]	—	Suppressed Event Train	—
		measurement	—	—	—
		sample	—	—	—
		count	—	—	—
		gateTime	—	—	1.0e-8
		nominal	—	—	0.1
		condition	—	—	GE
		NOGO	—	—	—
		GO	—	—	—
		UL	—	—	—

Name	Type	Terminal	Inputs	Output	Formula
		LL	—	—	—
		Signal [In]	radar_TX_Signal	—	—

```
<RMS name="Event_Train" gateTime="1.0e-8" nominal="0.1" condition="GE"
In="radar_TX_Signal"/>
```

Suppressed Event Train	ProbabilityEvent	Event [Out]	—	Car Pulse	—
		seed	—	—	0
		probability	reply_eff	—	—
		Signal [In]	Event Train	—	—

```
<ProbabilityEvent name="Suppressed_Event_Train" probability="(reply_eff)"
In="Event_Train"/>
```

Car Pulse	Attenuator	Signal [Out]	—	Radar RX Signal	—
		gain	atten	—	—
		Signal [In]	radar_TX_Signal	—	—
		Gate[In]	Suppressed Event Train	—	—

```
<Attenuator name="Car_Pulse" gain="atten" Gate="Suppressed_Event_Train"
In="radar_TX_Signal"/>
```

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	—	—

```
<SignalDelay name="Radar_RX_Signal" acceleration="(range_accn*2/3.0e8)"
delay="(range*2/3.0e8)" rate="(range_rate*2/3.0e8)" In="Car_Pulse"/>
```

10.14.1.4 Rules and notes

Any additional information that is needed, such as the rules and notes that accompany the tabular format, is included in the XML definition bounded by the <description> tags.

```
<description>
   <notes>
This annex describes a transmitted signal as a reference. Thus the TSF library provides a
description for both the transmitted (Radar_TX_Signal) and received (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 (giving rise to a
"reply efficiency").

To achieve this 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 provides an event while the continuous rms value is greater than a
nominal threshold value. This 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.
   </notes>
   <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.
   </rules>
</description>
```

10.14.1.5 Complete XML description for RADAR_RX_SIGNAL

The above extracts do not provide the complete XML description for the SignalFunction RADAR_TX_SIGNAL so here is the complete XML description.

```
   <TSF name="RADAR_RX_SIGNAL" uuid="{7D866851-EDFA-4A99-9801-658439350C9C}">
      <interface>
         <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
         elementFormDefault="qualified">
            <xs:element name="RADAR_RX_SIGNAL">
               <xs:annotation>
                  <xs:documentation>The Radar_RX_Signal provides an appropriate delayed
                  signal response to an input Radar signal.
                  </xs:documentation>
               </xs:annotation>
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="SignalFunctionType">
                        <xs:attribute name="atten" type="Ratio" default="1">
                           <xs:annotation>
                              <xs:documentation>Atten</xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="range" type="Distance">
                           <xs:annotation>
                              <xs:documentation>Range of simulated target
                              </xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="range_accn" type="Acceleration" default="0">
                           <xs:annotation>
                              <xs:documentation>Rate of change of rate change
                              </xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="range_rate" type="Speed" default="0">
                           <xs:annotation>
                              <xs:documentation>Rate of change of target range
                              </xs:documentation>
                           </xs:annotation>
                        </xs:attribute>
                        <xs:attribute name="reply_eff" default="100%">
                           <xs:annotation>
                              <xs:documentation>Proportion of Tx pulses returned
                              </xs:documentation>
                           </xs:annotation>
                           <xs:simpleType>
                              <xs:restriction base="Ratio">
                                 <xs:minInclusive value="0 "/>
                                 <xs:maxInclusive value=" 100%"/>
                              </xs:restriction>
                           </xs:simpleType>
                        </xs:attribute>
                     </xs:extension>
                  </xs:complexContent>
               </xs:complexType>
            </xs:element>
         </xs:schema>
      </interface>
      <model>
```

```
        <Signal Out="Radar_RX_Signal" In="radar_TX_Signal" xmlns="STDBSC"
         xsi:schemaLocation="STDBSC stdbsc.xsd"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
           <RMS name="Event_Train" gateTime="1.0e-8" nominal="0.1" condition="GE"
           In="radar_TX_Signal"/>
           <ProbabilityEvent name="Suppressed_Event_Train" probability="reply_eff"
           In="Event_Train"/>
           <Attenuator name="Car_Pulse" gain="atten" Gate="Suppressed_Event_Train"
           In="radar_TX_Signal"/>
           <SignalDelay name="Radar_RX_Signal" acceleration="(range_accn*2/3.0e8)"
           delay="(range*2/3.0e8)" rate="(range_rate*2/3.0e8)" In="Car_Pulse"/>
        </Signal>
     </model>
     <description>
        <notes>

This annex describes a transmitted signal as a reference. Thus the TSF library provides a
description for both the transmitted (Radar_TX_Signal) and received (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 (giving rise to a
"reply efficiency").

To achieve this 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 provides an event while the continuous rms value is greater than a
nominal threshold value. This 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.
        </notes>
        <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.
        </rules>
     </description>
  </TSF>
```

10.14.2 XML for MODULATED_SIGNAL

Extracts from an XML description for MODULATED_SIGNAL are given below the equivalent tabular description (see Table 40 and Table 41). In the following examples, additional blank lines are included for clarity.

10.14.2.1 Interface data

Table 40—MODULATED_SIGNAL interface

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Voltage	—	—
Carrier frequency	car_freq	Frequency	—	—
Depth of modulation	mod_depth	Ratio	—	0 to 1
Modulation Signal	mod_Signal	SignalFunction	—	—

XML Equivalent

```
<xs:attribute name="car_ampl" type="Voltage">
```

```
      <xs:annotation>
         <xs:documentation>Carrier amplitude</xs:documentation>
      </xs:annotation>
   </xs:attribute>

   <xs:attribute name="car_freq" type="Frequency">
      <xs:annotation>
         <xs:documentation>Carrier frequency</xs:documentation>
      </xs:annotation>
   </xs:attribute>

   <xs:attribute name="mod_depth">
      <xs:annotation>
         <xs:documentation>Depth of modulation</xs:documentation>
      </xs:annotation>
      <xs:simpleType>
         <xs:restriction base="Ratio">
            <xs:minInclusive value=" 0"/>:
            <xs:maxInclusive value=" 1"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:attribute>
```

10.14.2.2 Signal model

Table 41—MODULATED_SIGNAL model

Name	Type	Terminal	Inputs	Output	Formula
AM Signal	AM	Signal [Out]	—	MODULATED_SIGNAL	—
		modIndex	mod_depth	—	—
		Carrier [In]	Carrier	—	—
		Signal [In]	mod_Signal	—	—
Carrier	Sinusoid	Signal [Out]	—	AM Signal	—
		amplitude	car_ampl	—	—
		frequency	car_freq	—	—
		phase	—	—	0 rad

XML Equivalent

```
<model>
   <Signal Out="modulated_Signal"In="mod_Signal" xmlns="STDBSC" xsi:schemaLocation="STDBSC
stdbsc.xsd"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <Sinusoid name="Carrier" amplitude="car_ampl" frequency="car_freq"/>
      <AM name="AM_Signal" modIndex="mod_depth" Carrier="Carrier" In="mod_Signal"/>
   </Signal>
</model>
```

10.14.2.3 Complete XML description for MODULATED_SIGNAL

A complete XML description for the SignalFunction MODULATED_SIGNAL entry is shown here. Note that the uuid is a globally unique identifier, which may be provided by one of several proprietary tools.

```
<TSF name="MODULATED_SIGNAL" uuid="{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}">
   <interface>
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
         <xs:element name="MODULATED_SIGNAL">
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="SignalFunctionType">
                     <xs:attribute name="car_ampl" type="Voltage">
                        <xs:annotation>
                           <xs:documentation>Carrier amplitude</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="car_freq" type="Frequency">
                        <xs:annotation>
                           <xs:documentation>Carrier frequency</xs:documentation>
                        </xs:annotation>
                     </xs:attribute>
                     <xs:attribute name="mod_depth">
                        <xs:annotation>
                           <xs:documentation>Depth of modulation</xs:documentation>
                        </xs:annotation>
                        <xs:simpleType>
                           <xs:restriction base="Ratio">
                              <xs:minInclusive value=" 0"/>:
                              <xs:maxInclusive value=" 1"/>
                           </xs:restriction>
                        </xs:simpleType>
                     </xs:attribute>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
      </xs:schema>
   </interface>
   <model>
      <Signal Out="modulated_Signal"In="mod_Signal" xmlns="STDBSC"
xsi:schemaLocation="STDBSC stdbsc.xsd"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         <Sinusoid name="Carrier" amplitude="car_ampl" frequency="car_freq"/>
         <AM name="AM_Signal" modIndex="mod_depth" Carrier="Carrier" In="mod_Signal"/>
      </Signal>
   </model>
</TSF>
```

10.14.3 TSF model for linear sweep

The TSF model for the LINEAR_SWEEP outlined in 10.9 is provided as follows:

```
<tsf:TSF name="LINEAR_SWEEP" uuid="{43C5B0F1-7202-4F77-A2B2-3AA654EC5F6D}">
  <tsf:interface>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
      <xs:element name="LINEAR_SWEEP">
        <xs:annotation>
          <xs:documentation>
            <!-- A linear sweep starting at the 'start_freq' and increasing by
                 the frequency value 'freq_change'. -->
          </xs:documentation>
        </xs:annotation>
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="SignalFunctionType">
              <xs:attribute name="sweep_time" type="Time">
                <xs:annotation>
                  <xs:documentation>
                    <!-- 'sweep_time' is the duration of the sweep. -->
```

```
                  </xs:documentation>
                </xs:annotation>
              </xs:attribute>
              <xs:attribute name="start_time" type="Time" default="0 s" >
                <xs:annotation>
                  <xs:documentation>
                    <!-- 'start_time' defines when the sweep starts' -->
                  </xs:documentation>
                </xs:annotation>
              </xs:attribute>
              <xs:attribute name="car_ampl" type="Physical">
                <xs:annotation>
                  <xs:documentation>
                    <!-- 'car_ampl' is the amplitude of  the sinusoidal carrier -->
                  </xs:documentation>
                </xs:annotation>
              </xs:attribute>
              <xs:attribute name="start_freq" type="Frequency">
                <xs:annotation>
                  <xs:documentation>
                    <!-- 'start_freq' is the frequency at which the sweep starts -->
                  </xs:documentation>
                </xs:annotation>
              </xs:attribute>
              <xs:attribute name="freq_change" type="Frequency">
                <xs:annotation>
                  <xs:documentation>
                    <!-- 'freq_change' is the change of frequency during the sweep -->
                  </xs:documentation>
                </xs:annotation>
              </xs:attribute>
              <xs:attribute name="dc_offset" type="Physical" default="0">
                <xs:annotation>
                  <xs:documentation>
                    <!-- 'dc_offset' is the dc offset level of the signal -->
                  </xs:documentation>
                </xs:annotation>
              </xs:attribute>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </tsf:interface>
  <tsf:model>
    <Signal Out="Linear_Sweep">
      <SingleRamp name="Sweep_Rate" amplitude="1" riseTime="(sweep_time)"
                  carrierFrequency="(start_freq)" frequencyDeviation="(freq_change)"
                  In="Sweep_Rate" />
      <Constant name="DC_Offset" amplitude="(dc_offset)" />
      <Sum name="Linear_Sweep" In="Sweep DC_Offset" />
    </Signal>
  </tsf:model>
</tsf:TSF>
```

10.14.4 TSF model for logarithmic sweep

The TSF model for the LOG_SWEEP outlined in 10.10 is provided as follows:

```
<tsf:TSF name="LOG_SWEEP" uuid="{E89DE4F3-42CC-479E-B726-A1D0A71B3F98}">
  <tsf:interface>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
      <xs:element name="LOG_SWEEP">
        <xs:annotation>
          <xs:documentation>
            <!-- A logarithmic sweep starting at the 'start_freq' and increasing by
```

```
                    the frequency value 'freq_change'. -->
                </xs:documentation>
              </xs:annotation>
              <xs:complexType>
                <xs:complexContent>
                  <xs:extension base="SignalFunctionType">
                    <xs:attribute name="sweep_time" type="Time">
                      <xs:annotation>
                        <xs:documentation>
                          <!-- 'sweep_time' is the duration of the sweep. -->
                        </xs:documentation>
                      </xs:annotation>
                    </xs:attribute>
                    <xs:attribute name="start_time" type="Time" default="0 s">
                      <xs:annotation>
                        <xs:documentation>
                          <!-- 'start_time' defines when the sweep starts' -->
                        </xs:documentation>
                      </xs:annotation>
                    </xs:attribute>
                    <xs:attribute name="dc_offset" type="Physical" default="0">
                      <xs:annotation>
                        <xs:documentation>
                          <!-- 'dc_offset' is the dc offset level of the signal -->
                        </xs:documentation>
                      </xs:annotation>
                    </xs:attribute>
                    <xs:attribute name="car_ampl" type="Physical">
                      <xs:annotation>
                        <xs:documentation>
                          <!-- 'car_ampl' is the amplitude of  the sinusoidal carrier -->
                        </xs:documentation>
                      </xs:annotation>
                    </xs:attribute>
                    <xs:attribute name="start_freq" type="Frequency">
                      <xs:annotation>
                        <xs:documentation>
                          <!-- 'start_freq' is the frequency at which the sweep starts -->
                        </xs:documentation>
                      </xs:annotation>
                    </xs:attribute>
                    <xs:attribute name="freq_change" type="Frequency">
                      <xs:annotation>
                        <xs:documentation>
                          <!-- 'freq_change' is the change of frequency during the sweep -->
                        </xs:documentation>
                      </xs:annotation>
                    </xs:attribute>
                  </xs:extension>
                </xs:complexContent>
              </xs:complexType>
            </xs:element>
          </xs:schema>
        </tsf:interface>
        <tsf:model>
          <Signal Out="Log_Sweep">
            <SingleRamp name="Sweep_Rate" amplitude="1" riseTime="(sweep_time)"
                        startTime="(start_time)" />
            <Attenuator name="Ln_ramp" gain="(ln(start_freq+freq_change))"
                                     In="Sweep_Rate" />
            <E name="Exponent" In="Ln_ramp" />
            <Attenuator name="Scaled_Exp" gain="(1/(start_freq+freq_change))" In="Exponent" />
            <Constant name="DC_Offset" amplitude="(dc_offset)" />
            <FM name="Sweep" amplitude="(car_ampl)" carrierFrequency="(start_freq)"
                             frequencyDeviation="(freq_change)" In="Scaled_Exp" />
            <Sum name="Log_Sweep" In="DC_Offset Sweep" />
          </Signal>
        </tsf:model>
      </tsf:TSF>
```

10.15 Use of IDL to specify TSF signal model information

The IDL provides a language neutral description of the interfaces to the BSCs and TSF models. The BSCs are fully described in Annex D of IEEE Std 1641-2010. Annex F of IEEE Std 1641-2010 provides IDL descriptions for all the TSF models. Any new TSF model generated by a user should be provided with the appropriate IDL description. This will ensure that all implementations of that signal are the same.

As an example, the relationship between the tabular definitions for RADAR_RX_SIGNAL in Annex E of IEEE Std 1641-2010 and the IDL description in Annex F of IEEE Std 1641-2010 is illustrated.

10.15.1 IDL for RADAR_RX_SIGNAL

As with the section on XML, this is not intended to be a tutorial for the use of IDL, but to show the relationship of the entries in the tabular format and the IDL description. Extracts from the IDL description are given together with the equivalent extract form the tabular description. For full information on the IDL refer to *Distributed Computing Environment Specifications* [B2].

10.15.1.1 General information

The TSF Class name is retained from Annex E of IEEE Std 1641-2010, i.e., RADAR_RX_SIGNAL.

```
//RADAR_RX_SIGNAL
    [
        object,
        uuid(7D866851-EDFA-4A99-9801-658439350C9C),
        dual,
        helpstring("IRADAR_RX_SIGNAL Interface"),
        pointer_default(unique)
    ]
```

10.15.1.2 Interface data

The five attributes from the interface properties table (see Table 21) are each listed between the braces of the interface field. The description is given as text in a helpstring in each case. The attribute name is taken from the Name column. The type is taken from the Type column. The default is taken from the Default column. The range details are omitted.

Description	Name	Type	Default	Range
Atten	atten	Ratio	1	

```
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);
```

Range of simulated target	range	Distance		

```
    [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);
```

Rate of change of rate change	range_accn	Acceleration	0	

```
    [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);
```

Rate of change of target range	range_rate	Speed	0	

```
    [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);
```

Proportion of Tx pulses returned	reply_eff	Ratio	100%	0% to 100%

```
    [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);
};
```

The final entry in the interface properties table is not an attribute but an input SignalFunction that is not included in the IDL description. The IDL description closes with the following entry.

```
[
    uuid(7D866851-EDFA-4A99-9801-658439350C90),
    helpstring("RADAR_RX_SIGNAL class"),
    noncreatable
]
coclass RADAR_RX_SIGNAL
{
    [default] interface IRADAR_RX_SIGNAL;
};
```

10.15.1.3 Complete IDL description for RADAR_RX_SIGNAL

The complete IDL description for the SignalFunction RADAR_TX_SIGNAL is shown here.

```
//RADAR_RX_SIGNAL
[
    object,
    uuid(7D866851-EDFA-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;
};
```

10.15.2 IDL for MODULATED_SIGNAL

Extracts from an IDL description for MODULATED_SIGNAL are given below the equivalent tabular description (see Table 42). In the following examples, additional blank lines are included for clarity.

10.15.2.1 Interface data

Table 42—MODULATED_SIGNAL interface

Description	Name	Type	Default	Range
Carrier amplitude	car_ampl	Voltage	—	—
Carrier frequency	car_freq	Frequency	—	—
Depth of modulation	mod_depth	Ratio	—	0 to 1
Modulation Signal	mod_Signal	SignalFunction	—	—

IDL Equivalent

```
interface IMODULATED_SIGNAL : ITSF
{
    enum {MODULATED_SIGNAL_BASE=(TSF_BASE+256)};

    [propget, id(MODULATED_SIGNAL_BASE+1), helpstring("Carrier amplitude")]
        HRESULT car_ampl([out, retval] Voltage *pVal);
    [propputref, id(MODULATED_SIGNAL_BASE+1), helpstring("Carrier amplitude")]
        HRESULT car_ampl([in] Voltage newVal);

    [propget, id(MODULATED_SIGNAL_BASE+2), helpstring("Carrier frequency")]
        HRESULT car_freq([out, retval] Frequency *pVal);
    [propputref, id(MODULATED_SIGNAL_BASE+2), helpstring("Carrier frequency")]
        HRESULT car_freq([in] Frequency newVal);

    [propget, id(MODULATED_SIGNAL_BASE+3), helpstring("Depth of modulation")]
        HRESULT mod_depth([out, retval] Ratio *pVal);
    [propputref, id(MODULATED_SIGNAL_BASE+3), helpstring("Depth of modulation")]
        HRESULT mod_depth([in] Ratio newVal);

    };
```

10.15.2.2 Complete IDL description for MODULATED_SIGNAL

The above extracts do not provide the complete IDL description for the SignalFunction MODULATED_SIGNAL so here is the complete IDL description. Note that the uuid is a globally unique identifier, which may be provided by one of several proprietary tools.

```
//MODULATED_SIGNAL
    [
        object,
        uuid(xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx),
        dual,
        helpstring("IMODULATED_SIGNAL Interface"),
        pointer_default(unique)
    ]
    interface IMODULATED_SIGNAL : ITSF
    {
        enum {AM_SIGNAL_BASE=(TSF_BASE+256)};

        [propget, id(MODULATED_SIGNAL_BASE+1), helpstring("Carrier amplitude")]
            HRESULT car_ampl([out, retval] Voltage *pVal);
        [propputref, id(MODULATED_SIGNAL_BASE+1), helpstring("Carrier amplitude")]
            HRESULT car_ampl([in] Voltage newVal);
        [propget, id(MODULATED_SIGNAL_BASE+2), helpstring("Carrier frequency")]
            HRESULT car_freq([out, retval] Frequency *pVal);
        [propputref, id(MODULATED_SIGNAL_BASE+2), helpstring("Carrier frequency")]
            HRESULT car_freq([in] Frequency newVal);
        [propget, id(MODULATED_SIGNAL_BASE+3), helpstring("Depth of modulation")]
            HRESULT mod_depth([out, retval] Ratio *pVal);
        [propputref, id(MODULATED_SIGNAL_BASE+3), helpstring("Depth of modulation")]
            HRESULT mod_depth([in] Ratio newVal);
    };
    [
        uuid(xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx),
        helpstring("MODULATED_SIGNAL class"),
        noncreatable
    ]
    coclass MODULATED_SIGNAL
    {
        [default] interface IMODULATED_SIGNAL;
    };
```

11. Digital Signals

In order to understand how digital signals are represented in the STD standard, it is important to understand the different types of stream (see 4.5). A digital stream is a signal representing the digital states only, without any reference to a physical state that may be used to carry the digital information. This digital stream then has to be converted to a physical signal before it can be applied to a UUT.

11.1 Possible states for digital stream

Figure 111 shows the states available to elements of a digital stream, which may be H, L, X, or Z.

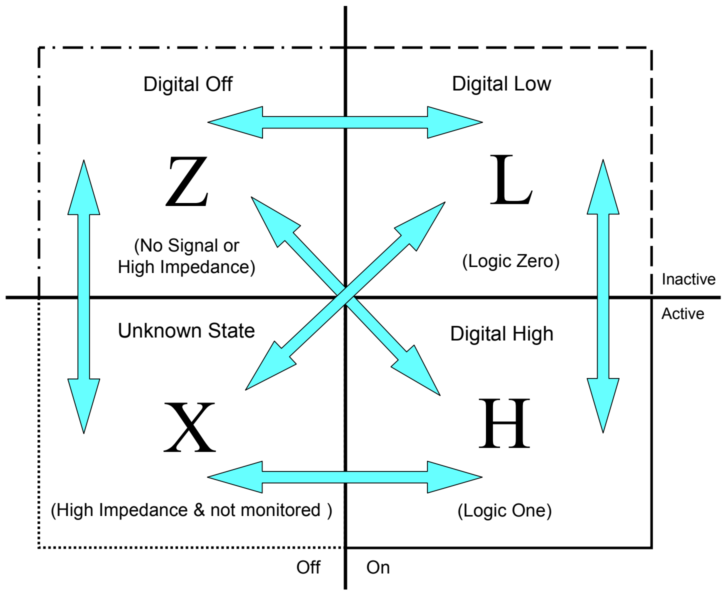


Figure 111 —State diagram for a digital stream

A digital source may adopt any of the four states:

- — Digital Off (Z)—There is no signal.
- — Digital High (H)—The logic one state
- — Digital Low (L)—The logic zero state
- — Unknown State (X)—The digital signal is present but it is not possible to know what it is. This may be considered to be equivalent to a high impedance state where the output may float to any value. (For a sensor this may be considered the same as the “Don’t Care” state).

11.2 Generating a digital stream

The Encode BSC may be used to convert a block of digital data into a digital stream. Figure 112 shows the digital stream that is generated from the data "HLXZHLXZ".

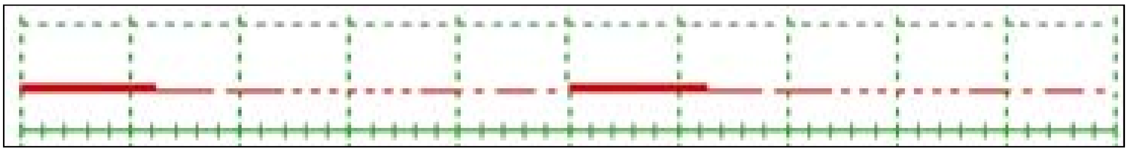


Figure 112 —A digital stream ("HLXZHLXZ")

Thus an element in a digital stream may be in any of the four quadrants of the diagram, representing H, L, X & Z. This is digital data only without any physical information.

NOTE—In the stream diagrams the following conventions are used:

- — A continuous solid line represents a Digital High (H)
- — A dashed line represents a Digital Low (L)
- — A dotted line represents an Unknown state (X)
- — A dash-dot line represents a Digital Off state (Z)

Before it can be passed to and from a UUT, a digital stream must be turned into a signal stream, i.e. the digital information must be represented by a physical signal. A digital stream cannot be connected directly to a UUT. Figure 113 shows the digital pattern "HLXZHLXZ" as a signal stream in which the digital stream H is converted to "Signal with Value" (5 V), L is converted to "Signal with Value" (0 V) and both X & Z are converted to "Signal with No Value" (X—a dotted line in the figure). When the digital stream has finished, it returns to No Signal (Z).

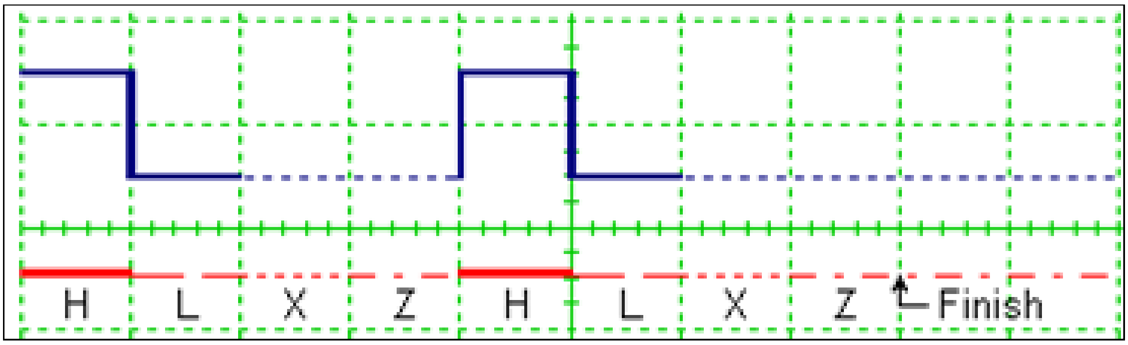


Figure 113 —Converting a digital streams into a physical signal

Note that the actual conversion depends on the type of physical signal generated—in this case a pulse class of non-return-to-zero. Another possibility could be different frequencies or tones, where one frequency represented H, another represented L, a third tone represented the digital streams X & Z states and no signal resulted in a no frequency signal.

This is consistent with what happens with the existing digital BSCs such as SerialDigital which can return to the No Signal state when the digital information is complete. Note that the SerialDigital and ParallelDigital BSCs provide a physical signal stream carrying digital data. They do not provide pure digital data only.

In a real hardware implementation, the digital X and Z states will normally be implemented as a high impedance signal (a signal stream X state).

Digital Streams may also be controlled by event streams, and the effect of an event stream on a digital stream may be illustrated by a similar table as used for the other streams (see Table 43).

Table 43—Effect of events on a digital stream

Data	Gate	Output
Z	On	Z
H	On	H
L	On	L
X	On	X
Z	Off	Z
H	Off	X
L	Off	Z
X	Off	X

This table shows the effect of an event state on the gate of an Encode BSC.

11.3 Converting digital data into a physical digital signal.

Converting digital data into a physical digital signal is a two stage process. Initially the data has to be converted into a digital stream, and then this digital stream has to be converted into a physical signal that can be applied to a UUT via a connector.

11.3.1 Converting data into a digital stream

The Encode BSC is used to convert digital data in to a digital stream. It allows any data representation to be packaged and streamed as a sequence of bits. It turns messages into a bit stream represented by events where the event state represents the individual bit state; this stream of events is a digital stream.

The Encode BSC has 5 attributes as follows:

a) data—this is the information to be streamed. This data may be of any type (i.e. datatype) but must be stated. When using XML descriptions, the attribute datatype shall be used to define a valid XML datatype. As an alternative, the Uniform Resource Identifier (URI) location of information may be specified.

b) width—this is of type integer and specifies the number of output channels. Zero indicates that the number of outputs is the minimum required to represents a complete data symbol. The default value is zero.

c) repetition—this is of type integer and specifies the number of times the data is output. Zero indicates that the sequence is repeated indefinitely. The default is one.

d) datatype— this is of type string and defines the type of the data

e) encoding—this is of type string and defines the character set used if the dataype is char. The default is UTF-8.

When the data is supplied as part of a complete test requirement, the type of the data will be implicit within the chosen carrier language. The datatype and encoding attributes are important when the data is expressed in XML, so that it is clear how to convert the data into the appropriate format for the chosen carrier language.

Thus the same digital data could be presented in different ways. The following data are equivalent:

— data="11010100" datatype=" boolean"

— data="HHLHLHLL" datatype="digitalString"

— data="D4" datatype="hexBinary"

— data="212" datatype="int"

— data="Ô" datatype="string" encoding="UTF-8"

— data="Ô" datatype="string" encoding="Windows-1252"

— data="Ō" datatype="string" encoding="Windows-1257"

Since the width attribute does not necessarily need to match the data information type, the data are converted into a stream of bits and each bit assigned to consecutive channels so that the MSB is in the channel number corresponding to the channel width. If the number of bits is less than the number of channels, the unused channels following the LSB are effectively filled with Z (no signal). If the number of bits exceeds the number of channels then the overflowing bits are assigned to the next parallel word in the stream staring at the channel number corresponding to the channel width.

The most convenient arrangement is to arrange the data so the number of bits is the same as the channel width. With any other arrangement it would be easy to lose synchronization. Although using width=1 appears to provide serial data, it would not be possible to determine where one data item ends and the next one begins, unless the stop and start bits were included in the data.

Table 44 shows the different bit streams that the same block of data provides depending on the values selected for the **width** and **repetition** attributes.

Table 44—Digital streams for the same data with different attributes

Attributes	Channel	Bit stream
data = "11010100" width = "4" repetition = "2"	1 (LSB)	0 1 0 1 →
	2	0 0 0 0 →
	3	1 1 1 1 →
	4 (MSB)	0 1 0 1 →
data = "11010100" width = "8" repetition = "2"	1 (LSB)	0 0 →
	2	0 0 →
	3	1 1 →
	4	0 0 →
	5	1 1 →
	6	0 0 →
	7	1 1 →
	8 (MSB)	1 1 →
data = "11010100" width = "2"	1 (LSB)	0 1 1 1 →
	2 (MSB)	0 0 0 1 →
data = "11010100" width = 3 repetition = "3"	1 (LSB)	0 0 1 0 1 1 1 0 →
	2	0 1 1 1 0 0 0 1 →
	3 (MSB)	1 0 0 0 1 0 1 1 →
data = "11010100" width = "3"	1 (LSB)	Z 1 0 →
	2	0 0 1 →
	3 (MSB)	0 1 1 →
data = "11010100" width = "1"	1 (MSB goes first)	0 0 1 0 1 0 1 1 0 0 1 0 1 0 1 1 →

Attributes	Channel	Bit stream
repetition = "2"		

Figure 114 shows an Encode BSC. Each pattern is delivered when **In** goes Active (1,H). If the input becomes tri-state (Gated Off), the output is Gated off. Therefore the rate at which the events on the **In** change control the data rate of the output digital stream.

Digital signals are unique in that their values do not take on physical values; rather, they take on event states that can be converted into physical values.

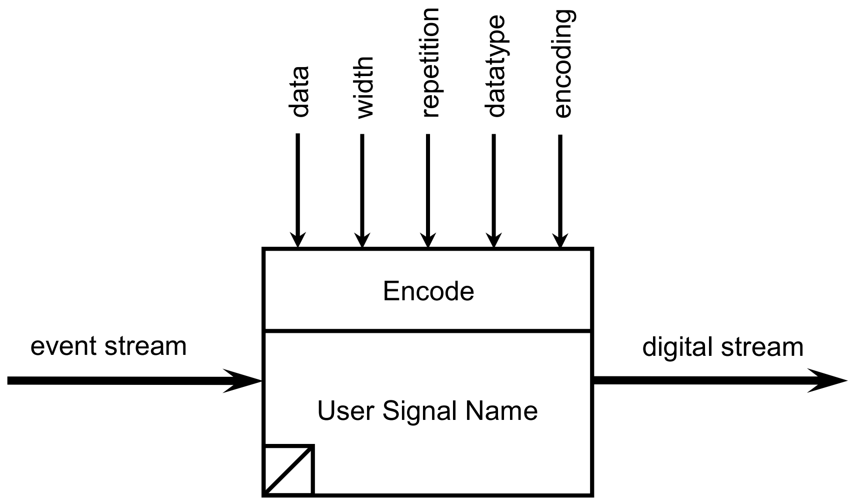


Figure 114 —Encode BSC

Although it is the event on the input going to the Active state that clocks the data out into the digital stream, any stream that carries event information may be used. This includes another digital stream.

11.3.2 Converting the digital stream into a physical digital signal

There are two Control BSCs that may be used to convert digital streams into physical signals. The simplest of these is the SelectIf BSC, which uses a digital stream to select inputs with physical values, hence creating a physical signal.

This may be illustrated by using the example in Table 44 where data="11010100", width="1", repetition="2". This digital stream is then passed through the BasicNRZ TSF from Annex F of the STD standard (see Figure 115)

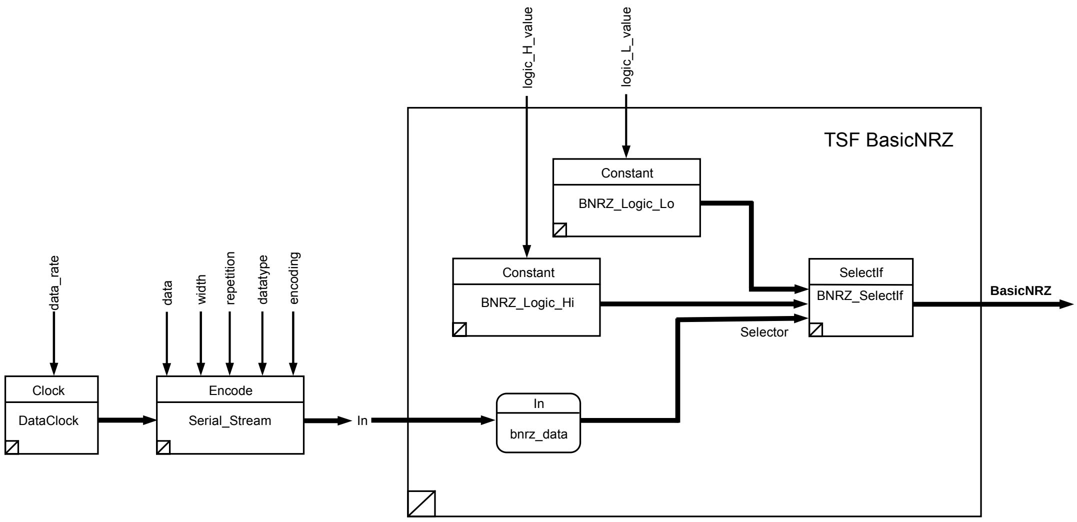


Figure 115—Generating a BasicNRZ digital signal

Using the data from Table 44 it is possible to describe how the BasicNRZ signal is created using the model in Figure 115. Table 45 shows the values used for the attributes and Table 46 shows the values for each part of the model as the data starts to flow, i.e. for the first 6 bits of the stream (110101).

Table 45—Attribute values for BasicNRZ signal

Attribute	Value
data_rate	100 kHz
data	11010100
width	1
repetition	2
datatype	boolean
encoding	
logic_H_value	5 V
logic_L_value	0 V

Table 46—Data flow through digital signal model

Model element	Flow of data															
DataClock	I	A	I	A	I	A	I	A	I	A	I	A	I	A	I	A
data	1		1		0		1		0		1		0		0	
Serial_Stream (In)	Z	H		H		L		H		L		H		L		
bnrz_data	Z	H		H		L		H		L		H		L		
BNRZ_Logic_Lo	0 V															
BNRZ_Logic_Hi	5 V															
BNRZ_SelectIf	Z	5 V				0 V		5 V		0 V		5 V		0 V		
BasicNRZ signal	Z	5 V				0 V		5 V		0 V		5 V		0 V		

The DataClock event stream clocks the data out of the Encode BSC on every transition from Inactive to Active. The digital stream "Serial_Stream" starts in the Z state, then follows the data as it is clocked through, outputting the Boolean 1 as a H and the Boolean 0 as an L. This digital stream is then input to the TSF and arrives at the Selector input of the SelectIf BSC. At each transition at the Select input of H to L and L to H the alternate **In** is selected, resulting in alternate output values of 5 V and 0 V for the input Hs and Ls.

11.3.3 Converting a parallel digital stream into a physical signal

The SelectIf may only be used with a single channel input, but if a multiple channel digital stream is used the effect is as if multiple SelectIf BSCs were to be used. This may be illustrated by using the example in Table 44 where data="11010100", width="4", repetition="2". This digital stream is then passed through a SelectIf BSC in the same way as shown for the BasicNRZ TSF in Figure 115. The attribute values would be very similar to those for the previous example (see Table 45) with the exception that the value for the width attribute would be width=4. The output is shown in Table 47, where it may be seen that the result is the same as if four separate SelectIf BSCs were used in parallel.

Table 47 —Data flow for 4 bit wide digital signal

Element	Channel info	Flow of data									
Data clock		I	A	I	A	I	A	I	A	I	A
Digital data	1 (LSB)	1		0		1		0		—	
	2	1		1		1		1		—	
	3	0		0		0		0		—	
	4 (MSB)	1		0		1		0		—	
Digital stream (SelectIf Selector)	1 (LSB)	Z	H		L		H		L		Z
	2	Z	H		H		H		H		Z
	3	Z	L		L		L		L		Z
	4 (MSB)	Z	H		L		H		L		Z
SelectIf In(1)	Logic Low Value	0 V									
SelectIf In(2)	Logic High Value	5 V									
SelectIf output	1 (LSB)	Z	5 V		0 V		5 V		0 V		0 V
	2	Z	0 V		5 V		0 V		5 V		5 V
	3	Z	5 V		0 V		5 V		0 V		0 V
	4 (MSB)	Z	5 V		0 V		5 V		0 V		0 V

The Data clock event stream clocks the data out of an Encode BSC on every transition from Inactive to Active. The digital stream starts in the Z state, then follows the data as it is clocked through as a four-bit wide parallel stream. This digital stream is then passed to the Selector input of the SelectIf BSC. Although the digital stream is a single channel, each element of the stream is processed separately and in parallel by the SelectIF. At each transition at the Select input of H to L and L to H the alternate **In** is selected, resulting in alternate output values of 5 V and 0 V for each bit in the channel. The final output is a four-bit wide physical signal representing the original four-bit wide data.

11.4 Using the SelectCase BSC

11.4.1 General

The SelectIf BSC provides a facility where the digital stream may be used to select one of many signals. The Selector input channels are used as a binary address to indicate which of the **In** signals is to be passed to the output. Thus, if the Selector input has four channels it will allow up to sixteen input channels to be selected. A mask is available which allows any (or all) of the Selector channels to be masked out.

11.4.2 A simple digital to analog converter (DAC)

Figure 116 shows a SelectIf BSC being used as a very simple DAC. The Selector input has a 4-bit digital stream as its input. A mask is applied to mask out the MSB so that only the three least significant bits are used. As each digital stream word is clocked through the SelectCase BSC the appropriate input signal is passed to the output.

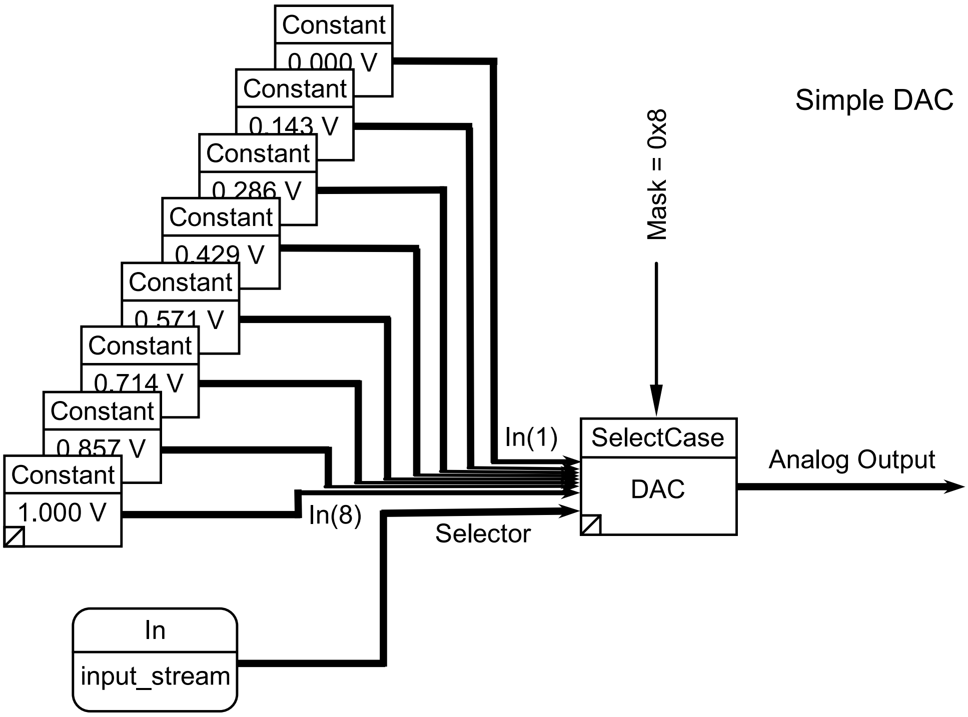


Figure 116 —Simple DAC using SelectCase

The operation of the model is illustrated in Table 48. As each word in the digital stream arrives at the Selector input the MSB is masked out. The remaining 3 bits are converted to a one-based address. That means address 0 (000) selects **In(1)**, address 1 (001) selects **In(2)** , and address 7 (111) selects **In(8)**. Addresses 8 through 15 select **In(1)** through **In(8)** because the MSB is masked out and does not affect the input selection. The signal on the selected **In** is output to the single SelectCase output channel.

Table 48 —Operation of SelectCase BSC

Element		**Flow of data**											
Pattern number		—	1	2	3	4	5	6	7	8	9	10	11
Digital stream	1 (LSB)	Z	L	L	H	H	H	L	H	H	H	L	L
(Selector input)	2	Z	L	H	H	L	L	L	L	L	H	H	L
	3	Z	H	H	H	H	L	L	L	H	H	H	H
	4 (MSB)	Z	L	L	L	L	L	L	L	L	L	L	L
Masked address		—	4	6	7	5	1	0	1	5	7	6	4
Input selected		—	**In(5)**	**In(7)**	**In(8)**	**In(6)**	**In(2)**	**In(1)**	**In(2)**	**In(6)**	**In(8)**	**In(7)**	**In(5)**
SelectCase output	(V)	Z	0.071	0.357	0.500	0.214	-0.375	-0.500	-0.375	0.214	0.500	0.375	0.071

This model converts the digital stream into an analog signal between -0.5 V and 0.5 V in linear steps of 0.143 V (see Table 49). The signal is a 5 kHz sinusoid, but as the available amplitude steps are large the output signal is distorted.

Table 49—Signal values at SelectCase inputs

Input	Signal value (V)
In(1)	-0.500
In(2)	-0.375
In(3)	-0.214
In(4)	-0.071
In(5)	0.071
In(6)	0.214
In(7)	0.357
In(8)	0.500

The 3-bit values are calculated as follows. The actual values for a 3 kHz sinusoid sampled at 5 µs intervals are calculated from Equation (4).

$$a = A\sin(\omega t) \quad (4)$$

where

A is the amplitude of the sinusoid (which must be 0.5 or less for this TSF example)
ω is $2\pi \times$ the frequency of the sinusoid
t is time.

Equation (4) provides a series of actual instantaneous amplitude values as shown in Table 50. The octal value for the nearest 3-bit code is then obtained from Equation (5).

$$Val_{\mathrm{Oct}} = \mathrm{int}(7a) \quad (5)$$

where

Val_{Oct} is the octal representation of the closest available value to the actual signal amplitude
a is the instantaneous amplitude of the sinusoid
int is a function that provides the nearest integer number.

These octal values are also shown in Table 50. When the sinusoid is regenerated, the only output values available are those shown in Table 49, hence the waveform is generated using those values and outputs the signal as illustrated in Figure 117.

Table 50—Generating the analogue outputs

Time (us)	Actual sinusoid amplitude (V)	Nearest available code Octal (binary)	Amplitude output by DAC (V)
0	0.000	4 (010)	0.071
5	0.405	6 (110)	0.357
10	0.476	7 (111)	0.500
15	0.155	5 (101)	0.214
20	-0.294	1 (001)	-0.375
25	-0.500	0 (000)	-0.500
30	-0.294	1 (001)	-0.375
35	0.155	5 (101)	0.214
40	0.476	7 (111)	0.500
45	0.405	6 (110)	0.357
50	0.000	0 (000)	0.071

Although this example is simple with just a few large steps, a similar model may be created with a larger address range and much smaller voltage increments, e.g., an audio CD uses two channels each with 16 bits. 16 bit encoding provides 16^2 or 256 increments. This technique also allows for non-linear digital to analog conversions, such as those used in the telephone industry, where audio is converted to a linear 12 bit signal and then further encoded using only 8 bits. Although the transmitted audio is "distorted", the resultant speech is perfectly understandable.

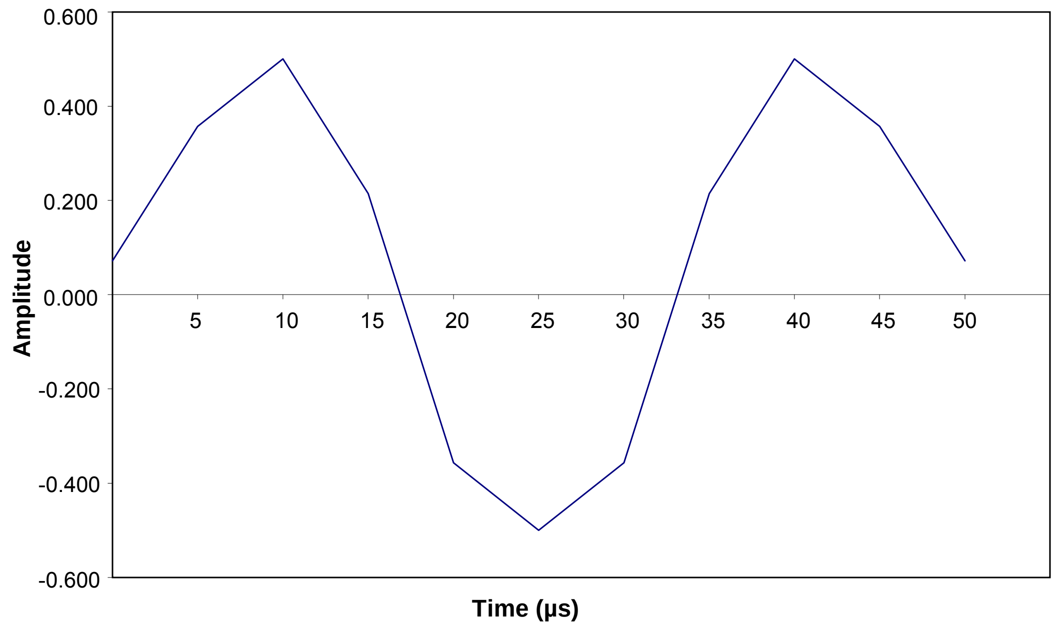


Figure 117 —Distorted sinusoid generated from 3-bit code

11.5 Extracting digital data from a physical digital signal.

11.5.1 Using an Instantaneous BSC

The process of extracting digital data from a physical digital is also a two stage process. First the physical signal has to be converted into a digital stream and then the stream can be decoded to provide the digital data.

The physical signal can be converted to events using the Instantaneous BSC. For example, in the case of a digital voltage signal, the "nominal" attribute can be used to represent the threshold value for zeros and ones, and the "condition" attribute can be used to trigger the events that become the digital stream.

The digital stream is then converted into data using the Decode BSC. The Decodes BSCs attributes define the datatype and encoding of the resulting data. The data extracted form the digital signal is saved in the **measurements** array, with the last pattern in **measurement**.

For simple single threshold digital logic, one multichannel Instantaneous BSC and one Decode BSC will suffice to extract the data. If different H and L thresholds are required, two of each BSC type will be required. If the signal includes indeterminate digital signal levels the resulting saved data would not be the same. The differences between the two sets of data would indicate indeterminate states. This process could be further developed to use Event logic between the Instantaneous BSCs and a Decode BSC such that the indeterminate levels were detected before passing the data to the Decode BSC.

11.5.2 Using the Decode BSC alone

The Decode BSC is a Sensor, so it inherits all the properties of the generic Sensor class. This means that it is possible to perform the data decoding in a single operation. The nominal and condition attributes may be used to convert the digital signal into a digital stream and the digital stream is then decoded into a data array. If dual thresholds are required, then two Decode BSCs will be needed.

12. More about Events and their interaction

12.1 Interaction between streams

12.1.1 Using an event stream to control a source

The way in which signal streams are affected and controlled by event streams is covered, in reasonable depth, in 9.4. So, a simple example will suffice to illustrate the relationship. The example in Figure 118 shows a source (Sinusoid) synchronized and gated by events.

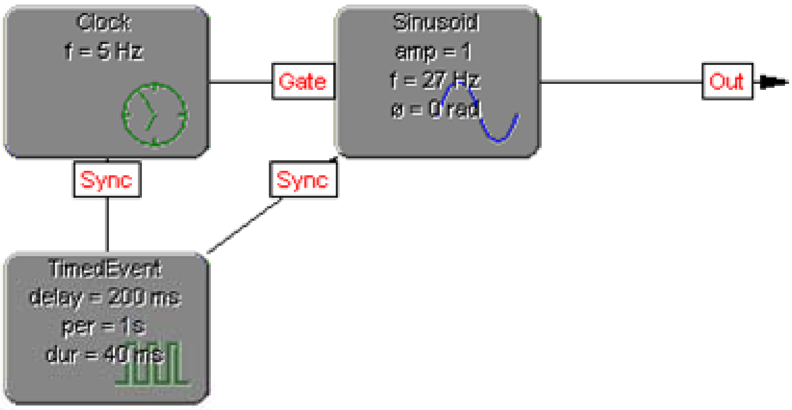


Figure 118 —Sinusoid controlled by event stream

Looking at the waveform in Figure 119 (upper trace), it may be seen that the signal is initially in the “No Signal” (Z) state. After it has received the synchronization signal (center trace), it becomes active and is in the “Signal with Value” (V) state or the “Signal with No value” (X) state, according to the state of the gate event (lower trace).

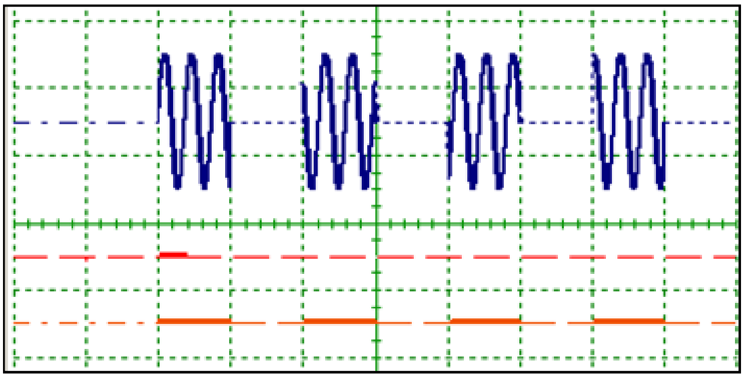


Figure 119 —Waveform from Sinusoid controlled by event stream

Note that the No Signal (Z) state is represented by the ‘dot-dash’ line and the Signal with No Value (X) state is represented by the dotted line. This behavior is explained by the state diagram. Initially the signal (in the Z state) is Inactive. Once started it remains active until it is eventually destroyed when no longer required. While active, it may be switched between the X and V states by being gated on and off by an event stream.

This is illustrated in Table 51 which shows the output of a SUM BSC according to the information on the gate input. If there is no signal on the gate input it is the same as being permanently gated on.

Table 51—Effect of events on a signal stream

Input	Gate	Output
Z	A or X	Z
V	A or X	V
X	A or X	X
Z	Z or I	Z
V	Z or I	X
X	Z or I	X

12.1.2 Events on Events

Events, when interconnected, behave differently. This is because event streams are transmitted through event BSCs as well as being used to gate (and synchronize) event BSCs. Consider the arrangement shown in Figure 120. After a time of 200 ms a Clock is started which is followed by a NotEvent BSC.

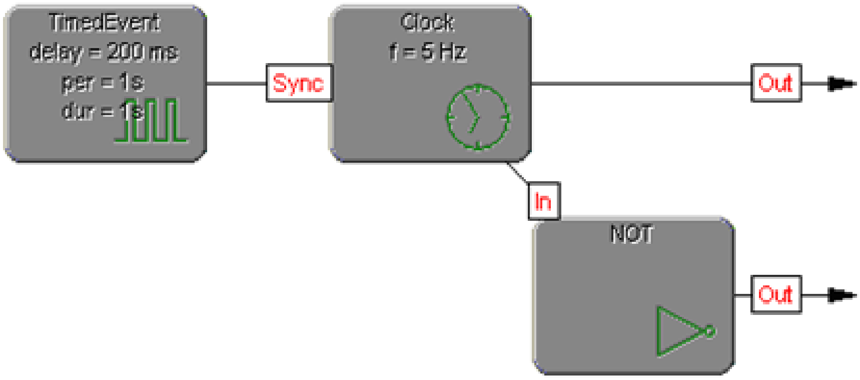


Figure 120 —Event generator with Clock followed by NotEvent

The event streams are shown in Figure 121 with the Clock output shown in the upper trace and the NotEvent output in the lower trace. The outputs of the BSCs are initially both in the “No Event” (Z) state until they are started. At that point both switch to the gated on states but have the opposite “Event Inactive” (I) and the “Event Active” (A) states.

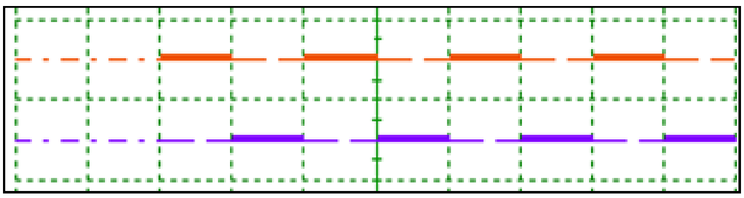


Figure 121 —Waveform showing Clock followed by NotEvent

If a gate event is then added to one of the BSCs (say the Clock) with the gate in the Active state, then the result will be the same. When the gate becomes Inactive, the Clock and NotEvent BSC outputs both assume the “No Event” (Z) state. This is illustrated in Figure 122 where the bottom trace shows the gate signal. When the gate signal becomes inactive, both the top traces show the dot-dash waveform indicating the "No Event" (Z) state.

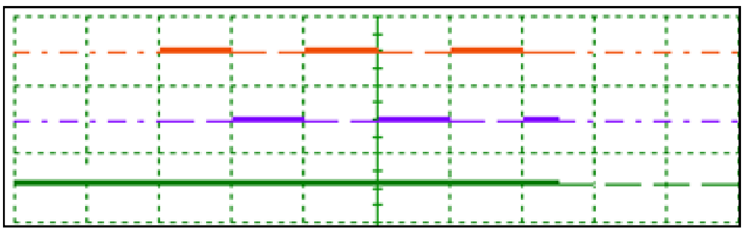


Figure 122 —Waveform from Figure 120 showing effect of a gate on Clock BSC

Thus it appears that the fourth state, the "Event Off" (X) state, is not available. This is not the case for event conditioner BSCs, as shown in a later section. Therefore the effect of gating an event stream by other events, which is shown in Table 52, includes the X state.

Table 52—Effect of events on an event stream

Input	Gate	Output (OrEvent)	Output (NotEvent)
Z	A or X	Z	Z
A	A or X	A	I
I	A or X	I	A
X	A or X	X	X
Z	Z or I	Z	Z
A	Z or I	Z	Z
I	Z or I	Z	Z
X	Z or I	Z	Z

12.1.3 Signal streams carry event information

A signal stream carries event information as illustrated in the example in Figure 123 in which the sinusoidal signal from the earlier example (see Figure 118) is used to gate a Constant BSC. The OrEvent is just used as a linking component. The Constant BSC once gated on, remains on continuously (center trace in Figure 124) and does not follow the pattern of the sinusoid signal.

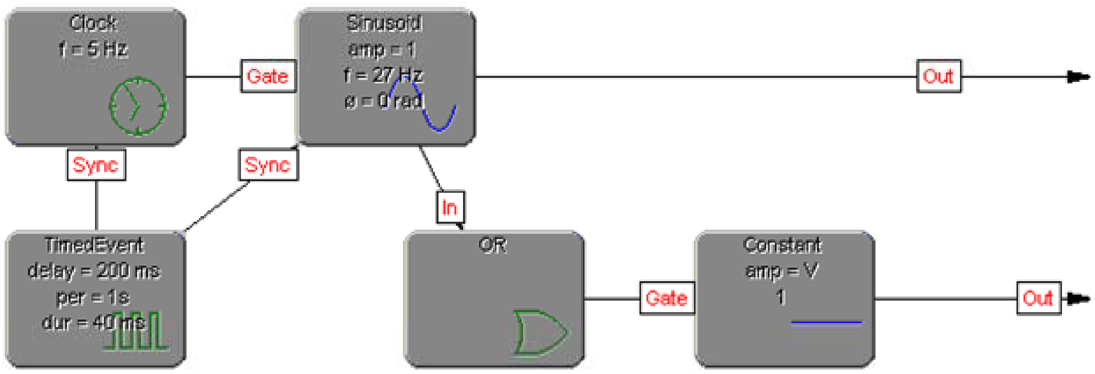


Figure 123 —Constant gated by Sinusoid

This is because the event information (lower trace) carried by the physical signal stream starts in the inactive (Z) state and then becomes active and remains active, i.e. both the X (Signal with No Value) & V (Signal with Value) properties of the signal stream carry active event states.

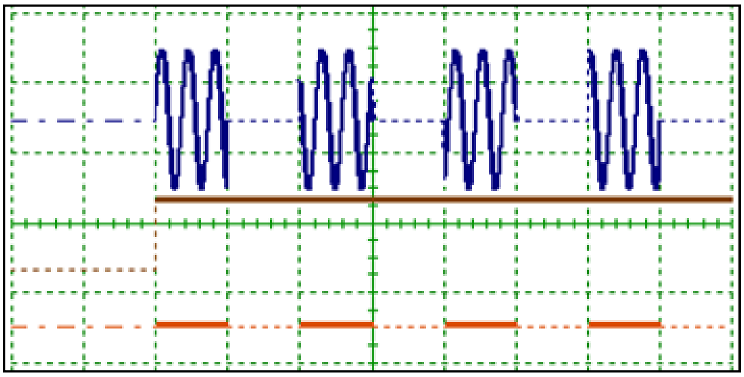


Figure 124 —Waveform showing Constant gated on continuously by Sinusoid

12.1.4 Recovering Event information from signal streams

When an event conditioner BSC (such as an OrEvent BSC) is used to follow a Physical signal BSC, the fourth (X) state is used to retain the X state of the signal stream. It was shown in the previous example (see Figure 123 and Figure 124) that the event stream recovered from the Sinusoid had no effect when gating the Constant BSC (because both X and A are active states), i.e. a SignalFunction cannot tell the difference on its gate input between an event stream of all A states and an event stream of mixed A and X states.

Now, if this event stream is passed through a NotEvent BSC (Figure 125) the original gating information is recovered (but inverted).

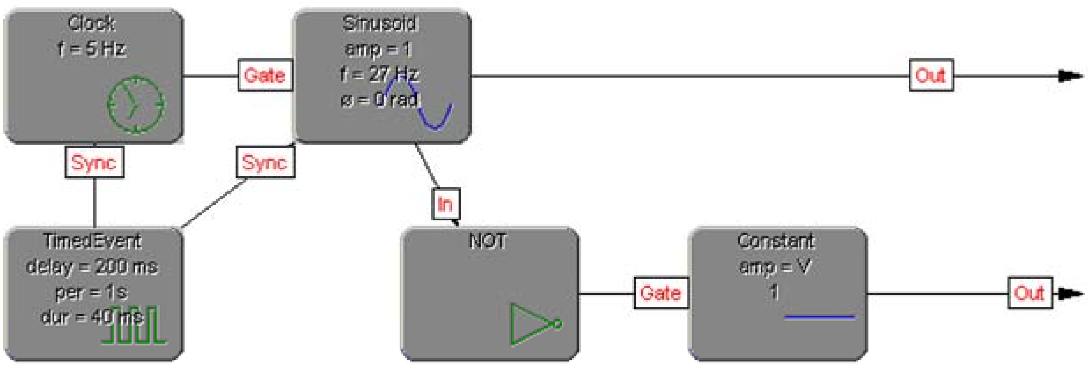


Figure 125 —Recovering event data via a NotEvent

This is illustrated in Figure 126 in which the lower trace shows the event information recovered via the NotEvent BSC. The center trace shows the output of the Constant BSC gated by the NotEvent BSC. The Z state (shown with the dot-dash line) and the I state (shown with dashes) both gate the Constant off, then the X state (shown with the dotted line) gates the Constant on.

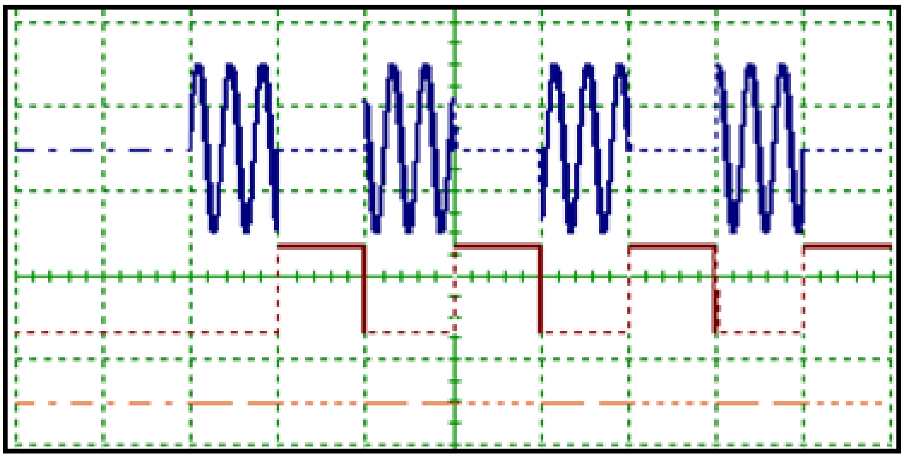


Figure 126 —Waveform showing Constant gated by recovered event data

The recovery of event information from a signal stream is summarized in Table 53, using both an OrEvent and a NotEvent.

Another way to determine how the event information is recovered, is to translate the signal stream state into the event state using the equivalent quadrant of the state diagrams, and then perform the event operation. That is, convert V to A, X to X and Z to Z, then refer to the Table 52.

Table 53—Recovery of event information from a signal stream

Physical	OrEvent	NotEvent
Z	Z	Z
V	A	I
X	X	X

Figure 2, Figure 3 and Figure 4 together with Table 51, Table 52 and Table 53 illustrate the interactions between signal streams and event streams.

12.2 Recovering Event information from digital streams

Retrieving the event data from a digital signal produces a similar pattern as the recovered data from the signal stream, but there is an extra entry for the digital Low state (see Table 54).

Table 54—Recovery of event information from a digital stream

Digital Stream	OrEvent	NotEvent
Z	Z	Z
H	A	I
L	I	A
X	X	X

In fact, the digital stream may be considered the same as an event stream. It has the same four possible states and they do not represent physical signals. They may be used interchangeably and both may be used to gate signal streams (physical signals). Furthermore, signal streams are also the same as far as the event information is concerned. They carry additional information about the active signal (such as voltage, frequency, etc), but the event states as represented by the continuous, dashed, dotted, and dot-dashed lines are equivalent. Any Inactive state (Z, I or L) will gate a SignalFunction off and any Active State (A, X, H or V) will gate a SignalFunction on.

Therefore, it is possible to draw up a single table to represent the states of these signals and then to indicate what happens when any of them are gated on or off (see Table 55).

Table 55—Effect of gating OrEvent & Sum BSCs

Input	Gate	Output (OrEvent)	Output (SUM)
Z	A, H or X	Z	Z
H (or A or V)	A, H or X	H (or A or V)	H (or A or V)
L (or I)	A, H or X	L (or I)	L (or I)
X	A, H or X	X	X
Z	Z, L or I	Z	Z
H (or A or V)	Z, L or I	Z	X
L (or I)	Z, L or I	Z	Z
X	Z, L or I	Z	X

The table shows what happens to any stream when it passes through an event conditioner SignalFunction (such as an OrEvent BSC) and through a combiner SignalFunction (such as a SUM BSC). It is the type of SignalFunction that determines the effect of the gate, not the type of signal.

This also shows that all of the streams act in the same way and to some extent may be considered the same. The important difference is that only signal streams, i.e. streams of physical signals, may be applied to a UUT.

13. Test Procedure Language (TPL)

13.1 Introduction to TPL

The TPL provides a mechanism for users who want to document test requirements in a textual format. Using the TPL is similar in concept to using ATLAS signal statements and may be used as a method of adding IEEE 1641 programming facilities to legacy ATLAS systems. The TPL provides verbs giving users the ability to manipulate test objects (such as signal definitions contained within the TSFs) in test procedures or signal sequences. TPL not only describes the signals but also includes an amount of action (sequence) information.

The TPL provides a textual method of incorporating the required test signals and measurement criteria into a carrier language and yet still have a standardized method of writing signal oriented statements. Users can define test requirements in terms of these TPL test objects embedded within the carrier languages. In this way, different languages may use their own syntax and language semantics and still define the same test requirement because they reference the same external test objects.

The TPL statements will have to be converted into native language code. Therefore, within a TPL environment, there will be some translation mechanism to convert from this neutral format of the signal statements into the preferred carrier language format before the test statements can be compiled and executed. This preserves the best compromise between coding test requirements entirely in a proprietary test language and using a specific test-only language with restrictive programming features. The examples in 8.5 through 8.5.2 provide a background of how the TPL maps into a typical carrier language.

Using TPL, a test requirement may be written within any carrier language, and may at a later stage be ported to a different test system. There are two methods of achieving this portability. One would be to install the same carrier language compiler or interpreter onto the replacement test system and use the TPL program unchanged. The second would be to translate the carrier language content of the test requirement into a different carrier language, keeping the same TPL signal-oriented statements. Either method retains the advantage of using TPL test statements that are supported by the definitions within the STD standard.

13.2 Simple test requirement in TPL

The following test requirement is a series of simple tests written in the TPL. In this example, BASIC is used as the carrier language. In order to highlight the different components of the test requirement, different fonts are used for the TPL segments (Roman) and the carrier language segments (*Italic*). The program uses the following arbitrary conventions:

— Carrier language variable name are in Title Case, such as *`DCPowerValue`*

— TPL events are in title case preceded “e” as in `eRisingEdge`.

— Resources, such as sources, sensors and clocks, include uppercase as in `PulseCOUNTER`.

These conventions are used solely to help the reader identify the meaning of names and are not a recommendation. Users and organizations are free to adopt any convention that suits their particular situation or environment.

13.2.1 UUT for example test program 1

This UUT, represented in Figure 127, and its test program demonstrate some of the TPL statements and their use within a carrier language. It is not intended to represent a realistic UUT.

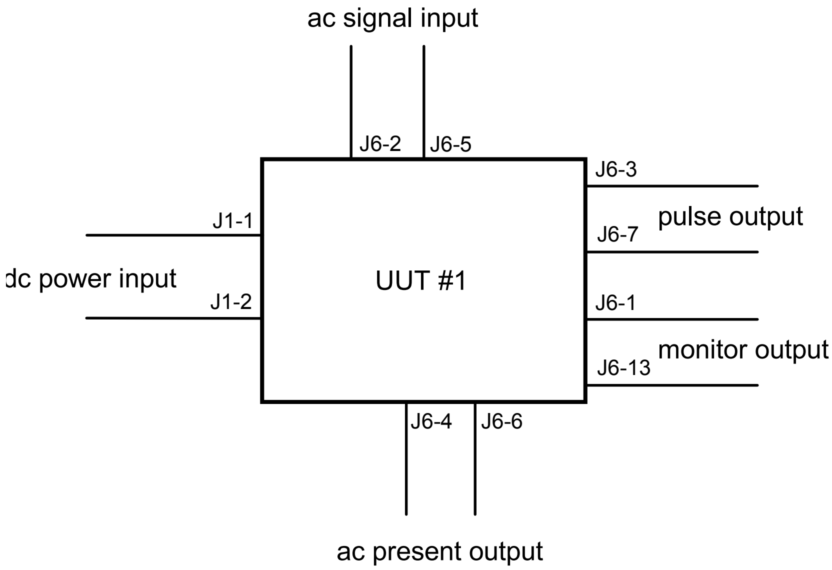


Figure 127 —UUT for example TPL program

The operation of the UUT is such that when power is applied to the dc power input (pins J1-1 and J1-2), a pulse stream is output from the pulse output (pins J6-3 and J6-7). When there is an appropriate ac signal input (pins J6-2 and J6-5), then an output voltage appears on the ac present output (pins J6-4 and J6-6). The monitor output (pins J6-1 and J6-13) provides an output voltage when the dc power input is at or above its minimum value.

The test program includes comments to explain the tests being performed.

13.2.2 Example test program 1

```
'*********************************************************************
'*   Test Methodology:          TPL                                  *
'*   UUT Type:                  Imaginary UUT                        *
'*   Programmer:                A N Other                            *
'*   Date:                      23 August 2004                       *
'*********************************************************************
'
'
'*********************************************************************
'* In this Example:                                                  *
'*   the Roman Text is Test Procedure Language Code                  *
'*   the Italic Text is Carrier Language (including comments)        *
'*                                                                   *
'*   The following conventions are used with names:                  *
'*   ThisCASE text      Resources (Sources, Sensors, Clocks, etc)    *
'*   Title Case text    Carrier language variables                   *
'*   Title Case (with an initial "e")  indicate events of all types  *
'*                                                                   *
'*********************************************************************
'
'
'*********************************************************************
'* Begin the preamble part that contains the declarative statements *
'*********************************************************************
```

```
'
    Dim Result As Single
    Dim IMeas, VCheck As Single
    Dim DropOutVolts As Single
    Dim PValue, FreqValue, TimeR, TimeF As Single
    Dim EighthEdgeOccurred As Boolean
    Dim DropOut As Boolean

    Dim NoOfPulses As Integer
    Dim TimeCount As Single
    Dim TimeToCarryOn As Single
    Dim ElapsedTime As Single

    Dim dcPowerValue As Single
    Dim Counter As Single
    Dim TimeToSwOff As Single


    EighthEdgeOccurred = False                       'initialize as False
    Drop-Out           = False                       'initialize as False

   'Open a file for results to be saved in
    Open "ResultsFile" For Output As #1

   'The following Setups identify events and define resources.
   'They may be placed immediately before they are required (enabled)

   '<TPL>
   'signal events
    Setup ampl inst GT 8.0 V as event ePowerOn;
    Setup ampl inst GT 2.5 V as event eRisingEdge;

    Setup ampl inst GT 2.5 V as event eEighthEdge;

    Setup ampl inst LT 6.0 V  as event eMidVolts;
    Setup ampl inst LT 0.01 V as event eLowVolts;

    Setup ampl inst GT 2.5 V  as event eACPresent;
    Setup ampl inst LT 2.5 V  as event eACAbsent;

   'event counter
    Setup Events of eRisingEdge
          as counter PulseCOUNTER;

   'evented event
    Setup from ePowerOn to eEighthEdge
            as event eTimeTo8thEdge;

   'define a clock to give an output every 1μs
    Setup 1 μs as clock UsecGENERATOR;

   'define an event counter to count outputs from μsec clock)
    Setup Events of UsecGENERATOR
          as counter UsecCLOCK;

   'Time Interval measurement
    Setup Interval range 0 μs to 10e8 μs
          from ePowerOn to eEighthEdge;
          as timer UsecTIMER;

   'This statement establishes a power supply resource.
   'However, there is no particular reason to define it prior to use.
```

```
    Setup DC_SIGNAL ampl dcPowerValue V
          errlmt ±0.08 V range 0 V to 12 V
          as source dcPOWER;

'*********************************************************************
'   This the start of the procedural part of the program
'*********************************************************************

   'Enable resources prior to power-up of UUT
    Enable UsecCLOCK;
    Enable UsecGENERATOR Sync with ePowerOn;

    Enable PulseCOUNTER;
    Enable UsecTIMER;
   '<TPL/>

    dcPowerValue = 12.0                  'Specify up DC voltage

   '<TPL>
   'Enable various events prior to power on of UUT

   'Enable ePowerOn Event
    Enable ePowerOn    on hi J1-1 lo J1-2;

   'Enable eRisingEdge Event
    Enable eRisingEdge on hi J6-3 lo J6-7;

   'Power on to LRU - this will trigger ePowerOn event
    Connect dcPOWER     to hi J1-1 lo J1-2;
   '<TPL/>

    Do
      '<TPL>
       Read PulseCOUNTER into NoOfPulses;
      '<TPL/>
    Loop While NoOfPulses < 7            'wait for the 7th edge

   '<TPL>
   'Enable eEighthEdge Event
   '(i.e. next eRisingEdge will be eEighthEdge)
    Enable eEighthEdge on hi J6-3 lo J6-7;
   '<TPL/>

    Do
      '<TPL>
       Read PulseCOUNTER into NoOfPulses;
      '<TPL/>
    Loop While NoOfPulses < 8            'wait for the 8th edge

    EighthEdgeOccurred = True

   '<TPL>
   'get the time from PowerOn to the eighth edge
    Read UsecTIMER into ElapsedTime;

   'Disable the Events
    Disable ePowerOn;
    Disable eRisingEdge;
    Disable eEighthEdge;
    Disable PulseCOUNTER;
   '<TPL/>

   'Save results.
```

```
      Print #1, "Time from power on to 8th pulse = " ;ElapsedTime


      TimeToCarryOn = ElapsedTime + 200000 'Add 200 ms
                                           'i.e. add 200,000 µs
                                           'wait for TimeToCarryOn
      Do
        '<TPL>
         Read UsecCLOCK into TimeCount;
        '<TPL/>
      Loop While TimeCount < TimeToCarryOn

     '<TPL>
      Setup SQUARE_WAVE dc_offset range 0 V to 2 V
         as sensor OffsetMETER;
      Connect hi J6-3 lo J6-7 to OffsetMETER;
      Read OffsetMETER into Result;
      Compare OffsetMETER <= 1.75 V;
     'The Compare statement will have set GO/NOGO flag
      Disconnect OffsetMETER;
      Reset OffsetMETER;
     '<TPL/>

     'Save results.
      Print #1, "DC Offset for Output = " ; Result

     'This subroutine call will measure the shape of the square wave

      Call MeasSquareWave (PValue, FreqValue, TimeR, TimeF)

     'Save results.
      Print #1, "Peak value of SquareWave = " ; PValue
      Print #1, "Frequency  of SquareWave = " ; FreqValue
      Print #1, "Rise Time  of SquareWave = " ; TimeR
      Print #1, "Fall Time  of SquareWave = " ; TimeF

     'If the result of the Compare was good
     'apply an ac signal and check that the offset is around zero

      If GO Then
        '<TPL>
         Setup AC_SIGNAL ampl 5 V
                         freq 400 Hz
               as source SigGEN;
         Connect SigGEN to hi J6-2 lo J6-5;

         Setup SQUARE_WAVE dc_offset into RESULT range –1 V to 1 V
               as sensor OffsetMETER;
         Connect hi J6-3 lo J6-7 to OffsetMETER;
         Read OffsetMETER into Result;
         Compare OffsetMETER LL -0.05 V UL +0.05 V;
         Disconnect OffsetMETER;
         Reset OffsetMETER;
        'Now check peak voltage of square wave
         Setup SQUARE_WAVE ampl pk range 0 V to 9 V
               as sensor PkMETER;
         Connect hi J6-3 lo J6-7 to PkMETER;
         Read PkMETER into PValue;
         Compare PkMETER LL 7.0 V UL 7.8 V;
         Disconnect PkMETER;
         Reset PkMETER;
        'Remove ac signal
         Disconnect SigGEN;
```

```
        '<TPL/>

        'Save results.
         Print #1, "DC Offset  with ac input = " ; Result
         Print #1, "Peak value with ac input = " ; PValue

      Else

        'If the original GO/NOGO test failed,
        'apply a link between TP2 and TP5 and recheck the dc offset
        '<TPL>
         Setup IMPEDANCE ampl 0 Ohm errlmt +0.05 Ohm - 0.0 Ohm,
                         ampl inst_max 1 A
               as source Link;
         Connect LinK to hi TP2 lo TP5;

         Setup SQUARE_WAVE dc_offset range 0 V to 2 V
               sensor OffsetMETER;
         Connect hi J6-3 lo J6-7 to OffsetMETER;
         Read OffsetMETER into Result;
         Compare OffsetMETER <= 1.75 V;
         Disconnect OffsetMETER;
         Reset OffsetMETER;

         Disconnect Link;
         Reset Link;
        '<TPL/>

        'Save results.
         Print #1, "DC Offset with link = " ; Result

      End If

     '<TPL>
     'Check the current consumption
      Setup DC_SIGNAL ampl range 0 mA to 200 mA
            sensor AMMETER;
      Connect via J1-1 to AMMETER ;
      Read AMMETER into IMeas;

     'Check the monitor output
      Setup DC_SIGNAL range 0 V to 12 V
            sensor VoltMETER;
      Connect hi J6-1 lo J6-13 to VoltMETER;
     '<TPL/>

     'Save results.
      Print #1, "Current consumption = " ; IMeas

     'Reduce the DC Power Supply voltage until
     'the signal at the monitor output drops below 2.54 V

      For DropOutVolts = 12 To 0 Step -0.05

        '<TPL>
         Change dcPOWER DC_SIGNAL ampl DropOutVolts V;
         Read VoltMETER into VCheck;
        '<TPL/>

         If VCheck < 2.54 Then             'dropout occurred
            DropOut = True
            Exit For                       'leave the For loop

```

```
            End If

           '<TPL>
           'Read the input current
            Read AMMETER into IMeas;
           '<TPL/>
           'Check if current is below 25 mA
            If IMeas < 0.025 Then Exit For    'leave the For loop

         Next                                 'value of DropOutVolts

        'Save results.
         If DropOut = True Then
            Print #1, "DropOut voltage = " ; DropOutVolts
         Else
            Print #1, "DropOut failed"
         End If

        '<TPL>
         Disconnect AMMETER;
         Reset AMMETER;
         Disconnect VoltMETER;
         Reset VoltMETER;

        'Ensure DC Power amplitude is 0V
         Change dcPOWER DC_SIGNAL ampl 0 V;

         Enable eRisingEdge on hi J6-3 lo J6-7;
         Enable ePowerOn    on hi J1-1 lo J1-2;

         Enable eACPresent  on hi J6-4 lo J6-6;
         Enable eACAbsent   on hi J6-4 lo J6-6;

         Enable PulseCOUNTER;
         Enable eTimeTo8thEdge;

        'AC_SIGNAL from Source SigGEN is still Setup from last use
        'Connect SigGEN to pins gated from Power-on to 8th Edge
         Connect SigGEN to hi J6-2 lo J6-5
                 gate with eTimeTo8thEdge;

        'Setup an event to be counted
         Setup ampl inst GT 2.5 V
               as event eCountEvent;

         Enable eCountEvent on hi J6-3 lo J6-7;

        'Setup EventCOUNTER
         Setup Events range max 1000
              of event eCountEvent
              as counter EventCOUNTER;

         Enable EventCOUNTER
                gate from eACPresent
                     to   eACAbsent;

        'Switch on the Power, this will start process
         Change dcPOWER DC_SIGNAL ampl 12 V;
        '<TPL/>

         Do
           '<TPL>
            Read PulseCOUNTER into NoOfPulses;
```

```
         '<TPL/>
       Loop While NoOfPulses < 7           'wait for the 7th edge

      '<TPL>
       Enable eEighthEdge on hi J6-3 lo J6-7;
      '<TPL/>

       Do
         '<TPL>
          Read PulseCOUNTER into NoOfPulses;
         '<TPL/>
       Loop While NoOfPulses < 8           'wait for the 8th edge

      '<TPL>
       Read EventCOUNTER into Counter;

      'Verify no of counts
       Compare EventCOUNTER LL 7 UL 9;

      'On switch off, check the time for monitor output to disappear
       Setup Interval range 0 s to 5 s
             as timer TimerCOUNTER;
       Enable eMidVolts on hi J1-1 lo J1-2;
       Enable eLowVolts on hi J6-1 lo J6-13;
       Enable TimerCOUNTER;

      'GATE Read between events
       Read TimerCOUNTER into TimeToSwOff
            gate from eMidVolts to eLowVolts;

      'Turn Power Off, this will start process
       Change dcPOWER DC_SIGNAL ampl 0 V;

       Wait_For 5 s;
      'The reading will now be available

       Disable eMidVolts;
       Disable eLowVolts;

       Disconnect dcPOWER;
       Reset dcPOWER;
       Disconnect all;
       Reset all;
      '<TPL/>

      'Save results.
       Print #1, "Time to switch off = " ; TimeToSwOff
       Close #1

    End                                    'End of Test Program


   '____________________________________________________________________

   'Define 'MeasSquareWave' as a 'Sub' Procedure

    Sub MeasSquareWave (ValP, ValF, ValRt, ValFt As Single)

      '<TPL>
       Setup SQUARE_WAVE ampl pk range 0 V to 9 V
             sensor PkMETER;
       Connect hi J6-3 lo J6-7 to PkMETER;
       Read PkMETER into ValP;
```

```
    Disconnect PkMETER;
    Reset PkMETER;

    Setup SQUARE_WAVE freq range 2.4 kHz to 2.6 kHz
          sensor FreqMETER;
    Connect hi J6-3 lo J6-7 to FreqMETER;
    Read FreqMETER into ValF;
    Disconnect FreqMETER;
    Reset FreqMETER;

    Setup SQUARE_WAVE rise_time range 10 µs to 200 µs
          sensor EdgeTIMER1;
    Connect hi J6-3 lo J6-7 to EdgeTIMER1;
    Read EDGETIMER1 into ValRt;
    Disconnect EdgeTIMER1;
    Reset EdgeTIMER1;

    Setup SQUARE_WAVE fall_time range 10 µs to 200 µs
          sensor EdgeTIMER2;
    Connect hi J6-3 lo J6-7 to EdgeTIMER2;
    Read EdgeTIMER2 into ValFt;
    Disconnect EdgeTIMER2;
    Reset EdgeTIMER2;
   '<TPL/>

End Sub
'____________________________________________________________________
```

13.3 Further test requirement in TPL

This example is a series of tests on a power supply unit.

13.3.1 Power supply UUT for example test program 2

The UUT is a unit that takes a three-phase ac input and provides a set of dc outputs. There are also some three-phase outputs derived directly from the inputs. The test program is a limited subset of tests that may be applied to a power unit and that are designed to highlight some specific features of the STD standard and the TPL.

The tests include measuring the output voltages of all the ac and dc outputs. The power supply requires the application of loads before the dc outputs will operate correctly; therefore, a new TSF signal model is required to represent a constant current load. The diagram in Figure 128 shows the connections into and out of the power supply.

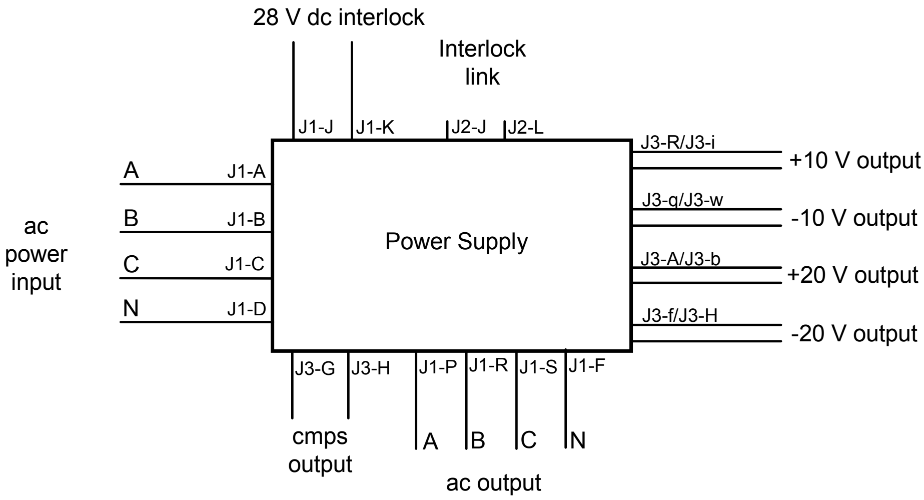


Figure 128 —Power supply UUT for TPL program

The power supply receives ac power from a three-phase and neutral supply. The supply provides four dc outputs: a +10 V supply, a –10 V supply, a +20 V supply, and a –20 V supply. Before the outputs are active, a 28 V dc interlock signal must be applied, and a link must be made between the Interlock link pins. When the supply is operating correctly, there is an output of 10 V on the cmps output. When the supply fails (due to an overvoltage condition on the +10 V output), the cmps output goes to zero. The ac output is transformer coupled to the input.

13.3.2 TSF signal model for example test program 2

This power supply UUT requires loads to be applied to each dc output to maintain the outputs operating correctly. Although there is a BSC for a load, this is a passive load model with specified impedance. Power supply testing requires constant current loads to be applied; therefore, a new TSF signal model is defined. This signal model is the constant BSC that can provide a current of specified amplitude. See Clause 10 for further information about the creation of a TSF signal model. The interface properties for the model are shown in Table 56, and the description of the model is shown in Table 57.

Table 56 —CONSTANT_I_LOAD interface

Description	Name	Type	Default	Range
Load current	ampl	Current	0	-20 A to +20 A

Table 57 —Constant current load model

Name	Type	Terminal	Inputs	Output	Formula
Constant Current	Constant	Signal [Out]	—	CONSTANT_I_LOAD	—
		amplitude	ampl	—	—

13.3.3 Example test program for power supply UUT

The test program is not comprehensive, nor is it to be taken as a good example of a power supply test program. It is only an example of a program that includes many commonly used TPL statements.

This test program also includes comments to explain the tests being performed.

```
'*********************************************************************
'*   Test Methodology:          TPL                                  *
'*   UUT Type:                  Power Supply                         *
'*   Programmer:                N G Neer                             *
'*   Date:                      26 August 2004                       *
'*********************************************************************
'
'
'*********************************************************************
'* In this Example:                                                  *
'*   the Roman Text is Test Procedure Language Code                  *
'*   the Italic Text is Carrier Language (including comments)        *
'*                                                                   *
'*   The following conventions are used with names:                  *
'*   ThisCASE text      Resources (Sources, Sensors, Clocks, etc)    *
'*   TitleCase text     Carrier language variables                   *
'*   eTitleCase (with an initial "e")  indicate events of all types *
'*                                                                   *
'*********************************************************************
'
'
'*********************************************************************
'* Begin the preamble part that contains the declarative statements *
'*********************************************************************
'
    Dim FailFlag As Boolean
    Dim PhaseA, PhaseB, PhaseC As Single
    Dim TenVPosLoad As Single
    Dim TenVNegLoad As Single
    Dim TwentyVPosLoad As Single
    Dim TwentyVNegLoad As Single
    Dim PowerA, PowerB, PowerC As Single
    Dim TotalPower As Single
    Dim TimeValue As Single
    Dim CmpsChange As Boolean

    FailFlag = False                     'initialize as False

   'Open a file for results to be saved in
    Open "ResultsFile" For Output As #1

   'The following Setups identify events and define resources.
   'They may be placed immediately before they are required (enabled)

   '<TPL>
   'signal events
    Setup ampl inst GT 14.0 V as event eOverVoltage;
    Setup ampl inst LT  4.0 V as event eOutputFail;

'*********************************************************************
'   This the start of the procedural part of the program
'*********************************************************************

   '<TPL/>

    TenVPosLoad    = +1.48                'Minimum Load for +10 V
    TenVNegLoad    = -0.736               'Minimum Load for -10 V
    TwentyVPosLoad = +0.38                'Minimum Load for +20 V
    TwentyVNegLoad = -0.065               'Minimum Load for -20 V
```

```
'<TPL>
'Setup ac supply

 Setup AC_SIGNAL ampl 115 V errlmt ± 2 V
 range 0 V to 160 V
 as source acPOWER;

'Setup 28 V Interlock supply

 Setup DC_SIGNAL ampl 28 V
 as source InterLOCK;

'Setup Measurement Resource for ac signals

 Setup AC_SIGNAL ampl range 0 V to 160 V
 as sensor acVoltMETER;

'Setup Measurement Resource for dc signals

 Setup DC_SIGNAL ampl range -30 V to +30 V
 as sensor dcVoltMETER;

'Connect the three phase supply
 Connect acPOWER ThreePhaseWye to
 a J1-A b J1-B c J1-C n J1-D;

 Wait_For 100 ms;

'Connect the interlock link
 Connect J2-J and J2-L as connection Link;

'Apply the minimum loads
 Call ApplyLoads
    (TenVPosLoad, TenVNegLoad, TwentyVPosLoad, TwentyVNegLoad)

'Connect interlock supply – power supply switches on
 Connect InterLOCK to hi J1-J lo J1-K;

 Wait_For 200 ms;

'Check the three phase outputs
 Connect hi J1-P lo J1-F to acVoltMETER;
 Read acVoltMETER into PhaseA;
 Compare acVoltMETER LL 110 V UL 120 V;
 Disconnect acVoltMETER;
'<TPL/>

 If NOGO Then FailFlag = True

'<TPL>
 Connect hi J1-R lo J1-F to acVoltMETER;
 Read acVoltMETER into PhaseB;
 Compare acVoltMETER LL 110 V UL 120 V;
 Disconnect acVoltMETER;
'<TPL/>

 If NOGO Then FailFlag = True

'<TPL>
 Connect hi J1-S lo J1-F to acVoltMETER;
 Read acVoltMETER into PhaseC;
 Compare acVoltMETER LL 110 V UL 120 V;
 Disconnect acVoltMETER;
```

```
'<TPL/>

 If NOGO Then FailFlag = True
'Save results.
 Print #1, "Three Phase Output Voltages"

 Print #1, "AC Output Phase A = " ; PhaseA; " V"
 Print #1, "AC Output Phase B = " ; PhaseB; " V"
 Print #1, "AC Output Phase C = " ; PhaseC; " V"

 Print #1, "Outputs with Minimum Load"

'<TPL>
'Check the output dc voltages
 Call OutputVoltageChecks

'Full Load for +10 V = +14.8 A
'Full Load for -10 V = -7.36 A
'Full Load for +20 V = +3.80 A
'Full Load for -20 V = -0.65 A

'Apply full loads
 Call ChangeLoads (+14.8, -7.36, +3.8, -0.65)
'<TPL/>

 Print #1, "Outputs with Full Load"

'<TPL>
'Check the output dc voltages
 Call OutputVoltageChecks

'Check the input ac power
 Setup AC_SIGNAL ampl range 0 V to 300 W
 as sensor acPowerMETER;

 Connect hi J1-P lo J1-F to acPowerMETER;
 Read acPowerMETER into PowerA;
 Disconnect acPowerMETER;
 Connect hi J1-R lo J1-F to acPowerMETER;
 Read acPowerMETER into PowerB;
 Disconnect acPowerMETER;
 Connect hi J1-S lo J1-F to acPowerMETER;
 Read acPowerMETER into PowerC;
 Disconnect acPowerMETER;
 Reset acPowerMETER;
'<TPL/>

 InputPower = PowerA + PowerB + PowerC
 Print #1, "AC Input Power = ";InputPower; " W"

 If InputPower > 680 Then FailFlag = True

'<TPL>
'Now check for cmps output to go low
'when overvoltage occurs.
 Setup Interval Range 0 ms to 300 ms
 from eOverVoltage
 to   eOutputFail
 as timer CmpsTIME;

'Setup the overvoltage input
 Setup DC_SIGNAL ampl 15 V
 as source OverVOLTAGE;
```

```
'Setup the cmps output check
 Connect hi J3-G lo J3-H to dcVoltMETER;

'Enable eOverVoltage Event
 Enable eOverVoltage on hi J3-R lo J3-i;

'Enable eOutputFail Event
 Enable eOutputFail on hi J3-G lo J3-H;

'Enable CMPSTIME timer
 Enable CmpsTIME
 timeout 350;

'Apply the overvoltage condition
 Connect OverVOLTAGE to hi J3-R lo J3-i;

'Allow time for events to happen
 Wait_For 400 ms;

'Get time value from times
 Read CmpsTIME into TimeValue;

'Check that cmps output did change.
 Read dcVoltMETER into Result;
 Compare dcVoltMETER LL -0.2 V UL +0.4 V;

 Disable eOverVoltage;
 Disable eOutputFail;
'<TPL/>

 If NOGO Then
    CmpsChange = False
    FailFlag = True
    Print #1, "cmps output failure (no change in level) "

 Else

   '<TPL>
    Compare CmpsTIME <= 0.198 s;
   '<TPL/>
    CmpsChange = True
    If Nogo Then
       FailFlag = True
       Print #1, "cmps output failure (response time too long) "
    Else
       Print #1, "cmps output response time good "
    End If

    Print #1, "cmps Response Time = "; TimeValue; " s"
 End If

'<TPL>
'Check that +20 V output has gone to zero
 Connect hi J3-A lo J3-b to dcVoltMETER;
 Read dcVoltMETER into Result;
 Compare dcVoltMETER LL -0.05 V UL =0.15 V;
 Disconnect dcVoltMETER;
'<TPL/>

 If NOGO Then
    FailFlag = True
    Print #1, "Overvoltage Trip Failure "
```

```
    End If

   '<TPL>
   'Release the overvoltage condition
    Disconnect OverVOLTAGE;
   'Check that +20 V output has recovered
    Connect hi J3-A lo J3-b to dcVoltMETER;
    Read dcVoltMETER into Result;
    Compare dcVoltMETER LL 18.22 V UL 21.88 V;
    Disconnect dcVoltMETER;
   '<TPL/>

    If NOGO Then
       FailFlag = True
       Print #1, "+20 V Output did not recover "
       Print #1, "+20 V Output voltage = "; Result; " V"
    End If

   '<TPL>
   'End of Tests
   'Remove interlocks and power
    Disconnect InterLOCK;
    Disconnect Link;
    Disconnect LoadPOS10V;
    Disconnect LoadNEG10V;
    Disconnect LoadPOS20V;
    Disconnect LoadNEG20V;

    Disconnect acPower;

   'Reset resources
    Reset eOverVoltage;
    Reset eOutputFail;

    Reset acPower;

    Reset InterLOCK;

    Reset LoadPOS10V;
    Reset LoadNEG10V;
    Reset LoadPOS20V;
    Reset LoadNEG20V;

    Reset dcVoltMETER;
    Reset acVoltMETER;

    Reset all;

   '<TPL/>
   'Save overall result.
    If FailFlag = True Then
       Print #1, "UUT Fails Tests - Refer to detailed results above"
    Else
       Print #1, "UUT Passes all Tests"
    End If
    Close #1

 End                                         'End of Test Program


'____________________________________________________________________

'Define "ApplyLoads" as a "Sub" Procedure
```

```
 Sub ApplyLoads (Load1, Load2, Load3, Load4 As Single)

    '<TPL>
     Setup CONSTANT_I_LOAD ampl Load1 A
     errlmt ±2% range 0 A to 20 A
     as source LoadPOS10V;

     Setup CONSTANT_I_LOAD ampl Load2 A
     errlmt ±2% range 0 A to 10 A
     as source LOADNEG10V;

     Setup CONSTANT_I_LOAD ampl Load3 A
     errlmt ±2% range 0 A to 5 A
     as source LoadPOS20V;

     Setup CONSTANT_I_LOAD ampl Load4 A
     errlmt ±2% range 0 A to 1 A
     as source LoadNEG20V;

     Connect LoadPOS10V to hi J3-R lo J3-i;
     Connect LoadNEG10V to hi J3-q lo J3-w;
     Connect LoadPOS20V to hi J3-A lo J3-b;
     Connect LoadNEG20V to hi J3-f lo J3-H;
    '<TPL/>

 End Sub
'____________________________________________________________________

'Define "ChangeLoads" as a "Sub" Procedure

 Sub ChangeLoads (Load1, Load2, Load3, Load4 As Single)

    '<TPL>
     Change LoadPOS10V
     Load1 A;

     Change LoadNEG10V
     Load2 A;

     Change LoadPOS20V
     Load3 A;

     Change LoadNEG20V
     Load4 A;
    '<TPL/>

 End Sub
'____________________________________________________________________

'Define "OutputVoltageChecks" as a "Sub" Procedure

 Sub OutputVoltageChecks (ValP, ValF, ValRt, ValFt As Single)

    '<TPL>
     Connect hi J3-R lo J3-I to dcVoltMETER;
     Read dcVoltMETER into Result;
     Compare dcVoltMETERt LL 8.51 V UL 10.19 V;
     Disconnect dcVoltMETER;
    '<TPL/>

     If NOGO Then FailFlag = True
    'Save results.
```

```
      Print #1, "+10 V Supply Output = " ; Result; " V"

     '<TPL>
      Connect hi J3-q lo J3-w to dcVoltMETER;
      Read dcVoltMETER into Result;
      Compare dcVoltMETER LL -10.19 V UL -8.51 V;
      Disconnect dcVoltMETER;
     '<TPL/>

      If NOGO Then FailFlag = True
     'Save results.
      Print #1, "-10 V Supply Output = " ; Result; " V"

     '<TPL>
      Connect hi J3-A lo J3-b to dcVoltMETER;
      Read dcVoltMETER into Result;
      Compare dcVoltMETER LL 18.22 V UL 21.88 V;
      Disconnect dcVoltMETER;
     '<TPL/>

      If NOGO Then FailFlag = True
     'Save results.
      Print #1, "+20 V Supply Output = " ; Result; " V"

     '<TPL>
      Connect hi J3-f lo J3-H to dcVoltMETER;
      Read dcVoltMETER into Result;
      Compare dcVoltMETER LL -21.88 V UL -18.22 V;
      Disconnect dcVoltMETER;
     '<TPL/>

     If NOGO Then FailFlag = True
    'Save results.
     Print #1, "-20 V Supply Output = " ; Result; " V"

 End Sub
'_______________________________________________________________________
```

13.4 Examples of test statements in TPL

The previous two example programs illustrate the use of many of the TPL statements in the context of a test requirement embedded in a typical carrier language. The following examples are of separate test statements that illustrate more of the optional parts of a TPL statement.

The use of optional content in a TPL statement enables a test requirement developer to add more detailed information about a test should it be required. In many instances, all that may be required is to make a simple statement such as “Apply 10 V” where the accuracy of the signal is not particularly significant. In this case, the test requirement implementer will be expected to judge from previous experience and knowledge of the type of UUT. In other situations, the test requirement may be such that it is necessary to make a more detailed statement such as “Apply 10.05 V +0.01 V –0.005 V with a current limit of 2.5 A.” The TPL provides the test requirement developer with the ability to specify tests with the necessary level of detail.

13.4.1 Setup source statements

13.4.1.1 Simple setup source statement

```
Setup DC_SIGNAL ampl 10 V
as source InputSignal;
```

This signal requirement simply asks for a 10 V dc signal and names it “InputSignal.”

13.4.1.2 Setup source with specified accuracy and additional attributes

```
Setup DC_SIGNAL dc_ampl 10.05 V errlmt +0.01 V –0.005 V,
                ac_ampl pk-pk 50 mV errlmt ± 1 mV,
                freq 60 Hz
as source SignalWithRipple;
```

This signal requirement refers to the signal model for DC_SIGNAL in Annex E of IEEE Std 1641-2010 and specifies a voltage of 10.05 V with an accuracy of +10 mV and –5 mV, together with a superimposed 60 Hz ac ripple voltage of 50 mV peak to peak with an accuracy of ± 1 mV.

13.4.1.3 Setup source with Gate

```
Setup AC_SIGNAL ampl 25 V errlmt 1,
                freq 200 Hz errlmt 10 Hz
gate with GatingEvent
as source acBurst;
```

This signal requirement refers to the signal model for AC_SIGNAL in Annex E of IEEE Std 1641-2010 and specifies an ac signal of 25 V with an accuracy of ±1 V. If not specified, the units for accuracy are automatically taken from the units of the signal value. The frequency is specified to be 200 Hz with an accuracy of ± 10 Hz. The signal will be gated on during the period that the event “GatingEvent” is present. In other words, the AC_SIGNAL starts at the beginning of the event “GatingEvent” and stops at the end of “GatingEvent.”

An alternative method of specifying the gate period would be to use two separate events: one for starting the signal (StartEvent) and the other for stopping the signal (StopEvent). This would be written as follows:

```
Setup AC_SIGNAL ampl 25 V errlmt 1,
                freq 200 Hz errlmt 10 Hz
gate from StartEvent to StopEvent
as source acBurst;
```

13.4.1.4 Setup source with Sync event

```
Setup AC_SIGNAL ampl 25 V errlmt 1,
                freq 200 Hz errlmt 10 Hz,
                phase 3.14159 rad
sync to SyncEvent
as source SyncedAC;
```

This signal will start when the event “SyncEvent” occurs, and the ac signal will always start with the phase angle specified. Each time the synchronizing event occurs, the ac signal will be resynchronized to restart at the specified phase angle.

It is possible to include both synchronizing and gating events. In this case, the signal will start on the first synchronizing event during the gate period and will be resynchronized if more synchronizing events occur during the gate period.

```
Setup AC_SIGNAL ampl 25 V errlmt 1,
                freq 200 Hz errlmt 10 Hz,
                phase 3.14159 rad
sync to SyncEvent
gate from StartEvent to StopEvent
as source SyncedACBurst;
```

13.4.1.5 Setup source with variable

```
Setup AC_SIGNAL ampl SupplyVoltage V errlmt 0.5 V
                                     range 0 V to 125 V,
             freq SupplyFreq Hz errlmt 10 Hz range 45 Hz to 65 Hz
as source acSupply;
```

This signal requirement illustrates the use of variables for passing the magnitudes of the voltage and frequency. The variables SupplyVoltage and SupplyFreq must be valid carrier language variables suitable for the value to be passed, which in this case are decimal numbers.

In this case, the range that the values may assume has also been included. This range provides more information for the implementation of the test requirement. Without the range values, it may not be easy to determine what sort of supply may be required. The implementer would have to search through the test requirement looking for all the possible values for SupplyVoltage and SupplyFreq.

13.4.2 Setup sensor statements

Setup statements for sensors are very similar to the setups for source signals. The simplest form allows the user to specify the signal and the attribute to be measured. In most cases, it is more expedient to supply additional information about the signal and measurement. This information helps the implementer ensure that the correct measuring instruments are available.

13.4.2.1 Simple setup sensor statement

```
Setup AC_SIGNAL ampl
sensor acMeter;
```

This signal requirement is valid but not particularly helpful. The implementer will have to refer to the rest of the test requirement or maybe even other documentation about the UUT in order to determine, for example, whether the signal to be measured is voltage or current and whether the signal is large or small. There is a considerable difference in measuring microvolts and kilovolts.

13.4.2.2 Setup source statement with additional information

```
Setup AM_SIGNAL mod-depth 30% errlmt ±3% range 0% to 100%,
```

```
                    mod-freq 200 Hz,
                    car-ampl 1.2 V,
                    car-freq 25 kHz
    as sensor ModMeter;
```

This statement provides considerably more information about the signal to be measured. The inclusion of the <mValue> expression provides more information about the measurement. It states that the depth of modulation of an AM signal is to be measured, but also gives the expected value, the required accuracy as ± 3%, and the range in which the measured value will fall as 0% to 100%. The expected value is given by the <mValue> expression, which in this example is 30%.

Other information about the signal is also provided, namely the modulation frequency, carrier frequency, and carrier amplitude. This additional information makes it easier to select a suitable measuring instrument.

As a minimum, a test requirement developer should always consider giving the expected value range for the measurement.

13.4.2.3 Different methods of observing an attribute

There are different ways of observing an attribute. For example, the amplitude of an ac signal may be expressed as an rms value, a peak-to-peak value, an average value, or one of several others. This is specified by the <mQualifier> expression, which in this example is pk-pk. If the <mQualifier> expression is omitted, it is assumed that an rms measurement is to be made.

```
    Setup AC_SIGNAL ampl pk_pk range 0v to 10 V,
                    freq 1 kHz
    as sensor acVoltmeter;
```

This statement is a setup to measure the peak-to-peak amplitude of an ac signal.

13.4.2.4 Gating and synchronization of measurements

The setup sensor statement allows the specification of the point at which a measurement is to occur by using a **Sync** event and the period over which the measurement is to be made by using a **Gate** event. If a measurement is to be made and no gate period is specified, it is assumed that the measurement period is adequate to obtain the required result. For example, when measuring rms, it is assumed that the signal is observed for long enough for a correct rms value to be ascertained. An instantaneous value may be observed at a specific instant as defined by an event as referenced with the sync keyword.

13.4.2.5 Setup sensor for undefined signal

A Setup statement specifies a measurement that is to be made on an unspecified signal. It is possible to measure the voltage of a signal without knowing exactly the characteristics of the signal. A signal output from a UUT may be an ac signal, a modulated signal of some kind (AM, FM, or PM), or perhaps a square wave signal, and so on. It is still possible to measure the peak-to-peak or rms voltage of that signal. In this case, the keyword Undefined_Signal is used in place of a signal name.

```
    Setup Undefined_Signal Voltage trms range 0 V to 25 V
    as sensor RMSMeter;
```

13.4.3 Setup event statements

Several different types of events are available. A signal-based event depends upon some attribute of a signal acquiring some value. An event-based event is dependent upon one or more other events. A time-based event is dependent upon clocked time. All events may be gated or synchronized to other events.

13.4.3.1 Signal-based event

```
    Setup ampl inst GT 8.0 V as event ePowerOn;
```

This event requirement (taken from an earlier example) specifies that an event called “ePowerOn” will occur when an amplitude achieves a value of greater than 8.0 V.

13.4.3.2 Event-based event

```
    Setup from StartEvent to StopEvent
    gated with AnotherEvent
    as event NewEvent;
```

The event “NewEvent” will occur when “StartEvent” occurs provided the gating event “AnotherEvent” is present. A further event will occur each time “StartEvent” occurs following a “StopEvent,” again, provided that the gating event “AnotherEvent” is present.

13.4.3.3 Time-based event

```
    Setup after  2 ms
          for   10 µs
          every 10 ms
          8 times
       sync to StartEvent
    as event TimeEvent;
```

This event requirement specifies an event that will occur every 10 ms for eight times. The event will last for 10 µs after a delay of 2 ms. The sequence of timed events will be triggered by the external event “StartEvent” as shown in Figure 129.

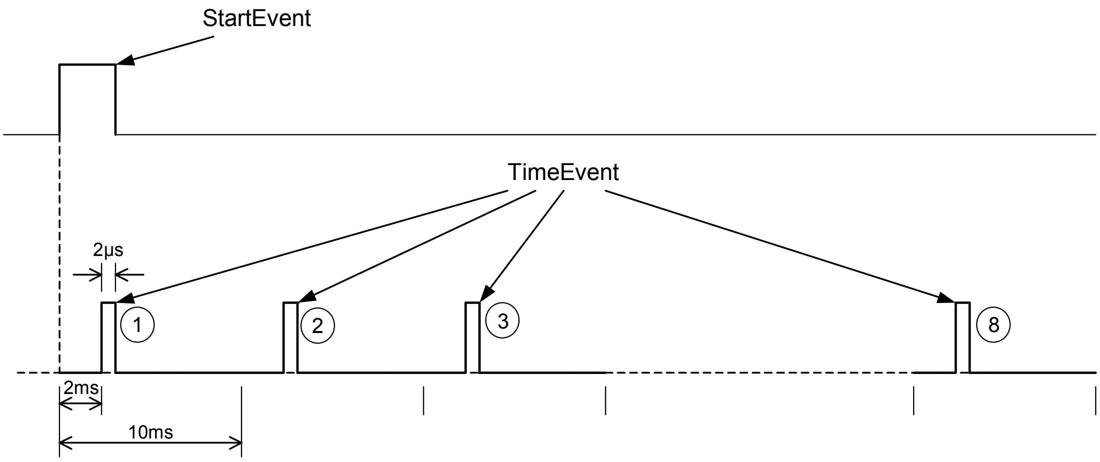


Figure 129 —Time-based events

13.4.4 Setup clock statement

The setup clock statement specifies a stream of events at the given frequency or period. An even mark-to-space ratio is assumed; in other words, if the clock is used as a gating signal, the gate will be "on" for the first half-period and "off" for the second half-period.

```
    Setup clock 5 kHz as clock SyncClock;
```

or

```
    Setup clock 200 μs as clock SyncClock;
```

Either of these statements will define the same clock, which may be gated or synchronized as any other source.

13.4.5 Setup time interval measurement statement

This setup time interval statement specifies the measurement of the time between any two events. These could be two different events or consecutive occurrences of the same event.

```
    Setup Interval errlmt 5% range 0 ms to 25 ms
    from Event1 to Event2
    as timer TimerCounter;
```

This statement specifies a timer (TimerCounter) with an accuracy of 5% that must be able to measure up to 25 ms. The time between the occurrences of "Event1" and "Event2" will be measured.

If "Event2" was omitted, the time between two occurrences of "Event1" would be measured.

```
    Setup Interval errlmt 5% range 0 ms to 25 ms
    from Event1;
```

These measurements may be gated or synchronized by other events.

13.4.6 Setup event counter

An event counter will count the number of occurrences of the specified event. This counter may be gated on and off so that the number of events that occur in a specified period may be counted.

```
Setup Events errlmt ±2 range MAX 500 of CountEvent
gate with EventEnable;
```

This event counter requirement will set up the counting of the number of “CountEvent” events that occur when gated by the event “EventEnable.” The accuracy required is ± 2 counts and not more than 500 “CountEvent” events are expected.

13.4.7 Reset statement

The Reset statement resets and releases the previously setup signal resource. It only requires the previously defined resource name. Several or all of the resources may be reset in a single statement. The statement does not imply that the resources will be reset in a particular order or with any specific timing.

The simplest form of the statement resets a single resource:

```
Reset acBurst:
```

To reset several resources a list is specified:

```
Reset acPower, dcVoltmeter, acVoltmeter;
```

To reset all resources use this statement:

```
Reset all;
```

13.4.8 Connect statements

Connect statements may be used to connect sources and sensors to the UUT and to connect UUT pins together.

13.4.8.1 Connect sources

The connect source statement causes a named source that has been established by a Setup statement to be connected to the UUT and initiates the signal unless the signal is subject to other synchronization or gating inputs. This statement may also be used to specify gate and synchronization inputs.

```
Connect acPOWER ThreePhaseWye to
a J1-A b J1-B c J1-C n J1-D;
```

This Connect statement, taken from an earlier example, connects an ac signal as a three-phase power input to the UUT. The ThreePhaseWye connection class means that four connections are required. These connections have the ConnectionClassPinNames of a, b, c, and n representing the three live phases and neutral. In this example the UUT pins names are J1-A, J1-B, J1-C, and J1-D.

Connections may be made to more than one pin at a time. For example, if a UUT has two pins for each connection in a power input, these may be specified in a single connect statement:

```
Connect dcPower to
```

```
hi PL1-1, PL1-2 lo PL1-5, PL1-6;
```

This statement connects the hi pin of the power signal to both pins PL1-1 and PL1-2, and the lo pin of the power signal is connected to both pins PL1-5 and PL1-6.

13.4.8.2 Connect sensors

The connect sensor statement is similar to the connect source. It causes a previously defined sensor that has been established by a Setup statement to be connected to the UUT and prepares the sensor to take a reading. If no **Gate** or **Sync** events have been defined, the measurement is taken immediately. In the case of the connect sensor statement, the order of the pins and sensor name is reversed. The following statement is taken from an earlier example and connects a pair of UUT pins to an instrument that measures ac volts:

```
Connect hi J1-R lo J1-F to acVoltmeter;
```

The following statement connects an ammeter to the UUT to measure the current flowing into (or out of) the UUT:

```
Connect via J1-1 to AMMETER;
```

The sensor ammeter will have already been set up as a sensor to measure current. As a current measurement requires a serial connection, the ammeter is connected to a single UUT pin (J1-1) using the ConnectionClassPinName "via."

13.4.8.3 Connecting pins together

Connecting UUT pins together is often required. The following statement shows two such pins being connected together. In this case the connection is given a name "Link," as it is simpler to refer to a connection name when removing the connection.

```
Connect J2-J and J2-L as connection Link;
```

13.4.9 Disconnect statement

The Disconnect statement removes a connection between the UUT and the resource. The Disconnect statement may reference the signal name or the connection name. When a connection is removed, the connection name (if used) will be released. Any related signal name will not be released because the signal will remain "Setup" until a reset statement releases it. All of the connections remaining at any time may be removed by a single statement.

These statements will disconnect all the resources used in the previous connect statements.

```
Disconnect Link;
Disconnect AMMETER;
Disconnect acVoltmeter;
Disconnect acPOWER;
Disconnect dcPower;
```

This set of Disconnect statements may be replaced by the single statement:

```
Disconnect all;
```

Using the keyword "all" in a disconnect statement will disconnect all the currently connected resources, but there is no implication that the connections will be released in a particular order or with any specific timing.

13.4.10 Enable statements

The Enable statements apply to all events, clocks, timers, counters, and so on. They play a similar part in the TPL as the Connect statements do for the signal resources. There are two formats for the Enable statements, one for enabling signal based events and one for all other events.

13.4.10.1 Enable event – general case

The simplest form of enable event statement switches on the event monitor so that the events may be observed. The events being enabled may be further gated or synchronized with other events. Several events may be enabled in a single statement, but any further gating or synchronization will apply to all named events.

```
Enable Event1, Event2;
```

This statement enables the two named events ("Event1" and "Event2") so that they may be observed.

```
Enable CmpsTIME
timeout 350;
```

If a timeout is specified in an Enable statement, it provides a maximum time that the resource monitor will have to wait for the event to occur. This limit prevents a program from hanging if something fails to occur.

13.4.10.2 Enabling signal-based events

The Enable statement for signal-based events enables the event and associates the event with the pins on which the signal is to be observed.

```
Enable eOverVoltage on hi J3-R lo J3-i;
```

13.4.11 Disable statement

The Disable statement switches off the event monitor that is associated with the specified event. All of the currently enabled resources may be disabled in a single statement. The statement does not imply that the resources will be disabled in a particular order or with any specific timing.

```
Disable eOverVoltage;
Disable all;
```

The two statements above will first disable the event "eOverVoltage" and then disable all of the remaining enabled resources.

13.4.12 Read statement

The Read statement prepares the sensor to take a reading. If no **Gate** or **Sync** events have been defined, the measurement is taken immediately, and the statement returns the measured attribute of the setup and connected sensor. The Read statement also provides the name of the variable into which the measured value is to be stored. This statement may be further gated or synchronized using events. A timeout value allows the user to specify the maximum time the system has to allow for the signal to be read.

```
Read acVoltmeter into Result;
Read TimerCounter into TimeVar
timeout 200 ms;
```

The first of the two Read statements takes the value measured by the sensor acVoltmeter and stores it in the carrier language variable Result. The second statement takes the value measured by the timer TimerCounter and stores it in the carrier language variable TimeVar. If the timer has not obtained a valid value after 200 ms, the program will continue, and the value in TimeVar will be indeterminate.

The variable name chosen must be a valid name according to the requirements of the carrier language. It may need to be pre-declared as the appropriate type. For example, most measurements will require a real (or decimal) variable in which to save the result.

If a sensor is set up to take a sequence of measurements, they are considered to be one operation. A **Sync** event starts the sequence of measurements, and another **Sync** event will restart the sequence and discard the original measurements from the incomplete sequence. When a complete sequence is attained, the results are returned into the results variable, and any further **Sync** events are ignored.

13.4.13 Change statement

The Change statement facilitates the changing of the numeric value of one or more attributes of a source signal. This statement is useful in situations where it is necessary to vary a parameter without disconnecting and reconnecting signals. For example, it may be required to reduce a power supply voltage and check that the UUT still behaves correctly on reduced power. Only the numeric value of previously established attributes may be changed. It is not possible to change one attribute for another.

```
Change dcSource 5 V;
```

The above Change statement changes the voltage of a named source dcSource from the previous value to 5 V. The dc source must have already been set up and connected to the UUT.

```
Change dcSource VoltageValue;
```

The above Change statement changes the voltage of a dc source from the previous value to the value provided in the variable VoltageValue. If this statement was used within a loop structure in the carrier language, it would be possible to ramp up or ramp down the voltage supplied by the dc source.

13.4.14 Compare statement

The Compare statement compares the value from a signal sensor with defined limits and sets GO and NOGO Boolean flags accordingly. A Read statement must have been executed.

```
Read EventCOUNTER into Counter;
Compare EventCOUNTER LL 7 UL 9;
```

The value associated with EventCOUNTER is compared with limits of 7 and 9. If the value is 7, 8, or 9, then the GO flag will be set to True, and the NOGO flag set to False. For any other value, the GO flag will be set to False, and the NOGO flag set to True.

As a Read statement must have previously been completed, the value in the read variable Counter could be tested in the carrier language.

13.4.15 Wait_For statement

The Wait_For statement causes the program to pause for the given period of time. This instruction could also be done within the carrier language, but using the TPL statement makes the timing implementation independent by specifying the time in seconds or associated units. For example, to pause execution for 100 ms to allow circuit elements to settle the statement would be stated as follows:

```
Wait_for 100 ms;
```

13.5 Quantities, units, and unit symbols

The STD standard defines the quantities that may be used in TPL statements in a table. The available units and their symbols are also listed in the table. It is recommended that only SI units be used for all test requirements. The STD standard supports the non-SI units that are used in customary practice for some technologies, such as knots for airspeed rather than meters per second.

13.5.1 Support for legacy test requirements

It is accepted that existing test requirements may use non-SI units, and the STD standard provides support for programs that use non-SI units. For reasons of traceability back to the original specification documents, it is often not possible to translate non-SI-based values to SI-based values.

13.5.1.1 Use of unit symbols

Wherever possible, the correct unit symbol (as defined in IEEE Std 260.1™–2004 [B8]) should be used in all test requirements. It may not be possible to use the designated symbol. For example, it may not be possible to enter the µ symbol or the ± symbol; in these cases, it is acceptable to use the alternative symbols “u” and “+-,” respectively.

Some programming languages that may be used as carrier languages may not allow these symbols or may use them as reserved or special symbols. If so, it is also possible to use the alternative symbols.

13.5.2 Entering special symbols

The entry of some symbols using a standard keyboard may require the use of an extended method. For convenience, the decimal codes for some special symbols are listed in the tables below. To enter the symbol in a commonly used operating system, press the [Alt] key and enter the decimal code on the numeric keypad including the leading zero. All the symbols listed in Table 58 are available in the ISO/IEC 8859-1:1997 [B14], windows-1252 [B4], and Unicode [B17] character sets.

Table 58—Decimal codes for symbols

Symbol	Decimal code
° (degree)	0176
± (plus-minus)	0177
^{2} (squared)	0178

Symbol	Decimal code
^{3} (cubed)	0179
µ (mu)	0181
• (middle dot)	0183

From experience, the keyboard codes in Table 59 may be used with the same commonly used operating system dependent upon the code pages loaded. These codes are entered by pressing the [Alt] key and entering the decimal code on the numeric keypad. The symbols for pi (π) and omega (Ω) may also be entered using these codes.

Table 59—Keyboard entry codes for symbols

Symbol	Keyboard code
π (pi)	227
µ (mu)	230
Ω (omega)	234
± (plus-minus)	241
° (degree)	248
• (middle dot)	250
^{2} (squared)	253

14. Signal Modeling Language (SML)

14.1 Introduction to SML

The SML is a derived version of the functional programming language Haskell 98. Information about Haskell 98 is freely available. Any user who requires information about the definition and use of Haskell 98 is referred to *Haskell 98 Language and Libraries, The Revised Report,* [B15] and *A Gentle Introduction to Haskell 98* [B3].

This clause is provided for users who wish to acquire a better understanding of the mathematical underpinning to the BSCs.

14.1.1 Objectives of SML

The SML has the following objectives:

— Provide a precise, consistent, complete, and traceable way to define signals and methods.

— Utilize basic signals as components of more complex functions.

— Permit signals that have been defined to be viewed.

— Allow easy use of signals via a defined interface.

— Support reusability of signal definitions.

— Provide a mathematically supportable basis for signal definitions.

The SML not only supports the definition of signal models but also provides a means of evaluating a resource that implements a signal.

14.1.2 SML support of test requirement creation

The SML permits signal definitions to be created by permitting new signals to be created from existing signals. The new signal may be viewed and analyzed to ensure that it meets the requirements of the new test situation.

The SML provides a mechanism for describing the properties of the BSCs that form the basis of all signals defined in the STD standard.

14.2 Using SML to define a BSC

As this subclause is a brief illustration of how the SML is used to define signals, it will be limited to a few simple examples of the definition of BSCs. Each example will show the textual BSC definition from Annex B of IEEE Std 1641-2010, the SML definition from Annex A of IEEE Std 1641-2010, and the plot of the signal.

14.2.1 Sinusoid BSC example

14.2.1.1 Textual definition

The textual BSC definition taken from Annex B of IEEE Std 1641-2010 is as follows:

Sinusoid<type: Physical> ::Periodic

a) *Definition*—A **Sinusoid** is a signal where the amplitude of the dependent variable is given by the formula in Equation (6):

$$e = A\sin(\omega t+\varphi) \tag{6}$$

where
- A is the amplitude
- ω is $2\pi \times$ frequency
- t is time
- 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.

14.2.1.2 SML definition

The corresponding SML definition of the sinusoid BSC taken from Annex A of IEEE Std 1641-2010 is as follows:

Sinusoid ::Periodic

```
>bscSinusoid =
> (\amplitude frequency phase ->
>   sine amplitude frequency phase
> )
```

14.2.1.3 Creating plot file

This SML definition can be used within a Haskell 98 tool to generate a file that can be used to plot the function. The waveform amplitude, frequency, and initial phase angle are defined in the Haskell 98 code that was executed. Examining Figure 130 will show that the amplitude is 1.5 V, the initial phase angle is 90°, and the frequency is 50 Hz (5 cycles are present in the 0.1 s timeframe). This information aligns with the information in the Haskell 98 code namely "bscSinusoid (V 1.5) (HZ 50) (DEG 90)." The plot was calculated for a time of 0.1 s.

Extract of code to create plot:

```
> sigSinusoid = (bscSinusoid (V 1.5) (HZ 50) (DEG 90))::(SignalRep Time Voltage)
> plotSinusoid = plotExample "Sinusoid" sigSinusoid 0.0 0.1 (-2) 2.0
```

The function bscSinusoid is defined separately to avoid a single, large expression. Extracts from a separate file are shown below where the function bscSinusoid is defined. The code shown is incomplete, and further information is imported from other modules such as information about physical types.

Extract 1:

```
>data BSC a b =
>-- Source
>---Periodic
>    Sinusoid {amplitude::b, frequency::Frequency, phaseAngle::PlaneAngle}
```

Extract 2:

```
> instance Signal BSC where
>---Periodic
>    toSig (Sinusoid amplitude frequency phase) = bscSinusoid amplitude frequency phase
```

Extract 3:

A.10.1.2.1 Sinusoid ::Periodic

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

The definition of Sinusoid in Annex A of IEEE Std 1641-2010 is reflected in extract 3.

14.2.1.4 Obtaining the plot

The example plot shown in Figure 130 was generated using the command line below in the plotting tool. The values 0 – 0.1 were used for time coordinates and –2.0 to 2.0 for the amplitude coordinates of the graph. In this example, the “with lines” attribute causes the plotted points to be joined so that the output waveform is more distinctive.

```
plot [0.0:0.1] [-2.0:2.0] "Sinusoid.plt" with lines
```

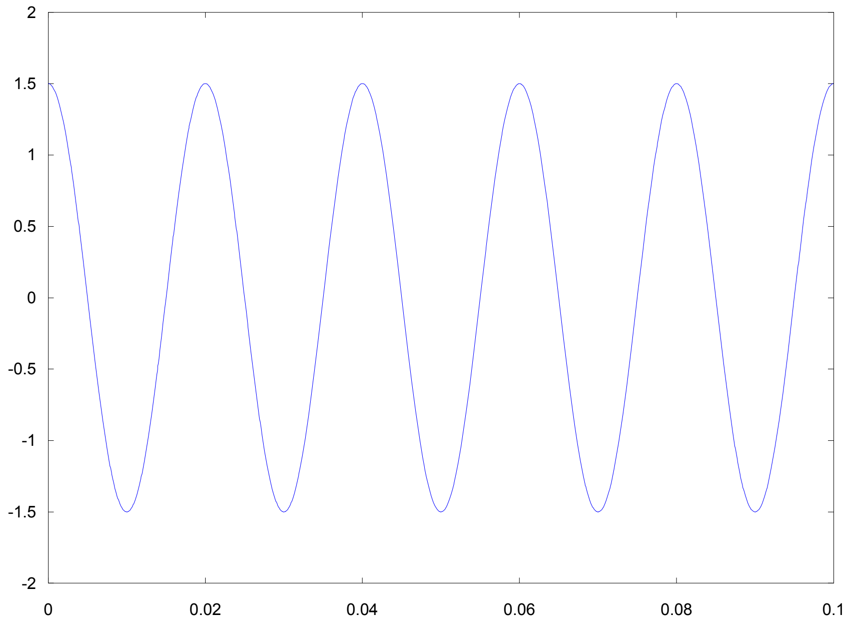


Figure 130 —Plot of sinusoid BSC from SML information

14.2.2 AM BSC example

14.2.2.1 Textual definition

The textual BSC definition taken from Annex B of IEEE Std 1641-2010 is as follows:

AM ::Modulator

a) *Definition*—**AM** is a modulator where the amplitude of the carrier varies with the amplitude of the modulating input signal.

b) *Attributes*

modIndex <ratio>—modulation index (depth of modulation) (default = 0.3)

Carrier <SignalFunction>—sinusoidal signal to be modulated

c) *Description*—The formula for AM signal is given in Equation (7):

$$e = E_c(1+\text{modIndex} \times m(t))\sin\omega_c t \tag{7}$$

where

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

As an example, the output for an AM-modulated sinusoid signal is given by Equation (8):

$$e = E_c(1+m_a\sin\omega_m t)\sin\omega_c t \tag{8}$$

where

E_c	is the carrier amplitude (unmodulated)
m_a	is the depth of modulation (≡ modulation index)
ω_m	is $2\pi \times$ modulating frequency
ω_c	is $2\pi \times$ carrier frequency
t	is time

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

14.2.2.2 SML definition

The corresponding SML definition of the sinusoid BSC taken from Annex A of IEEE Std 1641-2010 is as follows:

AM ::Modulator

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

14.2.2.3 Creating plot file

This SML definition can be used within a Haskell 98 tool to generate a file that can be used to plot the function. The waveform amplitude, frequency, and initial phase angle are defined in the Haskell 98 code that was executed. Examining Figure 131 will show that the amplitude is 1.5 V, the initial phase angle is 90°, and the frequency is 50 Hz (5 cycles are present in the 0.1 s timeframe). The information in the Haskell 98 code includes a definition of the carrier with an amplitude of 1 V and a frequency of 500 Hz and a definition of the modulation with an amplitude of 1 V and a frequency of 50 Hz. The definition of the AM signal sigAM defines the modulation depth of 0.3 (30%) and references the carrier and modulation definitions. The plot was calculated for a time of 0.1 s.

Extract of code to create plot:

```
> sigSinusoid = (bscSinusoid (V 1.5) (HZ 50) (DEG 90))::(SignalRep Time Voltage)
> sigCarrier = (bscSinusoid (V 1.0) (HZ 500) (DEG 0))::(SignalRep Time Voltage)
> sigAM = bscAM 0.3 sigCarrier sigSinusoid
> plotAM = plotExample "AM" sigAM 0.0 0.1 (-2) 2
```

The function bscAM is defined separately to avoid a single, large expression. Extracts from a separate file are shown below where the function bscAM is defined. The code shown is incomplete, and further information is imported from other modules such as information about physical types.

Extract 1:

```
>data BSC a b =
>--Conditioner
>---Modulator
>    AM {modIndex::Float, carrier::(SignalRep a b), signal::(SignalRep a b)}
```

Extract 2:

```
> instance Signal BSC where
>---Modulator
>    toSig (AM modIndex carrier signal) = bscAM modIndex carrier signalPeriodic
```

Extract 3:

```
A.10.2.3.2 AM ::Modulator

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

The definition of AM in Annex A of IEEE Std 1641-2010 is reflected in extract 3.

14.2.2.4 Obtaining the plot

The example plot shown in Figure 131 was generated using the command line below in the plotting tool. The values 0 – 0.1 were used for time coordinates and –2.0 to 2.0 for the amplitude coordinates of the graph. In this example, the “with lines” attribute causes the plotted points to be joined so that the output waveform is more distinctive.

```
plot [0.0:0.1] [-2.0:2.0] "AM.plt" with lines
```

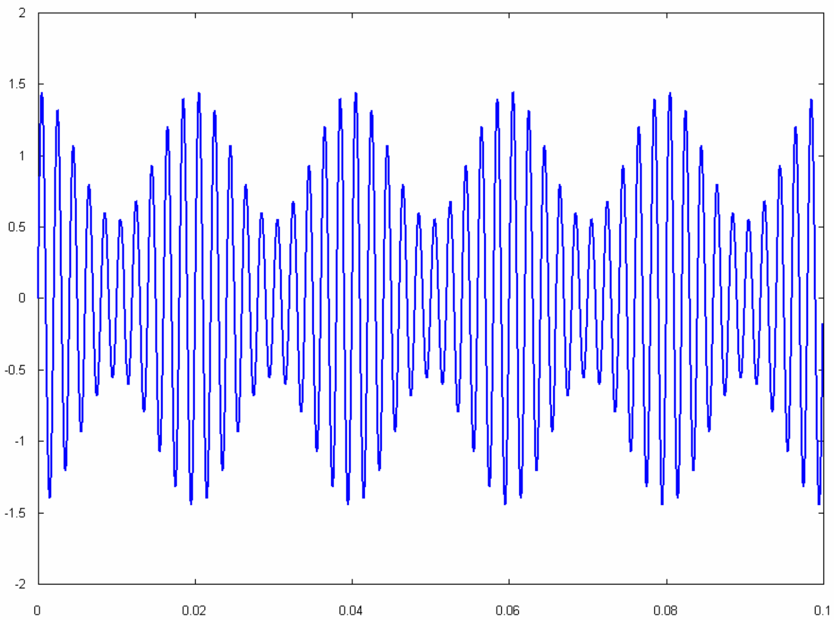


Figure 131 —Plot of AM BSC from SML information

14.2.3 Other functions

Some of the function definitions have been ignored in this brief outline. For example, "sine" and "mulSig" are defined in a separate "Pure" module (see Annex A of IEEE Std 1641-2010).

Annex A

(informative)

Glossary

For the purposes of this guide, the following terms and definitions apply. These and other terms within IEEE standards are found in the *Authoritative Dictionary of IEEE Standards Terms* [B5].

class: A template for the creation of an object instance. The class defines the properties of the object.

component: One of the parts that make up a system. A component may be hardware or software and may be subdivided into other components. Components communicate their functionality through their interface definitions.

function: The action or purpose which a specific item is intended to perform or serve.

method: A procedure implementing one of the operations supported by an object class.

model: A mathematical or physical representation of the system relationships.

property: A kind of responsibility that is an inherent or distinctive characteristic or trait that manifests some aspect of an object's knowledge or behavior.

semantics: The relationship of symbols or groups of symbols to their meanings within a language. For the test procedure language (TPL), semantics is the connotative meaning of words in a TPL statement. For software, semantics is the relationships of symbols and their meaning independent of the manner of their interpretation and use. For meta-languages, semantics is the discipline for expressing the meanings of computer language constructs in a meta-language.

sensor: A transducer that converts a parameter at a test point to a form suitable for measurement by the test equipment.

subclass: One of the classes, designated as such, whose attribute types are a superset of those of another class.

syntax: The structural or grammatical rules that define how the symbols in a language are to be combined to form words, phrases, expressions, and other allowable constructs.

system architecture: The structure and relationship among the components of a system. The system architecture may also include the system's interface with its operational environment.

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

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

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

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

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

transducer: A device to receive energy from one system and supply energy (of either the same or of a difference kind) to another system, in such a manner that the desired characteristics of the energy input appear at the output.

unit under test (UUT): The entity to be tested. It may range from a simple component to a complete system.

Annex B

(informative)

Intrinsic measurement

B.1 Introduction

Clause 7 described the two types of measurement, Intrinsic and Generic, available in IEEE 1641. This Annex provides more information about intrinsic measurement and explains in detail how to interpret intrinsic measurement information expressed in XML. It also includes details of all the information required to make such a measurement and what the default values are if specific information is not provided.

intrinsic—adjective, belonging naturally; essential.

This type of measurement performs the specific measurement specified by the BSC, no matter what the signal being measured. This is because all the Sensor BSC subclasses except Measure specifically define the measurement method.

B.2 Essential aspects and parts of a measurement

B.2.1 Information required to complete a measurement

There are five sets of information required in order to complete a measurement definition:

a) Method—what is the measurement being performed
b) Type—what is the type (and refType) of the signal
c) When—when is the measurement to be performed
d) Value—what results are required from the measurement
e) Capability—expected measurement value and associated information

These are always present even if they are not initially apparent in the measurement information. If they are not specifically stated, they will be obtained by inference or from defaults.

B.2.2 Default attributes for sensors

If an attribute is not specified in a measurement, it may be assumed to have a value from the following list:

— type = Voltage
— refType = Time
— measuredVariable = Dependent
— samples = 1

— gateTime = 0
— condition = NONE
— nominal = <no value> (0 is assumed if a value is required)
— UL = <no value> (0 is assumed if a value is required)
— LL = <no value> (0 is assumed if a value is required)
— As = <none>

B.2.3 The breakdown of a measurement

Most measurements are made using the dependent variable with the independent variable (refType) being time. Therefore, this discussion will focus on measurements of this type when describing the meaning of measurement information. The information provided is equally true for different dependent and independent types.

The contents of an XML measurement string is interpreted as follows:

B.2.3.1 Method (measurement method)

The measurement method is determined by the BSC class, e.g. RMS, Average, Instantaneous, etc.

B.2.3.2 Type (signal type)

a) Specified type
 If it is provided, the type is as specified, e.g. <… type = "Current" …>.

b) Inferred type
 If no specific type is provided, the type may determined from the nominal value, e.g. if the XML string contains <… nominal = "3 mA" …>, the type is Current.

c) Default type
 If the type is not specified or cannot be determined from the nominal value, then the default Type is assumed, i.e. the type is Voltage.

d) Translated type
 Only for translations between allowed types.

 If the type of the nominal value is different to the specified or default type, it may be a translated type, e.g. watts (W) to volts (V). The type of the signal remains as determined but the values are translated.

B.2.3.3 When

a) Condition
 The condition attribute determines when signal is evaluated.

When a condition attribute is used, the measurement(s) will only be taken and the abstract signal will only be output while the condition is satisfied. The event output is active while the condition is satisfied, otherwise it is inactive.

The nominal value provides the reference for the condition.

b) Gate time
The period over which the measurement is evaluated or the signal monitored.

1) gateTime = 0
The measurement is taken over a period that is implementation dependent. The implementation provides the best result that it is able

2) gateTime = *n*
The measurement is taken over the period defined by the gateTime (*n*).

3) gateTime = -*n*
The measurement is taken over the period of the gate event

c) External gate event
If an external gate event is used to control the measurement, the measurement may only be executed if the requirements of the gate event are satisfied as described in 7.2.2

An external gate event has precedence over the condition. If a measurement has both an external event input and a condition set, the condition will only be evaluated and the measurement completed when the gate event occurs.

B.2.3.4 Value

a) Samples
Samples determines how many measurements are recorded

1) Samples = 1 (default)
A measurement is taken and the result stored in the read-only attribute "measurement".

2) Samples = *n* (where *n* >1)
A series of *n* measurements are taken and stored in the read-only array "measurements". The last value measured is also stored in the read-only attribute "measurement".

3) Samples = 0
The BSC is used as a monitor only. No measurements are recorded and the attributes "measurement" and "measurements" do not contain any valid values. The attribute "count" will contain the value 0 (zero).

b) Measured variable
The value or values returned by the measurement will be either the values related to the signal's physical or the point at which the measurement was achieved.

1) measuredVariable = Dependent (default)

The measurement values are of the type of the signal being evaluated.

2) measuredVariable = Independent

The measurement values are of the refType, i.e. the independent variable associated with the measurement.

The descriptions below assume the refType is time for the same of simplicity and understanding. The principles hold true for any other refType.

B.2.3.5 Capability (nominal values)

a) Compatible nominal values

Nominal values that are directly compatible with the measurement method and the signal type provide additional information for the measurement.

1) magnitude

The magnitude gives an indication of the expected value. This is for indication only and has no effect on the measured value.

2) unit

The unit of a compatible nominal value must be compatible with the type of the measured signal.

3) qualifier

The qualifier of a compatible nominal value must be compatible with the measurement method

4) errlmt

The errlmt value defines the acceptable uncertainty for the measurement method.

5) range

The range attribute provides the range of values over which the measurement system is expected to operate. The measurement is not constrained but may not provide the correct value if it outside of the specified range.

6) res

The res attribute provides the minimum resolution required of the measurement system.

b) Incompatible nominal values

Nominal values that are not directly compatible with the measurement and the signal type are assumed to be Capability attributes, i.e. they provide additional information about the signal being measured but do not contribute any information about the measurement itself. A nominal value is incompatible if one or both of ii) and iii) below are true.

A capability attribute describes an aspect of the measured signal that may help with suitable resource allocation.

1) magnitude

The magnitude gives an indication of the magnitude of the capability attribute of the expected signal. This is for indication only and has no effect on the measured value.

2) unit

The unit of the capability attribute may be incompatible with the type of the measured signal.

3) qualifier

The qualifier of the capability attribute may be incompatible with the measurement method

4) errlmt

The errlmt value defines an uncertainty in the specified magnitude of the capability attribute.

5) range

The range attribute provides a range of values over which the capability attribute may be found. The capability attribute is not constrained to the range specified but is expected to remain within the specified range.

6) res

The res attribute does not provide any useful information about a capability attribute.

B.2.4 Returned values

The measurement returns values as a result of measurements in read-only attributes as follows:

a) Measurement—the last (or only) measurement value taken
b) Measurements—an array containing the values of the measurements taken
c) Count—the number of measurements taken
d) GO—set by comparing measured values against the LL and UL attributes
e) NOGO—set by comparing measured values against the LL and UL attributes
f) HI—set by comparing measured values against the UL attribute
g) LO—set by comparing measured values against the LL attribute

B.3 Interpreting measurement information

Table B.1 lists all the parts of a measurement where the default value is provided if one is available. This table may be used as a baseline for interpreting any intrinsic measurement by entering the values from the measurement expression into the table and using the defaults for any essential missing data.

Table B.1—Default intrinsic measurement information

Information	Default	Notes
Measurement method	<none>	From BSC class
Signal attributes		
type	Voltage	Specific, from Nominal, or default

Information	Default	Notes
refType	Time	
measuredVariable	Dependent	
samples	1	
gateTime	0	Default = Implementation dependant
condition	NONE	
nominal	<no value>	
UL	<no value>	0 assumed if value required
LL	<no value>	0 assumed if value required
As	<none>	
Measurement Results		Read-only attributes
Measurement	<qualifier> voltage	method, type (if samples ≠ 0)
Measurements	<qualifier> voltage or n/a	n/a if samples = 1 (default) or 0
Events	1 when measurement complete	Default for samples = 1 (default)
Output abstract signal	Voltage @ measured level	Abstract signal (at magnitude)
GO	1 if within limits, else 0	
NOGO	0 if within limits, else 1	
HI	1 if > UL, else 0	
LO	1 if < LL, else 0	
Measurement info.		
Expected value	Any	From Nominal (if compatible type)
Min value expected	Any	From Nominal (if compatible type)
Max value expected	Any	From Nominal (if compatible type)
Permitted uncertainty	Any	From Nominal (if compatible type)
Required resolution	Any	From Nominal (if compatible type)
Capability info.		
Characteristic (type)	None provided	From Nominal (if different type)
Observation method	None provided	From Nominal (if different type)
Value	None provided	From Nominal (if different type)
Max value	None provided	From Nominal (if different type)
Uncertainty in value	None provided	From Nominal (if different type)

B.4 Examples showing breakdown of derivation of attributes

The following examples of measurement expressions in XML each have a brief interpretation of their meaning, followed by a table that shows each attribute is derived. If there is no information in the XML expression from which to derive the attribute value, the default is used.

B.4.1 Simple RMS measurement

```
<RMS />
```

Interpretation—Take an rms voltage measurement and create an event when taken (see Table B.2).

Table B.2—Derivation of attribute values for simple rms measurement

Attribute	Value	Derivation
Signal attributes		
method	RMS	BSC class
type	Voltage	default
refType	Time	default
measuredVariable	Dependent	default
samples	1	default

Attribute	Value	Derivation
gateTime	Implementation dependant	default = 0
condition	NONE	default
nominal	<no value>	default
UL	<no value>	default 0 assumed if required
LL	<no value>	default 0 assumed if required
As	<none>	default
Read-only attributes		
Measurement	rms voltage	default (method, type)
Measurements	n/a	default (samples = 1)
Events	1 when measurement complete	default (samples = 1)
Output abstract signal	Voltage @ rms level	default
GO	1 if within limits, else 0	
NOGO	0 if within limits, else 1	
HI	1 if > UL, else 0	
LO	1 if < LL, else 0	
Measurement info.		
Expected value	Any	default
Min value expected	Any	default
Max value expected	Any	default
Permitted uncertainty	Any	default
Required resolution	Any	default
Capability info.		
Characteristic (type)	none provided	
Observation method	none provided	
Value	none provided	
Max value	none provided	
Uncertainty in value	none provided	

B.4.2 RMS monitor

```
<RMS samples="0" />
```

Interpretation—Output an abstract signal of rms voltage values that would have been taken (see Table B.3).

Table B.3—Derivation of attribute values for rms monitor

Attribute	Value	Derivation
Signal attributes		
method	RMS	BSC class
type	Voltage	default
refType	Time	default
measuredVariable	Dependent	default
samples	0	Samples=0
gateTime	Implementation dependant	default = 0
condition	NONE	default
nominal	<no value>	default
UL	<no value>	default
LL	<no value>	default
As	<none>	default
Read-only attributes		
Measurement	none	samples = 0
Measurements	none	samples = 0
Events	none	samples = 0

Attribute	Value	Derivation
Output abstract signal	Voltage @ rms level	default
GO	0	samples = 0
NOGO	0	samples = 0
HI	0	samples = 0
LO	0	samples = 0
Measurement info.		
Expected value	Any (used for abstract output only)	
Min value expected	Any (used for abstract output only)	
Max value expected	Any (used for abstract output only)	
Permitted uncertainty	Any (used for abstract output only)	
Required resolution	Any (used for abstract output only)	
Capability info.		
Characteristic (type)	none provided	
Observation method	none provided	
Value	none provided	
Max value	none provided	
Uncertainty in value	none provided	

B.4.3 RMS measurement with an expected value

```
<RMS nominal="0.03 +- 3% range MAX 10"/>
```

Interpretation—Take an rms voltage measurement with an uncertainty better than 3%. The expected measurement value is 30 mV but could be as high at 10 V (see Table B.4).

Table B.4—Derivation of attribute values for rms measurement with an expected value

Attribute	Value	Derivation
Signal attributes		
method	RMS	BSC class
type	Voltage	default
refType	Time	default
measuredVariable	Dependent	default
samples	1	default
gateTime	Implementation dependant	default = 0
condition	NONE	default
nominal	<no value>	default
UL	<no value>	default
LL	<no value>	default
As	<none>	default
Read-only attributes		
Measurement	rms voltage	default (method, type)
Measurements	n/a	default (samples = 1)
Events	1 when measurement complete	default (samples = 1)
Output abstract signal	Voltage @ rms level	default
GO	1 if within limits, else 0	
NOGO	0 if within limits, else 1	
HI	1 if > UL, else 0	
LO	1 if < LL, else 0	
Measurement info.		
Expected value	0.03 V	nominal
Min value expected	Any	default
Max value expected	10 V	nominal

Attribute	Value	Derivation
Permitted uncertainty	±3%	nominal
Required resolution	Any	default
Capability info.		
Characteristic (type)	none provided	
Observation method	none provided	
Value	none provided	
Max value	none provided	
Uncertainty in value	none provided	

B.4.4 RMS measurement with type conversion

```
<RMS nominal="3 mW +- 3% range MAX 1 W" type= "Voltage" />
```

Interpretation—Take an rms voltage measurement and convert the result to power (mW) with an uncertainty better than 3%. The expected measurement value is 3mW but could be as high at 1W (see Table B.5).

Table B.5—Derivation of attribute values for rms measurement with type conversion

Attribute	Value	Derivation
Signal attributes		
method	RMS	BSC class
type	Voltage	default
refType	Time	default
measuredVariable	Dependent	default
samples	1	default
gateTime	Implementation dependant	default = 0
condition	NONE	default
nominal	<no value>	default
UL	<no value>	default
LL	<no value>	default
As	<none>	default
Read-only attributes		
Measurement	rms voltage	default (method, type)
Measurements	n/a	default (samples = 1)
Events	1 when measurement complete	default (samples = 1)
Output abstract signal	Voltage @ rms level	default
GO	1 if within limits, else 0	
NOGO	0 if within limits, else 1	
HI	1 if > UL, else 0	
LO	1 if < LL, else 0	
Measurement info.		
Expected value	3 mW, i.e. 0.387 V (using 50 Ω)	Nominal (50 Ω is default)
Min value expected	Any	default
Max value expected	1 W, i.e. 7.071 V (using 50 Ω)	Nominal (50 Ω is default)
Permitted uncertainty	±3%, i.e. 1.5% of voltage value (approx)	nominal
Required resolution	Any	default
Capability info.		
Characteristic (type)	none provided	
Observation method	none provided	
Value	none provided	
Max value	none provided	
Uncertainty in value	none provided	

B.4.5 RMS current measurement with capability information

```
<RMS nominal="av 30 mA +- 3% range MAX 1A" />
```

Interpretation (see Table B.6):

— Take an rms current measurement, the expected **average** current characteristic of the input signal is 30mA, within 3%, but could be as high as 1A;

— This nominal value is used as Capability of the signal. The required accuracy for the measurement is not specified and may be any value. No expected value, range or resolution for the measured value is provided.

Table B.6—Derivation of attribute values for current measurement with capability information

Attribute	Value	Derivation
Signal attributes		
method	RMS	BSC class
type	Current	implied from nominal
refType	Time	default
measuredVariable	Dependent	default
samples	1	default
gateTime	Implementation dependant	default = 0
condition	NONE	default
nominal	<no value>	default
UL	<no value>	default
LL	<no value>	default
As	<none>	default
Read-only attributes		
Measurement	rms voltage	default (method, type)
Measurements	n/a	default (samples = 1)
Events	1 when measurement complete	default (samples = 1)
Output abstract signal	Voltage @ rms level	default
GO	1 if within limits, else 0	
NOGO	0 if within limits, else 1	
HI	1 if > UL, else 0	
LO	1 if < LL, else 0	
Measurement info.		
Expected value	Any	default
Min value expected	Any	default
Max value expected	Any	default
Permitted uncertainty	Any	default
Required resolution	Any	default
Capability info.		
Characteristic (type)	Current	nominal
Observation method	Average	nominal
Value	30 mA	nominal
Max value	1 A	nominal
Uncertainty in value	± 3%	nominal

B.4.6 RMS voltage measurement with capability information

```
<RMS nominal="av 30 mA +- 3% range MAX 1A" type= "Voltage" />
```

Interpretation (see Table B.7):

— Take an rms voltage measurement, the expected **average** current characteristic of the input signal is 30mA but could be as high as 1A.

— This nominal value is used as Capability of the signal. The required accuracy for the measurement is not specified and may be any value. No expected value, range or resolution for the measured value is provided.

Table B.7—Derivation of attribute values for voltage measurement with capability information

Attribute	Value	Derivation
Signal attributes		
method	RMS	BSC class
type	Voltage	type= "Voltage"
refType	Time	default
measuredVariable	Dependent	default
samples	1	default
gateTime	Implementation dependant	default = 0
condition	NONE	default
nominal	<no value>	default
UL	<no value>	default
LL	<no value>	default
As	<none>	default
Read-only attributes		
Measurement	rms voltage	default (method, type)
Measurements	n/a	default (samples = 1)
Events	1 when measurement complete	default (samples = 1)
Output abstract signal	Voltage @ rms level	default
GO	1 if within limits, else 0	
NOGO	0 if within limits, else 1	
HI	1 if > UL, else 0	
LO	1 if < LL, else 0	
Measurement info.		
Expected value	Any	default
Min value expected	Any	default
Max value expected	Any	default
Permitted uncertainty	Any	default
Required resolution	Any	default
Capability info.		
Characteristic (type)	Current	nominal
Observation method	Average	nominal
Value	30 mA	nominal
Max value	1 A	nominal
Uncertainty in value	± 3%	nominal

B.4.7 RMS voltage measurement with capability information

```
<Average name="AverageVoltageMeas" samples="2" gateTime="25 ms"
nominal="trms 8 V +-0.5 V range MAX 10 V" condition="GT" UL="+1 V"
LL="- 1 V" In="TwoWireInp" />
```

Interpretation (see Table B.8):

— Take two average voltage measurements, each to be measured with a gateTime of 25 ms. Verify that the average values measured are between the limits of -1 V and +1 V. The signal is to be evaluated when the trms value of the signal is greater than 8 V.

— The expected **trms** voltage characteristic of the input signal is 8 V ±0.5 V, but could be as high as 10 V.

— This nominal value is used as Capability of the signal. The required accuracy for the measurement is not specified and may be any value. No expected value, range or resolution for the measured value is provided.

Table B.8—Derivation of attribute values for voltage measurement with capability information

Attribute	Value	Derivation
Signal attributes		
method	Average	BSC class
type	Voltage	default
refType	Time	default
measuredVariable	Dependent	default
samples	2	samples="2"
gateTime	25 ms	gateTime="25 ms"
condition	GT	condition="GT"
nominal	8 V trms	nominal="trms 8 V"
UL	+1 V	UL="+1 V"
LL	–1 V	LL="-1 V"
As	<none>	default
Read-only attributes		
Measurement	Average voltage	samples = 2
Measurements	Array of two values	samples = 2
Events	2 (one with each measurement)	samples = 2
Output abstract signal	Voltage @ average level	default
GO	2 if both meas. within limits, else 1 or 0	UL & LL specified
NOGO	0 if both meas. within limits, else 1 or 2	UL & LL specified
HI	0 if both meas. < UL, else 1 or 2	UL specified
LO	0 if both meas. > LL, else 1 or 2	LL specified
Measurement info.		
Expected value	Any	default
Min value expected	Any	default
Max value expected	Any	default
Permitted uncertainty	Any	default
Required resolution	Any	default
Capability info.		
Characteristic (type)	Voltage	nominal
Observation method	trms	nominal
Value	8 V	nominal
Max value	10 V	nominal
Uncertainty in value	± 0.5 V	nominal

B.5 Creating IEEE 1641 measurements for legacy ATLAS systems

B.5.1 Introduction

A selection from the list of measurements used by a legacy ATLAS system is used here to illustrate the measurements that can be defined using IEEE 1641. The list is presented using the original ATLAS terms; the first column shows a list of ATLAS NOUN-MODIFIERS, and the second column indicates which ATLAS NOUNs the measurement applies to. In ATLAS, a NOUN MODIFIER is always linked to a specified NOUN, but that is not always necessary in IEEE 1641. Often the measurement technique is independent of the type of signal being measured; effectively meaning the ATLAS NOUN may be ignored: these are the intrinsic measurements and will be the same for all occurrences of the NOUN-MODIFIER. In the cases where the type of the signal is important to the measurement, i.e. where the ATLAS NOUN has to be considered, this gives rise to an IEEE 1641 generic measurement.

B.5.2 List of typical ATLAS measurements

Table B.9 identifies which type of IEEE 1641 measurement is required for each example. Note that some of the ATLAS measurements are not pure IEEE 716 ATLAS constructions but are system specific modified ATLAS. For the VOLTAGE-AV intrinsic measurement, an example TSF is provided.

Table B.9—Legacy ATLAS measurements and the IEEE 1641 equivalents

Legacy ATLAS NOUN-MODIFIER	Legacy ATLAS NOUN	Equivalent IEEE 1641 measurement type
AMPLITUDE-PK-NEG	Any valid NOUN	Intrinsic
AMPLITUDE-PK-POS	Any valid NOUN	Intrinsic
CAR-FREQ	Any valid NOUN	Intrinsic
CAR-FREQ	AM SIGNAL	Generic
CAR-FREQ	FM SIGNAL	Generic
CAR-FREQ	PM SIGNAL	Generic
COUNT	Any valid NOUN	Intrinsic
CURRENT	Any valid NOUN	Intrinsic
CURRENT-TRMS	Any valid NOUN	Intrinsic
DC-OFFSET	Any valid NOUN	Intrinsic
DC-OFFSET	PULSED DC SIGNAL	Generic
DISTORTION	AC SIGNAL	Generic
DISTORTION	DC SIGNAL	Generic
FALL-TIME	Any valid NOUN	Intrinsic
FREQ-DEV	FM SIGNAL	Generic
FREQUENCY	Any valid NOUN	Intrinsic
HARM-***-POWER	Any valid NOUN	Intrinsic
HARM-***-VOLTAGE	Any valid NOUN	Intrinsic
MOD-AMPL	AM SIGNAL	Generic
MOD-AMPL	FM SIGNAL	Generic
MOD-AMPL	PM SIGNAL	Generic
MOD-AMPL	PAM SIGNAL	Generic
MOD-FREQ	AM SIGNAL	Generic
MOD-FREQ	FM SIGNAL	Generic
MOD-FREQ	PM SIGNAL	Generic
MOD-FREQ	PAM SIGNAL	Generic
TIME	Any valid NOUN	Intrinsic
VOLTAGE	Any valid NOUN	Intrinsic
VOLTAGE-AV	Any valid NOUN	Intrinsic
VOLTAGE-INST-NEG	Any valid NOUN	Intrinsic
VOLTAGE-INST-PK-NEG	Any valid NOUN	Intrinsic

VOLTAGE-INST-PK-POS	Any valid NOUN	Intrinsic
VOLTAGE-INST-POS	Any valid NOUN	Intrinsic
VOLTAGE-PK-NEG	Any valid NOUN	Intrinsic
VOLTAGE-PK-PK	Any valid NOUN	Intrinsic
VOLTAGE-PK-POS	Any valid NOUN	Intrinsic
VOLTAGE-TRMS	Any valid NOUN	Intrinsic

B.5.3 Example IEEE 1641 equivalent to ATLAS VOLTAGE-AV

In this example, VOLTAGE_AV provides the average voltage value (noun modifier VOLTAGE-AV) of any signal, irrespective of the signal (ATLAS noun).

Figure B.1 shows the VOLTAGE_AV TSF, which comprises a single "Average" BSC and an input.

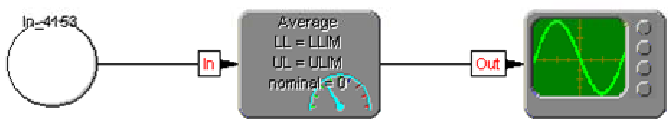


Figure B.1—Model of average voltage measurement TSF

Figure B.2 shows the VOLTAGE_AV TSF being used to measure the average voltage of an incoming signal. The measurement is independent of the signal type. The example shows a signal which combines a dc component of 1.6 V and an ac component of 2 V, resulting in an ac signal with a dc offset.

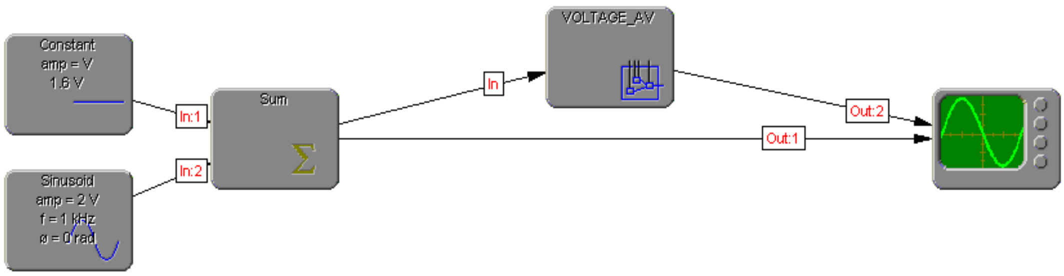


Figure B.2—TSF used for measurement

The measured value is the average voltage (1.6 V) and is shown by the center trace. The gate time (ATLAS GATED-BY) determines how quickly and accurately a result is obtained and this is illustrated by the trace with the decaying waveform, which shows that several cycles are required before the correct average value is determined. This can be seen more clearly in Figure B.3, which is an enlarged view of the 'scope display.

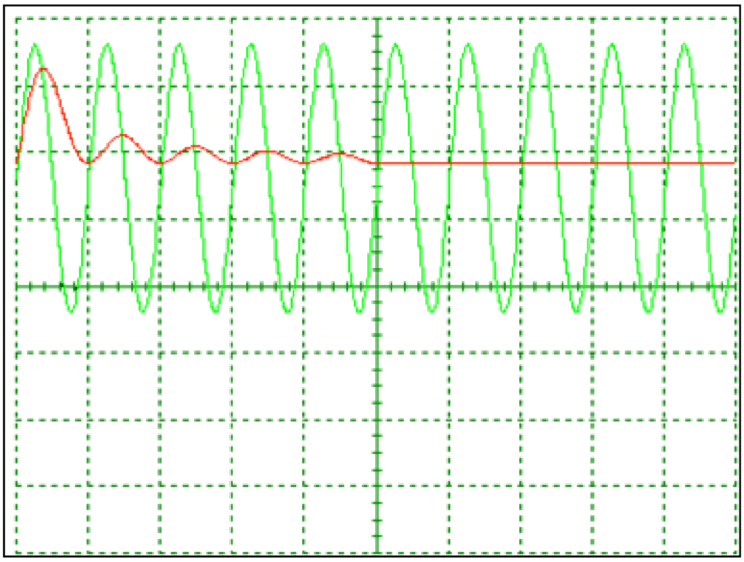


Figure B.3—Enlarged view of average voltage measurement

Figure B.4 shows the VOLTAGE_AV TSF in XML format. The single “Average” BSC may be seen in the model part of the TSF—<Average name="AverageVoltage_4742" …/>. The TSF interface allows the user to specify the gate time and upper and lower limits for the measurement. The PASS/FAIL flags are set according to whether the measured result falls within the specified limits.

```xml
<?xml version="1.0" ?>
<tsf:TSF name="VOLTAGE_AV" . . . . . >
  <tsf:interface>
    <xs:schema
      xmlns:xs=http://www.w3.org/2001/XMLSchema
      elementFormDefault="qualified">
      <xs:element name="VOLTAGE_AV">
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="SignalFunctionType">
              <xs:attribute name="GATED_BY" type="Time">
              </xs:attribute>
              <xs:attribute name="ULIM" type="Voltage">
              </xs:attribute>
              <xs:attribute name="LLIM" type="Voltage">
              </xs:attribute>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:schema>
  </tsf:interface>
  <tsf:model>
    <Signal Out="AverageVoltage_4154" In="In_4153">
      <Average type="Voltage" name="AverageVoltage_4154"
         samples="1" gateTime="GATED_BY"
         UL="ULIM" LL="LLIM" In="In_4153" />
    </Signal>
  </tsf:model>
</tsf:TSF>
```

Figure B.4—XML version of average voltage measurement TSF

Intrinsic measurements will give the correct result on any signal of the appropriate type, i.e. the VOLTAGE_AV TSF will provide the average voltage of any voltage signal.

Annex C

(informative)

Generic measurement

C.1 Introduction

This Annex provides more information about generic measurement and explains how to interpret generic measurement information expressed in XML. It also includes details of all the information required to make such a measurement and what the default values are if specific information is not provided.

generic—adjective, referring to a class or group; not specific.

The Sensor BSC subclass Measure uses the context of the measurement information to determine which measurement method (or methods) is to be used.

C.2 Information required to complete a measurement

Fundamentally, the same five sets of information that were defined for intrinsic measurement are required to completely define a generic measurement, but the information is obtained in a different way. There are two ways of defining generic measurements; by using “lazy” specifications, or by the use of As, with or without attributes.

C.2.1 Lazy specifications

A lazy specification may be identified by the use of the Measure BSC and no reference signal identified by the As attribute.

a) Method (measurement method): is determined from the qualifier of the nominal value if present, or uses the default—Instantaneous.

b) Type (signal type): is determined from the unit of the quantity associated with the nominal value if present, or uses the default—Voltage.

c) When: determined in the same way as for intrinsic measurement

d) Value: determined in the same way as for intrinsic measurement

e) Capability (nominal values): provides direct information for the measurement. The two most important attributes are unit, which determines the type of the measured signal, and qualifier, which determines the measurement method.

C.2.2 Use of As

As is used to identify a signal that is used as a model against which the signal being measured is referenced or compared. This reference signal may be used with or without attributes; if attributes are defined, they define the values to be measured and if no attributes are defined, the error between the input signal and the reference signal is measured.

a) Method (measurement method): determined in the same way as for lazy generic measurement

b) Type (signal type): is determined from the type of the attribute of the As signal. If no attribute is specified the unit type is determined from of the nominal value, and if no nominal value is provided, the default is used—Voltage

c) When: determined in the same way as for intrinsic measurement

d) Value: determined in the same way as for intrinsic measurement

e) Capability (nominal values): provides direct information for the measurement in the same way as for lazy generic measurement.

C.3 Interpreting measurement information

Table C.1 lists all the parts of a measurement where the default value is provided if one is available. This table may be used as a baseline for interpreting any generic measurement by entering the values from the measurement expression into the table and using the defaults for any essential missing data.

Table C.1—Default generic measurement information

Information	Default	Notes
method	Instantaneous	From Nominal qualifier
Signal attributes		
type	Voltage	From type of 'As' signal, unit of Nominal, or default
refType	Time	
measuredVariable	Dependent	
samples	1	
gateTime	0	Default = Implementation dependant
condition	NONE	
nominal	<none>	
UL	<no value>	0 assumed if value required
LL	<no value>	0 assumed if value required
As	<none>	
Measurement Results		Read-only attributes
Measurement	instantaneous voltage	method, type (if samples ≠ 0)
Measurements	n/a	n/a if samples = 1 (default) or 0
Events	1 when measurement occurs	Default for samples = 1 (default)
Output abstract signal	Voltage @ instantaneous level	Abstract signal (at magnitude)
GO	1 if within limits, else 0	
NOGO	0 if within limits, else 1	
HI	1 if > UL, else 0	
LO	1 if < LL, else 0	
Measurement info.		
Expected value	Any	From Nominal (if compatible type)
Min value expected	Any	From Nominal (if compatible type)
Max value expected	Any	From Nominal (if compatible type)
Permitted uncertainty	Any	From Nominal (if compatible type)
Required resolution	Any	From Nominal (if compatible type)
Capability info.		
Characteristic (type)	none provided	From Nominal (if different type)
Observation method	none provided	From Nominal (if different type)
Value	none provided	From Nominal (if different type)
Max value	none provided	From Nominal (if different type)
Uncertainty in value	none provided	From Nominal (if different type)

C.4 Examples showing breakdown of derivation of attributes

The following examples of measurement expressions in XML each have a brief interpretation of their meaning, followed by a table that shows each attribute is derived. If there is no information in the XML expression from which to derive the attribute value, the default is used.

C.4.1 Simple 'lazy' measurement

```
<Measure />
```

Interpretation—Take an instantaneous voltage measurement and create an event when taken (see Table C.2).

Table C.2—Derivation of attribute values for simple 'lazy' measurement

Attribute	Value	Derivation
Signal attributes		
method	Instantaneous	default (no qualifier provided)
type	Voltage	default
refType	Time	default
measuredVariable	Dependent	default
samples	1	default
gateTime	Implementation dependant	default = 0
condition	NONE	default
nominal	<none>	default
UL	<no value>	default 0 assumed if required
LL	<no value>	default 0 assumed if required
As	<none>	default
Read-only attributes		
Measurement	instantaneous voltage	method, type
Measurements	n/a	default (samples = 1)
Events	1 when measurement occurs	default (samples = 1)
Output abstract signal	Voltage @ instantaneous level	default
GO	1 if within limits, else 0	
NOGO	0 if within limits, else 1	
HI	1 if > UL, else 0	
LO	1 if < LL, else 0	
Measurement info.		
Expected value	Any	default
Min value expected	Any	default
Max value expected	Any	default
Permitted uncertainty	Any	default
Required resolution	Any	default
Capability info.		
Characteristic (type)	none provided	
Observation method	none provided	
Value	none provided	
Max value	none provided	
Uncertainty in value	none provided	

C.4.2 Simple 'lazy' measurement with expected value

```
<Measure nominal="1.5 V"  />
```

Interpretation—Take an instantaneous voltage measurement and create an event when taken (see Table C.3).

Table C.3—Derivation of attribute values for measurement with expected value

Attribute	Value	Derivation
Signal attributes		
method	Instantaneous	default (no qualifier provided)
type	Voltage	implied from nominal
refType	Time	default
measuredVariable	Dependent	default
samples	1	default
gateTime	Implementation dependant	default = 0
condition	NONE	default
nominal	1.5 V	nominal
UL	<no value>	default 0 assumed if required
LL	<no value>	default 0 assumed if required
As	<none>	default
Read-only attributes		
Measurement	instantaneous voltage	method, type
Measurements	n/a	default (samples = 1)
Events	1 when measurement occurs	default (samples = 1)
Output abstract signal	Voltage @ instantaneous level	default
GO	1 if within limits, else 0	
NOGO	0 if within limits, else 1	
HI	1 if > UL, else 0	
LO	1 if < LL, else 0	
Measurement info.		
Expected value	1.5 V	nominal
Min value expected	Any	default
Max value expected	Any	default
Permitted uncertainty	Any	default
Required resolution	Any	default
Capability info.		
Characteristic (type)	none provided	
Observation method	none provided	
Value	none provided	
Max value	none provided	
Uncertainty in value	none provided	

C.4.3 Simple 'lazy' monitor

```
<Measure samples="0"  />
```

Interpretation—Output an abstract signal of instantaneous voltages values (see Table C.4).

Table C.4—Derivation of attribute values for simple 'lazy' monitor

Attribute	Value	Derivation
Signal attributes		
type	Voltage	implied from nominal
refType	Time	default
measuredVariable	Dependent	default
samples	0	samples="0"
gateTime	Implementation dependant	default = 0
condition	NONE	default
nominal	<none>	default
UL	<no value>	default 0 assumed if required
LL	<no value>	default 0 assumed if required
As	<none>	default
Read-only attributes		
Measurement	none	samples = 0
Measurements	none	samples = 0
Events	none	samples="0"
Output abstract signal	Voltage @ instantaneous level	default
GO	0	samples = 0
NOGO	0	samples = 0
HI	0	samples = 0
LO	0	samples = 0
Measurement info.		
Expected value	Any	default
Min value expected	Any	default
Max value expected	Any	default
Permitted uncertainty	Any	default
Required resolution	Any	default
Capability info.		
Characteristic (type)	none provided	
Observation method	none provided	
Value	none provided	
Max value	none provided	
Uncertainty in value	none provided	

C.4.4 Generic average current measurement

```
<Measure nominal="av" type="Current"/>
```

Interpretation—Take an average current measurement and create an event when taken (see Table C.5).

Table C.5—Derivation of attribute values for average current measurement

Attribute	Value	Derivation
Signal attributes		
method	Average	qualifier
type	Current	type="Current"
refType	Time	default
measuredVariable	Dependent	default
samples	1	default
gateTime	Implementation dependant	default = 0
condition	NONE	default
nominal	<none>	default

Attribute	Value	Derivation
UL	<no value>	default 0 assumed if required
LL	<no value>	default 0 assumed if required
As	<none>	default
Read-only attributes		
Measurement	average current	method, type
Measurements	n/a	default (samples = 1)
Events	1 when measurement occurs	default (samples = 1)
Output abstract signal	Current @ average level	default
GO	1 if within limits, else 0	
NOGO	0 if within limits, else 1	
HI	1 if > UL, else 0	
LO	1 if < LL, else 0	
Measurement info.		
Expected value	Any	default
Min value expected	Any	default
Max value expected	Any	default
Permitted uncertainty	Any	default
Required resolution	Any	default
Capability info.		
Characteristic (type)	none provided	
Observation method	none provided	
Value	none provided	
Max value	none provided	
Uncertainty in value	none provided	

C.4.5 Error measurement using 'As' signal

```
<Measure As="SpecifiedSignal"  />
```

Interpretation—Measure the error in the voltage of the input signal compared to the voltage of the specified signal and create an event when taken (see Table C.6).

Table C.6—Derivation of attribute values for C.4.5 error measurement using 'As' signal

Attribute	Value	Derivation
Signal attributes		
method	Instantaneous	default (no qualifier provided)
type	Voltage	default
refType	Time	default
measuredVariable	Dependent	default
samples	1	default
gateTime	Implementation dependant	default = 0
condition	NONE	default
nominal	<none>	default
UL	<no value>	default 0 assumed if required
LL	<no value>	default 0 assumed if required
As	SpecifiedSignal	default
Read-only attributes		
Measurement	instantaneous voltage (error in)	method, type
Measurements	n/a	default (samples = 1)
Events	1 when measurement occurs	default (samples = 1)

Attribute	Value	Derivation
Output abstract signal	Error Voltage @ instantaneous level	default
GO	1 if within limits, else 0	
NOGO	0 if within limits, else 1	
HI	1 if > UL, else 0	
LO	1 if < LL, else 0	
Measurement info.		
Expected value	Any	default
Min value expected	Any	default
Max value expected	Any	default
Permitted uncertainty	Any	default
Required resolution	Any	default
Capability info.		
Characteristic (type)	none provided	
Observation method	none provided	
Value	none provided	
Max value	none provided	
Uncertainty in value	none provided	

C.4.6 Measurement of current error using 'As' signal

```
<Measure nominal="trms 3.5 mA" As="SpecifiedSignal"  />
```

Interpretation—Measure the error in the current of the input signal compared to the current of the specified signal. Use the trms intrinsic measurement method (see Table C.7).

Table C.7—Derivation of attribute values for measurement of current error

Attribute	Value	Derivation
Signal attributes		
method	RMS	qualifier
type	Current	implied from nominal
refType	Time	default
measuredVariable	Dependent	default
samples	1	default
gateTime	Implementation dependant	default = 0
condition	NONE	default
nominal	<none>	default
UL	<no value>	default 0 assumed if required
LL	<no value>	default 0 assumed if required
As	SpecifiedSignal	default
Read-only attributes		
Measurement	rms current (error in)	method, type
Measurements	n/a	default (samples = 1)
Events	1 when measurement occurs	default (samples = 1)
Output abstract signal	Error Current @ rms level	default
GO	1 if within limits, else 0	
NOGO	0 if within limits, else 1	
HI	1 if > UL, else 0	
LO	1 if < LL, else 0	
Measurement info.		
Expected value	3.5 mA	nominal

Attribute	Value	Derivation
Min value expected	Any	default
Max value expected	Any	default
Permitted uncertainty	Any	default
Required resolution	Any	default
Capability info.		
Characteristic (type)	none provided	
Observation method	none provided	
Value	none provided	
Max value	none provided	
Uncertainty in value	none provided	

C.4.7 Measurement on an input signal determined by 'As' signal

```
<Sinusoid name="ReferenceSignal" amplitude="0.43 V" frequency="1.2 kHz"
/>
<Measure As="ReferenceSignal" attribute="ReferenceSignal.amplitude"  />
```

Interpretation—The reference signal is a sinusoid with an amplitude of 0.43 V and a frequency of 1.2 kHz. Make a measurement on the input signal equivalent to measuring the amplitude attribute of the reference signal (see Table C.8).

Table C.8—Derivation of attribute values for input signal measurement

Attribute	Value	Derivation
Signal attributes		
method	Instantaneous	referenced attribute
type	Voltage	referenced attribute
refType	Time	default
measuredVariable	Dependent	default
samples	1	default
gateTime	Implementation dependant	default = 0
condition	NONE	default
nominal	<none>	default
UL	<no value>	default 0 assumed if required
LL	<no value>	default 0 assumed if required
As	ReferenceSignal	default
Read-only attributes		
Measurement	instantaneous voltage	method, type
Measurements	n/a	default (samples = 1)
Events	1 when measurement occurs	default (samples = 1)
Output abstract signal	Voltage@ instantaneous level	default
GO	1 if within limits, else 0	
NOGO	0 if within limits, else 1	
HI	1 if > UL, else 0	
LO	1 if < LL, else 0	
Measurement info.		
Expected value	0.43 V	referenced attribute
Min value expected	Any	default
Max value expected	Any	default
Permitted uncertainty	Any	default
Required resolution	Any	default

Attribute	Value	Derivation
Capability info.		
Characteristic (type)	Frequency	referenced attribute
Observation method		
Value	1.2 kHz	referenced attribute
Max value	none provided	
Uncertainty in value	none provided	

C.5 Creating IEEE 1641 measurements for legacy ATLAS systems

C.5.1 Introduction

Annex B includes an explanation of how a series of measurements used by a legacy ATLAS system can be defined using IEEE 1641. The list included both intrinsic and generic measurements (see B.5.2).

In all the generic measurements listed in Table B.9, the signal attribute (ATLAS noun modifier) to be measured is described as part of an example signal (ATLAS noun) and is very dependant upon the signal defined. This requires the signal to be accurately modeled and the measured attribute is identified as an attribute of that signal. If the incoming signal (i.e. the signal being assessed) is the same as the defined signal, the resultant measurement will be accurate. If the incoming signal is completely different to the expected signal, the incoming signal will be Measured As if it is of the expected type and the result will be a measure of the best fit to the expected signal.

C.5.2 Example IEEE 1641 equivalent to ATLAS AM SIGNAL measurement

The MOD_AMPL_AM TSF describes a complete AM Signal and asks for the amplitude of the modulating signal that would have given rise to the incoming AM signal. This is illustrated in Figure C.1 where the second trace shows the AM Signal that is expected, as defined in the TSF model using the values entered into the TSF panel. The top trace shows the actual incoming AM Signal, which has a deeper modulation. This indicates that the amplitude of the actual modulating signal (third trace) is higher than the expected modulating signal (lower trace).

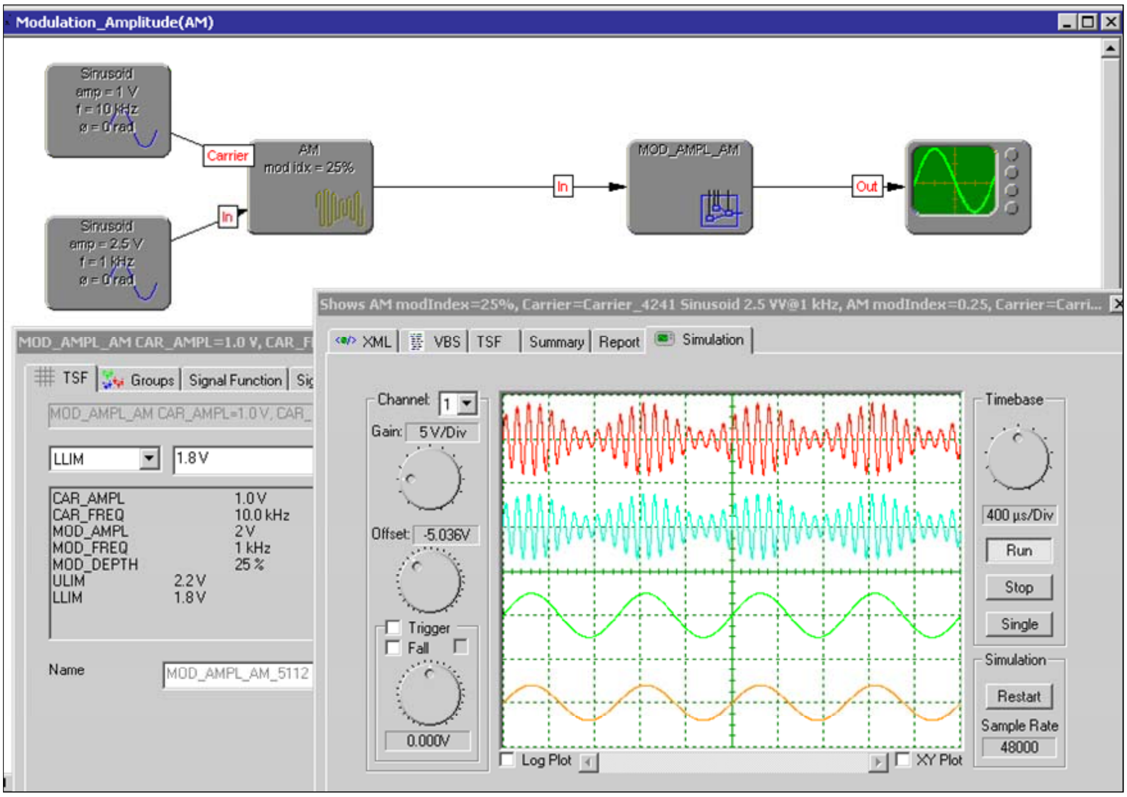


Figure C.1—Generic measurement of an AM signal

The operation of the TSF may be seen from the model part of the XML definition, which is shown in Figure C.2. The key parts are in the reference to the Measure BSC. Measure_4146 is the Measure BSC and the attribute being measured is the amplitude (.amplitude) of the modulating signal. The As refers to AM_4235 which is the definition of the expected AM Signal.

```
<tsf:TSF name="MOD_AMP_AM" . . . . >
 <tsf:model>
  <Signal Out="Measure_4146" In="In_4156">
   <Sinusoid name="Carrier_4241"
    amplitude="CAR_AMPL"
    frequency="CAR_FREQ"/>
   <Sinusoid name="Modulation_4237"
    amplitude="MOD_AMPL"
    frequency="MOD_FREQ"/>
```

Figure C.2—Model segment of TSF for measuring modulation amplitude

C.5.3 Example IEEE 1641 equivalent to ATLAS DISTORTION measurement

This DISTORTION_AC_SIGNAL TSF will determine the total distortion in an AC Signal. It shows the effect of not specifying an attribute in the 'As' Measure BSC field and is illustrated in Figure C.3. This TSF assesses the overall difference between the specified signal and the incoming signal. The expected (specified) signal is a sinusoid of amplitude 1 V, with a frequency of 1 kHz and an initial phase angle of -45 deg (the sinusoidal trace). The incoming signal is a distorted sine wave with the same fundamental frequency (trace with flattened peaks). This TSF shows the total distortion in the signal in the lower trace.

The source and comparison signals are offset on the screen so that the shape and amplitude of the distortion may be easily seen.

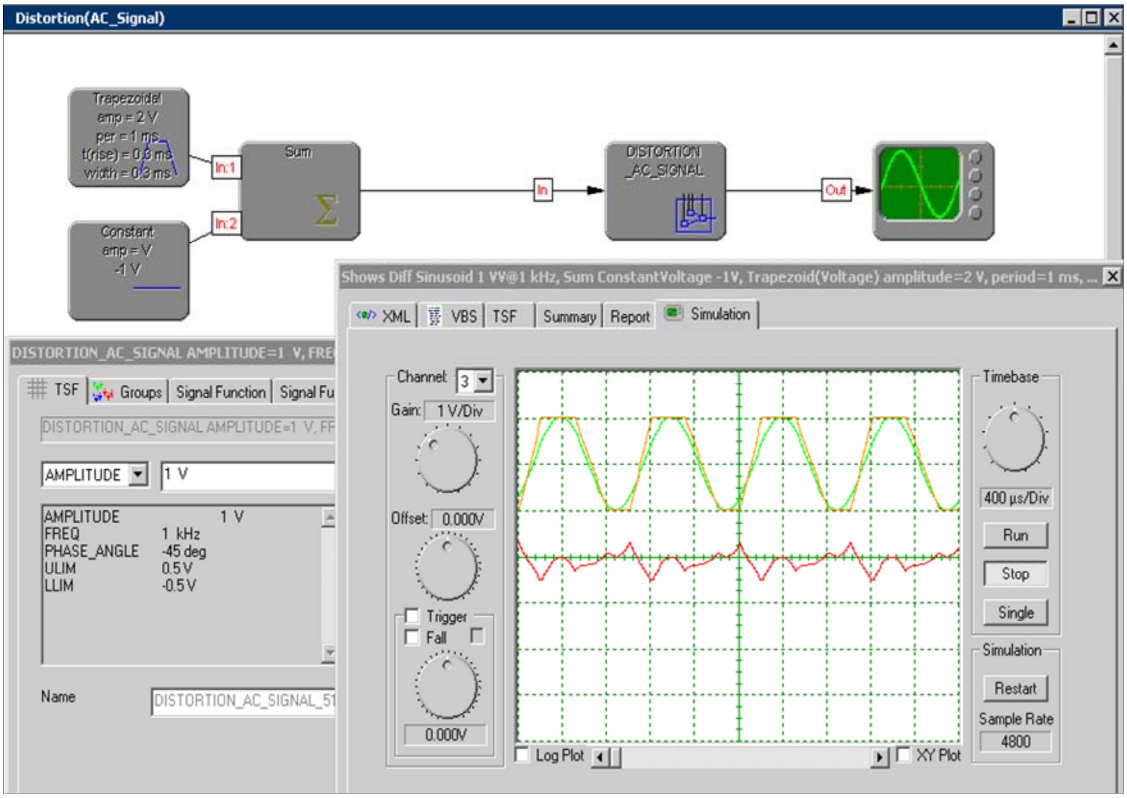


Figure C.3—Measurement showing difference between expected and actual signal

As with intrinsic measurement, this model describes the measurement that must be made and not the instrument that may be used to perform the measurement. The instrumentation used to perform the measurement will have to be selected to be suitable for the type of signal being assessed as well as the attribute.

C.6 Dependence upon reference signal

Generic measurement depends upon the reference signal specified. This point can be illustrated by a few examples in which the amplitude of the signal is measured. The Measure subclass is defined in Annex B of IEEE Std 1641-20102010

a) Example 1—Measure the amplitude of a sinusoid signal (see Figure C.4).

```
<Measure As="Sinusoid" attribute='amplitude' In=.../>
```

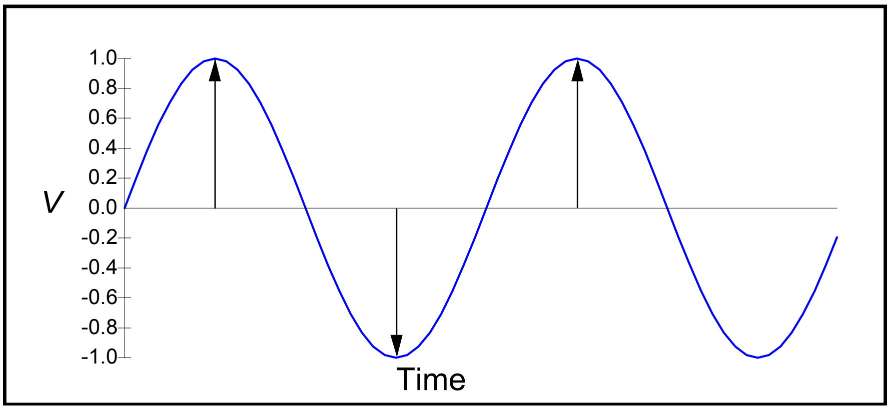


Figure C.4—Amplitude of sinusoid signal

b) Example 2—Measure the amplitude of a square wave signal (see Figure C.5).

```
<Measure As="SquareWave" attribute='amplitude' In=.../>
```

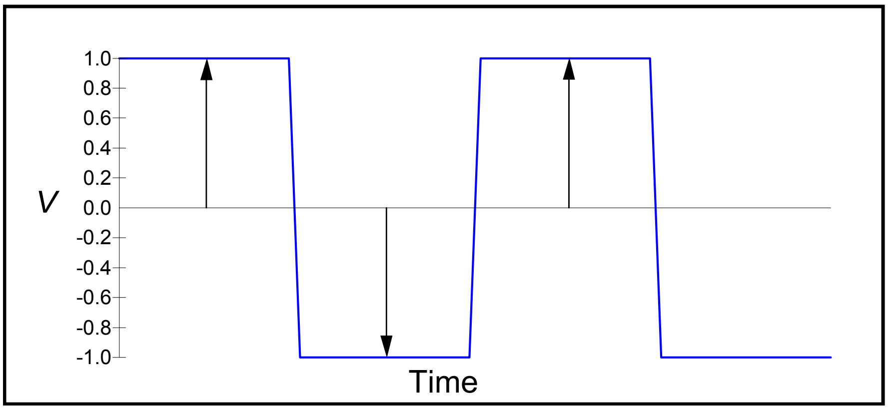


Figure C.5—Amplitude of square wave signal

c) Example 3—Measure the amplitude of a trapezoid signal (see Figure C.6).

```
<Measure As="Trapezoid" attribute='amplitude' In=.../>
```

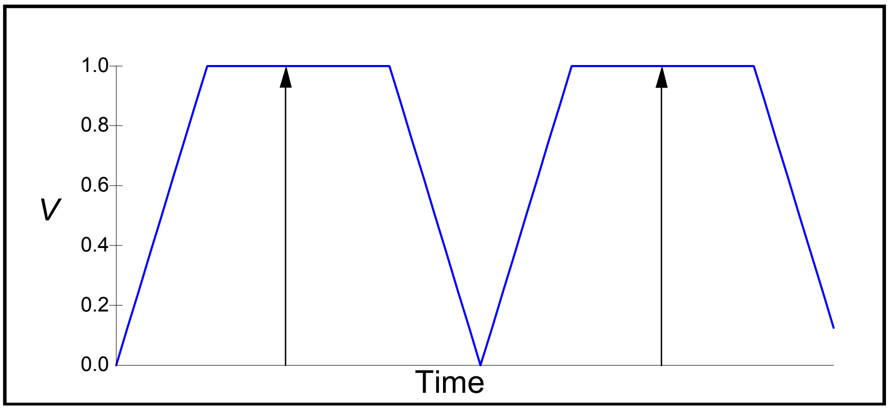


Figure C.6—Amplitude of trapezoid signal

The examples show that the term amplitude means different things depending on the context of the reference signal being specified. It does not refer to waveform being measured. It is important to measure the attribute actually required.

C.6.1 Obtaining best match and least error

When the waveform being measured does not match the reference signal, what should be measured? For example, if the input signal is a trapezoid but a square wave had been expected, what should be measured? The STD standard identifies the result that should be obtained, i.e., the value of the attribute that yields the minimum error between the input waveform and reference signal. By definition, this value is the same value that yields the minimum standard deviation, or the minimum rms difference.

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

The minimum rms value will be when the peak-to-peak amplitude of the square wave and the peak amplitude of the trapezoid are the same! This situation can be shown graphically (see Figure C.7). It can be seen that measuring the trapezoid (upper trace) as if it is a square wave will give the same value as measuring the equivalent square wave (lower trace). The value of a square wave amplitude is half the value of the trapezoid amplitude. Any change of either amplitude will create a greater error value. This result has a common-sense appeal.

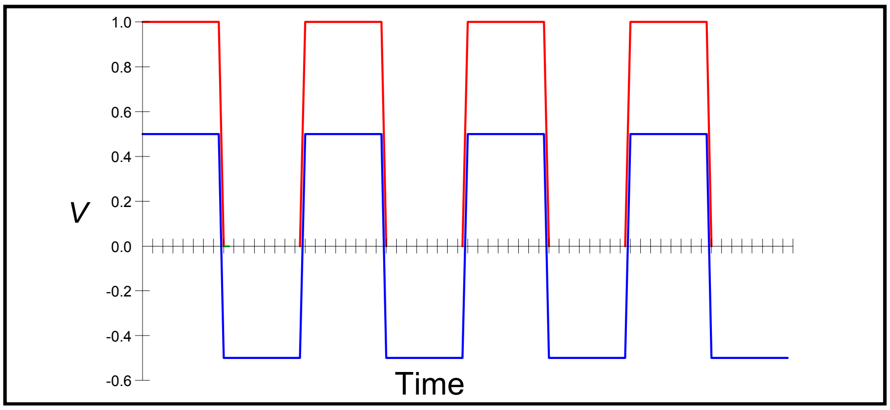


Figure C.7—Measuring trapezoid signal as square wave

The STD standard identifies what the measurement value means when the input waveform contains imperfections from the “perfect signal.” The rationale is that if it was known exactly what the input signal was, the input signal could have been used as the reference signal.

This situation is illustrated in Figure C.8, where the signal being measured is assumed to be a trapezoid signal (the reference signal). If the input signal is actually exponential in shape, it is difficult to see what the amplitude (of the equivalent trapezoid) actually is.

```
<Measure As=”Trapezoid” nominal='trms' attribute="amplitude"In=.../>
```

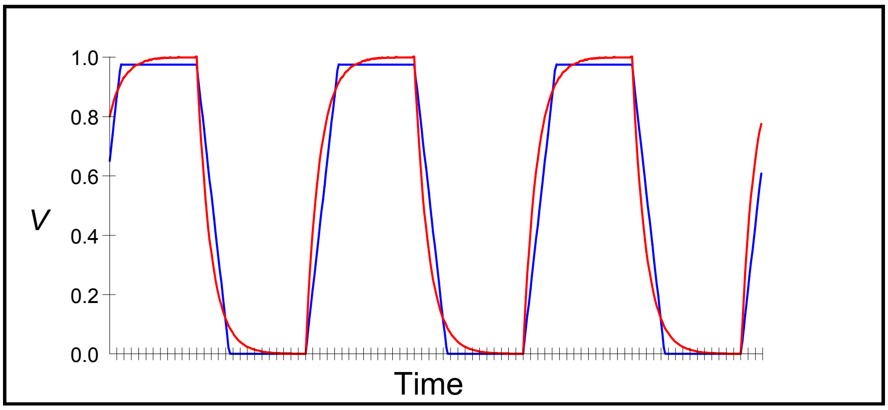


Figure C.8—Measuring exponential signal as trapezoid

The result returned is the amplitude of the trapezoid signal that best matches the exponential signal received, i.e., the trapezoid signal that gives the least error.

C.6.2 Measuring waveform aberration

The generic measure selects the reference waveform described by the reference signal that contains the least error (best match) with respect to the input waveform. The attributes of the reference signal that produce the least error become the measured attributes.

The generic measure can also be used to measure the actual waveform aberration by measuring the error (difference) between this reference waveform and the input waveform. Measuring the error is achieved by using the generic measure with a defined reference signal but not specifying any measure attributes. The difference or error between the best-match reference waveform and input waveform is returned.

For example, to measure the best-match value or rms error between the input signal and reference signal (see Figure C.8), use the following statement:

```
<Measure As=”Trapezoid” nominal=’trms’ In=.../>
```

By not identifying any particular attribute, the best match is determined, but the result of the measurement is the value of the error, i.e., the difference between the reference waveform and the input waveform.

This technique may also be used to collect a set of points representing the difference between the reference and input waveforms:

```
<Measure As=”Trapezoid” samples="1000" In=…/>
```

This technique may be used to show the instantaneous error aberration between the two waveforms in Figure C.8 graphically (see Figure C.9).

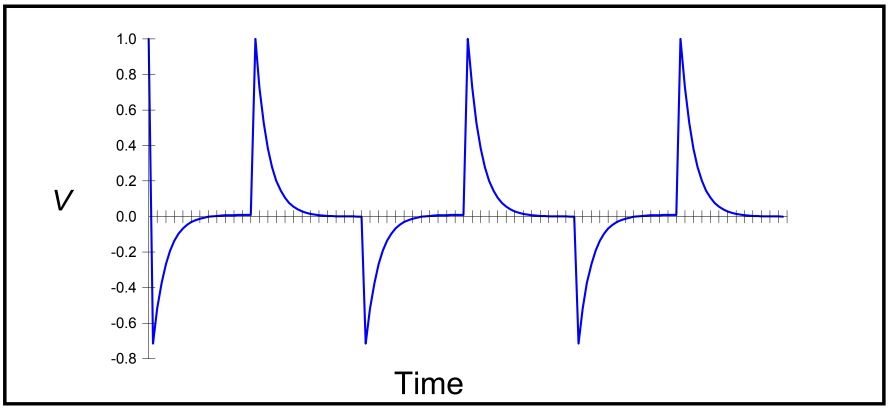


Figure C.9—Generic measurement of instantaneous error aberration

C.7 Measurements on a complex signal (square wave)

Various techniques provide the building blocks from which individual measurements can be defined or waveform points for analysis can be captured.

To put these techniques into context, consider a square wave example where the waveform to be measured has both ringing and droop. This situation is illustrated in Figure C.10.

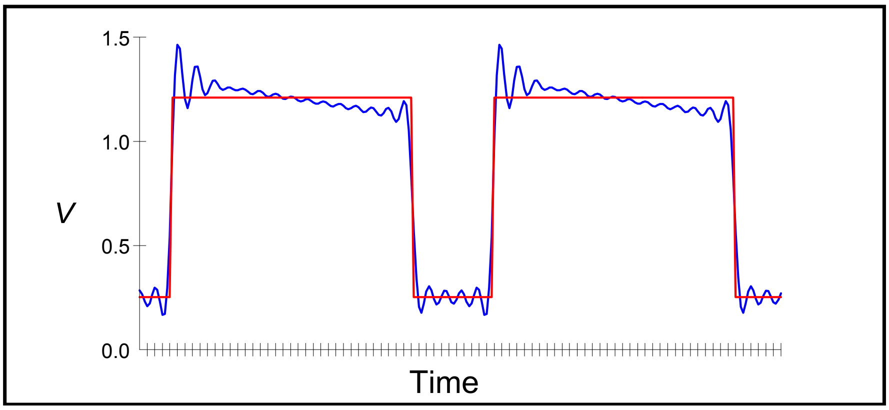


Figure C.10—Measurement of complex square wave

To measure the input waveform as if it is a newTsf:SQUARE_WAVE,[9] generic measurement is used. This technique gives rise to the values shown in Table C.9.

Table C.9—Generic measurements on square wave

Attribute	Name	Measurement value	XML expression
Square wave amplitude	ampl	0.478 24 V	<Measure As="SQUARE_WAVE" attribute="ampl" />
Square wave period	period	1 s	<Measure As="SQUARE_WAVE" attribute="period" />
DC offset	dc_offset	0.730 98 V	<Measure As="SQUARE_WAVE" attribute="dc_offset" />
Duty cycle	duty_cycle	75%	<Measure As="SQUARE_WAVE" attribute="duty_cycle" />
Waveform aberration		trms 0.084 67 V	<Measure As="SQUARE_WAVE" nominal="trms" />

Figure C.11 shows the error aberration for the square wave example together with the rms error.

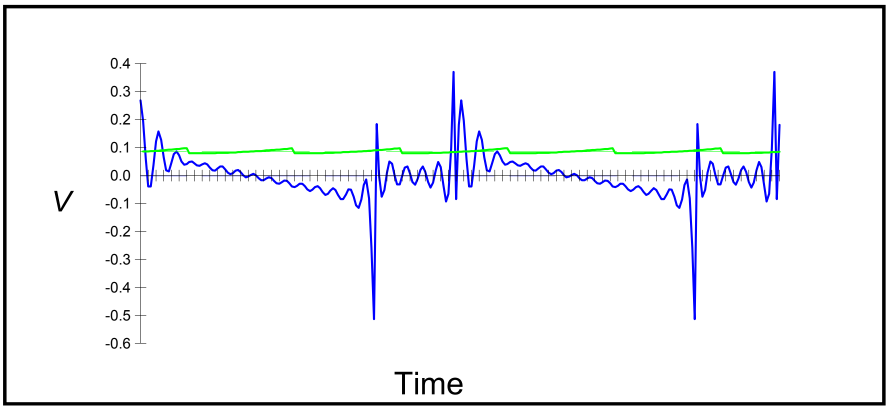


Figure C.11—Waveform aberration for complex square wave

[9] The tsf:SQUARE_WAVE in Annex E of the STD standard does not export the duty_cycle attribute. The new Tsf:SQUARE_ WAVE used in this example exports that attribute but is otherwise identical.

Annex D

(informative)

The role of Resource Adapter Information (RAI) in IEEE 1641

D.1 Introduction

This annex describes the role of the rules for maximizing the platform independence of test requirements using the rules defined in Annex K of the IEEE Std 1641. The effort to define these rules started with an attempt to define an interface between the UUT and the test equipment that would be entirely independent of the test system to be used., i.e. a Resource Adapter Interface. This led to the abbreviation RAI, which became the popular and accepted reference for this effort. It was soon realized that IEEE 1641 had all the ingredients required to provide the information required to allow multiple implementations, using different test executives, all using the same test information. Thus RAI became 'Resource Adapter Information' and provided a set of rules to ensure that any compliant test requirement will be truly independent of the test resources used to implement the tests.

RAI augments both IEEE 1641 and IEEE 1671 (ATML) standards by defining what information shall be present when using these formats. To this end, RAI identifies how test requirements shall be captured using IEEE-1641 signals, and any constraints that shall be imposed to enable portability of such signal definitions across independent ATS platforms. What it does not do is seek to standardize or control how tests are implemented on individual platforms. RAI identifies what should be present, but also allows for different initiatives and approaches to provide a test solution. The effect of which is that a requirement using RAI rules could be solved by a runtime application programming interface API, a development time compiler style solution, or any combination thereof.

D.2 Maximizing the platform independence of test requirements

IEEE 1641 provides a set of guiding principles together with a set of best practice rules that impose constraints on how to describe test requirements within a test program and against which these test requirements may be audited. An XML schema that encapsulates these rules may be created that allows test requirements to be validated. .

D.2.1 Guiding Principles

The following guiding principles should be adhered to when describing test requirements for implementation across multiple platforms:

— Test requirement information shall be complete and explicit.

— Test requirement information shall not be described in a manner that relates how a test platform will accomplish a requirement.

— Test requirement information shall not be described in a manner that relates knowledge of test system implementation.

D.2.2 Best Practice Rules

— Test Requirements shall use static signal definitions.

— Test Requirements shall only contain the UUT signal location.

— Test Requirements shall identify any signal synchronization required.

— Test Requirements shall define any signal triggering with relationship to the UUT events.

— Test Requirements shall identify any prerequisite signals required.

— Test Requirements shall include any environmental characteristics required.

D.3 Interpreting the principles and rules

Each of these rules has some further commentary in the STD standard to explain how to interpret and implement them. The key to successful implementation is to assume nothing! For example, many traditional test programs assume a natural sequencing of signals and measurements and rely upon implicit test system time factors to ensure that signals are applied and measurements are made correctly. RAI does not allow this. If there is any time restraint whatsoever as to when a signal may be applied or measured, then this must be defined explicitly.

D.3.1 Timing and synchronization

On the whole, legacy test equipment ran much more slowly than modern test systems. Test requirements that were well written using ATLAS that were intended to be portable, often unintentionally used the original test equipment's operation to pace the test sequences. When porting these programs to more modern equipment, problems often occur because the sequences run faster than the UUT can accept them. To create an RAI compliant test requirement, assume that the test equipment runs infinitely fast, i.e. with zero delay between signal & measurement steps. In this way the rest requirement will always be portable in the future, even when new (much faster) systems are used.

This does not mean that each time delay has to be defined to an impossible accuracy. Provide a time delay that fits the requirements of the UUT (not the test system) together with a reasonable tolerance that allows for both current and future test equipment.

D.3.2 Connections

Every signal and measurement must be referenced to a UUT connection, either directly or indirectly. This avoids making any assumptions as to how a test system may be interconnected or how it may be connected to the UUT. A future implementer will only have the UUT connections to refer to, and any other connection references will be confusing as well as invalid.

D.3.3 Test environment

Environmental conditions should also be included. Programs written to run within a "normal repair shop" environment may mean something different to another user with different test equipment. Providing a temperature and humidity range will always be more easily understood.

D.3.4 Prerequisite signals

Many test requirements have been traditionally written to run in a fixed sequence, so that even if the fault area was known, it is necessary to run a full test program to get to the relevant checks. This can be avoided by defining tests as stand alone tests. Stand alone tests are much more efficient if tests are run in a sequence determined by diagnostic requirements. In this case, prerequisite signals must be defined for each test. This does not mean that a final check of a UUT will not have to be run, in which case the most efficient test sequence order should be given such that many tests will already have the prerequisite signals in place.

D.4 Test requirement presentation

IEEE 1641 uses XML as the definitive medium for presenting test requirements. This has several advantages, including the fact that the code may be validated as already stated; ATML Test Description (IEEE 1671.1 [B11]) also uses XML and supports the use of IEEE 1641 for signal definition. Therefore, test requirements written using IEEE 1641 signals, within ATML test description tests, and compliant with RAI rules, provide the highest probability of being fully portable well into the future on any test equipment with the appropriate facilities.

Annex E

(informative)

Understanding IEEE 1641 capabilities

E.1 How to define and connect Loads

E.1.1 Introduction

There are two completely different kinds of load within IEEE 1641. One is a BSC and the other is a property of the Physical class. In addition, it is possible to define a signal of type Impedance, which adds a third possibility when a load is required.

The Load BSC provides an impedance, defined in terms of resistance and reactance, which be used to load a signal.

The load property represents the impedance, input or output, that is used when converting power, voltage, and current values as is often required in radio frequency (RF) applications.

This section explains how and when to use the Load BSC, how and when the load property should be invoked and when an Impedance signal is required. It also covers how to represent short circuits in IEEE 1641.

E.1.1.1 The Load BSC

The Load BSC is always used in conjunction with a connector. It is used to indicate that a load is required for the signal or UUT to operate correctly, i.e. it is information to the test implementer that a load is required at that point.

It is important to note that Load does not modify the signal (in the simulation sense) even though the signal may not be present or may not be correct if the load is not there. Although the Load is a Transformation BSC, it has a unity transfer function, i.e. the signal is not changed as it passes through the Load.

It may help to state where Load is not to be used in IEEE 1641—it is not an impedance element to be used for constructing networks.

E.1.1.2 The load property

The load property may be used with a signal of type Physical, e.g., with a signal of type Voltage. In RF applications, it is common to express an input (or output) signal with the unit symbols dBm or dBW. In most cases it is a voltage or current signal that is actually required, but it is expressed as a power with reference to an input or output impedance. The load property allows the value of that impedance to be expressed.

E.1.1.3 A signal of type Impedance

It is possible to invoke a signal of type Impedance. For example a Constant BSC may be used as a source with type Impedance and amplitude in ohms. Theoretically, any source BSC may be of type Impedance, but in nearly every case the Constant BSC will be the most suitable.

E.1.2 Engineering use of loads in test

The simplest way to explain the use of loads is with a set of examples or use cases.

Quite often the use of the impedance element is there to provide information to the engineer implementing the test and is not directly related to the signal itself.

There are several different ways in which a load is used with UUTs when testing.

E.1.2.1 Case 1—Load as in input to a UUT

The UUT requires an impedance element to be connected to two specified pins (see Figure E.1). No signal is to be applied and no measurements are to be made. This is just an element that the UUT needs in order to operate correctly.

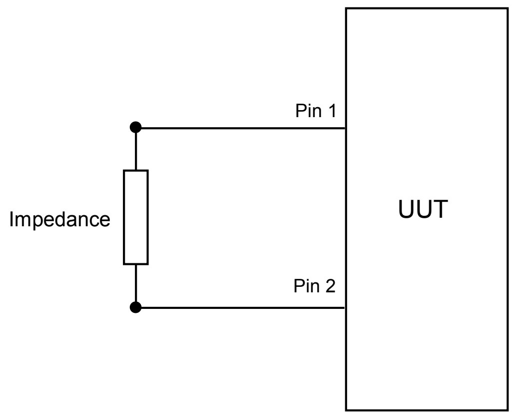


Figure E.1—Load impedance connected to a UUT

In the engineering sense the UUT is the source of an (unknown) signal which is being output into the load. What is actually happening is that a "signal" is being applied to the load by the UUT (see Figure E.2). This is true even if the impedance element is connected to an input of the UUT. An example is shown in the modified diagram below.

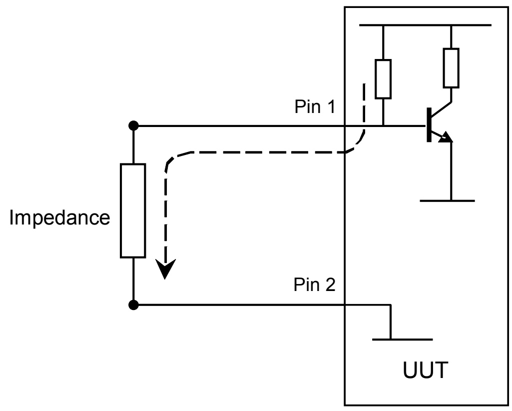


Figure E.2—Load effectively connected to output connections

This could be represented by a Load BSC connected as in case 2 below, but it would be better to consider that the UUT requires the application of a signal, i.e. a Constant BSC with physical type Impedance. This remains true even for the special case of a (shorting) link, where the impedance is zero, although in the case of a simple link the two pins could just be connected together.

E.1.2.2 Case 2—Load as an output to a UUT

Consider a UUT in which the output requires a load to be applied to two pins (see Figure E.3). A measurement may be taken of (say) the voltage at those UUT pins. The load has to be there to ensure the UUT is operating correctly. The measurement is taken across the UUT pins. These two may be considered two separate unrelated actions (even though the lack of the load may result in an incorrect measured value).

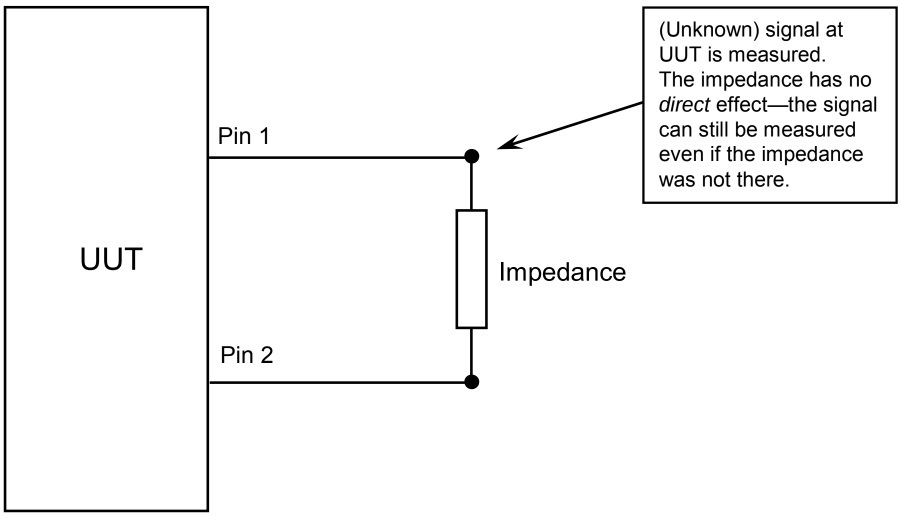


Figure E.3—Load connected to two output pins

In practice, the lack of the impedance may affect the value of the signal measured, but when the impedance is present it doesn't directly modify the signal.

A common example is where it is used to indicate that the UUT requires a matching impedance, such as a measuring instrument with an input impedance of 50 Ω. Note that as the measuring instrument is physically adding a load to the output pins, a Load BSC is required to represent that load. If the measuring instrument also translates between voltage and power values, the load property will also be utilized (see Use of the load property—Summary).

Although the Load BSC does not modify the signal, it is still a Transformation BSC and applies to the whole signal. It should not be thought of as a resistor with two terminals but as a complete load applied to a signal, however many terminals it has. This is illustrated in Figure E.4 and Figure E.5.

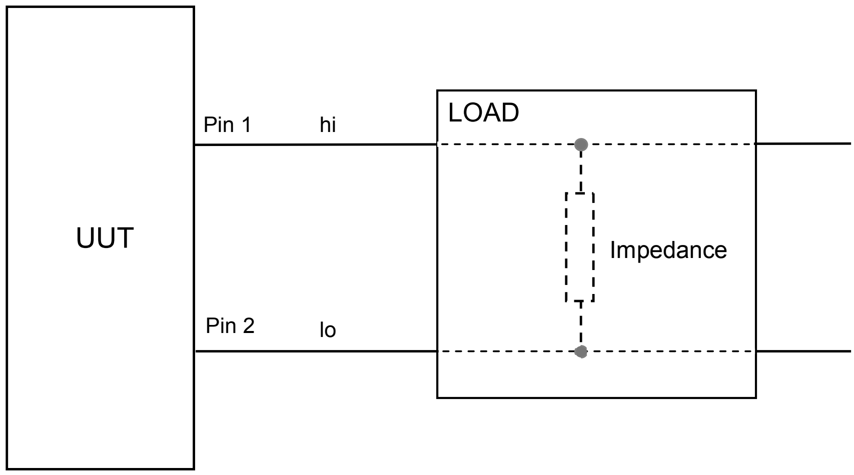


Figure E.4—Load as a transformation BSC with output port

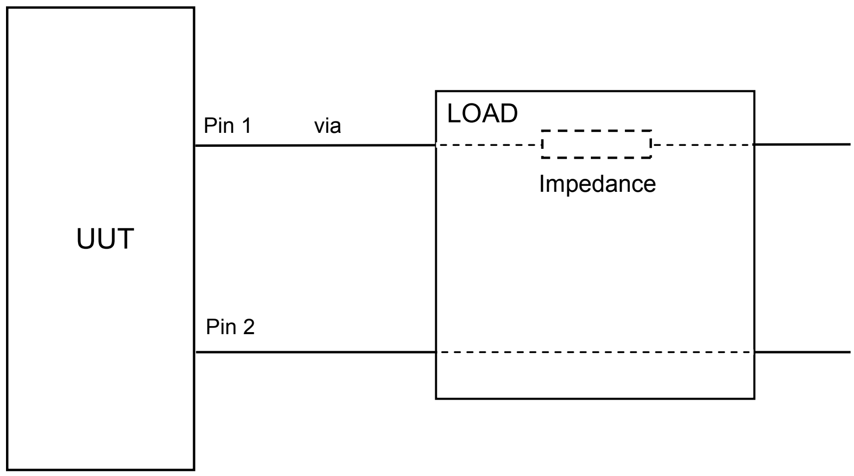


Figure E.5—Load used with a serial port (Via BSC)

E.1.2.3 Case 3—Load attached to an input to a UUT (parallel signal)

In this example a load is connected to two input pins on the UUT and a signal is applied to those same input pins (see Figure E.6). Again, these two may be considered two separate unrelated actions. The lack of this element should not result in an incorrect input value when the signal applied is defined independently of the total load.

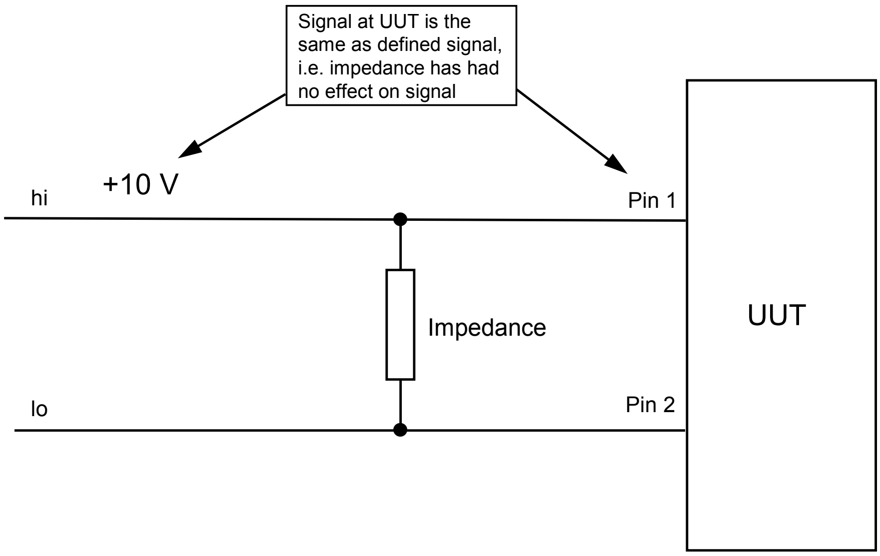


Figure E.6—Parallel Load on the input to a UUT

As an example, consider the situation where the impedance has to be there to ensure that the correct operation of the UUT or to prevent the input floating to an unsatisfactory value when the signal is inactive.

E.1.2.4 Case 4—Load attached to an input to a UUT (series signal)

This is similar to case 3 (where the signal is not affected by the impedance element) but for a current signal. The whole of the current flows through both the element and the UUT, i.e. the current is not changed (see Figure E.7).

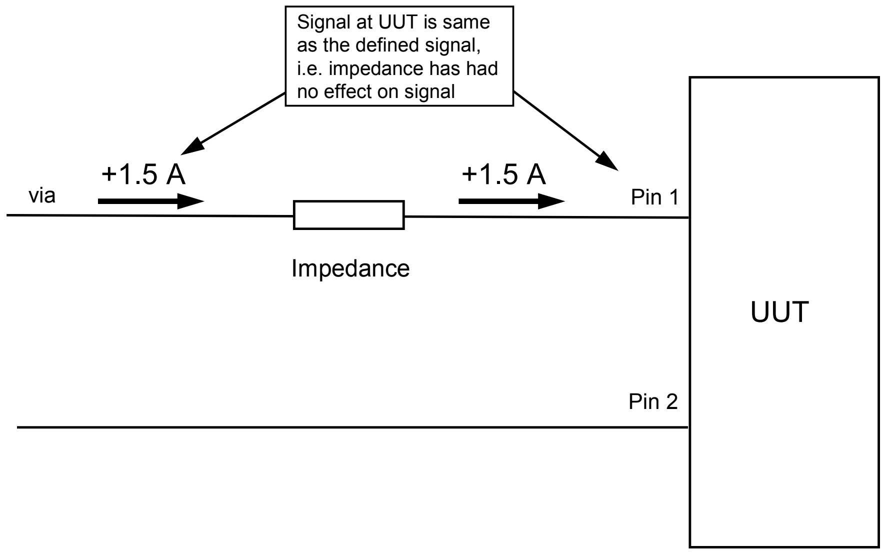


Figure E.7—Serial Load on the input to a UUT

E.1.2.5 Case 5—Load attached to an input to a UUT (parallel network element)

In this case the signal applied is affected by the load. An example is where the signal is defined in terms of current and part of the current will flow through the element and part through the UUT (see Figure E.8).

This is not a correct use of Load, as it is being used as a network element and the value of current flowing in the UUT (x amperes) is not known.

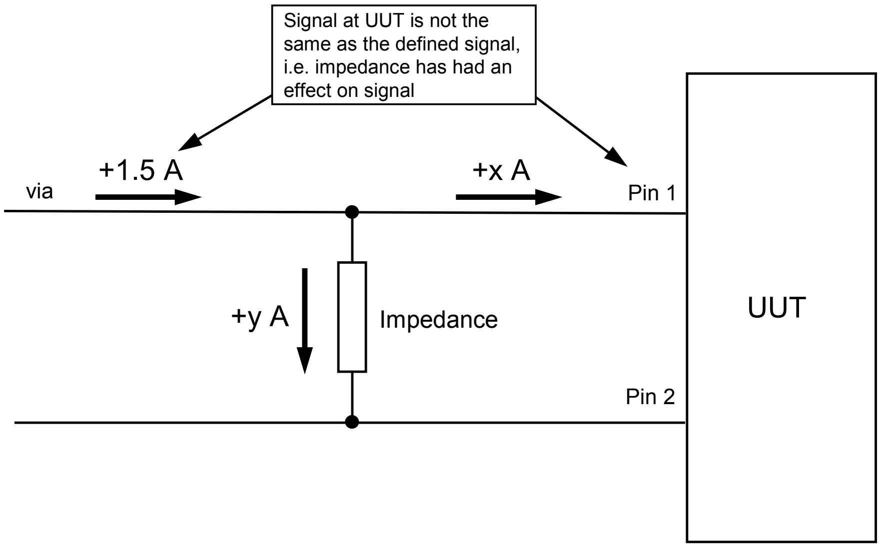


Figure E.8—Load as a parallel network element

In this situation, a completely different model may need to be used according to the specific situation. For example, if the current into the UUT (x A) is much lower than the bypass current (y A) then the model may represent a voltage source which is programmed as a current. In such cases, the signal being applied to the UUT is a function of the initial signal and the impedance, which should be dealt with by creating a TSF with the appropriate function code in the TSF. This situation is likely to arise quite infrequently and each occurrence should be examined on its own merits.

E.1.2.6 Case 6—Load attached to an input to a UUT (series network element)

This is another example similar to case 5; a load is used in series in between the applied signal and the UUT (see Figure E.9). In this case the load element should be considered part of the source signal TSF or as part of the UUT, depending on the specific circumstances. A Load BSC is not appropriate as it is being used as a network element and the voltage at the UUT (x volts) is not known.

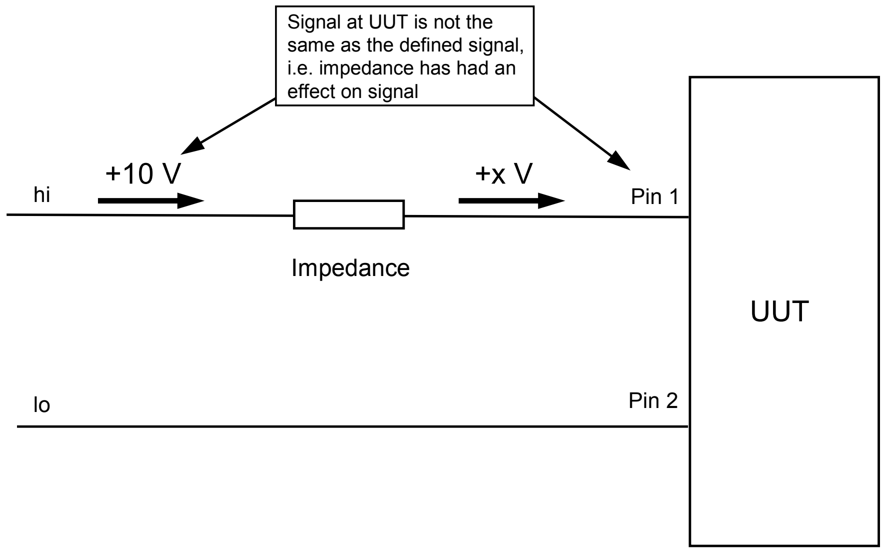


Figure E.9—Load as a serial network element

E.1.2.7 Instruments with significant output impedance

An example of case 6 is where the output impedance of a source instrument in significant. In many RF applications a UUT will have an input impedance of 50 Ω and the source instrument is assumed to have the appropriate output impedance. What is actually happening in this case is that the instrument sources a current, which is defined as a voltage translated according to the source instrument output impedance. These output or input impedances cannot be represented with a Load BSC.

A completely different model is required here, where a source current is programmed by a voltage divided by the source's output impedance. This conversion will be done mathematically in the source TSF. The effect of an incorrect UUT input impedance is then modeled correctly. In most cases, it is reasonable to assume that the UUT input impedance is correct (and is matched by the correct instrument output impedance) and that the effect of these impedances may be ignored.

Conversions between voltage and power will be done by invoking the load property of the physical signal.

E.1.2.8 Summary of differences between cases 1 to 6

Cases 2, 3 and 4 are similar in that the impedance does not directly affect the signal. In these cases the impedance has nothing to do with the signal definition, either for source or sensor. The impedance is only relevant to the requirements of the UUT. Therefore, in these cases a Load BSC may be used.

In case 1 the impedance itself may be considered to be the signal being applied. Therefore, in this case a Constant BSC of type Impedance may be used.

In cases 5 and 6 the impedance does affect the signal at the UUT directly. The defined signal is not applied to the UUT; i.e. the impedance has modified the signal before it is applied to the UUT. In these cases the Load BSC cannot be used as 1641 does not support the use of Loads to build networks. The solution is to examine the case in detail and to create a TSF that outputs the required signal using the appropriate function code in the TSF

E.1.3 Different kinds of load

The examples shown in E.1.2 are all related to the "TwoWire" use of a load. In engineering terms we use many special configurations of loads. These include three phase loads of different types (which may be balanced or unbalanced), plus other configurations for synchros, resolvers and busses. Several of these are illustrated below. Many of them are either the same as the two wire example, or multiple instances of the two wire example.

a) SinglePhase

Figure E.10 shows a Load BSC connected to a SinglePhase connector. This is the same as for TwoWire and TwoWireComp connectors. This is the simplest case with a single impedance of the value stated in the BSC attributes.

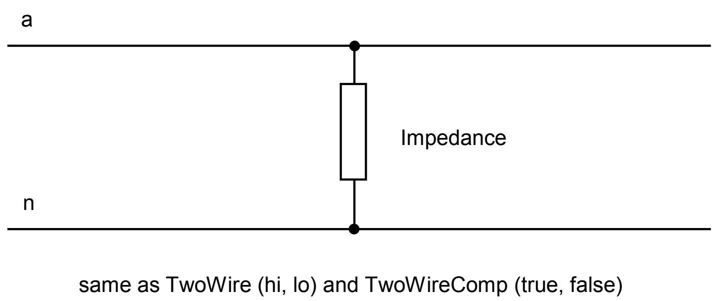


Figure E.10—Load BSC on a two-wire connection

b) TwoPhase

This example shows the case where a single Load BSC is connected to a TwoPhase connector (see Figure E.11). The Load BSC attributes for the impedance value (Resistance and Reactance) are the same for both impedance elements.

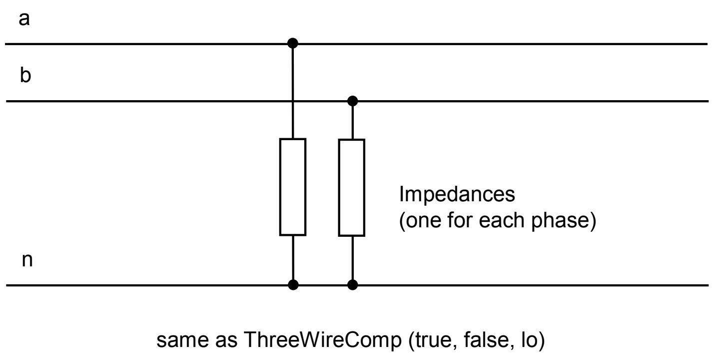


Figure E.11—Load BSC on a three-wire connection

c) ThreePhaseWye

This example is very similar to the TwoPhase example above except that there are three identical impedance elements (see Figure E.12).

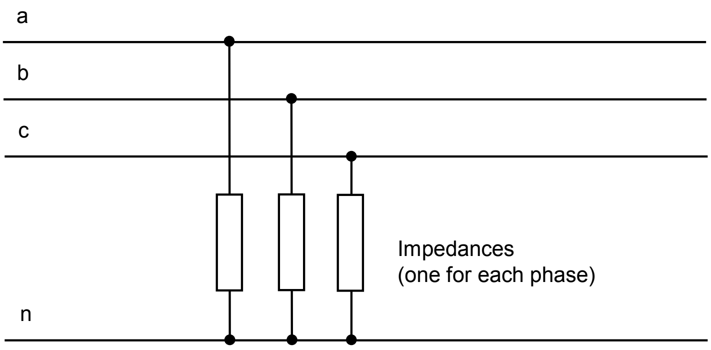


Figure E.12—Load BSC on a three phase wye connection

d) ThreePhaseWye (cont.)

Figure E.13 shows a different view of the ThreePhaseWye connection but in the more familiar "Y" configuration.

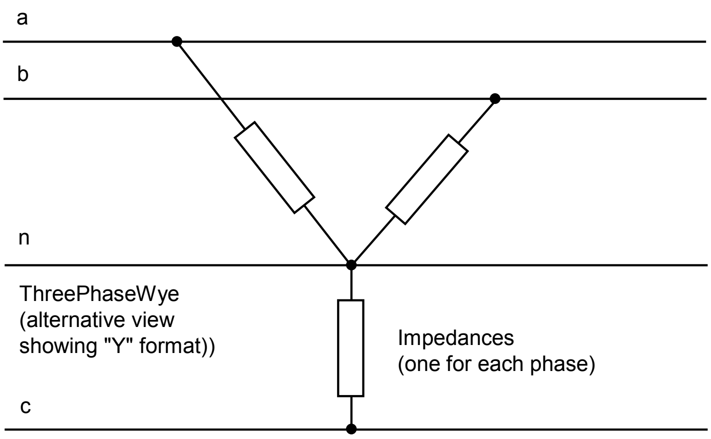


Figure E.13—Load BSC on a three phase wye connection (alternative view)

e) ThreePhaseDelta

In this example the Load BSC is connected to a ThreePhaseDelta connection (see Figure E.14). Again, the three impedance elements all have the same Resistance and Reactance values.

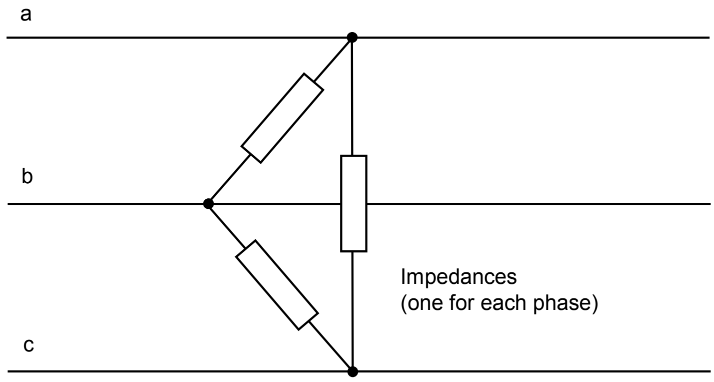


Figure E.14—Load BSC on a three phase delta connection

f) DigitalBus

The example of a Load BSC connected to a DigitalBus (see Figure E.15) shows multiple impedances, all of the same value. In this case the number of impedance elements will be the same as the number of digital channels as specified by the channelWidth attribute.

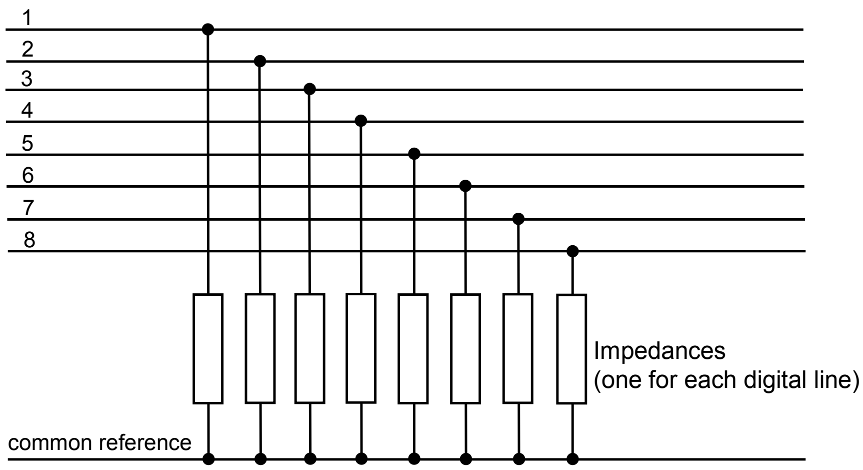


Figure E.15—Load BSC on a parallel digital connection

The above examples are treated as multiple versions of the cases given in the first section. This means that the Load BSC is suited for use where the impedances are used in the same way as illustrated in cases 2, 3 and 4 in E.1.2. All the impedances have the same value, which is that given to the instance of the Load BSC. The values of the impedances may well be the same, but this is not always the case in practice. For example, the three phase loads may be unbalanced, where the individual impedances have different values

In many cases, it would not matter if the impedance values were the same or different as the value has no direct effect on the signal. The problem is that there is no way of expressing different values in a Load BSC, as only one value may be given. When it is required to represent different values for multiple impedances in a Load, this is achieved by separating the signal into its component channels and using individual TwoWire Loads for each channel.

E.1.4 Summary of the uses of the Load BSC, the Load property, and type Impedance

E.1.4.1 Use of the Load BSC

a) The Load BSC (as defined by the SML) has no effect on a signal.

b) It may not be used as a network element.

c) It is used in association with a connector BSC listing the UUT pins. In simple terms, the Load is usually the first BSC immediately after the connector which lists the UUT pins.

d) It indicates that a Load is required to be connected to ensure the correct operation of a signal (1641 is a signal definition standard).

e) A Load BSC can only have a single impedance value, which applies to all impedances in a Load.

f) If different impedance values are required, separate model into multiple two wire loads.

g) If Load does not seem to fit the application, a different model entirely may be required.

E.1.4.2 Use of a source BSC of type Impedance

a) It is used as an input to a UUT.

b) It applies a signal of type Impedance, i.e. no assumptions may be made about a voltage or current at the UUT pins where it is connected.

c) The Constant BSC is normally used. This applies an impedance signal on two terminals which may be considered a model of a impedance element such as a resistor.

d) Any source BSC (e.g., Sinusoid, WaveformRamp) may be of type Impedance but many may not be realizable by real instruments.

E.1.4.3 Use of the load property

a) It is used when expressing a value of a signal of type Physical.

b) It is used to provide the appropriate impedance value for translation between valid units, i.e. between Power and Voltage.

E.2 Implementing short circuits in IEEE 1641

There are two ways in which a short circuit can be implemented. The first being the application of a constant source impedance of zero ohms, the second being the connection of a Load to the output of two pins. One is used on the input of a UUT and the other is used on the output of a UUT. In both cases they are used to indicate that a short is required on the UUT.

E.2.1 Short on a UUT input

In this case the Constant BSC is used as a source of impedance. This may be set to zero ohms and applied (as with any other source) to the UUT pins (see case 1 in E.1.2).

It would also be possible to apply a signal of zero volts between the UUT pins. This would have the same effect as short always has a potential of zero volts across it. This would not work if the short has a tolerance or is not precisely zero ohms, or if the UUT pins floated above zero volts.

The Load BSC should not be used as a load on the input of a UUT as it is a transformation BSC and itself expects an input.

E.2.2 Short on a UUT output

This may be implemented by using the Load BSC. The UUT is the input to the Load BSC as illustrated in case 2 in E.1.2. The Load is set to zero ohms and connected to the UUT output pins. The Load BSC does not require an output as no measurement is likely to be taken and the signal will not normally be passed to another element, both of which would be pointless after a short.

Note that the Load BSC will not actually change the output from the UUT so a simulation may not give a representative output in this situation.

It would always be possible to treat the UUT output as an input and use the Constant BSC as a source of impedance. This would be set to zero ohms and applied to the UUT pins as in the previous UUT input example. In this case the result of a simulation may or may not provide a representative result because effectively two signals of different types are being connected in parallel.

E.3 Signals and triggering

This section shows how one signal may be used to trigger another using IEEE 1641. The example used is a fragment from a test requirement in which a sinusoid is used to trigger the application of a dc signal.

E.3.1 Test requirement fragment

A sinusoid signal is expected, and after 3 rising edges of the sinusoid a dc voltage is to be provided. The example assumes no delay is required.

The relationship that the dc signal (DCSignal) has to the sinusoid (SineWaveSignal) is that it starts immediately after 3 rising edges of the sinusoid. The trigger (TriggerIntent) happens outside of the signal definition of the dc voltage. The timing relationship is understood to be a decision that will be accomplished at the system level, but there is no implication of how it will be accomplished. The test requirement is expressed in XML format from an informal source., but the basic intent of the program fragment is easily determined.

```
<TEST REQUIREMENT >
  <Capability>
    <SignalModel:Signal name="SineWaveSignal" >
      <SignalModel:Sinusoid name="SineWave" amplitude="pk 5 V"
       frequency="60 Hz"/>
    </SignalModel:Signal>
  </Capability>
  <ChildInfo>
    <ChildTR>
      <Capability>
        <SignalModel:Signal name="DCSignal" >
          <SignalModel:Constant name="DC" amplitude="2.5 V"/>
        </SignalModel:Signal>
      </Capability>
```

```
      </ChildTR>
      <TriggerIntent type="risingEdgeCountComplete" count="3" voltageLevel="3.5 V" />
    </ChildInfo>
  </TEST REQUIREMENT >
```

E.3.2 Possible implementations

Although the test requirement does not specify an implementation, it is reasonable to describe two different test systems that are both capable of accomplishing the requirement.

The first system has function generator (FG) with an output trigger that can put out a sinusoid and provide a rising edge trigger on a trigger port. The first system also has a dc source with an input trigger port. These can be configured so that the FG outputs a sinusoid and provides a separate trigger signal to the dc source after three rising edges of the sinusoid, so that the dc source can provide dc output.

The second solution includes a dc source with a trigger input and processing capable of counting edges. These can be configured so that the FG's output is provided to the trigger port of the dc source as well as its intended UUT location. In this situation, the dc source would process the sinusoid and know when the third edge occurs so that it can output its dc signal.

The requirement can be mapped to either of these system scenarios because it does not make system decisions in the description of the relationship between signals.

E.3.3 The problem

How can this requirement be defined in IEEE 1641 and not demand one or the other system implementations?

E.3.4 The basic solution

There is fundamentally no problem in describing such a signal using IEEE 1641 as it is currently defined.

The XML version of the problem is a segment from the overall test requirement. The XML as presented is not quite complete, because there is no link between the "SineWaveSignal" waveform (that will be used to determine the trigger point) and the trigger definition. The trigger is defined to be of type "risingedgecountcomplete", but there is no definition of this except in the accompanying text.

By taking both the XML description and the text together, it is possible to determine exactly what the test is and from that to define a solution using IEEE 1641.

E.3.4.1 The SineWaveSignal

```
  <Signal name="SineWaveSignal" Out="SineWave" >
    <Sinusoid name="SineWave" amplitude="pk 5 V" frequency="60Hz" />
  </Signal>
```

E.3.4.2 The DCSignal

```
  <Signal name="DCSignal" Out="DC" >
    <Constant name="DC" amplitude="2.5 V" />
  </Signal>
```

E.3.4.3 The TriggerIntent

```
<Signal name="TriggerIntent" Out="RisingEdgeCountComplete" >
  <Instantaneous name="TriggerLevel" nominal="inst 3.5 V" condition="GE" />
  <EventCount name="EventCounter" count="2" Gate="TriggerLevel" />
  <EventedEvent name="RisingEdgeCountComplete" In="EventCounter" />
</Signal>
```

E.3.4.4 The complete signal

At this point there three separate definitions with no link between them. These can be combined to provide desired overall Test Requirement. The XML for that would look like this:

```
<Signal name="DCSignal" Out="DC" >
  <Sinusoid name="SineWave" amplitude="pk 5 V" frequency="60Hz" />
  <Instantaneous name="TriggerLevel" gateTime="0.1 s" nominal="inst 3.5 V"
   condition="GE" In="SineWave" />
  <EventCount name="EventCounter" count="2" In="TriggerLevel" />
  <EventedEvent name="RisingEdgeCountComplete" In="EventCounter" />
  <Constant name="DC" amplitude="2.5 V" Sync="RisingEdgeCountComplete" />
</Signal>
```

In this XML description it is possible to identify the definition of the sine wave ("SineWave"), the definition of the trigger ("RisingEdgeCountComplete") and the fact that it is related to the sine wave by the combination of "TriggerLevel" and "EventCounter"). The definition of the dc voltage may also be identified ("DC") together with the fact that it is triggered by "RisingEdgeCountComplete".

This gives the output as shown in Figure E.16. The sinusoidal trace shows the output from the Sinusoid BSC ("SineWave"). The bottom trace shows the Instantaneous BSC ("TriggerLevel") being activated each time that "SineWave" signal is greater than 3 V. The EventCount BSC ("EventCounter") counts three "TriggerLevel" events and then triggers the Constant BSC ("DC") via the EventedEvent BSC ("RisingEdgeCountComplete").

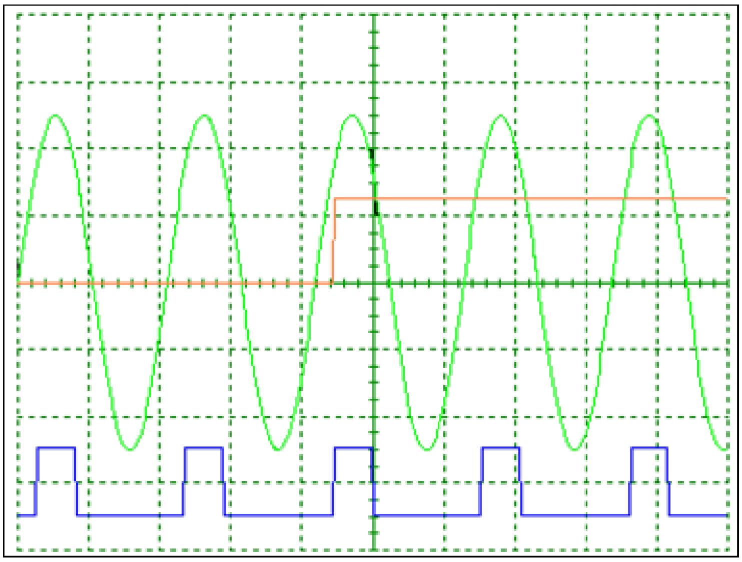


Figure E.16—dc voltage triggered by third cycle of sinusoid

Although this signal description appears to fully define the problem described above, it does have one shortcoming. The trigger is not quite correct as described as it counts the number of times that the "SineWave" has been greater than 3.5 V, and does not take into account the rising edge. This can be shown by offsetting the sinusoid by 90 deg as shown in Figure E.17, where the trigger fires on the second rising edge.

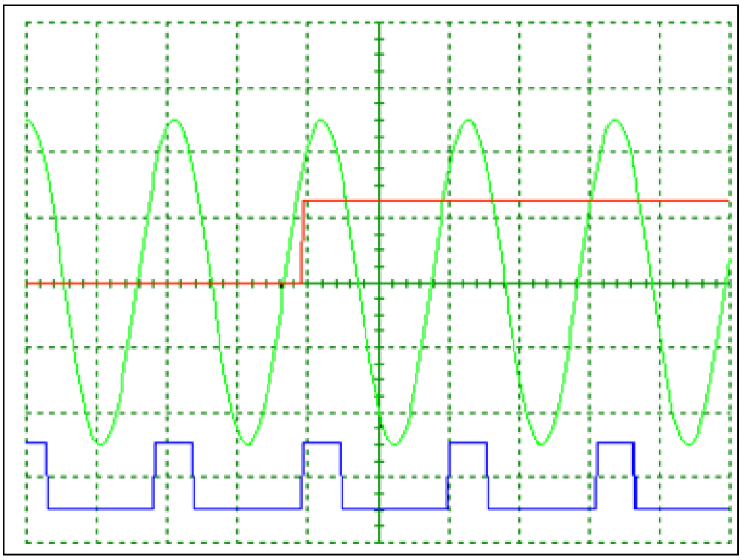


Figure E.17—Sinusoid starting with 90 deg offset

E.3.5 A complete solution

This is not necessarily a point at issue, but it is good practice to get the model correct, if only to illustrate that the IEEE 1641 XML for the trigger can accurately be made to describe the trigger point related to the signal that is being assessed.

The trigger is better described as follows:

```
<Signal name="TriggerIntent" Out="RisingEdgeCountComplete">
   <Instantaneous name="MustStartBelowTriggerLevel" gateTime="0.1s"
      nominal="inst 3.5 V" condition="LT" In="SineWave" />
   <Instantaneous name="TriggerLevel" gateTime="0.1 s" nominal="inst 3.5 V"
      condition="GE" Sync="MustStartBelowTriggerLevel" In="SineWave" />
   <EventCount name="EventCounter" count="2" In="TriggerLevel" />
   <EventedEvent name="RisingEdgeCountComplete" In="EventCounter" />
</Signal>
```

This ensures that the SineWave must be below 3.5 V before it starts counting events, as shown in Figure E.18.

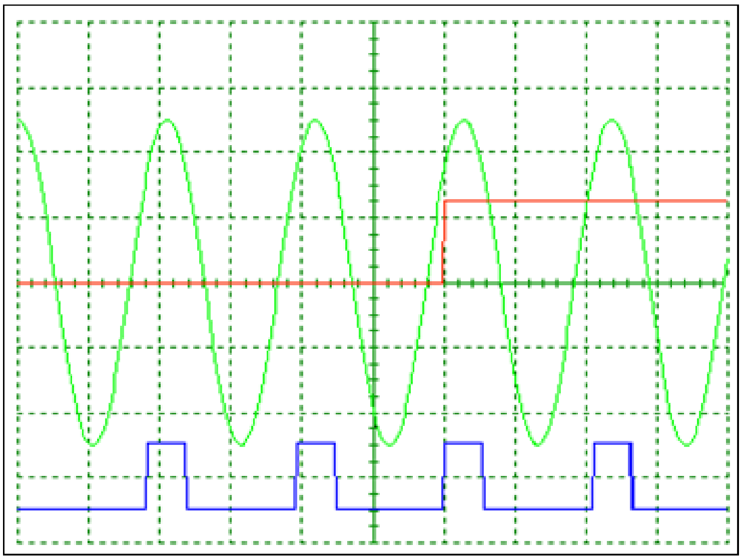


Figure E.18—Simulation showing trigger on rising edge

Although this looks complicated for the trigger description, it can be described once in a TSF model and reused. It is also accurate and leaves no room for misinterpretation.

The important issue here is that this describes exactly when the trigger is required related to the sinusoid, and is correct whether the sinusoid is output by a function generator (FG), an arbitrary waveform generator (ARB), or even the UUT. The two scenarios described in the original problem above are both possible implementations of the IEEE 1641 test description. This requires the implementer to understand the requirement (as clearly defined in IEEE 1641) and to implement the test using his knowledge of the way his particular test equipment works. The IEEE 1641 definition of the test requirement is not actually demanding any particular method of implementation.

Annex F

(informative)

Implementation of IEEE 1641—Application techniques

F.1 Implementing IEEE 1641—RF stimulus and measurement

F.1.1 Introduction

This example shows how IEEE 1641 may be used to document complex RF testing, specifically a complex RF measurement. This example was implemented using an ATS based on synthetic instrumentation. The UUT was representative of a spacecraft communications unit, operating in the microwave (Ku band) region. The test signals were defined and encapsulated as reusable test elements, i.e. IEEE 1641 TSFs, and called from a test sequence. The tests implemented were Spectrum Measurement, Gain Transfer and Spur Search.

In order to verify the 1641 signal & measurement definitions against those applied to the UUT, signal data captured from the test equipment was compared with simulation data, generated through SML.

F.1.2 Unit under test

The UUT chosen for this implementation was a filter, designed to model certain aspects of an up/down converter, used as part of a spacecraft microwave (Ku band) communication system. This particular unit had already been used to demonstrate the capabilities of the ATS and the selected tests already existed in the legacy test language. Therefore a set of results existed which could be used to validate the results.

F.1.3 The ATS synthetic instrumentation

There is a natural link between IEEE 1641 and synthetic instrumentation because of the SML that IEEE 1641 uses to define the behavior of its BSCs. Waveform arrays can directly be generated from any IEEE 1641 measurement or stimulus signal definition and then mapped onto the digitized arrays of a synthetic instrument. The architecture of the synthetic instrumentation used is shown in Figure F.1

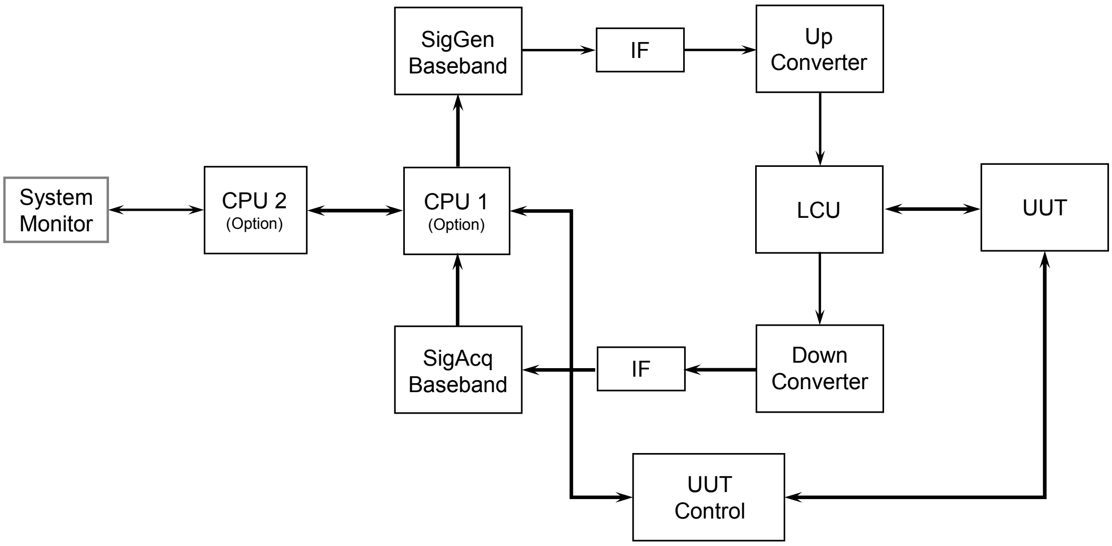


Figure F.1—Synthetic instrumentation architecture of the ATS used

F.1.4 Implemented tests

The tests selected were typical for a unit used in a Ku band communications system.

a) Spectrum Measurement—a common RF test (amplifier, etc) to characterize input/output pass-band frequency transfer functions (power out *vs*. frequency).

b) Gain Transfer—a common RF test (amplifier, etc) to characterize input/output power transfer functions (power in *vs*. power out).

c) Spur Search—a common RF test (amplifier, etc) to characterize the presence of unwanted spurious signal artifacts (spur frequency & level), including the exclusion of known wanted artifacts.

F.1.5 Spectrum measurement

This measurement required an input signal comprising a sinusoid and an output measurement system that included a matching impedance, and an average power measurement over a specified band.. The output spectrum was obtained to characterize the effect of the amplifier on the sinusoidal signal. Figure F.2 shows the source TSF with its output feeding into the input of the UUT, and the output of the UUT feeding into the measurement TSF.

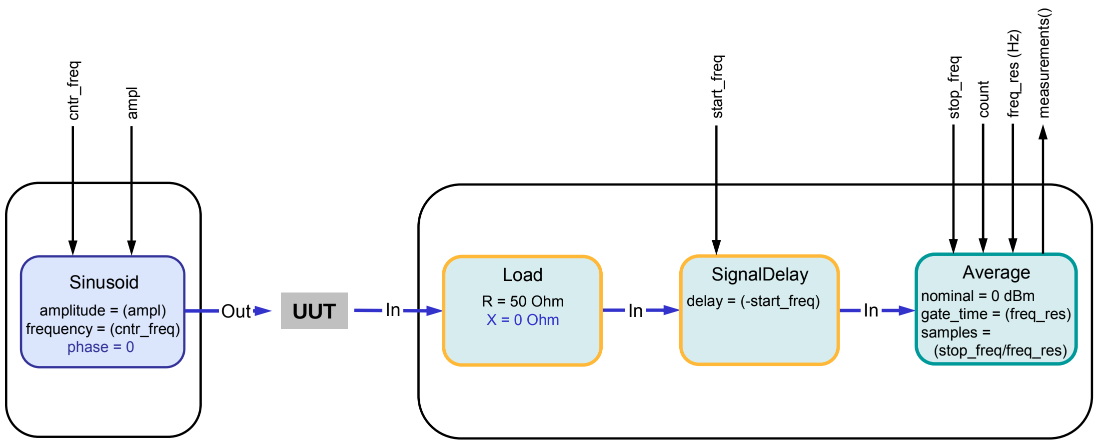


Figure F.2—Spectrum measurement setup

F.1.5.1 Source signal (CW_SIGNAL)

Figure F.3 shows the source signal TSF. It comprises a Sinusoid BSC, fixed in the Power domain with control over amplitude and frequency attributes. This provides a constant sine wave signal in which the frequency and power amplitude may be specified.

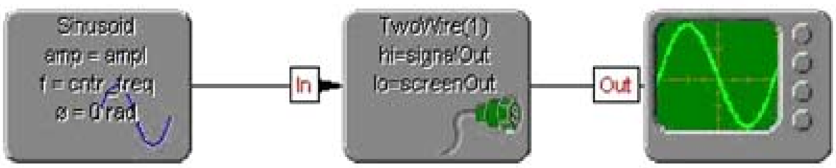


Figure F.3—Source signal TSF

The XML version of this TSF is provided below.

```
<?xml version="1.0" ?>
<tsf:TSF name="CW_SIGNAL" uuid="{F30F36BA-AF42-4489-B6B5-3E1ED7050952}"
 xmlns:tsf="STDTSF" xmlns:std="STDBSC" xmlns="STDBSC">
  <tsf:interface>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     elementFormDefault="qualified">
      <xs:element name="CW_SIGNAL">
        <xs:annotation>
          <xs:documentation>
           A sinusoid signal used to characterize a.power transfer function
          </xs:documentation>
        </xs:annotation>
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="SignalFunctionType">
              <xs:attribute name="cntr_freq" type="Frequency">
                <xs:annotation>
                  <xs:documentation>
                   The frequency of the applied signal.
                  </xs:documentation>
                </xs:annotation>
              </xs:attribute>
              <xs:attribute name="ampl" type="Power">
                <xs:annotation>
```

```
                    <xs:documentation>
                      The output power in the applied signal (into 50 Ohm)
                      </xs:documentation>
                  </xs:annotation>
                </xs:attribute>
                <xs:attribute name="signalOut" type="string">
                  <xs:annotation>
                    <xs:documentation>
                     The output signal connection.
                    </xs:documentation>
                  </xs:annotation>
                </xs:attribute>
                <xs:attribute name="screenOut" type="string">
                  <xs:annotation>
                    <xs:documentation>
                     The output ground connection.
                    </xs:documentation>
                  </xs:annotation>
                </xs:attribute>
              </xs:extension>
            </xs:complexContent>
          </xs:complexType>
        </xs:element>
      </xs:schema>
  </tsf:interface>
  <tsf:model>
    <Signal Out="TwoWire_4121">
      <Sinusoid name="CW_Sinusoid" amplitude="ampl" frequency="cntr_freq" />
      <TwoWire name="TwoWire_4121" lo="screenOut" hi="signalOut" channelWidth="1"
       In="CW_Sinusoid" />
    </Signal>
  </tsf:model>
</tsf:TSF>
```

F.1.5.2 Characterization measurement TSF (SPECTRUM_MEAS)

Figure F.4 shows the characterization TSF. This performs a frequency 'sweep' of the output of the amplifier and characterizes the output. The TSF comprises a permanent input Load of 50 Ω; followed by a SignalDelay in the frequency domain to exclude frequencies below the starting frequency, and an Average BSC to measure the power. The exclusion of frequencies below the starting frequency (start_freq) prevents the scanning of all frequencies from zero up to the starting frequency. This also suggests (but does not specify) that a down-converter could be used. The average power of each freq_res band, up to the stop_freq, is then measured to create a ‘zoom’ into the required frequency range. A nominal value of 0.0 dBm specifies that power should be returned as a gain in deciBels, with reference to 1 mW.

Figure F.4—Spectrum measurement TSF

The XML version of this TSF is as follows:

```
<?xml version="1.0" ?>
<tsf:TSF name="SPECTRUM_MEAS" uuid="{85093394-13CC-4A45-8D45-59ED389895EC}"
xmlns:tsf="STDTSF" xmlns:std="STDBSC" xmlns="STDBSC">
```

```
    <tsf:interface>
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
       elementFormDefault="qualified">
      <xs:element name="SPECTRUM_MEAS">
        <xs:annotation>
          <xs:documentation>
          An RF measurement used as part of test to characterize transfer function
          (power out vs. frequency)
          </xs:documentation>
        </xs:annotation>
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="SignalFunctionType">
              <xs:attribute name="start_freq" type="Frequency">
                <xs:annotation>
                  <xs:documentation>The starting frequency for the sweep
                  </xs:documentation>
                </xs:annotation>
              </xs:attribute>
              <xs:attribute name="stop_freq" type="Frequency">
                <xs:annotation>
                  <xs:documentation>
                   The stop frequency at the end of the sweep</xs:documentation>
                  </xs:annotation>
                </xs:attribute>
                <xs:attribute name="freq_res" type="Frequency">
                  <xs:annotation>
                    <xs:documentation>The frequncy resolution required</xs:documentation>
                  </xs:annotation>
                </xs:attribute>
                <xs:attribute name="signalIn" type="string">
                  <xs:annotation>
                    <xs:documentation>
                     The input signal connection.
                    </xs:documentation>
                  </xs:annotation>
                </xs:attribute>
                <xs:attribute name="screenIn" type="string">
                  <xs:annotation>
                    <xs:documentation>
                     The input ground connection.
                    </xs:documentation>
                  </xs:annotation>
                </xs:attribute>
              </xs:extension>
            </xs:complexContent>
          </xs:complexType>
        </xs:element>
      </xs:schema>
    </tsf:interface>
    <tsf:model>
      <Signal Out="AveragePower_4133">
        <TwoWire type="Power" reftype="Frequency" name="TwoWire_4125"
         lo="screenIn" hi="signalIn" channelWidth="1" />
        <Load type="Power" reftype="Frequency" name="Load_4126" resistance="50 Ohm"
         In="TwoWire_4125" />
        <SignalDelay type="Power" reftype="Frequency" name="SignalDelay_4129"
         delay="{-start_freq}" In="Load_4126" />
        <Average type="Power" reftype="Frequency" name="AveragePower_4133"
         samples="{stop_freq/freq_res}" gateTime="{freq_res}" nominal="0 dBm"
         In="SignalDelay_4129" />
      </Signal>
    </tsf:model>
  </tsf:TSF>
```

F.1.5.3 Executing the test

The test is then setup as shown in Figure F.2 The XML for the overall spectrum characterization is shown below. The output signal can be identified (CW_SIGNAL_4159) with its UUT connections PL1_1 and PL1_2. The input signal from the UUT may also be identified connected via PL4_1 and PL4_2.

```
<?xml version="1.0" ?>
<Signal Out="CW_SIGNAL_4159 SPECTRUM_MEAS_4164" xmlns="STDBSC" xmlns:this="MyTSFLib"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="MyTSFLib
 MyTSFLib.xsd">
  <this:CW_SIGNAL type="Power" name="CW_SIGNAL_4159" cntr_freq="1.5 GHz" ampl="0.5 dBm"
   signalOut="PL1_1" screenOut="PL1_2" />
  <this:SPECTRUM_MEAS type="Power" reftype="Frequency" name="SPECTRUM_MEAS_4164"
   start_freq="1.2 GHz" stop_freq="1.7 MHz" freq_res="0.5 MHz"
   signalIn="J4-1" screenIn="J4-2" />
</Signal>
```

The resultant waveform obtained by an implementation of this IEEE 1641 code is shown in Figure F.5.

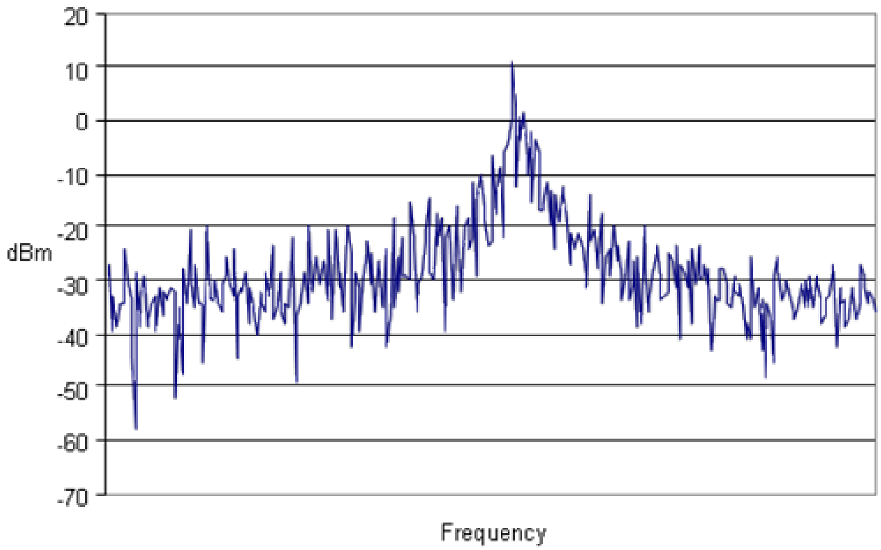


Figure F.5—Actual results from an implementation of a characterization measurement

F.1.6 Gain Transfer Test

The objective of this test is to measure the maximum power value in the whole spectrum. The same input as for the spectrum characterization is used but a new TSF is created for the output measurement. In this case, the Average BSC is replaced by a MaxInstantaneous BSC. A measurement of a single maximum instantaneous value is specified, representing the maximum power value in the spectrum. A nominal value of ‘0.0 dBm’ specifies that power should be returned as a gain in deciBels, with reference to 1 mW.

F.1.6.1 Maximum power measurement TSF (TONE_POWER)

Figure F.6 shows the maximum power TSF (TONE_POWER). This obtains the maximum power in the output signal over the bandwidth specified. The TSF comprises a permanent input Load of 50 Ω; followed by a SignalDelay (in the frequency domain) to exclude frequencies below those to be examined, and a MaxInstantaneous BSC to measure the maximum power within the bandwidth. As with the spectrum characterization, the exclusion of frequencies below the starting frequency (start_freq) prevents the scanning of all frequencies from zero up to the starting frequency and suggests (but does not specify) that a down-converter could be used. A nominal value of 0.0 dBm specifies that power should be returned as a gain in deciBels, with reference to 1 mW. The connection BSC (TwoWire) has been omitted in this case.

Figure F.6—Maximum power TSF

The XML version of this TSF is shown below. This example does not include a connection BSC. This does not affect the operation as the connection BSC may be added when the TSF is used.

```
<?xml version="1.0" ?>
<tsf:TSF name="TONE_POWER" uuid="{438D1283-B166-4265-8508-20CA66D5F2F6}"
xmlns:tsf="STDTSF" xmlns:std="STDBSC" xmlns="STDBSC">
  <tsf:interface>
    <xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema
     elementFormDefault="qualified">
      <xs:element name="TONE_POWER">
        <xs:annotation>
          <xs:documentation>
            Measures the Peak Power over a frequency band.
          </xs:documentation>
        </xs:annotation>
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="SignalFunctionType">
              <xs:attribute name="cntr_freq" type="Frequency">
                <xs:annotation>
                  <xs:documentation>
                    Centre frequency (where power is expected to be at a maximum)
                  </xs:documentation>
                </xs:annotation>
              </xs:attribute>
              <xs:attribute name="bandwidth" type="Frequency">
                  <xs:documentation>
                    The bandwidth over which power is to be measured.
                  </xs:documentation>
                </xs:annotation>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </tsf:interface>
  <tsf:model>
    <Signal Out="MaxInstantaneousPower_4199">
      <Load type="Power" reftype="Frequency" name="Load_4192" resistance="50 Ohm" />
      <SignalDelay type="Power" reftype="Frequency" name="SignalDelay_4195"
       delay="-{cnt_freq - {bandwidth/2}}" In="Load_4192" />
```

```
      <MaxInstantaneous type="Power" reftype="Frequency"
       name="MaxInstantaneousPower_4199" samples="1"
       gateTime="{cnt_freq + {bandwidth/2}}" nominal="0.0 dBm" In="SignalDelay_4195" />
    </Signal>
  </tsf:model>
</tsf:TSF>
```

F.1.6.2 Executing the test

The test is then setup in the same way as for the spectrum characterization measurement (see Figure F.2). The XML for the maximum power measurement is shown below. The output signal remains as for the previous example. The input signal from the UUT is connected via PL4_1 and PL4_2 (shown in this example as a separate BSC), and the maximum instantaneous power is then measured in the TONE_POWER TSF.

```
<?xml version="1.0" ?>
<Signal Out="CW_SIGNAL_4159 TONE_POWER_4206" xmlns="STDBSC" xmlns:this="MyTSFLib"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="MyTSFLib
 MyTSFLib.xsd">
  <this:CW_SIGNAL type="Power" name="CW_SIGNAL_4159" cntr_freq="1.5 GHz" ampl="0.5 dBm"
   signalOut="PL1_1" screenOut="PL1_2" />
  <TwoWire type="Power" reftype="Frequency" name="TwoWire_4211" lo="J4_2" hi="J4_1"
   channelWidth="1" />
  <this:TONE_POWER type="Power" reftype="Frequency" name="TONE_POWER_4206"
   cntr_freq="1.5 GHz" bandwidth="300 MHz" In="TwoWire_4211" />
</Signal>
```

F.1.7 Spur Search

Figure F.7 shows the TSF (SPUR_SEARCH) that will search for spurious outputs from the amplifier. After the SignalDelay in the frequency domain, which excludes frequencies below freq_start, pulses create masks for the frequency ranges of interest (mask_start to mask_stop). A specified number of MaxInstantaneous measurements (being the number of spurs to be detected) are made at the specified frequency resolution (freq_res), where the specified threshold is exceeded. The measurements provide the amplitude and frequency of the spurii, hence the two MaxInstantaneous BSCs, where one measures the dependent variable (the default) and the other the independent variable (measuredVariable="INDEPENDENT").

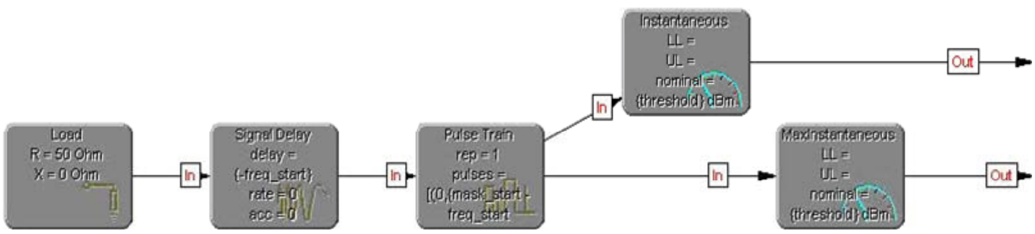


Figure F.7—Spur search TSF

The XML version of this TSF is shown below. The pulse definitions (PulseDefns) may be identified in the XML code. One shows a pulse width (pulseWidth) to be equal to (mask_start - freq_start), with the start and levelFactor using the default values. The other pulse has a start value equal to (mask_stop - freq_start), a pulseWidth equal to (freq_stop - freq_start), and a levelFactor of 1 (the default).

pulses="[(0, {mask_start - freq_start}, 1), ({mask_stop - freq_start}, {freq_stop - freq_start}, 1)]"

This TSF does not have built in connections, so must be used with a connection BSC.

```
<?xml version="1.0" ?>
<tsf:TSF name="SPUR_SEARCH" uuid="{DAC9975A-0A16-46FB-AA3B-D469A1288C87}"
 xmlns:tsf="STDTSF" xmlns:std="STDBSC" xmlns="STDBSC">
  <tsf:interface>
    <xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema
     elementFormDefault="qualified">
      <xs:element name="SPUR_SEARCH">
        <xs:annotation>
          <xs:documentation>
           This searches for spurii that are above a specified threshold
           value
          </xs:documentation>
        </xs:annotation>
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="SignalFunctionType">
              <xs:attribute name="freq_start" type="Frequency">
                <xs:annotation>
                  <xs:documentation>Minimum frequency to be scanned</xs:documentation>
                </xs:annotation>
              </xs:attribute>
              <xs:attribute name="freq_stop" type="Frequency">
                <xs:annotation>
                  <xs:documentation>Maximum frequency to be scanned</xs:documentation>
                </xs:annotation>
              </xs:attribute>
              <xs:attribute name="mask_start" type="Frequency">
                <xs:annotation>
                  <xs:documentation>
                   Start of frequency range of interest
                  </xs:documentation>
                </xs:annotation>
              </xs:attribute>
              <xs:attribute name="mask_stop" type="Frequency">
                <xs:annotation>
                  <xs:documentation>End of frequency range of interest</xs:documentation>
                </xs:annotation>
              </xs:attribute>
              <xs:attribute name="freq_res" type="Frequency">
                <xs:annotation>
                  <xs:documentation>Frequency resolution required</xs:documentation>
                </xs:annotation>
              </xs:attribute>
              <xs:attribute name="threshold" type="Power">
                <xs:annotation>
                  <xs:documentation>
                   Minimum power level required for a signal to be considerdd a spur
                  </xs:documentation>
                </xs:annotation>
              </xs:attribute>
              <xs:attribute name="no_spurs" type="int">
                <xs:annotation>
                  <xs:documentation>
                   Number of spurii to be searched for.
                  </xs:documentation>
                </xs:annotation>
              </xs:attribute>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </tsf:interface>
  <tsf:model>
    <Signal Out="MaxInstantaneousPower_4229 InstantaneousPower_4225">
      <Load type="Power" reftype="Frequency" name="Load_4221" resistance="50 Ohm" />
      <SignalDelay name="SignalDelay_4217" delay="{-freq_start}" In="Load_4221" />
      <PulseTrain type="Power" reftype="Frequency" name="PulseTrain_4224"
```

```
        pulses="[(0, {mask_start - freq_start}, 1),
                 ({mask_stop - freq_start}, {freq_stop - freq_start}, 1)]"
        repetition="1" In="SignalDelay_4217" />
       <MaxInstantaneous type="Power" reftype="Frequency"
        name="MaxInstantaneousPower_4229" measuredVariable="INDEPENDENT"
        samples="{no_spurs}" gateTime="{freq_res}" nominal="{threshold} dBm"
        condition="GT" In="PulseTrain_4224" />
       <Instantaneous type="Power" reftype="Frequency" name="InstantaneousPower_4225"
        samples="{no_spurs}" gateTime="{freq_res}" nominal="{threshold} dBm"
        condition="GT" In="PulseTrain_4224" />
    </Signal>
  </tsf:model>
</tsf:TSF>
```

F.1.7.1 Executing the test

The test is then setup in the same way as for the previous two measurements (see Figure F.2). The XML for the spur search is shown below. The output signal remains as for the previous example. The input signal from the UUT is connected via PL4_1 and PL4_2 (shown in this example as a separate BSC), and the spurii are then detected by the SPUR_SEARCH TSF.

```
<?xml version="1.0" ?>
<Signal Out="CW_SIGNAL__4159 SPUR_SEARCH__4260" xmlns="STDBSC" xmlns:this="MyTSFLib"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="MyTSFLib
 MyTSFLib.xsd">
  <this:CW_SIGNAL type="Power" name="CW_SIGNAL__4159" cntr_freq="1.5 GHz" ampl="0.5 dBm"
   signalOut="PL1_1" screenOut="PL1_2" />
  <TwoWire type="Power" reftype="Frequency" name="TwoWire__4211" lo="J4_2" hi="J4_1"
   channelWidth="1" />
  <this:SPUR_SEARCH type="Power" reftype="Frequency" name="SPUR_SEARCH__4260"
   freq_start="1.3 GHz" freq_stop="1.7 GHz" mask_start="1.35 GHz" mask_stop="1.65 GHz"
   freq_res="1 MHz" threshold="6 dBm" no_spurs="6" In="TwoWire__4211" />
</Signal>
```

F.2 Implementing IEEE 1641—Amplifier characterization using IEEE 1641

F.2.1 Introduction

This example discusses an implementation of a particular set of tests, for a UUT on a typical set of test resources. The UUT is an amplifier, typically used in communications, radar & instrumentation, in the range 0.02 GHz to 3 GHz. A 1 dB compression point test has been selected because, though the concept is relatively easy to understand, its implementation can be tedious, prone to error and there exist numerous techniques for its implementation. IEEE 1641 signal modeling provides a consistent definition of this measurement, which can be validated independently of its implementation. A gain (S-Parameter S21) has also been selected because it is a precursor to the 1 dB compression point measurement. A gain transfer test is included to show how a sweep of the output can be defined and ensure that the peak output power is determined, rather than the power at a specified frequency, which may not be the frequency where the maximum power is concentrated.

This example is based on a real, practical set of tests performed using IEEE 1641, but is presented here as generic guidance without reference to any specific tools or equipment. Prior to these tests being designed and implemented using IEEE 1641, they were performed using traditional programming methods and the results saved for comparison and verification purposes. Test programs were generated in C#, using IVI drivers for the two sets of test equipment. The test programs were executed against small commercial amplifier (the UUT) to create the base-line test results, for later validation of the 1641 test programs.

F.2.2 Source signal requirements (TSF design)

F.2.2.1 UUT Power Supply

Figure F.8 shows the UUT power supply unit (PSU) TSF. At its most basic, this is a constant voltage applied to the UUT power pins through a TwoWire connection. In this model the maximum permissible voltage is specified through the 'errlmt MAX' modifier of the TSF's amplitude property, as 13 V.

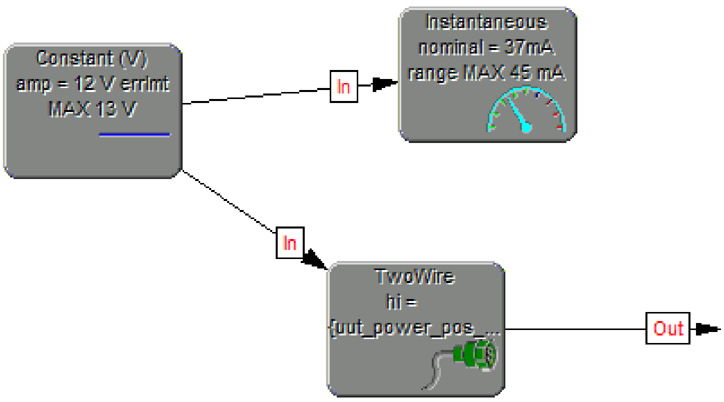


Figure F.8—UUT power supply TSF

The current that a UUT might consume is not important to the signal model of the power supply, because IEEE 1641 defines tests independently from their implementation, i.e. the test requirement is for a constant voltage. The amount of current drawn is not part of the signal model; but a practical problem exists that if the wrong resource is chosen, the constant voltage may not be maintained. This information is often required in order to maximize the efficiency of allocation of a suitable resource. For example, a precision voltage source and a heavy duty power supply may both be capable of supplying a constant 12 V; the former might have a very small current capacity, whereas the latter may require a minimum current drain in order to provide proper regulation.

In order to indicate the amount of current possibly required by the UUT, a concept of "capability"' has been added to the UUT power supply TSF. This stipulates that, during the test, if the instantaneous current were to be measured, it would nominally be 37 mA, with a maximum of 45 mA (i.e. 37 mA range MAX 45 mA). The Instantaneous BSC does not have an Out connection, so it is not part of the test requirement, but it does represent capability requirement of the chosen resource.

F.2.2.2 RF stimulus

Both the 1 dB Compression Point and Gain tests require a known RF continuous wave (CW) stimulus signal to be applied to the UUT before any measurements may be taken. Figure F.9 shows the RF CW stimulus TSF.

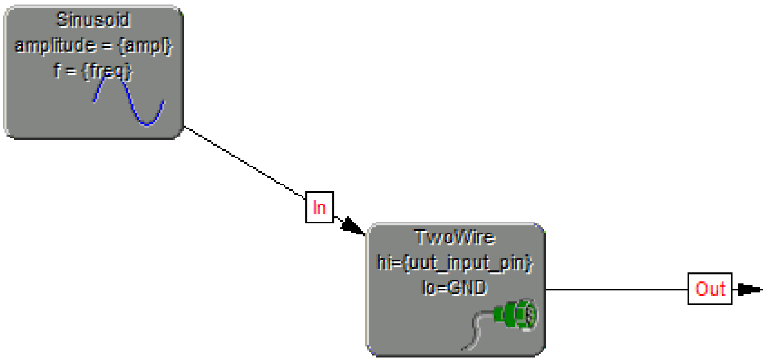


Figure F.9—RF CW stimulus TSF

F.2.2.2.1 Physical Type of the CW signal

When referring to RF signals, values are often expressed as a power level, e.g., 10 dBm. The unit symbol dBm is commonly used shorthand for dB (1 mW) and is acceptable in IEEE 1641. Care is required when expressing a power signal requirement in IEEE 1641. What is actually required is a voltage signal that will deliver certain amount of power into an input with a specified impedance, usually 50 Ω. This is illustrated in Figure F.10 and Figure F.11.

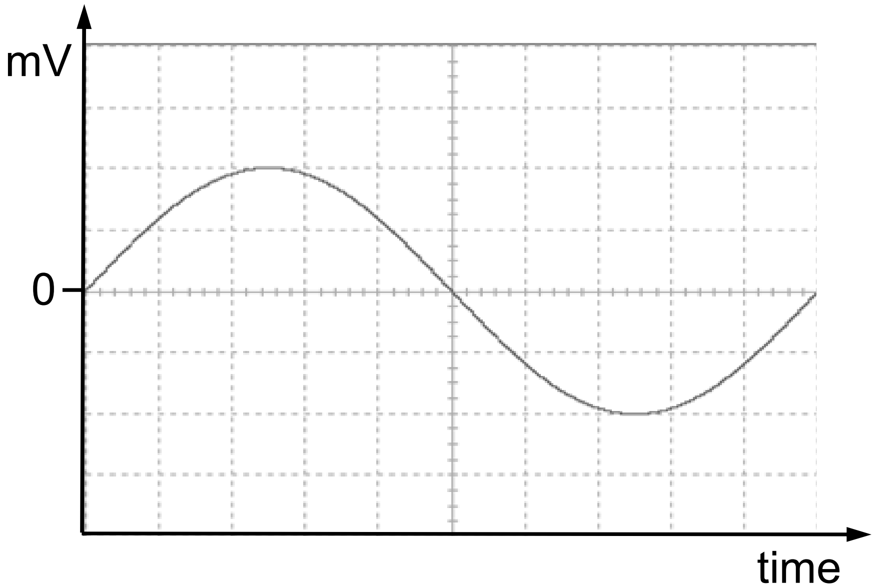


Figure F.10—Example of output from Sinusoid BSC

Figure F.10 illustrates a Sinusoid of type Voltage at a specified frequency. If a Sinusoid of type Power were specified with the same specified frequency it would look exactly the same graphically, because all IEEE 1641 signals are modeled in the same way, irrespective of type. For type = Voltage, this is reasonably intuitive. For type = Power, the sinusoid in Figure F.10 is not what we require. Figure F.11 shows what we might have expected from a Power signal; i.e. the product of voltage and current at twice the expected frequency.

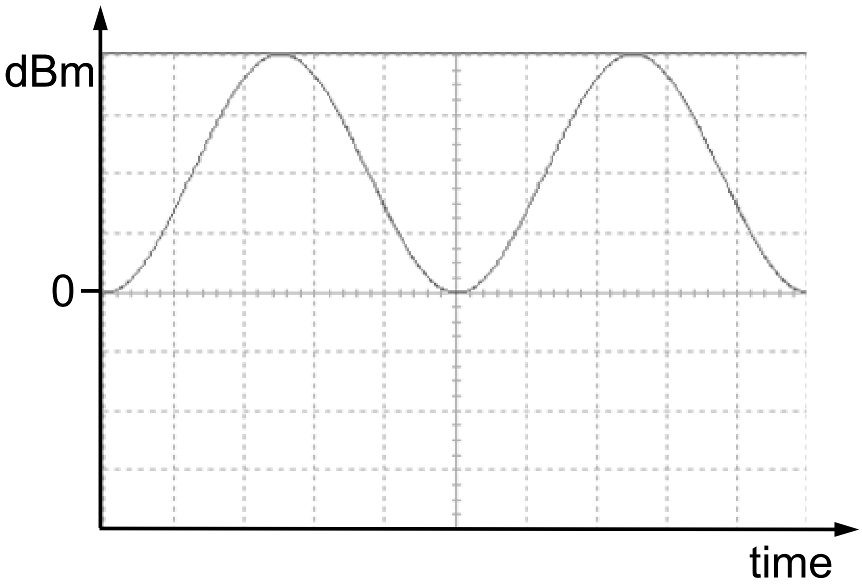


Figure F.11—Example of Sinusoidal Power

F.2.2.2.2 Impedance Matching

In order to apply a voltage, such that a given power is transferred, it is required that the impedance of the recipient of the power transfer (in this case the UUT) be known. IEEE 1641 defines a value modifier, 'load', that may be used to specify the impedance of a recipient of a power (or similar) transfer, where 'load' has a default value of 50 Ω. So the signal is specified as a signal of type Voltage with the amplitude provided in dBm with a load modifier of 50 Ω.

When measuring values on an output that requires a matching impedance a Load BSC may be used in the measurement BSC adjacent to the connection BSC. For example, if a 50 Ω matching impedance is required, a Load BSC with a value of 50 Ω is placed immediately after the TwoWire connection BSC.

F.2.2.3 A TSF model for the 1 dB compression point

An approach may be used that describes this TSF model such that it defines the point where 1 dB compression occurs, rather than a process to locate it. To locate the 1 dB compression point requires a series of measurements, with each iteration based on the previous measurement. By defining the target requirement rather than the process, the implementation independence of the signal-based test definition is maintained.

Figure F.12 illustrates the formula for determining the 1 dB compression point. The solid black line shows power (in) vs. power (out) for the UUT. It is known that this line will curve downwards as the amplifier runs into 'compression'. The heavier dashed line is calculated to be 1 dB below a line projected upwards from the 'small signal gain' (linear region) until it intersects with the measured line. The dotted lines encapsulate the region that defines the error limit for the measurement.

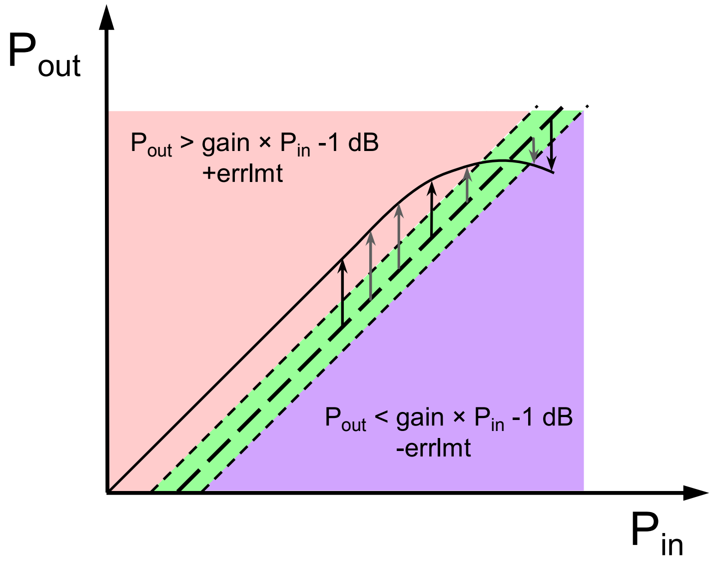


Figure F.12—1 dB compression point formula

The TSF defines attributes to indicate whether the 1 dB compression point has been reached or exceeded and an error limit that creates a region of success around that point.

To measure the 1 dB compression point, a test program typically takes measurements, denoted by the vertical arrows in the diagram, along the solid black line until the measured value is in the region denoted by the dotted lines. These measurements might be in a sequence of very small steps (small enough so that the measurement eventually just steps into the dotted region.; or, might be initially large, stepping back & forth, reducing in size each time the dotted region is stepped over. This illustrates the implementation independence of IEEE 1641.

Figure F.13 shows the TSF design for the 1 dB compression point test.

By including the RF CW stimulus TSF through a Sync connection, the TSF is defining that the stimulus signal must be applied to the UUT before the measurement is taken. A SignalDelay is used to define a minimum settling time for the UUT.

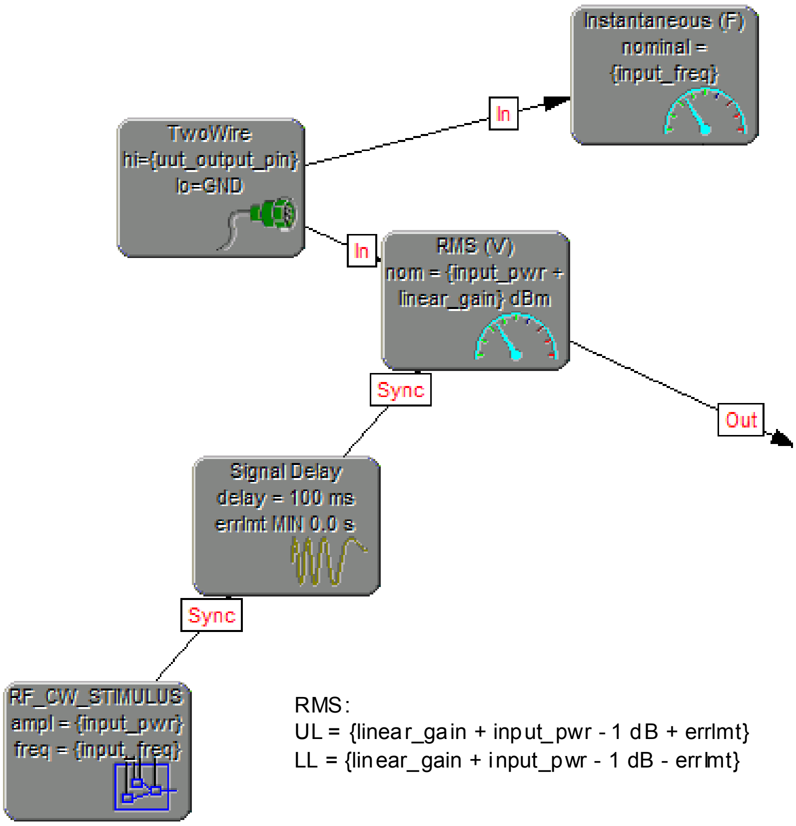


Figure F.13—1 dB compression point TSF

Figure F.14 illustrates the how correct RMS measurement of power may be achieved through a product of voltage and current measurements. This would, in fact, be mathematically equivalent to an IEEE 1641 average power measurement (i.e. not rms power). In practice, it is not necessary to measure the rms current, since this can be calculated from the voltage and input impedance. Also, where voltage and current are out of phase with each other, an average power model may be more appropriate.

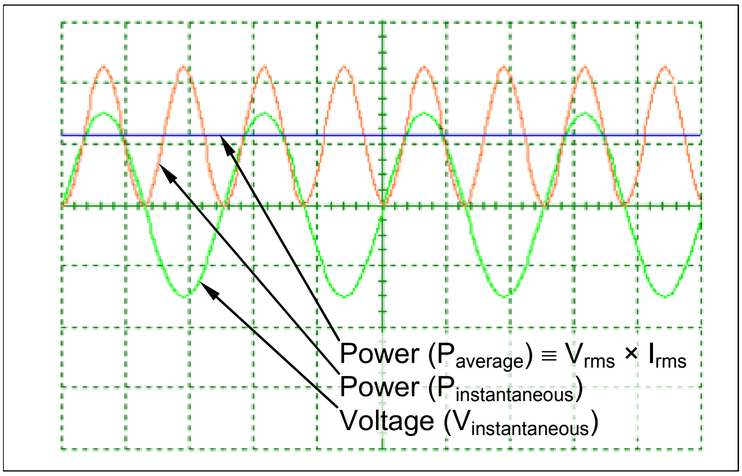


Figure F.14—Example average power *vs* V_{RMS} × I_{RMS}

Figure F.15 shows the TSF design to determine gain. The RF CW stimulus TSF is used to synchronize the measurement, the RMS measurement of type Voltage is used to measure power; and the Instantaneous frequency measurement is used to specify a capability requirement.

F.2.2.4 Gain TSF

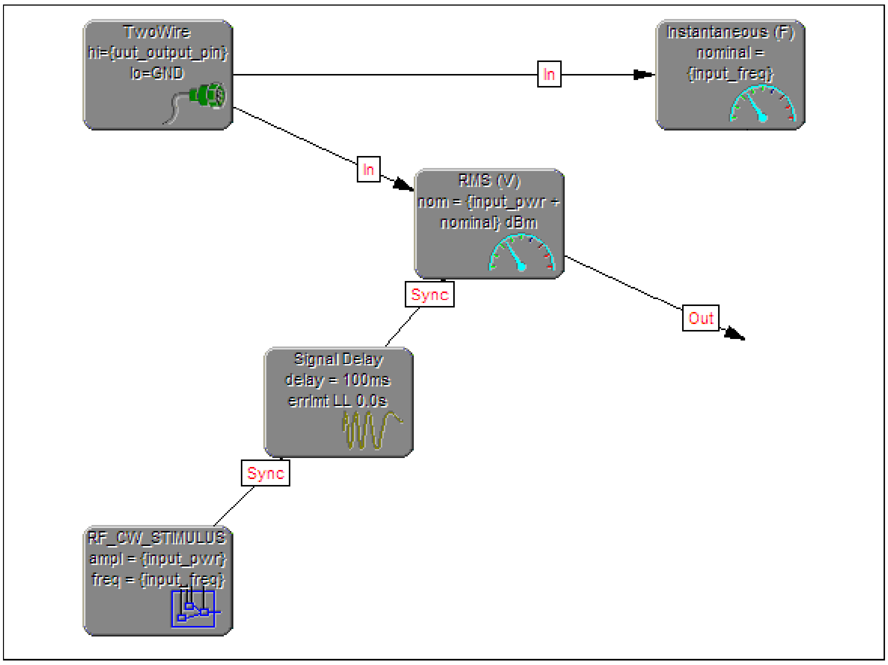


Figure F.15—Gain measurement TSF

F.2.2.5 Gain transfer TSFs

Two TSFs are required to implement gain transfer; one stimulus and one measurement. The stimulus TSF is same RF CW stimulus TSF as defined in F.2.2.2. The measurement TSF is shown in Figure F.16 and the measurement is made in the frequency domain (i.e. spectrum analysis). The formula to calculate the gain from these two TSFs resides in the test program rather than in the TSF models.

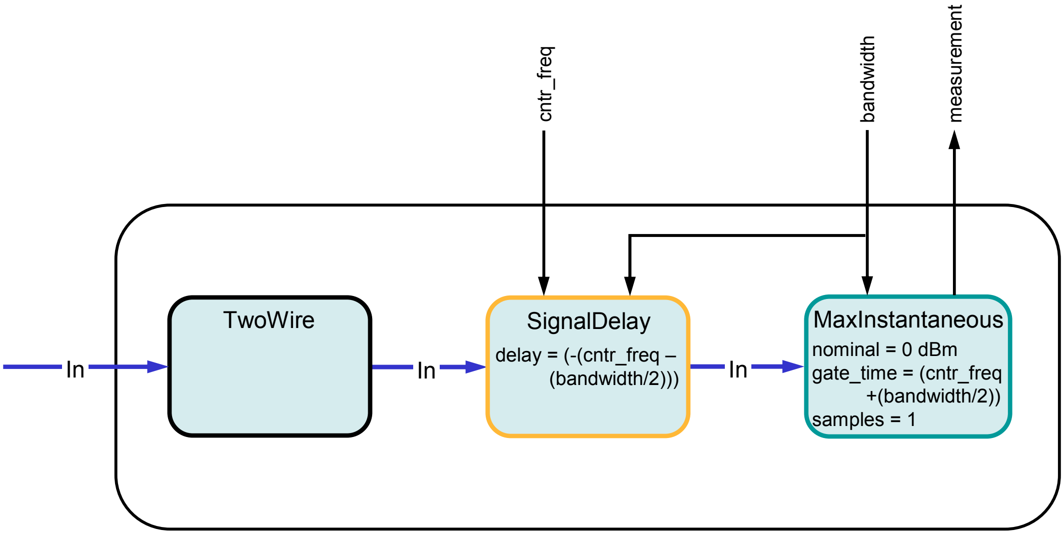


Figure F.16—Gain transfer measurement TSF

This TSF comprises a connection BSC, followed by a SignalDelay in the Frequency domain, to exclude frequencies below the start frequency of the sweep. This is followed by a measurement of a single MaxInstantaneous value, representing the maximum power value in the whole spectrum. A nominal value of "0.0 dBm" specifies that power should be returned as a gain in dB, with reference to 1 mW. This is similar to the TONE_POWER TSF used in F.1.6.1.

F.2.3 Simulation

It is possible to simulate tests in IEEE 1641 because of the underlying SML definitions of all BSCs. The simulator used must have been validated against the SML definitions. In order to fully simulate the tests, it is necessary to provide a model of the UUT as well as the signal requirement TSFs. Test programs must be written to call the TSFs in a suitable carrier language, e.g., C#. The TSF libraries may be exported in IDL by the modeling tool. The IDL is then compiled to create an IEEE 1641 compliant interface wrapper through which the C# test program can call the simulator processes. Figure F.17 illustrates the architecture of the simulation, showing the exported IDL interface, along with the XML TSF library, which contains the definitions of the TSFs signal models, used for simulation.

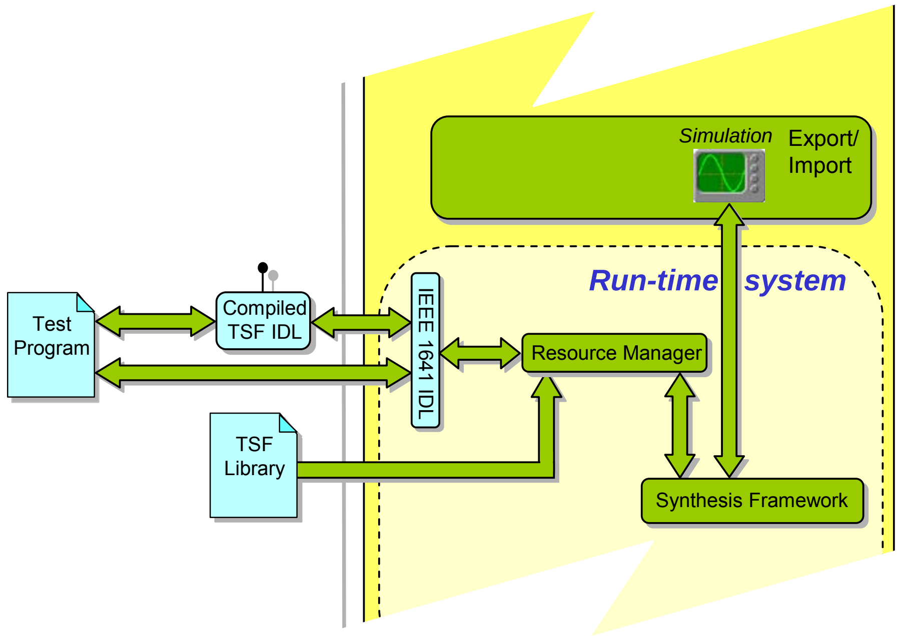


Figure F.17—IEEE 1641 simulator interface

F.2.3.1 UUT Modeling

In order to create the test simulations, a virtual UUT may be created using the Attenuator BSC, with a gain greater than 1, plus a Limit BSC, as shown in Figure F.18 The Attenuator is used to model the small signal gain (linear region of the amplifier) and the limit provides a basic model of the gain roll-off region of the amplifier.

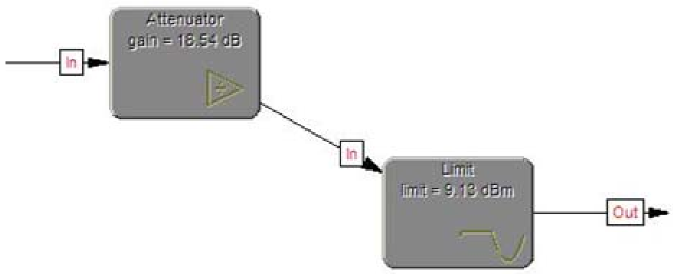


Figure F.18—Modeling the UUT using BSCs

F.2.3.2 Simulation results for 1 dB compression test

The UUT model serves to show that the 1 dB Compression & RF CW Stimulus TSFs are able to detect the output level of the UUT at the point where its gain falls 1 dB below its small signal gain. This is, naturally, the same value as that of the Limit BSC.

F.2.3.3 Gain Simulation Results

The UUT model also shows that the signal model correctly measures the value entered. This may seem obvious because the model for the UUT is a simple "perfect" amplifier, but it does check the TSF models for errors.

F.2.3.4 Gain Transfer Simulation

In order to provide a meaningful simulation of the Gain transfer measurement TSF, real waveform data was downloaded from a Spectrum Analyzer during an execution of a baseline test program created using traditional programming methods. Figure F.19 shows the simulation of the Gain transfer measurement TSF, using actual data from the gain test in this preparation phase. The real waveform data is the lower trace in the graph and the measured power is the dashed upper trace. Again, the simulated measured value for the gain was correctly determined (18.49 dBm in this example) and was consistent with the base-line test results.

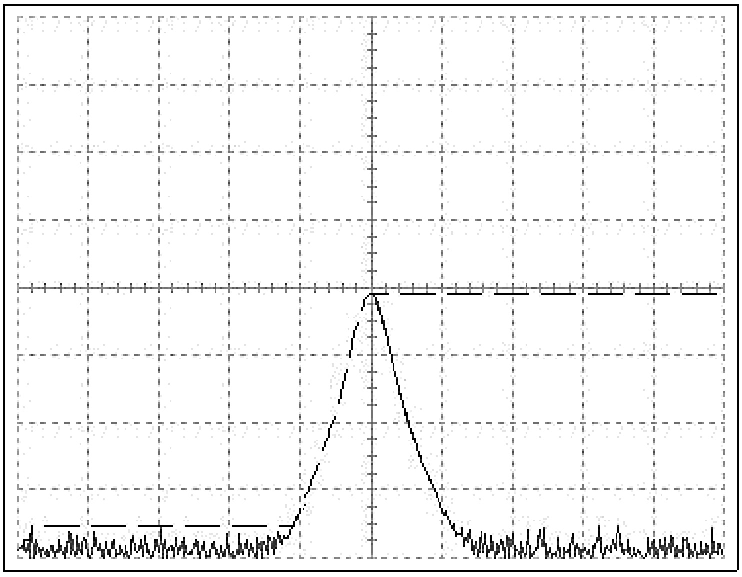


Figure F.19—Gain transfer measurement TSF Simulation

Further explanation is required here to explain how the saved data was used in the simulation. Figure F.20 shows where the real data was injected into the simulated measurement TSF in order to produce the correct measured result. The lower trace (the real waveform data) was used in a WaveformStep BSC and the output was connected into the MaxInstantaneous BSC. The top trace shows the output from the MaxInstantaneous BSC with the measured result being the peak value.

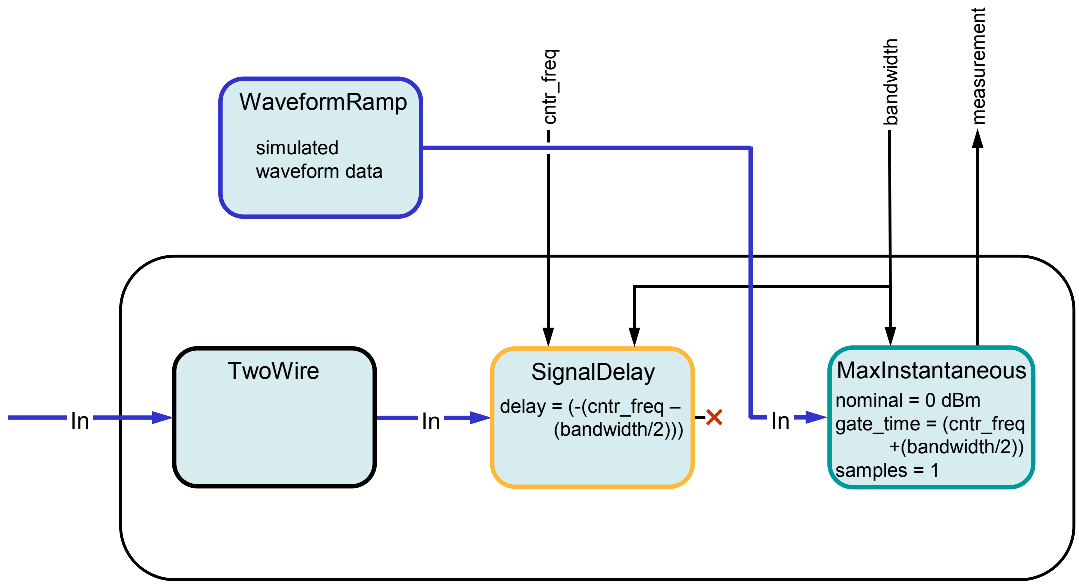


Figure F.20—Simulation of Gain transfer measurement

F.2.4 Validated test program

The 1641 test program used for the simulation may then be used to produce a validated test program, to implement the tests on instruments such as a Vector Signal Generator, a Spectrum Analyzer and a Vector Network Analyzer. An alternative would be to use Synthetic Instruments including a down-converter, digitizer and the appropriate measurement software.

F.2.4.1 IEEE ATML capability description

The use of an instrument capability description, in terms of IEEE 1641 signals, is fully documented in the ATML standards IEEE Std 1761.2 [B12] (Instrument Description) & IEEE Std 1671.6 [B13] (Test Station Description). The Instrument Description standard describes instruments on an individual basis and the Test Station Description standard deals with capabilities that may be the product of more than one instrument.

It is outside of the scope of this Guide to discuss in detail the different methods of describing the instruments and tests stations that may be used to implement the tests. But as an example, it is possible to define descriptions of two different test stations, one using the signal generator and spectrum analyzer, and one using the network analyzer, that use the same (or fully compatible) TSFs as were used in the test definitions.

F.2.4.2 Extracting signal-based capability information from instrument datasheets

There is significant complexity in determining signal-based capability information for such complex instruments as those referenced in this example and this is covered by the ATML standards. For convenience, a brief summary of the issues relevant to this example is provided:

— Spectrum Analyzer—for the measurements in this example, absolute and relative level accuracy are the relevant parameters. Since all measurements are done at one frequency, the level linearity is the key parameter for the accuracy of the gain measurements. Absolute level accuracy is relevant for the compression point measurement.

— Signal Generator—if the test system is calibrated, the signal generator does not contribute to measurement accuracy, because the actual output level of the signal generator is determined during calibration.

— Network Analyzer—the gain measurement is one of the fundamental measurements performed with a network analyzer. Gain accuracy translates to transmission measurement accuracy.

F.2.5 IEEE 1641 to driver translation

The programming interface elements defined by IEEE 1641 (e.g., Require, Run, Change, etc) along with elements for attributes, are used in an XML schema document to capture the native driver code required to implement each of those elements. This enables a translation between IEEE 1641 signal requirements into instrument driver specific code. An instance document using this XML schema is produced, describing the driver functions for the signal generator, spectrum analyzer and vector network analyzer, corresponding to the IEEE 1641 capabilities, defined in the ATML Test Station instance.

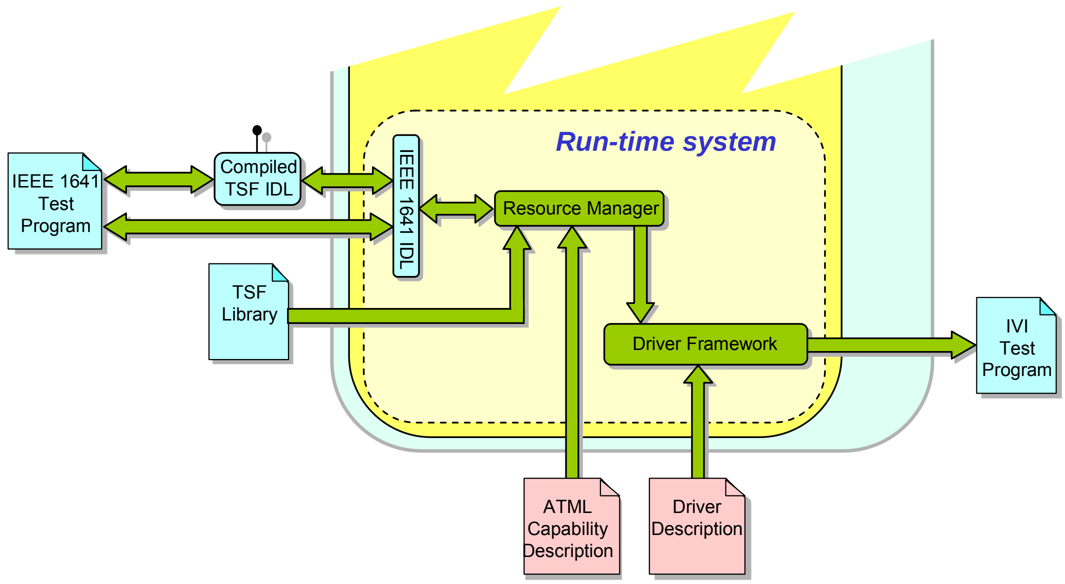


Figure F.21—Signal to Driver Translation Process

Translation from IEEE 1641 to driver code is achieved by matching the signal requirement in the 1641 test program (typically the TSF) to the signal capability in the ATML Test Station document. The resource identified in the Test Station document is located in the driver description and the driver commands are assembled. Figure F.21 shows an illustration of this process. When the test program is executed, the corresponding driver code is output, and may be substituted into the test program to produce to produce the final test program. Figure F.22 provides an example of such code that could be generated by a resource manager.

```
RsRFSigGen.new
RsSpecAn.new

RsSpecAn.Initialize("FSG", true, true, "")
IRsSpecAnBasic = RsSpecAn.Personality.Select("Basic")
RsRFSigGen.Initialize("SMJ", true, true, "")
IRsSpecAnBasic.Frequency.Span = 0
RsSpecAn.System.IO.WriteScpi("CALC:MARK:FUNC:SUM:RMS ON")
RsSpecAnBasic.Acquisition.DetectorType = RsSpecAnBasicDetectorTypeEnum.RsSpecAnBasicDetec
RsSpecAnBasic.SweepCoupling.SweepTime = 0.2
RsSpecAnBasic.Acquisition.SweepModeContinuous = false

RsRFSigGen.RF.Level = <-20>

RsRFSigGen.RF.Frequency = <1500000000>
IRsSpecAnBasic.Frequency.Center = <1500000000>

<nominal> = <20 dB>
RsRFSigGen.RF.OutputEnabled = true
RsRFSigGen.RF.WaitUntilSettled(<3000>)
IRsSpecAnBasic.Level.ReferenceLevel = (RsRFSigGen.RF.Level + <nominal> + 2)
IRsSpecAnBasic.Traces.Initiate
IRsSpecAn.System.WaitForOperationComplete(<3000>)

<measurement_complete> = true

RsRFSigGen.RF.OutputEnabled = false

<measurement> = RsSpecAn.System.IO.QueryScpi(20, "CALC:MARK:FUNC:SUMM:RMS:RES?") - Rsf
```

Figure F.22—Typical code generated by a Resource Manager

F.2.6 Executing the Test Program

The resultant test program can then be run on the selected test equipment with the confidence that the program is a true and accurate interpretation of the original test requirement.

F.2.7 Conclusions

A validated path may be used between IEEE 1641 signal-based test definitions and the resources used to implement them. Furthermore, the implementation of these tests through a standardized system of instrument drivers (such as IVI[10] drivers) presents a portable driver framework that is the native programming environment of choice for the implementer.

In an test requirement such as the example illustrated, the requirement for an input signal specified in terms of power may be better represented through the use of voltage, when applied with reference to a matching impedance.

TSFs (such as the 1 dB compression point TSF) can be implemented in different programmatic sequences, while retaining the same test definition. Different programmatic sequences open up the use of these TSFs to re-implementation to provide improved execution speed.

[10] The IVI Foundation is a consortium that promote standardized specifications for programming test instruments.

The concept of ‘capability requirement’ has been illustrated, which can capture information essential to resource allocation. For example, the current that the UUT might draw leads to a capability requirement on a power supply.

ATML Test Station Description may be used to provide automatic resource allocation, through software tools. This can be supplemented with an XML file describing driver functions, to automatically translate IEEE 1641 into driver code.

F.3 Implementing IEEE 1641—Envelope testing of waveforms

F.3.1 Introduction

It is possible to express a test in which a waveform is checked by specifying the waveform's envelope. The test can then determine whether the waveform remains within that specified envelope. The waveform itself is not specified, only the limits (envelope) which it must not cross.

F.3.2 Basic solution

This is achieved by defining the upper and lower bounds as waveforms. Either WaveformRamp or WaveformStep BSCs may be used. The initial example uses WaveformRamp as it is easier to use for an example with just a few points.

The input waveform is subtracted from the upper envelope, and if the waveform remains below this envelope the output will be zero or negative over the measurement period.

The input waveform is also subtracted from the lower envelope, and if the waveform remains above this envelope the output will be zero or positive over the measurement period.

If the waveform goes outside the envelope over the period of the measurement, an event will be raised which indicates the point at which the error occurred. This is shown graphically in Figure F.23).

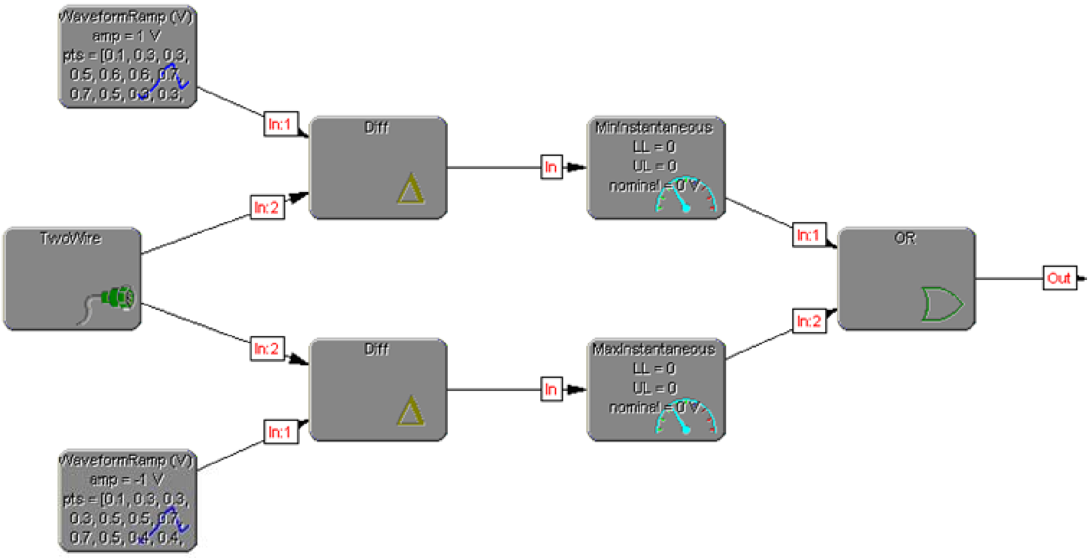


Figure F.23—Signal graph of measurement

This can be presented in XML as follows:

```
<Signal name="Waveform-out-of-envelope-Detector" Out="Any_Event_is_Fail">
   <WaveformRamp name="UpperWaveformEnvelope" amplitude="1 V"
     points="[0.1, 0.3, 0.3, 0.5, 0.6, 0.6, 0.7, 0.7, 0.5, 0.3, 0.3, 0.1]" />
   <WaveformRamp name="LowerWaveformEnvelope" amplitude="-1 V"
     points="[0.1, 0.3, 0.3, 0.3, 0.5, 0.5, 0.7, 0.7, 0.5, 0.4, 0.4, 0.1]" />
   <TwoWire name="SignalInput" channelWidth="1" />
   <Diff name="CompareSignal_to_UpperEnvelope"
     In="UpperWaveformEnvelope SignalInput" />
   <Diff name="CompareSignal_to_LowerEnvelope"
     In="LowerWaveformEnvelope SignalInput" />
   <MinInstantaneous name="LTzero_is_Fail" nominal="0 V" condition="LT"
     In="CompareSignal_to_UpperEnvelope" />
   <MaxInstantaneous name="GTzero_is_Fail" nominal="0 V" condition="GT"
     In=" CompareSignal_to_LowerEnvelope " />
   <OrEvent name="Any_Event_is_Fail" In="LTzero_is_Fail GTzero_is_Fail" />
</Signal>
```

Figure F.24 shows the upper and lower boundaries for the signal under test. This represented in the XML by the two WaveformRamp BSCs "UpperWaveformEnvelope" and "LowerWaveformEnvelope":

```
   <WaveformRamp name="UpperWaveformEnvelope" amplitude="1 V"
     points="[0.1, 0.3, 0.3, 0.5, 0.6, 0.6, 0.7, 0.7, 0.5, 0.3, 0.3, 0.1]" />
   <WaveformRamp name="LowerWaveformEnvelope" amplitude="-1 V
     points="[0.1, 0.3, 0.3, 0.3, 0.5, 0.5, 0.7, 0.7, 0.5, 0.4, 0.4, 0.1]" />
```

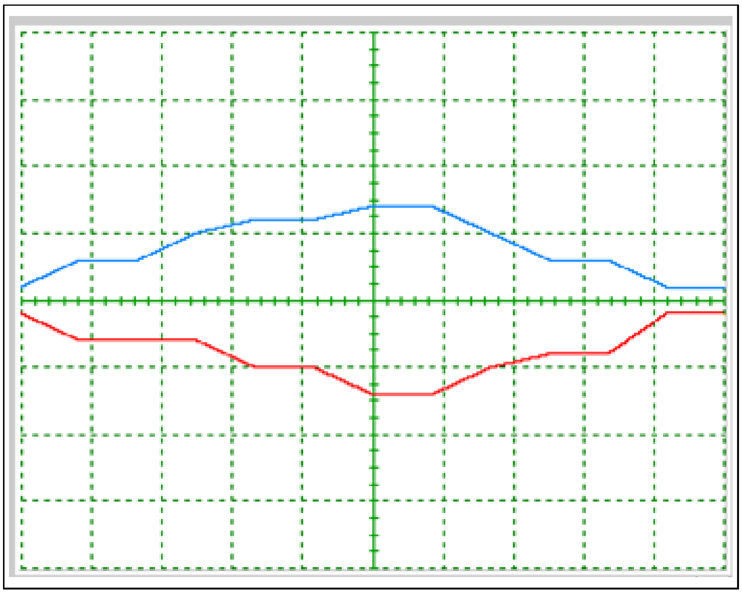


Figure F.24—Test envelope for waveform

Figure F.25 shows a waveform which is required to be checked to see that it falls within the envelope. The envelope is also shown for comparison. It may be seen that the waveform exceeds the envelope at a point on the lower boundary. This waveform is input via the TwoWire connection BSC "SignalInput".

The input waveform is subtracted from the defined boundaries by the two Diff BSCs. The following MinInstantaneous (and MaxInstantaneous) BSCs are used to raise an event if the waveform crosses the boundary. This is shown in the following XML; first for the upper boundary, and then for the lower boundary:

```
<Diff name="CompareSignal_to_UpperEnvelope"
  In="UpperWaveformEnvelope SignalInput" />
<MinInstantaneous name="LTzero_is_Fail" nominal="0 V" condition="LT"
  In="CompareSignal_to_UpperEnvelope" />

<Diff name="CompareSignal_to_LowerEnvelope"
  In="LowerWaveformEnvelope SignalInput" />
<MaxInstantaneous name="GTzero_is_Fail" nominal="0 V" condition="GT"
  In=" CompareSignal_to_LowerEnvelope " />
```

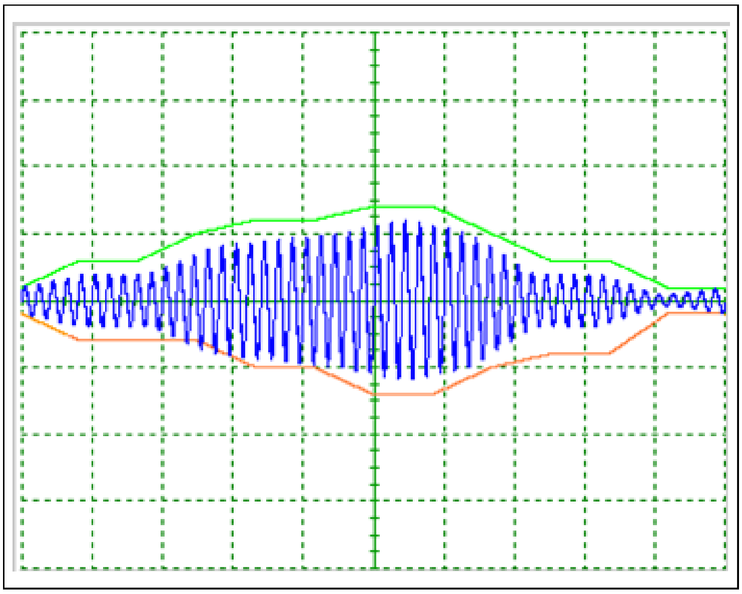


Figure F.25—Test waveform

Figure F.26 and Figure F.27 show the result of the comparison with the upper and lower boundaries respectively. In Figure F.27 it may be seen that the test fails because the waveform (after subtraction) goes positive. That is where the waveform goes below the lower boundary.

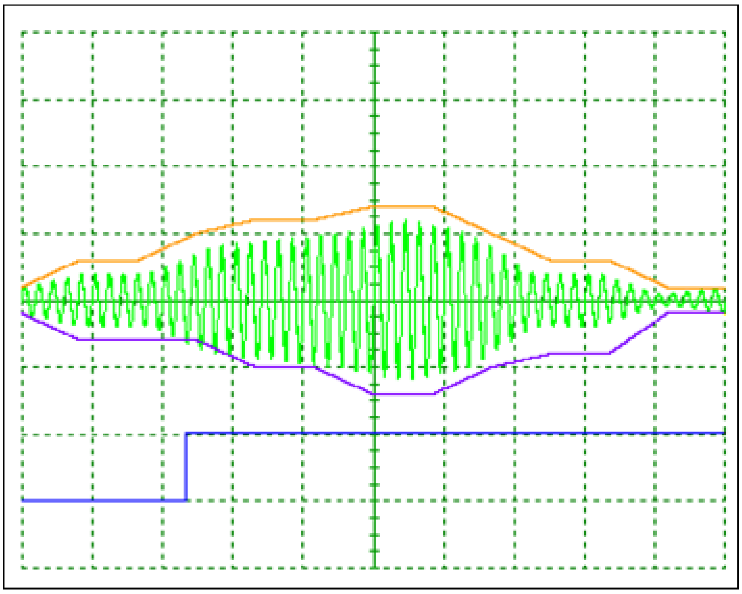


Figure F.26—Upper Boundary Test

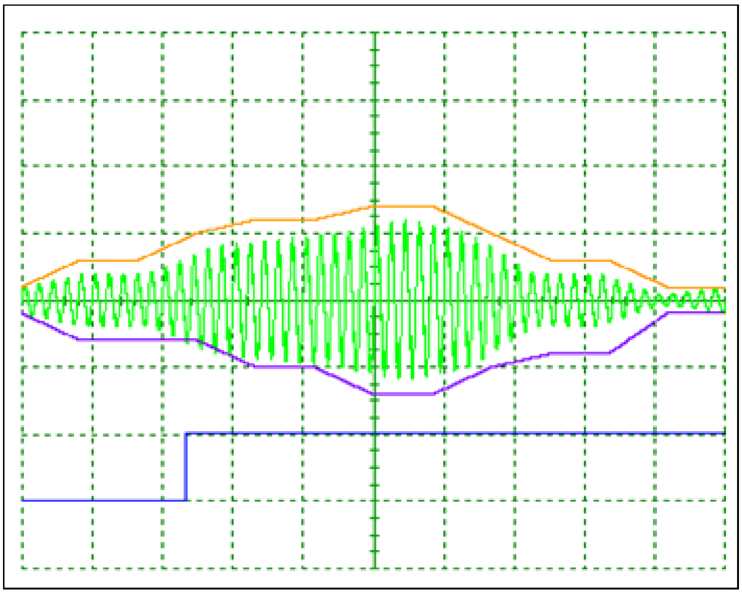


Figure F.27—Lower Boundary Test

The OrEvent BSC (see Figure F.23) is used to detect whether either sensor BSC detects a failure. The OrEvent provides a combined event to indicate a failure.

```
<OrEvent name="Any_Event_is_Fail" In="LTzero_is_Fail GTzero_is_Fail" />
```

shows the failure event adjacent to the input waveform and the envelope boundaries. The event occurs at the first point that the waveform exceeds the permissible envelope which is the point of failure of the test.

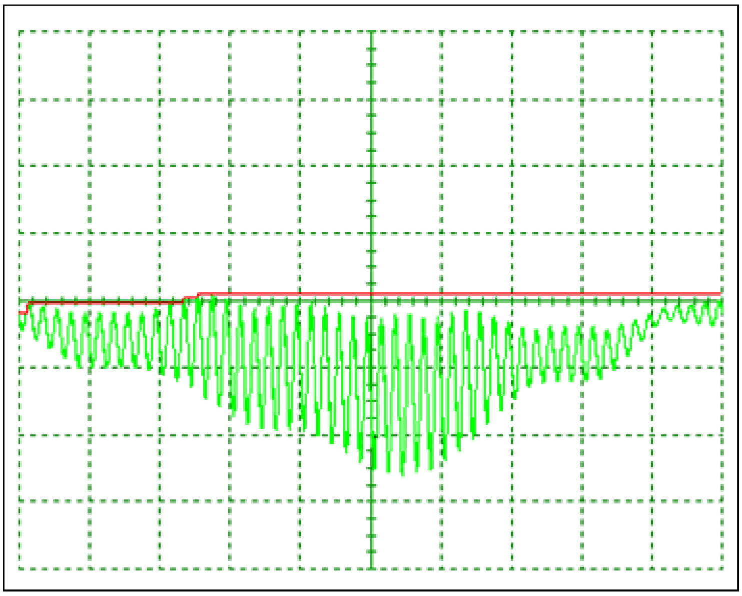


Figure F.28—Test showing point of failure

It is not necessary to generate the event to complete the test. The sensor BSCs may be set with the appropriate Upper Limit (UL = 0 V) and Lower Limit (LL = 0 V) to indicate a failure. The event shows exactly where the first failure occurs in the waveform.

F.3.3 Using WaveformStep BSCs

Figure F.29 and Figure F.30, together with the corresponding XML, show a similar test in which the envelope described with much more detailed WaveformStep boundaries. This is more in line with the way that instruments such as Arbitrary Waveform Generators operate.

The principle is the same as with the previous example.

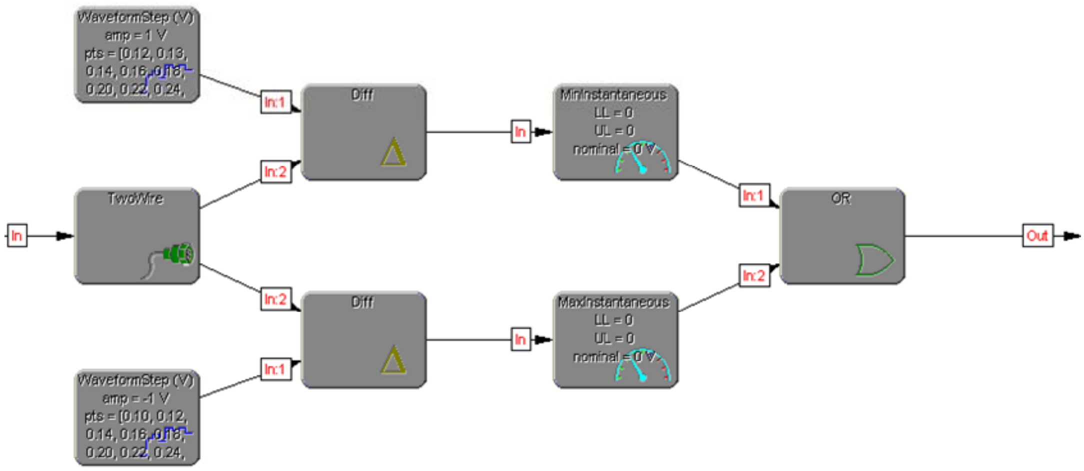


Figure F.29—Signal graph for WaveformStep version

XML Description of signal:

```
<Signal name="Waveform-out-of-envelope-Detector" Out="Any_Event_is_Fail" >
   <WaveformStep name="UpperWaveformEnvelope" amplitude="1 V"
     samplingInterval="8.26446280991736E-03"
     points="[0.12, 0.13, 0.14, 0.16, 0.18, 0.20, 0.22, 0.24, 0.26, 0.28, 0.30, 0.30,
              0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.32,
              0.34, 0.36, 0.38, 0.40, 0.42, 0.44, 0.46, 0.48, 0.50, 0.50, 0.51, 0.52,
              0.53, 0.54, 0.55, 0.56, 0.57, 0.58, 0.59, 0.60, 0.60, 0.60, 0.60, 0.60,
              0.60, 0.60, 0.60, 0.60, 0.60, 0.60, 0.60, 0.60, 0.61, 0.62, 0.63, 0.64,
              0.65, 0.66, 0.67, 0.68, 0.69, 0.70, 0.70, 0.70, 0.70, 0.70, 0.70, 0.70,
              0.70, 0.70, 0.70, 0.70, 0.70, 0.70, 0.68, 0.66, 0.64, 0.62, 0.60, 0.59,
              0.58, 0.57, 0.56, 0.55, 0.40, 0.40, 0.40, 0.40, 0.40, 0.40, 0.40, 0.40,
              0.40, 0.40, 0.40, 0.40, 0.38, 0.36, 0.34, 0.32, 0.30, 0.28, 0.26, 0.24,
              0.22, 0.20, 0.10, 0.10, 0.10, 0.10, 0.10, 0.10, 0.10, 0.10, 0.10, 0.10,
              0.10]" />
   <WaveformStep name="LowerWaveformEnvelope" amplitude="-1 V"
     samplingInterval="8.26446280991736E-03"
     points="[0.10, 0.12, 0.14, 0.16, 0.18, 0.20, 0.22, 0.24, 0.26, 0.28, 0.30, 0.30,
              0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30,
              0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.30, 0.32, 0.34,
              0.36, 0.38, 0.40, 0.42, 0.44, 0.46, 0.48, 0.50, 0.50, 0.50, 0.50, 0.50,
              0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.52, 0.54, 0.56, 0.58,
              0.60, 0.62, 0.64, 0.66, 0.68, 0.70, 0.70, 0.70, 0.70, 0.70, 0.70, 0.70,
              0.70, 0.70, 0.70, 0.70, 0.70, 0.70, 0.69, 0.68, 0.67, 0.66, 0.65, 0.64,
              0.63, 0.62, 0.61, 0.60, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.50,
              0.50, 0.50, 0.50, 0.50, 0.48, 0.46, 0.44, 0.42, 0.40, 0.38, 0.36, 0.34,
              0.32, 0.30, 0.20, 0.20, 0.20, 0.20, 0.20, 0.20, 0.10, 0.10, 0.10, 0.10,
              0.10]" />
   <TwoWire name="InputSignal" lo="Pin2" hi="Pin1" channelWidth="1" />
   <Diff name="CompareSignal_to_UpperEnvelope"
```

```
      In="UpperWaveformEnvelope InputSignal" />
    <Diff name="CompareSignal_to_LowerEnvelope"
      In="LowerWaveformEnvelope InputSignal" />
    <MinInstantaneous name="LTzero_is_Fail" nominal="0 V" condition="LT"
      In="CompareSignal_to_UpperEnvelope" />
    <MaxInstantaneous name="GTzero_is_Fail" nominal="0 V" condition="GT"
      In="CompareSignal_to_LowerEnvelope" />
    <OrEvent name="Any_Event_is_Fail" In="LTzero_is_Fail GTzero_is_Fail" />
</Signal>
```

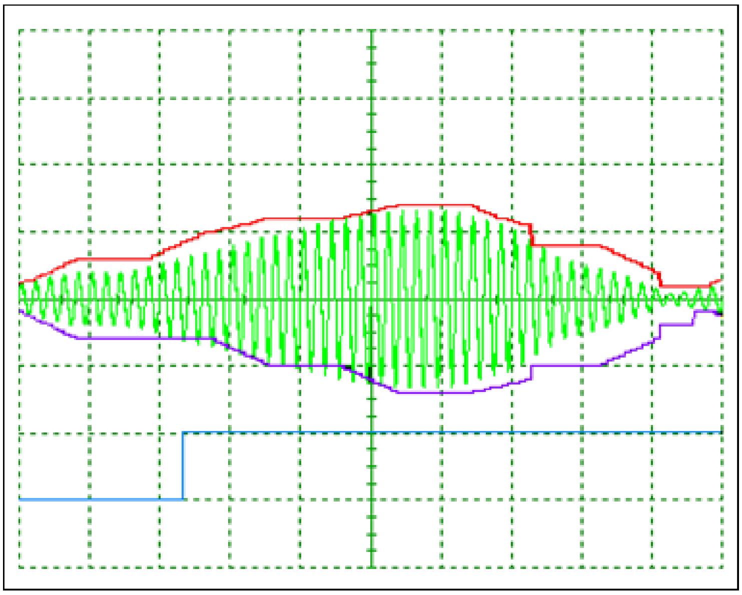


Figure F.30—Waveform and permissible envelope showing point of failure

The XML for the WaveformStep BSCs show a much larger array of data defining the upper and lower waveforms which form the boundaries of the envelope. The example shown is a "fixed" envelope. The test could be converted to a TSF, in which the envelope arrays could be passed to the model as attributes.

The disadvantage with this method is that there are two parallel measurements; the MaxInstantaneous and MinInstantaneous BSCs represent the two sensors which are operating in parallel. It is possible to perform this measurement with a single sensor with upper and lower limits.

F.3.4 Measurement using normalized limits

This method requires more signal processing but is perhaps a more elegant solution. The waveform is measured between normalized limits using a signal graph similar to Figure F.31 which give limits of 0 & 1 provided that the upper limit is never the same as the lower limit.

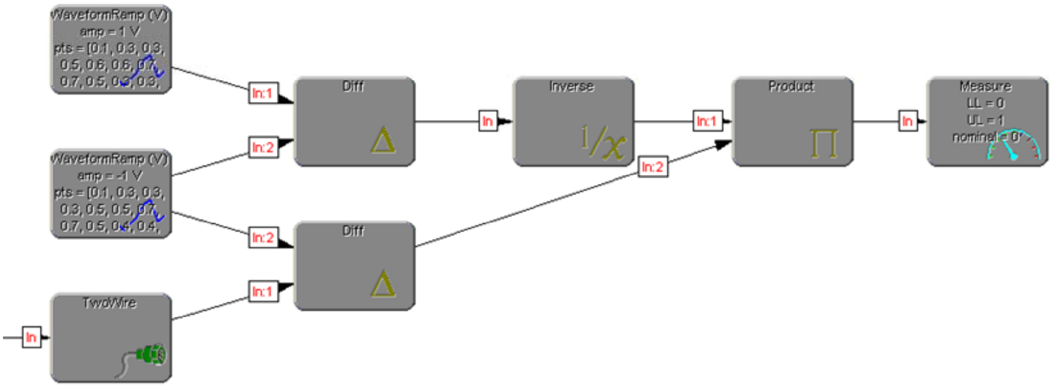


Figure F.31—Signal graph of normalized measurement

Figure F.32 shows the measurement task, with the unspecified waveform to be checked between the upper and lower boundaries.

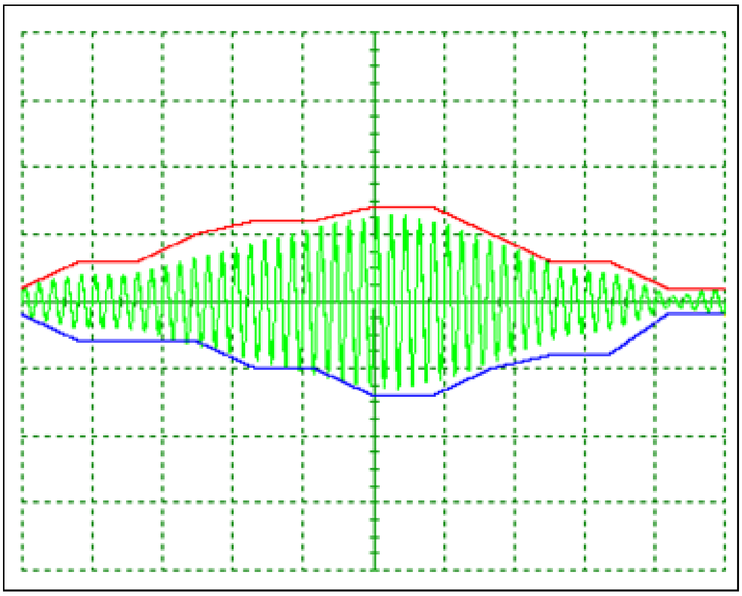


Figure F.32—Measurement task; check waveform remains between boundaries

The lower boundary waveform is subtracted from the input which gives the waveform shown in Figure F.33. The lower boundary waveform is subtracted from the upper boundary waveform to give the upper boundary to the modified waveform (top boundary trace in Figure F.33).

The modified waveform is then normalized by multiplying it with the inverse of the modified upper boundary. This gives the waveform shown in Figure F.34. This waveform may now be checked using a Measure BSC with upper and lower limits of zero and one respectively. If the normalized waveform goes outside of those limits then the test will have failed. The upper and lower waveform traces in Figure F.34 show the maximum excursions of the waveform. It may be seen that the lower waveform goes negative indicating that the test has failed, because the original waveform exceeded the lower boundary of the envelope at that point.

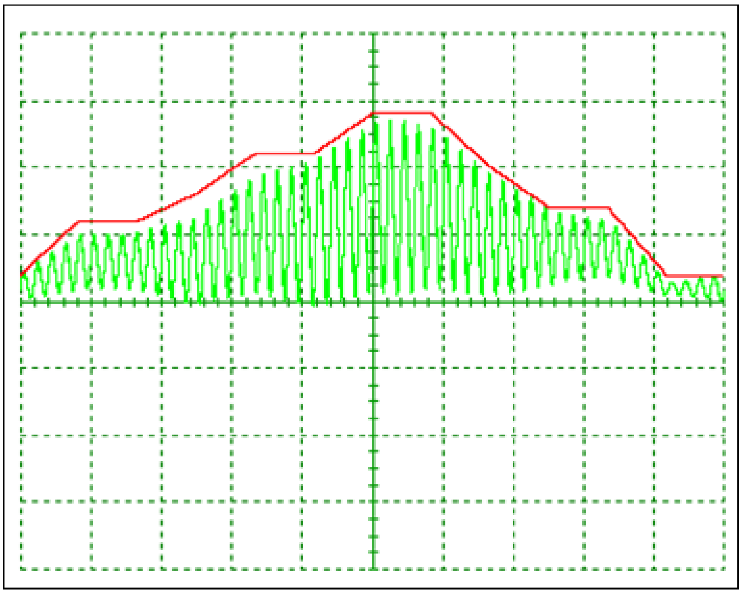


Figure F.33—Output from the two "Diff" BSCs

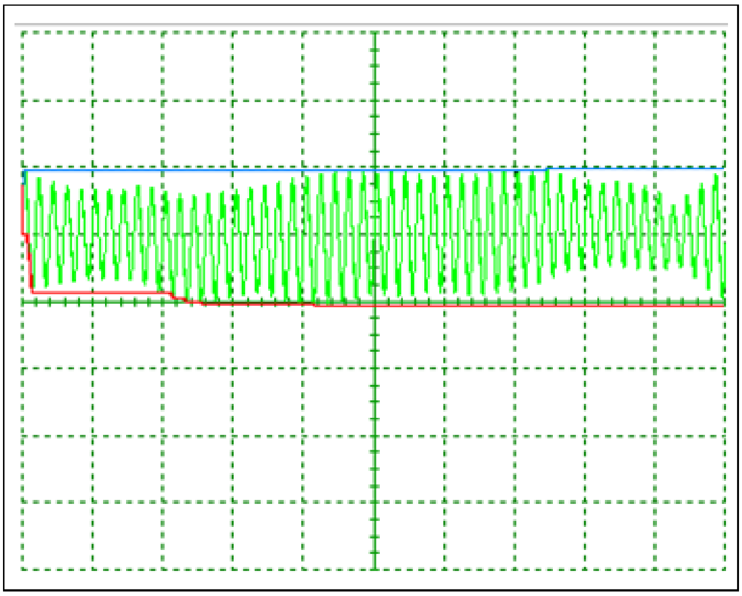


Figure F.34—Normalized measurement

This test technique may be represented as a TSF. The example here is not identical to the signal graph in figure 9 as it uses WaveformStep BSCs. The principles are the same and the TSF may be modified by a user to represent the exact test required.

TSF to measure a normalized waveform:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--  generated with newWaveX v2.8.1.3 (http://www.eads-ts.com)  -->
<tsf:TSFLibrary name="MyTSFLibrary" uuid="MyGUID" version="MyVersion"
```

```
    xmlns="STDBSC" xmlns:tsf="STDTSF"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <tsf:TSF name="Waveform_out-of-envelope_Detector"
       uuid="{AF44B930-15D4-4456-8989-395DB70ADF56}">
       <tsf:interface>
          <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
             elementFormDefault="qualified">
             <xs:element name="Waveform_out-of-envelope_Detector">
                <xs:annotation>
                   <xs:documentation>
                      <!-- 'Waveform_out-of-envelope_Detector' verifies that an incoming
waveform
                              fits within a specified envelope.  -->
                   </xs:documentation>
                </xs:annotation>
                <xs:complexType>
                   <xs:complexContent>
                      <xs:extension base="SignalFunctionType">
                         <xs:attribute name="input" type="BSTR">
                            <xs:annotation>
                               <xs:documentation>
                                  <!-- 'input' is the name of the connector pin to which
the
                                          incoming signal is connected  -->
                               </xs:documentation>
                            </xs:annotation>
                         </xs:attribute>
                         <xs:attribute name="reference" type="BSTR">
                            <xs:annotation>
                               <xs:documentation>
                                  <!-- 'reference' is the name of the connector pin to
which the
                                          incoming signal is referenced (usually the
ground pin).  -->
                               </xs:documentation>
                            </xs:annotation>
                         </xs:attribute>
                         <xs:attribute name="UpperBoundaryAmplitude" type="Physical">
                            <xs:annotation>
                               <xs:documentation>
                                  <!-- 'UpperBoundaryAmplitude' is the nominal amplitude
of the upper
                                          boundary of the envelope '  -->
                               </xs:documentation>
                            </xs:annotation>
                         </xs:attribute>
                         <xs:attribute name="UpperBoundaryArray"
type="SAFEARRAY(VARIANT)">
                            <xs:annotation>
                               <xs:documentation>
                                  <!-- 'UpperBoundaryArray'is an array containing the
points which describe
                                          the shape of the upper boundary of the envelope.
Each value in the
                                          array is relative to the amplitude
'UpperBoundaryAmplitude' -->
                              </xs:documentation>
                            </xs:annotation>
                         </xs:attribute>
                         <xs:attribute name="LowerBoundaryAmplitude" type="Physical">
                            <xs:annotation>
                               <xs:documentation>
                                  <!-- 'LowerBoundaryAmplitude' is the nominal amplitude
of the lower
                                          boundary of the envelope '   -->
                               </xs:documentation>
                            </xs:annotation>
                         </xs:attribute>
                         <xs:attribute name="LowerBoundaryArray"
type="SAFEARRAY(VARIANT)">
                            <xs:annotation>
```

```
                                    <xs:documentation>
                                       <!-- 'LowerBoundaryArray'is an array containing the 
points which describe 
                                            the shape of the lower boundary of the envelope. 
Each value in the 
                                            array is relative to the amplitude 
'LowerBoundaryAmplitude' -->
                                  </xs:documentation>
                                 </xs:annotation>
                              </xs:attribute>
                              <xs:attribute name="measurement_gate" type="Time">
                                 <xs:annotation>
                                    <xs:documentation>
                                       <!-- 'measurement_gate' is the total time over which the 
signal is measured 
                                            (equivalent to the "length" of the envelope)  --
>
                                    </xs:documentation>
                                 </xs:annotation>
                              </xs:attribute>
                           </xs:extension>
                        </xs:complexContent>
                     </xs:complexType>
                  </xs:element>
               </xs:schema>
            </tsf:interface>
            <tsf:model>
               <Signal Out="Normalized_Waveform_Measurement">
                  <TwoWire name="Signal_Input" lo="(reference)" hi="(input)" channelWidth="1" 
/>
                  <WaveformStep name="UpperWaveformEnvelope" 
                     amplitude="(UpperBoundaryAmplitude)" period="(measurement_gate)" 
                     samplingInterval="0s" points="[UpperBoundaryArray]" />
                  <WaveformStep name="LowerWaveformEnvelope" 
                     amplitude="(LowerBoundaryAmplitude)" period="(measurement_gate)" 
                     samplingInterval="0s" points="[LowerBoundaryArray]" />
                  <Diff name="Signal_Difference" In="Signal_Input LowerWaveformEnvelope" />
                  <Diff name="Boundary_Difference" 
                    In="UpperWaveformEnvelope LowerWaveformEnvelope" />
                  <Inverse name="Boundary_Inversion" In="Boundary_Difference" />
                  <Product name="Normalized_Signal" 
                    In="Boundary_Inversion Signal_Difference" />
                  <Measure name="Normalized_Waveform_Measurement" 
                    gateTime="(measurement_gate)" UL="1" LL="0" In="Normalized_Signal" />
               </Signal>
            </tsf:model>
         </tsf:TSF>
      </tsf:TSFLibrary>
```

This TSF has an interface (see table 1) that accepts the envelope definition and the input signal. The output of the TSF is a Pass/Fail status.

Table F.1—TSF Interface for normalized waveform measurement

Description	Name	Type	Default	Range
Signal Under Test	Signal_Input	SignalFunction		
Nominal amplitude of upper boundary	UpperBoundaryAmplitude	Physical		
Array of points defining upper boundary (relative values)	UpperBoundaryArray	array of real		
Nominal amplitude of lower boundary	LowerBoundaryAmplitude	Physical		
Array of points defining lower boundary (relative values)	LowerBoundaryArray	array of real		
time over which the signal is measured	measurement_gate	Time		

The number of points in the arrays will determine the accuracy of the envelope. Each point is a real value which specifies the instantaneous value relative to the nominal amplitude. For example, a nominal amplitude of -1 V will give an actual value of -1.3 V for an array value of 1.3 and an actual value of -0.8

V for an array value of 0.8. The measurement gate is the time over which the waveform is being measured, this is normally the same as the length of the envelope (in time).

F.4 Implementing IEEE 1641—Resource drivers & COTS languages

F.4.1 Introduction

IEEE 1641 is intended to be used within COTS programming languages & development environments to reach as many platforms as possible, through IDL, XML, and TPL. This subclause discusses the use of IDL and XML, since these are open to the greatest and most familiar choice of COTS test languages, compilers and development environments.

IEEE 1641 'Signal & Test Definition' is just that; a method of defining signals (to be applied or measured) and the structure of a test which encapsulates these signals to determine a pass/fail result.

Although signal definitions may be modeled using SML (e.g., to prove their defined behavior) the run-time system is treated as something of a 'black box'. An implementation of an IEEE 1641 run-time system must accept signal definitions through the methods defined in the standard, such as that defined by the IDL informative annex, then control the resources available to implement the test. This is not as complicated as it might first seem, as IEEE 1641 provides a number of information exchange formats which can be leveraged to support a software validation managed path from signal definition to a run-time system's source code project.

F.4.2 IEEE 1641 resource interface

IEEE 1641 defines a Require (<SignalDescriptor>, [UniqueID]) method that a run-time system must support, where the SignalDescriptor is any BSC, TSF, or XML signal definition (anonymous TSF), and the UniqueID allows specific resource information to be specified (typically as a result of some platform-specific pre-compilation activity). The Require method must return an interface that supports manipulation of the signal's attributes, in addition to the IEEE 1641 defined Run(), Change() & Stop() methods.

F.4.2.1 BSCs

In the case where BSCs are used to provide the SignalDescriptor, the standard explicitly defines the interface that must be returned by Require() as illustrated in thee following example of a test language using IEEE 1641 IDL:

```
// Define MyFMSignal
FM fm
Sinusoid sin
fm = Require(AM)
fm.modIndex = 8%
sin = Require(Sinusoid)
fm.In = sin.Out
fm.Out.Run
```

This example uses IDL calls used to construct and execute a test signal directly from BSCs. The key elements, which are explicitly defined by IEEE 1641 & therefore may be checked at compile time, are highlighted in emboldened italics.

F.4.2.2 XML signal definitions

To Require() an XML defined signal, it is possible either to specify the XML in-line with the code or as a URL. The next example shows IDL calls used to construct and execute a test signal using an XML (anonymous TSF) signal definition.

```
// Define MyFMSignal
SignalFunction mySignal
mySignal = Require(
 <Signal name="" Out="AM2" >
   <Sinusoid name="Sin32"/>
   <FM name="AM2" In="Sin32"/>
 </Signal>)
(FM)(mySignal.Item("FM2")).modIndex = 8%
mySignal.Out.Run
```

Again, the key elements, which are explicitly defined by IEEE 1641 & therefore may be checked at compile time, are highlighted in emboldened italics. It can be seen that considerably fewer elements may be checked at compile time. Although a compiler could be developed that could provide this additional checking, or the XML could be validated as part of a separate process. One benefit of an in-line XML checker is that it would validate the attribute values, which is otherwise not possible until run-time.

F.4.2.3 TSFs

TSFs provide for the definition of re-useable signals and measurements (e.g., TACAN, AM_SIGNAL, RADAR_RX, etc.) from BSCs, including their programming interfaces. Such TSFs typically make up a library to be used with a particular group of tests or test platform, but the signals, events, attribute types & connections are all defined within IEEE 1641 and may be similarly represented in a variety of formats such as XML or IDL. In use, TSFs are identical to BSCs but functionally more complex.

Since the use of TSFs allows the interface to test resources to be extended and customized, it is expected that TSFs are to be the typical building blocks for tests called by a test program. Therefore, the creation of resource interfaces that support the manipulation of TSF attributes is a key activity in the development of an IEEE 1641 compliant test.

The next example shows a typical programmatic use of a TSF, where key elements are either explicitly defined by IEEE 1641 or by a TSF library (and therefore implicitly defined by IEEE 1641):

```
MY_FM_SIGNAL myFMSignal
myFMSignal = Require(MY_FM_SIGNAL)
myFMSignal.modIndex = 8%
myFMSignal.Out.Run
```

Once again, the key elements are highlighted in emboldened italics. All the programmatic elements may be checked at compile time and the code is the most concise.

F.4.2.4 Benefits of TSFs in resource interfaces

By comparing the three examples given earlier, it is suggested that the use of TSFs presents two key benefits in writing test code for test resources:

— Maximized and earlier support for checking & debugging

— Concise, maintainable code.

A potential future benefit that might arise out of using in-line XML checkers is the ability to validate attribute values at compile time.

F.4.3 Review of late binding and early binding

Introduction

There are two fundamental ways to use software interfaces to programmatically control another application, such as a test program controlling a resource driver in an ATS run-time system. These are known as 'late binding' and 'early binding':

a) Late binding—a client application attempts to create a connection to a server application at run-time and to call its methods and attributes. It is not necessary to have previously confirmed the existence, methods or attributes of the server application.

b) Early binding—the compilation and usually writing process of a client application that makes reference to a server application's defined interface.

NOTE 1—Though the general terms 'late binding' & 'early binding' are sometimes associated with Basic language variants, neither this discussion nor IEEE 1641 are targeted or restricted to any particular language. For example, in COM 'late binding' and 'early binding' would refer to compiling without and with Dispatch IDs, respectively.

NOTE 2—The client application could perform run-time checking to confirm that the server application does exist and can provide the required methods and attributes.

Both late and early binding techniques are adopted in many test development environments.

F.4.3.1 Advantages of Late Binding

A late binding client does not incorporate information about a specific version of a server interface, so a late binding client/server relationship may be seen as more version-independent than an early bound client/server relationship. This may be seen simply as poor software configuration management.

Some software development environments don't actually support early binding, making late binding code projects more portable than early binding code projects.

Extra server interface information held in early binding code projects, which is not held in late binding code projects, may yield larger file sizes and longer compilation times.

F.4.3.2 Advantages of Early Binding

Early binding leads to faster executions speed for the client application, because the client already knows the location of the server application and its methods & attributes. In late binding, each of the server application's methods & attributes must be located before they can be called.

The server application's interface is known at the time the client application is being written, so debugging and error checking is greatly simplified & quicker because many COTS compilers can check the code automatically. The late binding alternative is manually checking the code, though some specialist compilers may perform this function.

Productivity enhancements in many COTS development environments, such as auto-complete, interface browsing, keyword help & chroma-coding, are supported by creating an early binding to a server application's interface.

F.4.4 Run-time and design-time resource allocation

There are two fundamental methods of allocating test resources on an ATS, as a result of a test requirement; run-time and design-time.

a) Run-time—test resources are automatically allocated while the test program is running, on the basis of a capability match between a test requirement and test resource. This may include compensation for path losses, according to the eventual match.

b) Design-time—test resources are matched to test requirements during an integration phase (this may include some automated processes) and fixed for the life of the test.

TSF oriented test programs will support both these scenarios; a required TSF is either available on the ATS and can be allocated; or is not available on the ATS, but the TSF library can be opened to retrieve the signal model, from which a match can be found.

F.4.5 Exchanging signal model information in different formats

Figure F.35 presents the general model of IEEE 1641 in terms of the four layers. The standard provides definitions for each of the layers such that a verifiable signal model can be obtained from any layer and through a choice of languages.

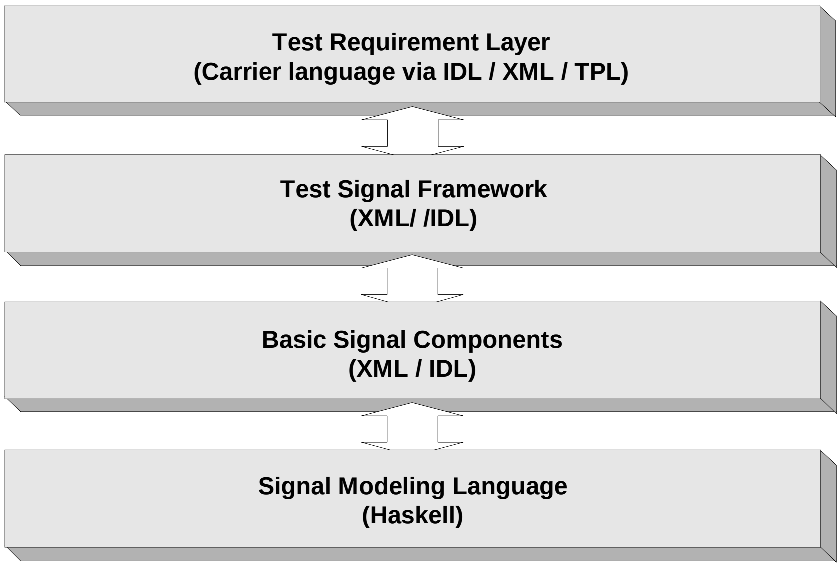


Figure F.35—IEEE 1641 Layers showing information formats

The following list provides additional notes about the layers and the information formats used:

a) Carrier Language— used to sequence the test signals and measurements defined through the layers below. At this layer a mix of all the formats (IDL, XML, and TPL) may be used. For example, the IDL Require() method may accept a uniform resource locator (URL) to an XML file, defining a <signal>.

b) TSF layer—defines both the signal model (in terms of BSCs) and the interface definition. Both signal model and interface can be defined together, using XML, in a TSF library. Furthermore, an XML schema description (XSD) of a TSF interface can be used to validate an XML description that uses that TSF (e.g., ATML Test Description [B11]), but this does not define the behavior of the TSF signal model. IDL can be used to define a TSF interface for compilation against early binding test code, but this does not define the behavior of the TSF signal model. For example, an integrator may want to validate a signal model of a TSF against a particular test resource; whereas, a test programmer might want to validate test code through an early binding compilation with a TSF IDL interface definition.

c) BSC layer—defines BSC interfaces in terms of IDL, XML and TPL, whereas their functional behavior is defined in the terms of the SML layer. For example, a signal modeling application would typically present the BSC layer to support a user in developing signals, but would need to link to the SML layer to generate simulations of the signal models, as described above.

d) SML—typically used only in signal modeling applications; possibly even only to validate the correct behavior of signal modeling applications, which might use quicker and more efficient synthesis frameworks.

Considering the possibilities listed, it may be seen that it is not simply the case that one format may be translated to another, either within the same layer or between layers.

F.4.6 Creating TSF programming interfaces – a worked example

This example is based upon the use of a tool that automates the creation of TSF models. It uses a hierarchical database architecture to encapsulate all the standard's informative data formats and is fronted by a graphical user interface to provide a visual signal modeling environment for TPS. The processes used are not specific to any particular tool, but represent a generic method of creating and using signal models.

F.4.6.1 Signal modeling

Figure F.36 shows a TSF that could be used to measure noise rejection in a radio receiver. The graphical representation is obtained from a drag-and-drop signal modeling environment that supports IEEE 1641 BSCs, and is part of a database architecture which captures the IEEE 1641 four layer model & interchange formats described earlier.

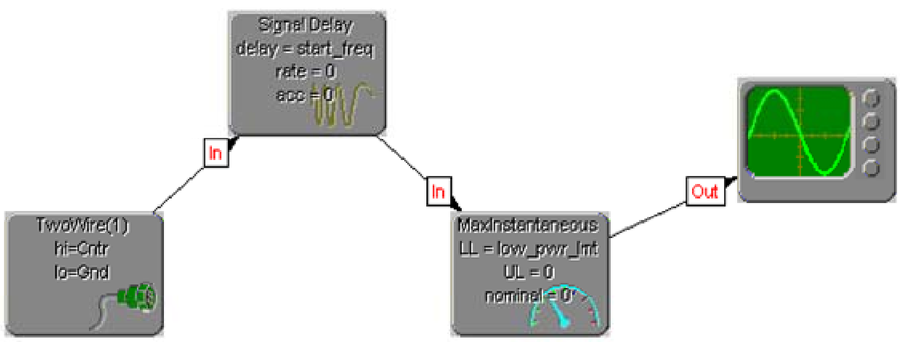


Figure F.36—An example of a TSF for measuring RF noise rejection

F.4.6.2 Signal Verification (Simulation)

Using the layered model, represented in the database architecture, it is possible to map the BSCs within the TSF model to SML, to produce a functional model of the TSF. Figure F.37 shows a simulation created from the functional model described above.

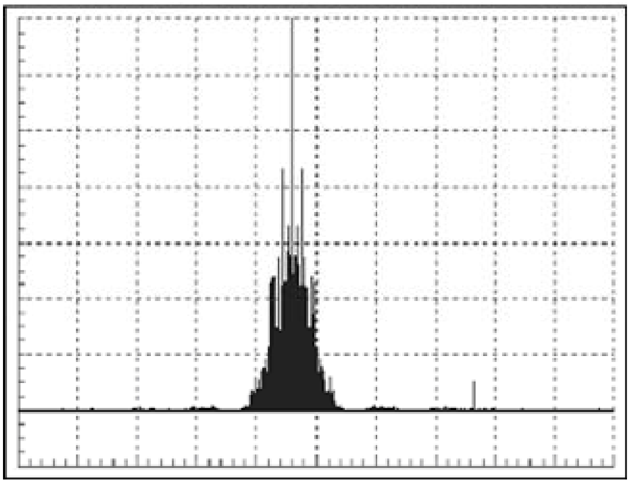


Figure F.37—SML simulation of the TSF model

For signal modeling purposes, this simulation might be used only to confirm that the behavior of the TSF as was intended by the designer; however, such a simulation might also be used to validate the correct operation of a test resource in an integration or even obsolescence management activity.

F.4.6.3 TSF interface definition

Although a TSF signal model refers to the TSF's interface attributes, these are defined separately, along with their types, ranges, default values and annotations. Figure F.38 shows the interface attributes for this example TSF, defined in the database.

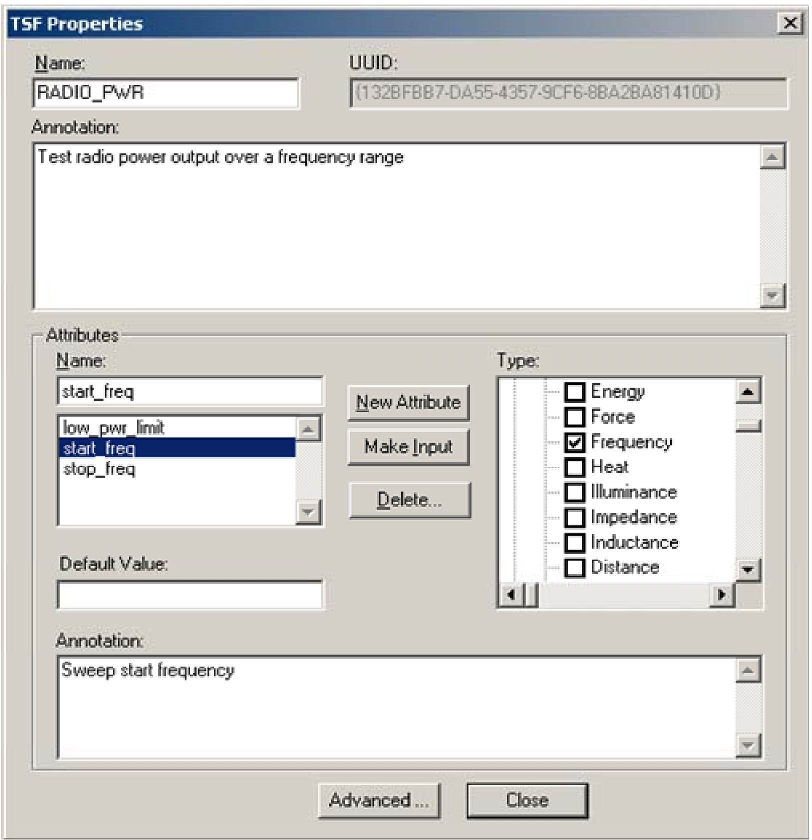


Figure F.38—Defining the TSF interface

F.4.6.4 TSF library generation

The complete TSF is shown below using XML notation. This carries all the essential information required to define the signal. In a user-friendly design environment, additional information is carried in a database to maintain the graphical layout that provides readability and simplifies the process for test engineers.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- created with newWaveX (http://www.newWaveX.com) -->
- <tsf:TSFLibrary xmlns:tsf="STDTSF" xmlns="STDBSC" xmlns:tsf716="STDTSFLib" name="MyTSFLib" uuid="{E6AE6F2C-2FFC-4DA8-8670-
  C48F3790E197}" version="1.0">
  <tsf:description />
  <!-- generated with newWaveX v3.0.11 (http://www.newWaveX.com) -->
- <tsf:TSF name="RADIO_PWR" uuid="{5CDA514B-EC0A-40AF-8283-A936A0A2F41E}" xmlns:tsf="STDTSF" xmlns="STDBSC">
  - <tsf:interface>
    - <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
      - <xs:element name="RADIO_PWR">
        + <xs:annotation>
        - <xs:complexType>
          - <xs:complexContent>
            - <xs:extension base="SignalFunctionType">
              - <xs:attribute name="low_pwr_limit" type="Power">
                - <xs:annotation>
                    <xs:documentation>Low power limit</xs:documentation>
                  </xs:annotation>
                </xs:attribute>
              + <xs:attribute name="start_freq" type="Frequency">
              + <xs:attribute name="stop_freq" type="Frequency">
              </xs:extension>
            </xs:complexContent>
          </xs:complexType>
        </xs:element>
      </xs:schema>
    </tsf:interface>
  - <tsf:model>
    - <Signal name="" Out="pwrSpectrum" xmlns="STDBSC">
        <TwoWire name="BNC" lo="Gnd" hi="Cntr" channelWidth="1" />
        <SignalDelay name="startFreq" delay="start_freq" In="BNC" />
        <MaxInstantaneous type="Power" reftype="Frequency" name="pwrSpectrum" samples="(stop_freq / 500k)"
          gateTime="500kHz" LL="low_pwr_limit" In="startFreq" />
      </Signal>
    </tsf:model>
  </tsf:TSF>
</tsf:TSFLibrary>
```

NOTE—the reftype (i.e. the independent variable) of the BSCs in the TSF signal is frequency. This explains how SignalDelay can be used to define the start frequency of a frequency sweep.

F.4.6.5 TSF reuse using the ATML standards

The TSF library code generated above is a complete definition of the TSF. It can be used to derive the TSF behavior, by extrapolation back to the SML for both the TPS designer and ATS integrator. It can also be used to generate a programming interface to support early binding solutions, such as compile-time debugging and development environments.

With the addition of an XML schema description, the TSF library can be reused in the ATML standards; Test Description [B11] and Instrument Description [B12]. Such an XSD is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated with newWaveX Signal Development  (http://www.newWaveX.com) -->
<xs:schema targetNamespace="MyTSFLib" elementFormDefault="qualified" version="0.0"
xmlns="STDBSC" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:import namespace="STDBSC" schemaLocation="STDBSC.xsd"/>
  <xs:annotation>
    <xs:documentation/>
  </xs:annotation>
  <xs:element name="RADIO_PWR" nillable="false">
    <xs:annotation>
      <xs:documentation>Tests radio power output over a frequency range
    </xs:documentation>
    </xs:annotation>
    <xs:complexType mixed="false">
      <xs:complexContent mixed="false">
        <xs:extension base="SignalFunctionType">
          <xs:attribute name="low_pwr_limit" type="Power">
```

```
              <xs:annotation>
                <xs:documentation>Low power limit</xs:documentation>
              </xs:annotation>
            </xs:attribute>
            <xs:attribute name="start_freq" type="Frequency">
            <xs:attribute name="stop_freq" type="Frequency">
          </xs:extension>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
  </xs:schema>
```

F.4.6.6 Resource driver programming interface

Once the test definition and the instrument description for the test and test system exist, it is then possible to obtain the software interface definition that is compliant with the standard, even though this TSF did not exist until we defined it. That interface is shown below, in terms of the IDL open standard.

```
[
 uuid(E6AE6F2C-2FFC-4DA8-8670-C48F3790E197),
 version(0.0),  helpstring("")
]

library MyTSFLib
{
 // Lines removed for brevity.
 //RADIO_PWR
 [
  object,
  uuid(5CDA514B-EC0A-40AF-8283-A936A0A2F41E),
  dual,
  helpstring("IRADIO_PWR Interface"),
  pointer_default(unique)
 ]

 interface IRADIO_PWR : IMyTSFLib
 {
  enum {RADIO_PWR_BASE=(MYTSFLIB_BASE+256)};

   [propget, id(RADIO_PWR_BASE+1), helpstring("Low power limit")]
    HRESULT low_pwr_limit([out, retval] Power *pVal);
   [propputref, id(RADIO_PWR_BASE+1), helpstring("Low power limit")]
    HRESULT low_pwr_limit([in] Power newVal);
   [propget, id(RADIO_PWR_BASE+2), helpstring("Sweep start frequency")]
    HRESULT start_freq([out, retval] Frequency *pVal);
   [propputref, id(RADIO_PWR_BASE+2), helpstring("Sweep start frequency")]
    HRESULT start_freq([in] Frequency newVal);
   [propget, id(RADIO_PWR_BASE+3), helpstring("Sweep stop frequency")]
    HRESULT stop_freq([out, retval] Frequency *pVal);
   [propputref, id(RADIO_PWR_BASE+3), helpstring("Sweep stop frequency")]
    HRESULT stop_freq([in] Frequency newVal);
 };

 [
  uuid(5CDA514B-EC0A-40AF-8283-A936A0A2F410),
  helpstring("RADIO_PWR class"),
  noncreatable
 ]

 coclass RADIO_PWR
 {
   [default] interface IRADIO_PWR;
 };
```

This IDL definition of the TSF's interface may usefully be used even before a driver is coded, since it can be compiled into a type library. A type library is useable to support test code compilation, even before the drivers are written, thus streamlining the test writing process. In the test development environment, this type library (in some cases this will need to be compiled as an empty driver stub) will support all the productivity features of modern COTS applications; auto complete, interface browsing, keyword help, chroma-coding, etc.

Sufficient information also exists in the TSF library to create a simulation of the eventual run-time system, via the mapping to SML. This is a further tool that can be used to aid the debugging process. Furthermore, the other informative formats, provided by the standard, enable the source signal model to be available to integration, obsolescence management and interoperability activities, throughout the life of the test.

F.4.6.7 Coding the Resource Driver

As has been illustrated, an IEEE 1641 compliant driver interface can be automatically obtained from a TSF library, using tools that capitalize upon the layered & multi-lingual structure of the standard. This interface can be expressed in a number of formats, such as XML, for use in the ATML group of standards. It is the IDL format that is most important for driver writing.

Compilation of IDL can be achieved on a variety of platforms, using COTS tools, to create a type library, with which a driver code stub can be created.

Implementation of the driver is dependent upon the specific resources available on a given test system, but the correct behavior of that implementation can be confirmed through the links to and use of synthesis and simulation provided by the standard's SML.

F.4.7 Conclusions

Of the various methods for programmatic control of test resources, IEEE 1641 TSFs through IDL provide the greatest support for design time checking & debugging. In addition to this, coding with TSFs yields the most concise and maintainable code. Some specialist compilers may perform this checking using other techniques.

Many additional features in commercially available development environments, such as auto-complete, interface browsing, keyword help & chroma-coding, are supported by TSF IDL and simplify the test development and integration process.

IEEE 1641 TSF oriented test programs will support both run-time and design time resource allocation through the XML defined signal models. Although, both allocation and coding may be greatly simplified where an ATS has a particular purpose or test domain for which a fixed set of TSFs can be provided.

IEEE 1641 TSF defined ATS may support both instrument obsolescence management and test interoperability through these TSFs and their rigorous definition via SML.

IEEE 1641 layered & multi-lingual structure allows a link between test descriptions, test code, instrument descriptions and resource drivers, which supports validation between any of these deliverable items.

F.5 Implementing IEEE 1641—Compilation techniques

F.5.1 Introduction

One of the major objectives of IEEE 1641 is to promote "openness", i.e. to provide a method of supporting fully transparent test requirements that may be fully understood and easily ported to different environments. This section discusses the use of IEEE 1641 with IDL and XML, as these provide access to the greatest choice of COTS test languages, compilers and development environments.

F.5.2 IEEE 1641 resource interface

IEEE 1641 defines a Require (<SignalDescriptor>, [UniqueID]) method, where the SignalDescriptor is any BSC, TSF, or XML signal definition (anonymous TSF), and the UniqueID allows specific resource information to be specified (typically as a result of some platform-specific pre-compilation activity). The Require method must return an interface that supports manipulation of the signal's attributes, in addition to the IEEE 1641 defined Run(), Change() & Stop() methods.

The following code is an example of the IEEE 1641 defined interface being used to construct and execute a test signal, where the test signal (FM_SIGNAL) is defined in a TSF library, loaded into the test system. The TSF library format is defined by 1641 and may be expressed using XML, XSD or IDL.

```
// Using my TSF library
FM_SIGNAL myFMSig
myFMSig = Require(FM_SIGNAL)
myFMSig.modIndex = 8%
myFMSig.Out.Run
```

IDL is an industry standard interface definition language, in this case used to define the object type, FM_SIGNAL. XML is used to describe the complete TSF behavior in terms of IEEE 1641 signals, enabling the test system to determine a resource capable of implementing that TSF, where the test resources are also described in terms of 1641 signals (e.g., using ATML Test Station Description [B13]).

The XML code below shows the same 1641 signal as part of an ATML Test Description. Test Description's "Operations" are based on IEEE 1641's TPL, an equivalent to 1641's IDL interface. Hence, 'OperationSetup' and 'OperationConnect' map onto the IDL Require and Run, respectively.

```
<!-- ATML Test Description -->
<Operation xsi:type="OperationSetup" … >
   <Sensor>
      <LocalSensorSignalReference localSignalID="ls1"/>
      <std:Signal>
         <myTSFLib:FM_SIGNAL name="myFMSig" modIdx="8 %"/>
      </std:Signal>
   </Sensor>
   ...

<Operation xsi:type="OperationConnect" ... >
   <Signal>
      <LocalSignalReference localSignalID="ls1"/>
   </Signal>
```

In this case, an XSD is used to define the interface (attributes) of elements with namespace prefix myTSFLib.

An ATML Test Description is not a test program, but describes a test that is to be implemented by a test program. The process of generating test programs from ATML Test Description will hinge on the ability to translate its signal parts into executable code.

The next example is a TPL version of the same test program.

```
//<TPL>
//Setup FM_SIGNAL modIdx 8 %
//          as source myFMSig;
//Connect myFMSig …
//          as connection toUUT;
//</TPL>
```

TPL is targeted at test programming languages that do not support IDL interfacing technologies (typically legacy), such as C, and is intentionally familiar to ATLAS users. In use, TPL is formed in the comment fields of the test program and must be translated to the same carrier language before the carrier's own compiler can process it.

Note that ATML Test Description's 'Operations' and 1641's TPL & IDL methods are equivalent, so the tests described in the three blocks of code shown above are equivalent.

F.5.3 TSF programming interface

The interface for running a test program is defined by IEEE 1641. The BSCs also have defined interfaces. TSFs, which may be quite complex in some cases, consist of a model and an interface. The interface may be expressed in a variety of formats, to suit the application. The TSF design tool will usually facilitate the export of the TSF data in XML, XSD or IDL files.

Figure F.39 shows a signal development system that links a graphical development and simulation environment with a variety of different file format exports. Behind the scenes, the graphical development environment is rule-based and ensures the validation of the signals that it is used to develop.

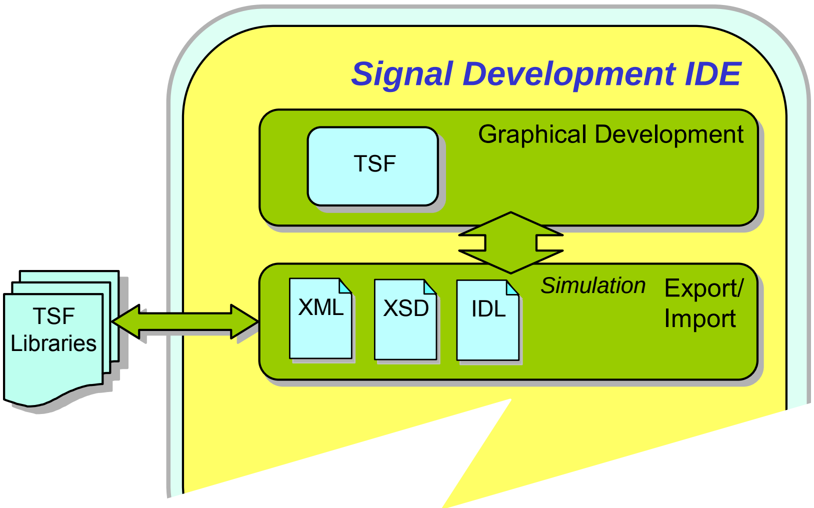


Figure F.39—TSF signal modeling tool

The exported IDL may be compiled using a COTS compiler to create a type library supporting the use of the TSF in test programs. The same type library may be used to provide a TSF wrapper interface to a translation framework, as shown in Figure F.40.

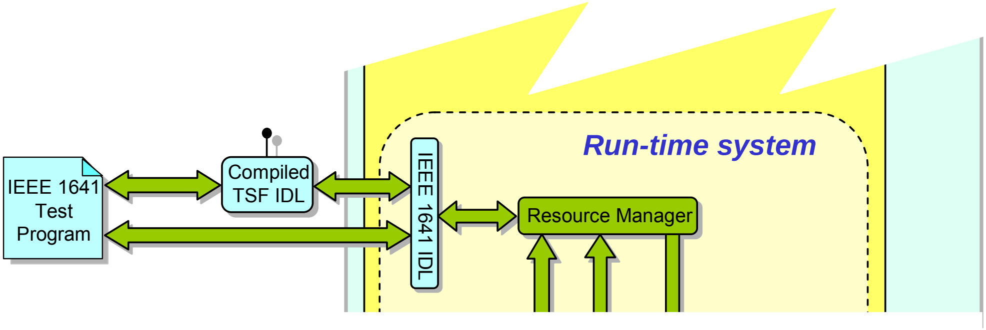


Figure F.40—IEEE 1641 core interface supplemented by an exported TSF interface

F.5.4 ATML capability description

IEEE ATML provides an XML format with which to describe a resource's capabilities in terms of signals, using IEEE 1641, a basic example of which is shown below. This capability description may be written in terms of the TSFs expected to be used on a test station or more generically, using Basic Signal Components.

```
 <hc:Capability name="FMSignal">
   <hc:Interface/>
   <hc:SignalDescription>
      <std:Signal Out="odcp1">
         <tsf:FM_SIGNAL name="fmSig" modIdx="range MAX 30 %"/>
      </std:Signal>
   </hc:SignalDescription>
</hc:Capability>
```

The first stage of implementing an IEEE 1641 test signal requirement is a mapping of the test signal onto a signal found in the capability description. This applies whether it occurs at run-time, during a pre-allocation step or during translation/compilation.

IEEE 1641 SML provides a means to validate that these capability descriptions against the performance of the resources they describe.

F.5.5 Mapping signals onto driver functions

F.5.5.1 1641 Interface

IEEE 1641's defined core programming interface provides a number of functions (see F.5.2). These include:

— Require <Signal>, [Allocation Information]

— Run [timeout]

— Change [timeout]

— Stop [timeout]

Furthermore, each BSC has its property interface defined by the standard, as illustrated in the following RMS example.

```
RMS:        nominal
                        condition
                        gateTime
                        samples
                        UL
                        LL
                        measuredVariable
```

In the case where the required signal is specified using XML (known as an ‘Anonymous TSF’), an additional function retrieves individual elements of that signal:

```
Item <Signal Name>
```

The returned signal element may then be referred to via its BSC interface.

These core functions and properties are all that is needed to interpret any IEEE 1641 signal, as any TSF interface wrapper (see F.5.3) can be mapped through these functions.

F.5.5.2 Driver Interface

Through IEEE 1641 Signal Modeling and the ATML Capability Description, it is possible to determine the driver functions that will implement their associated signal attributes. Figure F.41 illustrates the structure of the actions required to implement a given resource capability. Two additional actions, beyond those of IEEE 1641, are required to ‘Create’ and ‘Destroy’ instance data, in the case of unmanaged code.

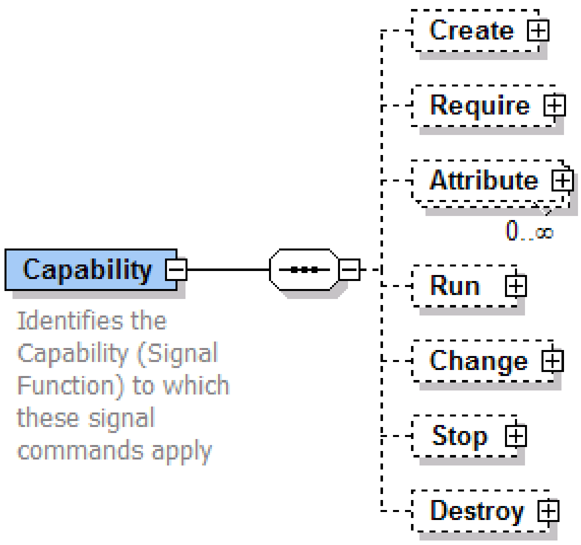


Figure F.41—Actions required to implement capabilities

Each action then breaks down into two types of driver command:

F.5.5.2.1 Command

Figure F.42 illustrates a possible structure of the driver commands required to implement an action, such as switch on an instrument's output or read back a value. Additionally, a formula may be required.

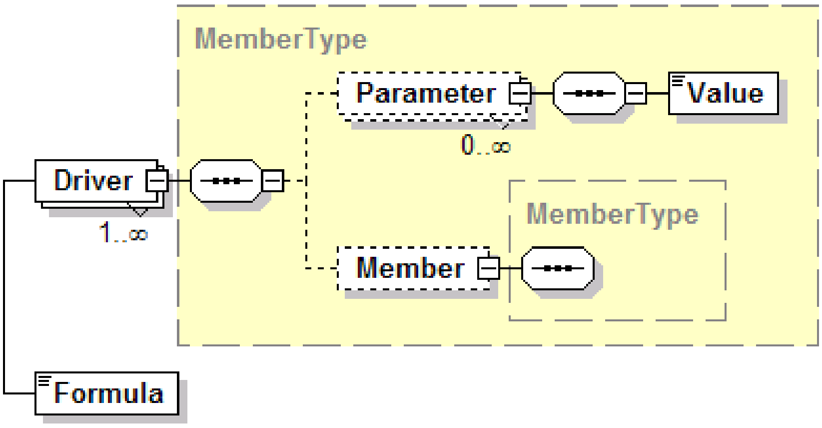


Figure F.42—Driver command

A formula may be specified using the same terms found in IEEE 1641, e.g.:

```
"{{MeasOutputPower.measurement} - {input_pwr}}"
```

As each of these attributes are mapped to actions within the same instance document, their driver commands can be determined.

F.5.5.2.2 Assignment

Figure F.43 illustrates a possible structure of the driver commands required to implement an assignment action. In this structure, assignment may either be a value or a value obtained through a further driver command.

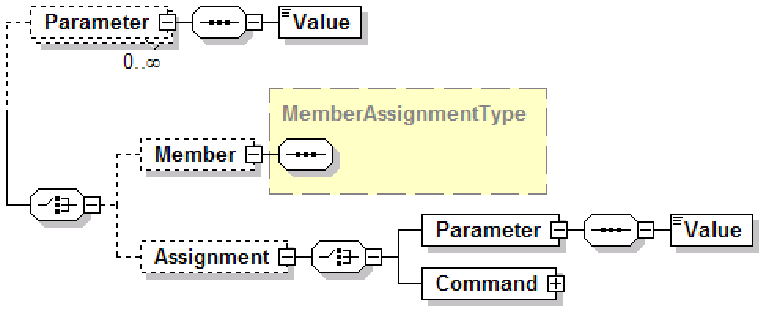


Figure F.43—Driver assignment

F.5.5.3 In use

The structure described above was implemented in the system used to perform the amplifier characterization tests described in F.2.

For example:

```
myODCP.input_pwr.magnitude = dPower;
```

Would return the driver string:

```
RsRFSigGen.RF.Level = <-20>
```

The calling program could equally be an application parsing an ATML Test Description document or an application interpreting TPL statements. The processes described here work equally for IDL COTS test programs, ATML Test Descriptions and TPL test programs.

For the translator, it is not possible to determine whether the assignment value, "-20", is a literal or a string, so it is returned as <-20>, denoting a possible variable.

F.5.6 Translation framework

The application framework for the implementation of the translation process, using the driver command structure described in F.5.3, is shown in Figure F.44.

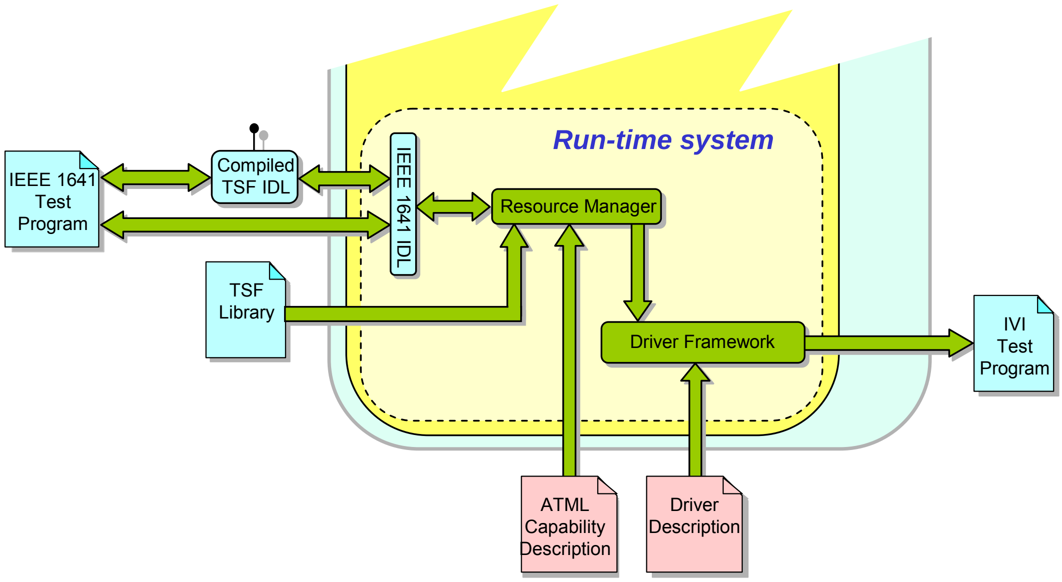
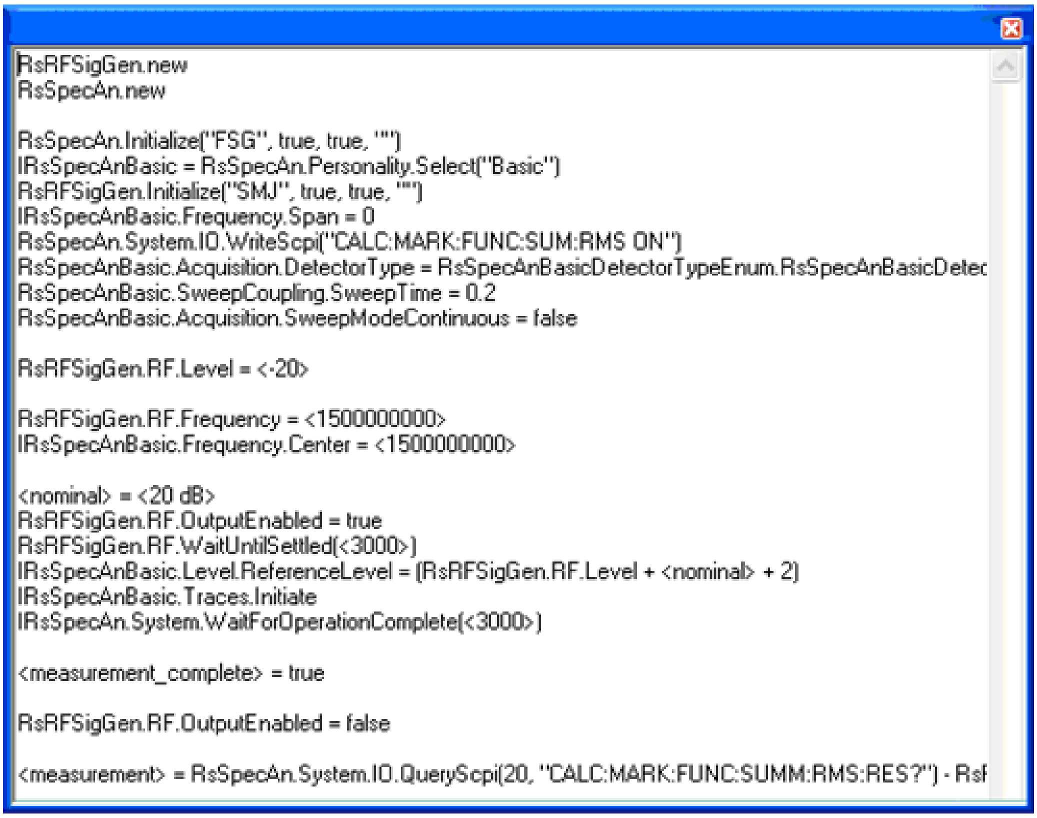


Figure F.44—Signal to driver translation framework

Translation is achieved by matching the signal requirement, supplied through the IDL interface, to the signal capability in the ATML Capability Description document. The resource identified is located in the in the Driver Description document and the driver commands are assembled.

The C# test programs, written for the amplifier characterization tests described in F.2. were executed against ATML Capability Descriptions and Driver Descriptions for a signal generator, spectrum analyzer & vector network analyzer. The sample of the driver code output from this process is shown in Figure F.45. This driver code may then be substituted into the test program to produce the final driver-based test program.



```
RsRFSigGen.new
RsSpecAn.new

RsSpecAn.Initialize("FSG", true, true, "")
IRsSpecAnBasic = RsSpecAn.Personality.Select("Basic")
RsRFSigGen.Initialize("SMJ", true, true, "")
IRsSpecAnBasic.Frequency.Span = 0
RsSpecAn.System.IO.WriteScpi("CALC:MARK:FUNC:SUM:RMS ON")
RsSpecAnBasic.Acquisition.DetectorType = RsSpecAnBasicDetectorTypeEnum.RsSpecAnBasicDetec
RsSpecAnBasic.SweepCoupling.SweepTime = 0.2
RsSpecAnBasic.Acquisition.SweepModeContinuous = false

RsRFSigGen.RF.Level = <-20>

RsRFSigGen.RF.Frequency = <1500000000>
IRsSpecAnBasic.Frequency.Center = <1500000000>

<nominal> = <20 dB>
RsRFSigGen.RF.OutputEnabled = true
RsRFSigGen.RF.WaitUntilSettled(<3000>)
IRsSpecAnBasic.Level.ReferenceLevel = (RsRFSigGen.RF.Level + <nominal> + 2)
IRsSpecAnBasic.Traces.Initiate
IRsSpecAn.System.WaitForOperationComplete(<3000>)

<measurement_complete> = true

RsRFSigGen.RF.OutputEnabled = false

<measurement> = RsSpecAn.System.IO.QueryScpi(20, "CALC:MARK:FUNC:SUMM:RMS:RES?") - RsI
```

Figure F.45—Sample driver code

F.5.7 Test program generation

The output generated in F.5.7 is not a syntactically correct test program and, of course, syntax varies across programming languages. What remains is for the full test program to be generated, applying the appropriate syntax to the output driver code.

The approach presented in example shows an implementation that can be applied across a broad range of test programming languages. Through the IEEE 1641 defined interfaces, this same translation module may be embedded in a wide variety of test program development environments, to add an IEEE 1641 to native driver translation service.

Annex G

(informative)

Bibliography

[B1] ARINC Specification 626-3, Standard ATLAS Language for Modular Test.[11]

[B2] Distributed Computing Environment (DCE) Specifications, The Open Group.

[B3] Hudak, P., Peterson, J., and Fasel, J. H., *A Gentle Introduction to Haskell 98*, 1999 (www.haskell.org).

[B4] Internet Assigned Numbers Authority (IANA), Registered character set, windows-1252.

[B5] IEEE Standards Dictionary: Glossary of Terms & Definitions, Standards Information Network, 2009.[12] [13]

[B6] IEEE/ASTM SI 10, American National Standard for Use of the International System of Units (SI): The Modern Metric System.

[B7] IEEE Std 181™-2011, IEEE Standard on Transmissions, Pulses, and Related Waveforms.

[B8] IEEE Std 260.1™-2004, IEEE Standard Letter Symbols for Units of Measurement (SI Units, Customary Inch-Pound Units and Certain Other Units).

[B9] IEEE Std 716™-1995, IEEE Standard Test Language for All Systems—Common Abbreviated Test Language for All Systems (C/ATLAS).

[B10] IEEE Std 1671™-2010, Standard for Automatic Test Markup Language for Exchanging Automatic Test Equipment and Test Information via XML.

[B11] IEEE Std 1671.1™-2009, IEEE Trial-Use Standard for Automatic Test Markup Language (ATML) for Exchanging Automatic Test Information via XML: Exchanging Test Descriptions.

[B12] IEEE Std 1671.2™-2008, IEEE Trial-Use Standard for Automatic Test Markup Language (ATML) for Exchanging Automatic Test Equipment and Test Information via XML: Exchanging Instrument Descriptions.

[B13] IEEE Std 1671.6™-2008, IEEE Trial-Use Standard for Automatic Test Markup Language (ATML) for Exchanging Automatic Test Information via XML: Exchanging Test Station Information.

[B14] ISO/IEC 8859-1:1997, Information Technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1.[14]

[11] ARINC publications are available from ARINC Research Corporation, Document Section, 2551 Riva Rd., Annapolis, MD 21401.

[12] IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).

[13] The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[14] ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch/). ISO/IEC publications are also available in the United States from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112, USA (http://global.ihs.com/). Electronic copies are available in the United States from the American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

[B15] Jones, S. P. (ed.), *The Haskell 98 Language and Libraries The Revised Report,* Cambridge University Press, 2003.

[B16] TIA-232, Interface Between Data Terminal Equipment (DTE) And Data Circuit-Terminating Equipment (DCE) Employing Serial Binary Data Interchange.[15]

[B17] The Unicode Consortium, "The Unicode Standard," Version 4.1.0, defined by *The Unicode Standard, Version 4.0* (Boston: Addison-Wesley, 2003), as amended by *Unicode 4.0.1* and by *Unicode 4.1.0* (http://www.unicode.org/versions/Unicode4.0.1 and http://www.unicode.org/versions/Unicode4.1.0).

[15] TIA publications are available from Global Engineering Documents, 15 Inverness Way East, Englewood, CO 80112-5704, USA (http://global.ihs.com/).